THE EXPERT'S VOICE®

SECOND EDITION

Pro Git

EVERYTHING YOU NEED TO KNOW ABOUT GIT

Scott Chacon and Ben Straub

Apress®

Pro Git

Scott Chacon, Ben Straub

Table of Contents

Licence

Ön söz: Scott Chacon

Pro Git'in ikinci buraxılışına xoş gəlmisiniz. İlk nəşr artıq dörd il əvvəl nəşr olundu. O vaxtdan bəri çox şey dəyişdi, amma bir çox vacib şey dəyişmədi. Əsas əmrlər və konsepsiyaların əksəriyyəti bu gün də qüvvədə olduğu üçün Git əsas komandası şeyi geriyə uyğun tutmaq üçün olduqca fantastik olduğundan, Git ətrafındakı cəmiyyətdə bəzi əlavə və dəyişikliklər olmuşdur. Bu kitabın ikinci n əşri bu dəyişiklikləri aradan qaldırmaq və yeni istifadəçi üçün daha faydalı ola bilməsi üçün kitabı yeniləmək üçündür.

İlk buraxılışı yazdığımda, Gitin istifadəsi hələ nisbətən çətin idi və daha çətin nüvəli hacker üçün çətinliklə tətbiq olundu. Müəyyən icmalarda buxarlanmağa başladı, lakin bu gün olduğu hər yerə yaxın bir yerə çatmamışdı. O vaxtdan bəri demək olar ki, hər bir açıq mənbəli icma bunu qəbul etdi. Git, Windows-da, bütün platformalar üçün qrafik istifadəçi interfeyslərinin partlaması, IDE d əstəyi və iş istifadəsində inanılmaz bir irəliləyiş əldə etdi. Dörd il əvvəlki Pro Git bunun heç birini bilmir. Bu yeni nəşrin əsas məqsədlərindən biri də Git cəmiyyətindəki bütün yeni sərhədlərə toxunmaqdır.

Git istifadə edən Açıq Mənbə cəmiyyəti də partladı. Əvvəlcə təxminən beş il əvvəl kitabı yazmaq üçün oturduğumda (ilk versiyanı çıxartmağım biraz vaxt aldı), GitHub adlı Git hosting veb saytını hazırlayan çox az bilinən bir şirkətdə işə başlamışdım. Nəşr zamanı saytdan bəlkə də bir neçə min nəfər istifadə edirdi və ondan yalnız dörd nəfər çalışırıq. Bu təqdimatı yazarkən GitHub, təxminən 5 milyon qeydiyyatdan keçmiş geliştirici hesabı və 230-dan çox işçisi olan 10 milyonuncu ev sahibliyi etdiyimiz layihəni elan edir. Sevin ya da nifrət edin, GitHub, Açıq Mənbə cəmiyyətinin geniş ərazilərini ilk nəşrini yazmağa oturduğum zaman ağlasığmaz bir şəkildə dəyişdirdi.

Pro Git-in orijinal versiyasında GitHub haqqında ev sahibliyi etdiyi Git nümunəsi olaraq kiçik bir hissə yazdım, heç vaxt çox rahat olmadım. Əsasən bir icma mənbəyi olduğunu hiss etdiyimi yazdığımı və eyni zamanda şirkətim haqqında danışdığımı çox sevmədim. Hələ də o maraqlar toqquşmasını sevməsəm də, GitHub-un Git cəmiyyətində əhəmiyyəti qaçınılmazdır. Git hosting nümunəsi əvəzinə kitabın bu hissəsini GitHub'ın nə olduğunu və ondan necə səmərəli istifadə edə cəyimi daha dərindən izah etməyə qərar verdim. Git'i necə istifadə edəcəyinizi öyrənmək ist əyirsinizsə, GitHub'ı necə istifadə edəcəyinizi bilmək, öz kodunuz üçün istifadə etməyinizə qərar verən Git sahibindən asılı olmayaraq dəyərli olan böyük bir cəmiyyətdə iştirak etməyə kömək edə cəkdir.

Son nəşrdən bu yana bir digər böyük dəyişiklik, Git şəbəkə əməliyyatları üçün HTTP protokolunun inkişafı və yüksəlməsidir. Kitabdakı nümunələrin çoxu daha sadə olduğundan SSH-dən HTTP-yə d əyişdirilmişdir.

Son bir neçə ildə Gitin nisbətən qaranlıq bir versiya idarəetmə sistemindən əsasən ticarət və açıq mənbəli versiya nəzarətinə qədər böyüməsini izləmək çox təəccüblü idi. Pro Git-in bu qədər yaxşı iş gördüyünə və eyni zamanda həm uğurlu həm də tamamilə açıq mənbəli bazarda olan az sayda texniki kitabdan biri olmağı bacardığına görə xoşbəxtəm.

Ümid edirəm Pro Git-in bu yenilənmiş nəşrini bəyənəcəksiniz.

Ön söz: Ben Straub

Bu kitabın ilk nəşri məni Gitə bağladı. Bu, əvvəllər gördüyüm hər şeydən daha təbii hiss edən proqram hazırlama tərzinə girişim idi. O vaxta qədər bir neçə ildir developer idim, amma bu məni olduğum yoldan daha maraqlı bir yola saldı.

İndi, illər sonra, Git-in böyük bir tətbiqinə köməkçi oldum, ən böyük Git hosting şirkətində çalışdım və insanlara Git haqqında öyrədərək dünyanı gəzdim. Scott ikinci nəşrdə işləməklə maraqlanacağımı soruşanda düşünməyə belə ehtiyac qalmadı.

Bu kitab üzərində işləmək çox xoşbəxtlik və imtiyaz oldu. Ümid edirəm ki, bu mənim kimi sizə kömək edəcəkdir.

İtfahlar

Həyat yoldaşım Becky'yə, o olmasaydı, bu macəranın əsla başlamazdı. - Ben Bu buraxılış mənim qızlarıma həsr edilmişdir. Bu illər ərzində məni dəstəkləyən həyat yoldaşım Jessica və qızım Josephine'ye, hansıki artıq nə olduğunu anlamayacaq çox yaşım olanda mənə dəstək olacaq. — Scott

Əməkdaşlar

Bu Open Source kitabı olduğundan illər ərzində düzəldilən bir neçə səhv və məzmun dəyişikliyi yaşandı. Pro Git-in Azərbaycan dilindəki versiyasını açıq mənbəli bir layihə kimi təqdim edən bütün insanlar burdadır. Bunu hər kəs üçün daha yaxşı bir kitab halına gətirməyə kömək etdiyiniz üçün hər kəsə təşəkkür edirəm.

```
Ali Mammadzada
Madina Jabrayil
Jean-Noël Avila
```

Giriş

Həyatınızın bir neçə saatını Git haqqında oxumağa sərf edəcəksiniz. Sizin üçün hazırladıqlarımızı izah etmək üçün bir dəqiqə vaxt ayıraq. Bu kitabın on fəsli və üç əlavəsinin qısa bir xülasəsi budur.

Fəsil 1 də Versiya Nəzarət Sistemlərini (VNS) və Git əsaslarını əhatə edəcəyik - heç bir texniki şey yoxdur, yalnız Git nədir, nə üçün VNS ilə dolu bir ölkədə meydana gəldi, onu nə ayırır və niyə çox insan istifadə edir. Sonra Git'i necə yükləyəcəyinizi və sisteminizdə hələ yoxdursa, ilk dəfə quracağımızı izah edəcəyik.

Fəsil 2 -də əsas Git istifadəsinə nəzər salacağıq - Git-in ən çox qarşılaşacağınız halların 80%-də necə istifadə ediləcəyi. Bu fəsli oxuduqdan sonra bir deponu klonlaya, layihənin tarixində nə baş verdiyini görə bilməli, sənədləri dəyişdirə və dəyişikliklərə qatqı təmin etməlisən. Kitab bu anda özbaşına yanarsa, başqa bir nüsxə almağa getməyiniz üçün Git'i istifadə edərək onsuz da olduqca faydalı olmalısınız.

Fəsil 3 tez-tez Git-in qatil xüsusiyyəti olaraq xarakterizə olunan Git-dəki branching modeli ilə əlaq ədardır. Burada Git-i paketdən həqiqətən nəyin fərqləndirdiyini öyrənəcəksiniz. İşinizi bitirdikdən sonra, Gitin branchingi-i həyatınızın bir hissəsi olmamışdan əvvəl necə yaşadığınız barədə düşünmək üçün sakit bir an keçirməyə ehtiyac hiss edə bilərsiniz.

Fəsil 4 serverdəki Git-i əhatə edəcəkdir. Bu fəsil, təşkilatınızda və ya əməkdaşlıq üçün şəxsi serverinizdə Git qurmaq istəyənlər üçündür. Başqasının sizin üçün idarə etməsinə icazə verm əyinizi istəsəniz, müxtəlif yerləşdirilmiş variantları da araşdıracağıq.

Fəsil 5 müxtəlif paylanmış iş axınlarını və Git ilə necə həyata keçiriləcəyini ətraflı şəkildə nəzərdən keçirəcəkdir. Bu fəsildə işinizi bitirdikdən sonra birdən çox uzaq depo ilə təcrübəli şəkildə işləyə, Git'i e-poçt üzərindən istifadə etməli və çoxsaylı uzaq branch-ları və qatqı patch-larını bacarıqla hoqqabazlıq etməlisiniz.

Fəsil 6 GitHub hosting xidmətini və alətlərini dərindən əhatə edir. Bir hesab üçün qeydiyyatdan keçməyi və idarə etməyi, Git depolarını, layihələrə töhfə vermək və özünüzə töhfələr qəbul etmək üçün ümumi iş axınlarını yaratmaq və istifadə etmək, GitHub-un proqramatik interfeysi və ümumiyyətlə həyatınızı asanlaşdırmaq üçün bir çox kiçik tövsiyələri əhatə edirik.

Fəsil 7 inkişaf etmiş Git əmrləri ilə bağlıdır. Burada qorxunc *reset* əmrinə yiyələnmək, səhvləri müəyyənləşdirmək üçün ikili axtarışdan istifadə etmək, tarixçəni redaktə etmək, reviziya seçimi və daha çox şey kimi mövzular haqqında məlumat əldə edəcəksiniz. Bu fəsildə Git haqqında bilikl əriniz tamamlanacaq, belə ki, həqiqətən ustadsınız.

Fəsil 8 xüsusi Git mühitinizi konfiqurasiya etməkdir. Buraya, xüsusi siyasətləri tətbiq etmək və ya təşviq etmək üçün çəngəl skriptlərin qurulması və istədiyiniz şəkildə işləyə bilmək üçün mühitin konfiqurasiya parametrlərindən istifadə daxildir. Xüsusi bir commit siyasətini tətbiq etmək üçün öz skriptlər qrupunuzu hazırlamağı da əhatə edəcəyik.

Fəsil 9 Git və digər VNS-lərdən bəhs edir. Buraya Git-in Subversion (SVN) dünyasında istifadəsi və digər VNS-lərdən Git-ə çevrilməsi daxildir. Bir çox təşkilat hələ də SVN-dən istifadə edir və dəyişm ək niyyətində deyil, amma bu vaxta qədər Git-in inanılmaz gücünü öyrənmiş olacaqsınız - və bu f əsildə hələ bir SVN serverindən istifadə etmək məcburiyyətində qalmağınızın öhdəsindən gələc

əyinizi göstərir. Hər kəsi dalmağa inandırdığınız təqdirdə bir neçə fərqli sistemdən layihələrin necə idxal ediləcəyini də əhatə edirik.

Fəsil 10 Git daxili hissələrinin qaranlıq, eyni zamanda gözəl dərinliklərinə baxır. Artıq Git haqqında hər şeyi bildiyinizə və güc və lütflə istifadə etdiyinizə görə Git-in obyektlərini necə saxladığını müzakirə etməyə davam edə bilərsiniz, obyekt modeli nədir, paketlərin detalları, server protokolları və s. Kitab boyu bu nöqtədə dərindən dalmaq istəsəniz, bu fəslin bölmələrinə istinad edəcəyik; ancaq bizim kimisinizsə və texniki detallara dalmaq istəyirsinizsə, əvvəlcə Fəsil 10-u oxumaq istəyə bilərsiniz. Bunu sizə tapşırırıq.

Əlavə **A** da Git-in müxtəlif spesifik mühitlərdə istifadəsinə dair bir sıra nümunələrə baxırıq. Git-d ən istifadə etmək istəyə biləcəyiniz və sizin üçün əlverişli olan bir sıra fərqli GUI və IDE proqramlaşdırma mühitlərini əhatə edirik. Git'i shell-nizdə, IDE'nizdə və ya mətn redaktorunuzda istifadə etməklə tanış olmaq istəyirsinizsə, buraya nəzər yetirin.

Əlavə **B**-də, libgit2 və JGit kimi alətlər vasitəsi ilə Git-in ssenarisini və genişləndirilməsini araşdırırıq. Mürəkkəb və sürətli xüsusi alətlər yazmaqla maraqlanırsınızsa və aşağı səviyyəli Git girişinə ehtiyacınız varsa, bu mənzərənin necə göründüyünü görə bilərsiniz.

Nəhayət, Əlavə **C**-də bütün əsas Git əmrlərini bir-bir nəzərdən keçiririk və kitabda harada yazdığımızı və onlarla nələr etdiyimizi nəzərdən keçiririk. Hər hansı bir xüsusi Git əmrini kitabda harada istifadə etdiyimizi bilmək istəyirsinizsə, buraya baxa bilərsiniz.

Gəlin başlayaq.

Başlanğıc

Bu fəsil Git ilə işə başlamaq haqqındadır. Versiya nəzarət alətləri ilə əlaqəli bəzi məlumatları izah etməyə başlayacağıq, sonra Git'in sisteminizdə necə işlədiləcəyinə və nəhayət işə başlamaq üçün necə qurulacağına nəzərdən keçirəcəyik. Bu fəslin sonunda Git'in niyə olduğunu, niyə istifadə etm əli olduğunuzu və bunun üçün niyə hazır olmalı olduğunuzu başa düşəcəksiniz.

Versiyaya Nəzarət Haqqında

"Versiya nəzarəti" nədir və ona niyə diqqət etməliyik? Versiya nəzarət sistemi fayla və ya fayllar qrupuna bütün vaxt ərzində edilən dəyişiklikləri qeydə alır. Buna görə də sonradan siz istədiyiniz versiyaya arxaya qayıda bilirsiniz. Bu kitabdakı nümunələr üçün versiyalarına nəzarət edilən fayllar olaraq bir proqramın qaynaq kodlarını istifadə edəcəksiniz ki, bu da reallıqda sizə komputerdəki istənilən fayl üzərində eyni əməliyyatları etməyə imkan verəcək.

Qrafik və ya veb dizaynersinizsə və şəkilin və ya layoutun hər bir versiyasını saxlamaq istə yirsinizsə(hansı ki, böyük ehtimalla istəyəcəksiniz), bir versiya nəzarət sistemi (VNS) istifadə etmək üçün çox ağıllı seçimdir. O sizə seçilmiş faylları əvvəlki vəziyyətinə qaytarmağa, bütün proyekti əvvəlki vəziyyətinə geri qaytarmağa, vaxt ərzində edilmiş dəyişiklikləri müqayisə etməyə, kimin etdiyi sonuncu dəyişikliyin sistemdəki problemə səbəb olduğuna, kimin problemi nə vaxt təqdim etdiyinə və daha çoxuna imkan yaradır. VNS-dən istifadə ümumilikdə, hər şey tərs getdiyində və ya faylları itirdiyinizdə asanlıqla bərpa edə biləcəyiniz mənasına gəlir. Əlavə olaraq, bütün bunları çox kiçik səy ilə edirsiniz.

Lokal Versiya Nəzarət Sistemi

Bir çox insanın versiyaya nəzarət metodu faylları digər qovluğa atmaqdır (əgər ağıllıdırlarsa, üz ərində vaxt yazılı bir qovluğa). Bu metod çox sadə olduğu üçün çox görülür, ancaq bu metod həm də inanılmaz dərəcədə səhvə meyillidir. Hansı qovluğun içində olduğunuzu unutmaq və istəm ədiyiniz fayla səhv şeyi yazmaq vəya digər fayllara kopyalamaq çox sadədir.

Bu problemi aradan qaldırmaq məqsədi ilə proqramistlər uzun zaman öncə sadə verilənlər bazasına sahib olan VNS-ləri proqramlaşdırdılar ki, fayllara edilən bütün dəyişiklikləri nəzarət altında saxlamaq mümkün olsun.

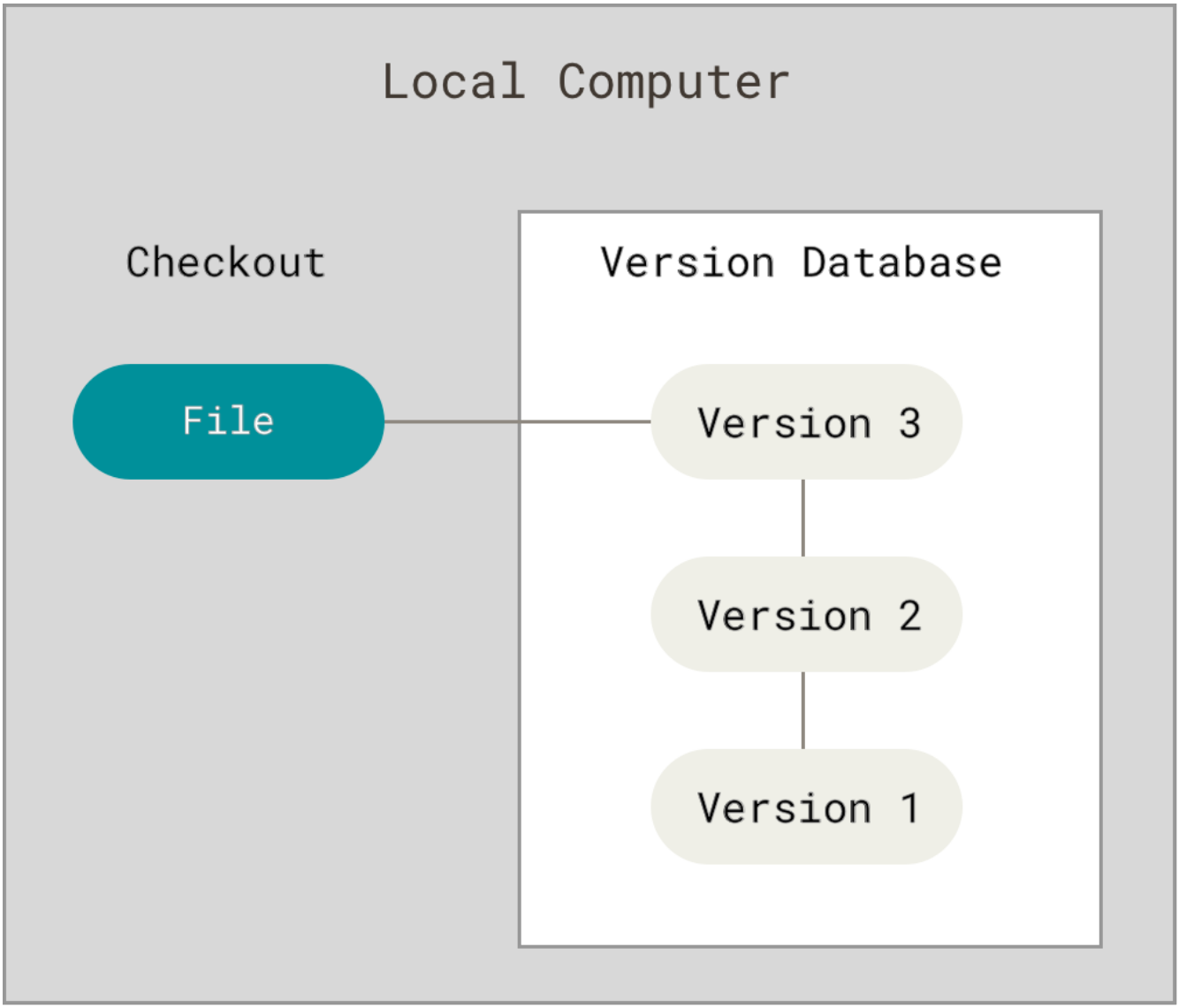


Figure 1. Lokal Versiya Nəzarəti.

VNS alətlərindən ən məhşurlarından biri RCS adlanırdı. O, bu gün də bir çox komputerdə yayılmışdır. RCS yamaq dəstələrini(fayllar arasındakı fərqləri) diskdə xüsusi formatda yadda saxlaması sayəsində fəaliyyət göstərir; İstənilən faylı göstərilmiş vaxta uyğun olaraq bütün yamaqları əlavə edərək yenidən yaratmağa qadirdir.

Mərkəzləşdirilmiş Versiya Nəzarət Sistemləri

İnsanların qarşılaşdığı əsas problemlərdən biri də budur ki, onlara digər sistemlərdəki developerl ərlə bir işləmək lazım olur. Bu problemlə başa çıxmaq üçün, Mərkəzləşdirilmiş Versiya Nəzarət Sistemləri (MVNS) yaradılmışdır. Bu sistemlər(CVS, Subversion və Perforce kimiləri) bütün versiyalanmış faylların saxlandığı tək bir serverə və həmin mərkəzləşdirilmiş yerdən fayllara baxan bir neçə müştəriyə sahib olurlar Bir neçə illərdir ki, versiya nəzarəti üçün bu bir standart olmuşdu.

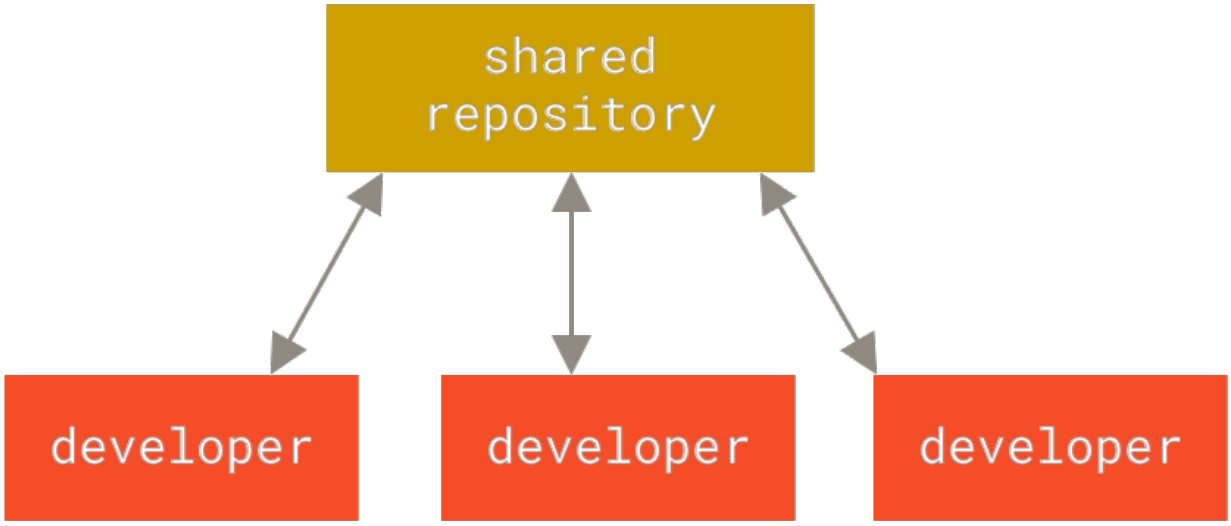


Figure 2. Mərkəzləşdirilmiş Versiya Nəzarəti.

Bu quruluş bir çox üstünlüyə sahibdir, xüsusiylə lokal VNS'lərdən. Məsələn, hamı bir-birinin proyekt üzərində hansı dərəcədə iş gördüyünü bilir. İdarəedicilərin kimin nə edə biləcəyinə dair yaxşı nəzarəti var və mərkəzləşdirilmiş nəzarət sistemlərini idarə etmək, hər müştəridəki müxtəlif lokal verilənlər bazalarını idarə etməkdən daha asandır.

Buna baxmayaraq, bu quruluşun həm də bəzi ciddi mənfi yönləri var. Ən barizi, mərkəzləşdirilmiş serverin təmsil etdiyi tək nöqsan nöqtəsidir. Əgər bu server 1 saatlığına çökərsə, həmin 1 saat ərzində heç kim heç bir işbirliyi görə bilməz ya da öz işlədikləri versiyalanmış dəyişiklikləri yadda saxlaya bilməz. Əgər mərkəzləşdirilmiş verilənlər bazasındakı sərt disk zədələnərsə və lazımi ehtiyatlar alınmaıbsa, siz hər şeyi itirirsiniz — insanların öz lokal komputerlərində saxladığı anlıq vəziyyətlərdən başqa proyektin bütün tarixi. Lokal versiya nəzarət sistemləri də eyni problemdən əziyyət çəkirlər — proyektin bütün tarixinin bir yerdə olduğu bütün vaxt ərzində siz hərşeyi itirm ək riskinə gedirsiniz.

Paylanmış Versiya Nəzarət Sistemləri

Burada işə Paylanmış Versiyaya Nəzarət Sistemləri (PVNS)lər qarışır. PVNS'də (Git, Mercurial, Bazaar ya da Darcs kimi), müştərilər sadəcə faylların anlıq vəziyyətini yadda saxlamır; bunun əv əzinə, onlar bütün anbarı fayllarının bütün tarixçəsi ilə bərabər olaraq kopyasını götürürlər. Bel əliklə, əgər server çökərsə və sistemlər həmin serverlə iş birliyi görürlərsə, müştərilərdəki anbarlardan hansısa biri serverə geri kopyalanaraq onu əvvəlki vəziyyətinə qaytara bilər.

Hər bir klon, həqiqətən də bütün məlumatın tam ehtiyatıdır.

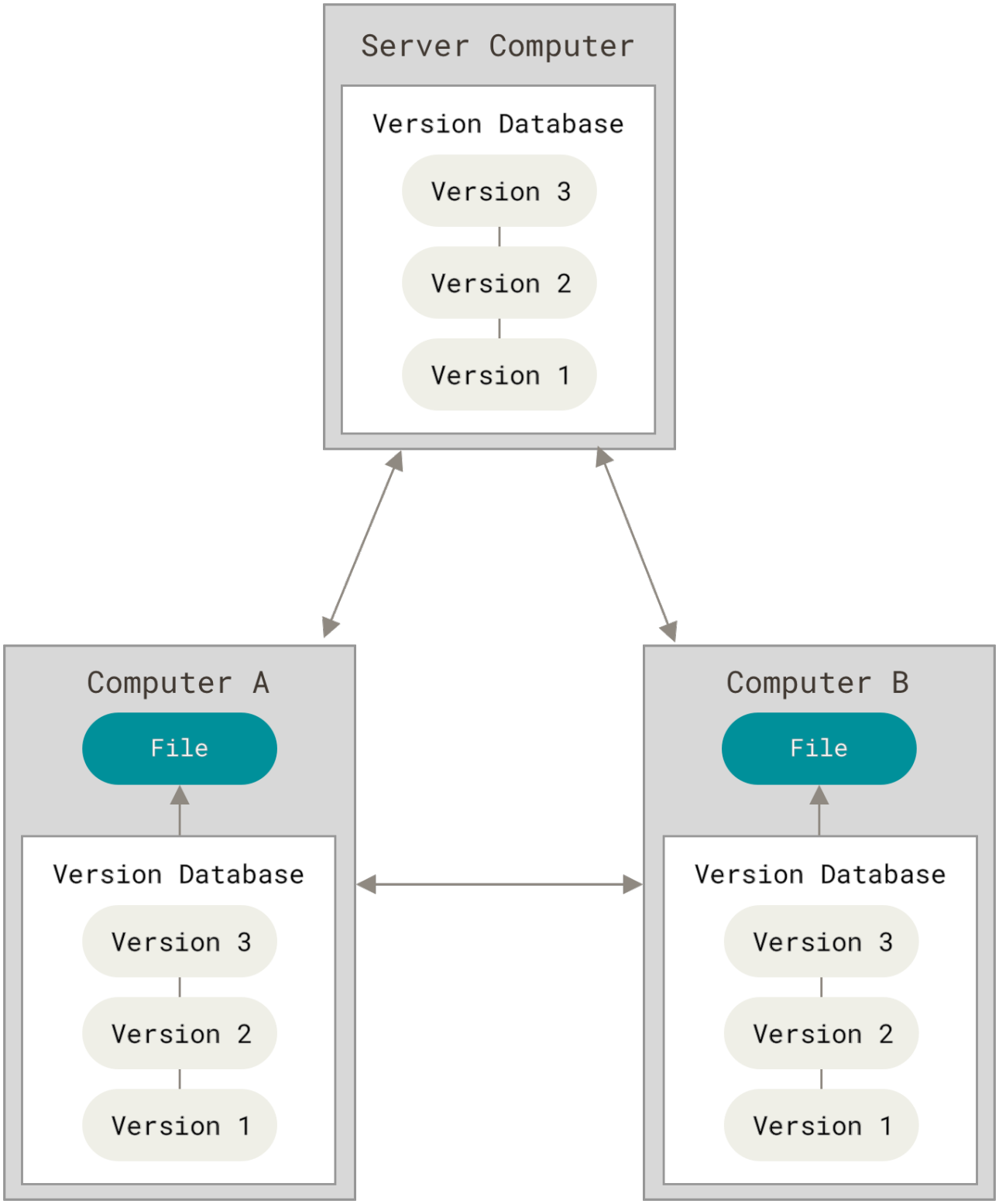


Figure 3. Paylanmış versiya nəzarəti.

Bundan başqa, bir çox sistemlər, bir çox uzaq anbarlarla rahat işləyə bilirlər, beləliklə, siz eyni proyektdə fərqli insanlar qrupu ilə müxtəlif yollarla işbirliyi edə bilərsiniz. Bu sizə ierarxik modellərdə olduğu kimi mərkəzləşdirilmiş sistemlərdə mümükün olmayan müxtəlif iş görmək metodlarını quraşdırmağa imkan verəcək.

Git'in Qısa Hekayəsi

Həyatdakı bir çox gözəl şey kimi, Git yaradıcı sonlanma və atəşli müzakirələrlə yarandı.

Linux kerneli kifayət qədər geniş əhatə dairəsinə sahib olan açıq qaynaq kodlu proqram t əminatıdır. Kernelin təmiri zamanlarının əksəriyyətində (1991-2002), proqram təminatına edilən dəyişikliklər yamaqlar və arxiv faylları əsasında olmuşdur. 2002-ci ildə, Linux kernel proyekti BitKeeper adlanan xüsusi paylanmış versiya nəzarət sistemindən istifadə etməyə başlamışdır.

2005-ci ildə, Linux kernelini kodlayan cəmiyyət ilə BitKeeper ticarət şirkəti arasında əlaqələr pozuldu və alətin havayı statusu geri qayıtdı. Bu Linux proqramçı cəmiyyətinə (və xüsusiylə Linuxun yaradıcısı olan Linus Torvals) BitKeeperdən öyrəndikləri dərslər əsasında özlərinə aid alət proqramlamağa başladılar. Yeni sistem üçün bəzi hədəflər aşağıdaki kimi idi:

- Sürət
- Sadə dizayn
- Qeyri-xətti proqramlaşdırma üçün güclü dəstək (yüzlərlə paralel budaqlar)
- Təmamilə paylanma
- Linux kerneli kimi iri həcmli proyektləri effektiv şəkildə idarə edə bilən (sürət və məlumat ölçüsü)

2005-ci ildə yarandığından bu yana, Git əvvəlki keyfiyyətləri saxlamaqla daha asan isifadə edilə biləcək şəkildə inkişaf edib. O heyrətamiz şəkildə sürətlidir, o böyük proyektlərlə həddən artıq effektlidir və qeyri-xətti proqramlaşdırma üçün inanılmaz budaq sistemi mövduddur. (Baxın Git'də Branch).

Git Nədir?

Beləliklə, Git qısaca nədir? Bu anlamaq üçün vacib bir hissədir, çünki Git'in nə olduğunu və bunun necə işlədiyinin əsaslarını başa düşsəniz, Git'dən səmərəli istifadə etmək çox güman ki, sizin üçün daha asan olacaq. Git'i öyrəndiyiniz zaman, CVS, Subversion və ya Perforce kimi digər VNS-lər haqqında bilə biləcəyiniz şeylərdən fikrinizi təmizləməyə çalışın - bu alətdən istifadə edərkən incə qarışıqlıqların qarşısını almağa kömək edəcəkdir. Git'in istifadəçi interfeysi digər digər VCS-lərə kifayət qədər bənzəsə də, Git məlumatları saxlayır və çox fərqli bir şəkildə düşünür və bu fərqləri anlamaq istifadə edərkən çaşqınlıq yaratmamağa kömək edəcək.

Snapshot'lar, Fərqlər Yox

Git və digər hər hansı bir VNS (Subversiya və daxil olan dostlar) arasındakı əsas fərq Git'in m əlumatları düşünməsidir. Konseptual olaraq, digər sistemlərin əksəriyyəti məlumatları fayl əsaslı dəyişikliklərin siyahısı kimi saxlayır. Digər sistemlər (CVS, Subversion, Perforce, Bazaar və s.) saxladıqları məlumatları sənədlər toplusu və zaman keçdikcə hər bir sənəddə edilən dəyişikliklər kimi düşünürlər (bu adətən *delta-based* version kontrolu kimi təsvir olunur).

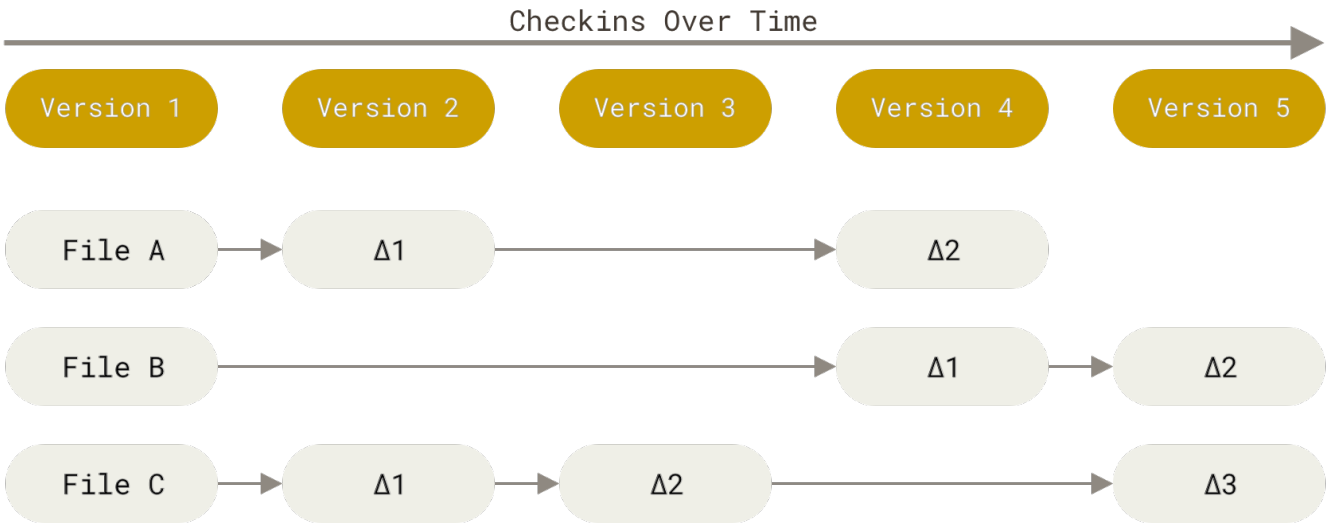


Figure 4. Hər bir sənədin əsas versiyasında dəyişiklik kimi məlumatların saxlanması

Git məlumatlarını bu şəkildə düşünmür və saxlamır. Bunun əvəzinə, Git, məlumatlarını daha çox miniatür bir fayl sisteminin bir sıra görüntüləri *(snapshot'ları) kimi düşünür. Git ilə hər dəfə etdiyiniz və ya proyektinizin vəziyyətini qoruduğunuz zaman, Git əsasən bütün sənədlərinizin o anda necə göründüyünün şəklini çəkir və həmin görünüşə istinad edir. Əgər fayllar dəyişməyibsə, səmərəli olmaq üçü Git yenə də saxlamır, sadəcə əvvəllər saxladığı eyni sənədlə əlaqələndirir. Git məlumatları daha çox **snapshot axını** kimi düşünür.

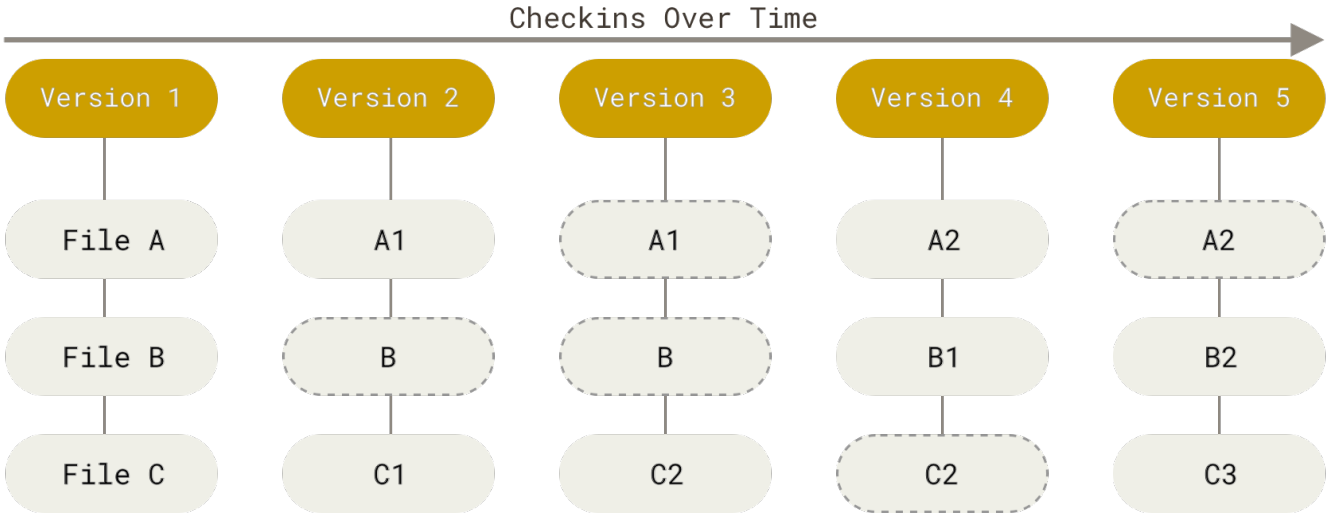


Figure 5. Zamanla məlumatları layihənin snapshot'ları olaraq saxlamaq

Bu Git və təxminən digər bütün VNS'lər arasında vacib bir fərqdir. Git, digər sistemlərin əvvəlki n əsildən kopyalanan versiya nəzarətinin demək olar ki, hər tərəfini yenidən nəzərdən keçirir. Bu, Git'i sadəcə bir VNS'dən çox üstün qurulmuş inanılmaz dərəcədə güclü alətlərə sahib mini bir fayl sistemi edir.

Git'də Branch bölümündə Git budaqlanmasını əhatə edərkən məlumatlarınızı bu şəkildə düşünər ək qazandığınız üstünlüklərin bir qismini araşdıracağıq.

Təxminən Hər Əməliyyat Lokaldır

Git'də əksər əməliyyatların həyata keçirilməsi üçün yalnız lokal sənədlər və mənbələrə ehtiyac duyulur - ümumiyyətlə şəbəkənizdəki başqa bir kompüterdən heç bir məlumat tələb olunmur. Ə ksər əməliyyatlarda şəbəkə gecikməsinin olduğu MVNS-ə alışmış olsanız, Git'in bu tərəfi sürət

tanrılarının Git'i bilinməyən güclərlə xeyir-dua verdiyini düşünməyə vadar edəcəkdir. Layihənin bütün tarixi lokal diskinizdə olduğundan əksər əməliyyatlar demək olar ki, ani görünür.

Məsələn, Git layihənin tarixini nəzərdən keçirmək üçün, tarixi əldə etmək və sizə göstərmək üçün serverə getməyə ehtiyac duymur - sadəcə lokal məlumat bazanızdan oxuyur. Bu o deməkdir ki, siz dərhal layihənin tarixini görə bilirsiniz. Bir ay əvvəl mövcud bir versiya ilə fayl arasında edilən d əyişiklikləri görmək istəyirsinizsə, Git bir ay əvvəl faylı axtara və uzaq bir serverdən bunu etməyi istəmək əvəzinə lokal fərq hesablamasını həyata keçirə bilər və ya lokal olaraq bunu etmək üçün uzaq serverdən faylın köhnə versiyasını çəkə bilər.

Bu da offline və ya VPN-dən kənarda olsanız edə bilməyəcəyiniz çox az şey olduğu deməkdir. Bir t əyyarədə və ya bir qatarda olsanız və bir az iş görmək istəsəniz, yükləmək üçün şəbəkə bağlantısına çatana qədər məmnuniyyətlə (*lokal* nüsxənizi xatırlayırsızmı?) işləyə bilərsiniz. Evə gedib VPN müştərinizlə düzgün işləyə bilmirsinizsə,o olmadan yenə də rahatlıqla işləyə bilərsiniz. Bir çox digər sistemlərdə bunu etmək ya mümkün deyil, ya da əziyyətlidir. Məsələn, Perforce-də serverə qoşulmadığınız zaman çox şey edə bilməzsiniz; Subversion və CVS-də sənədləri redaktə edə bilərsiniz, ancaq verilənlər bazanıza dəyişiklik edə bilməzsiniz (çünki verilənlər bazanız offlinedir).

Git'in Bütövlüyü Var

Git-dəki hər şey saxlanılmadan əvvəl yoxlanılır və sonra həmin yoxlama nömrəsi tərəfindən istinad edilir. Bu o deməkdir ki, Git-in xəbəri olmadan hər hansı bir fayl və ya qovluğun m əzmununu dəyişdirmək mümkün deyil. Bu funksionallıq ən aşağı səviyyələrdə qurulmuşdur və f əlsəfəsinin ayrılmaz hissəsidir. Git aşkarlanmadan məlumatı itirə və ya köçürmə zamanı fayl pozğunluğunu ala bilməzsiniz.

Git-in bu yoxlanış üçün istifadə etdiyi mexanizmə SHA-1 hash deyilir. Bu, altıbucaqlı simvollardan (0-9 və a – f) ibarət olan 40 simvollu bir string-dir və Git-də bir fayl və ya qovluq quruluşunun m əzmunu əsasında hesablanır. Bir SHA-1 hash aşağıdakı nümunəyə bənzəyir:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Bu hash dəyərlərini Git-də hər yerdə görəcəksiniz, çünki Git onları çox istifadə edir. Əslində, Git verilənlər bazasında hər şeyi fayl adı ilə deyil, tərkibindəki hash dəyəri ilə saxlayır.

Ümumiyyətlə Git Sadəcə Məlumat Əlavə Edir

Git-də hərəkətlər etdikdə demək olar ki, hamısı yalnız Git verilənlər bazasına *add* məlumat verir. Sistemi geri qaytarılmayan bir şey etmək və ya məlumatları hər hansı bir şəkildə məcbur silmək ç ətindir. Hər hansı bir VNS-də olduğu kimi hələ etmədiyiniz dəyişiklikləri itirə və ya qarışdıra bil ərsiniz, ancaq Git-də bir anlıq görüntüsünü aldıqdan sonra itirmək çox çətindir, xüsusən də veril ənlər bazasını mütəmadi olaraq başqa bir anbara köçürsəniz.

Git-dən istifadə etmək çox xoşdur, çünki ciddi şeyləri korlamaq təhlükəsi olmadan bir çox şeyləri sınaya biləcəyimizi bilirik. Git-in məlumatlarını necə saxladığı və itirilmiş məlumatları necə bərpa edə biləcəyinizə daha ətraflı baxmaq üçün Ləğv Edilən İşlər (Geri qaytarılan)-ə baxın.

Üç Əsas Vəziyyət

İndi diqqət yetirin - burada öyrənmə prosesinin qalan hissəsinin rahat keçməsini istəyirsinizsə, Git haqqında yadda saxlamağınız vacib olan şeydir. Git sənədlərinizin yerləşə biləcəyi üç əsas vəziyyə tə malikdir: *modified*, *staged* və *commissions*:

- Modified - faylı dəyişdirdiyinizi, lakin hələ verilənlər bazanıza verməmisiniz deməkdir.
- Staged - növbəti snapshot-a daxil olmaq üçün hazırkı versiyasında dəyişdirilmiş bir faylı qeyd etdiyiniz deməkdir.
- Commissions - məlumatların etibarlı olaraq lokal verilənlər bazasında saxlanıldığını bildirir.

Bu bizi Git layihəsinin üç əsas hissəsinə aparır: iş ağacı (the working tree), quruluş sahəsi (the staging area) və Git qovluğu (Git Directory).

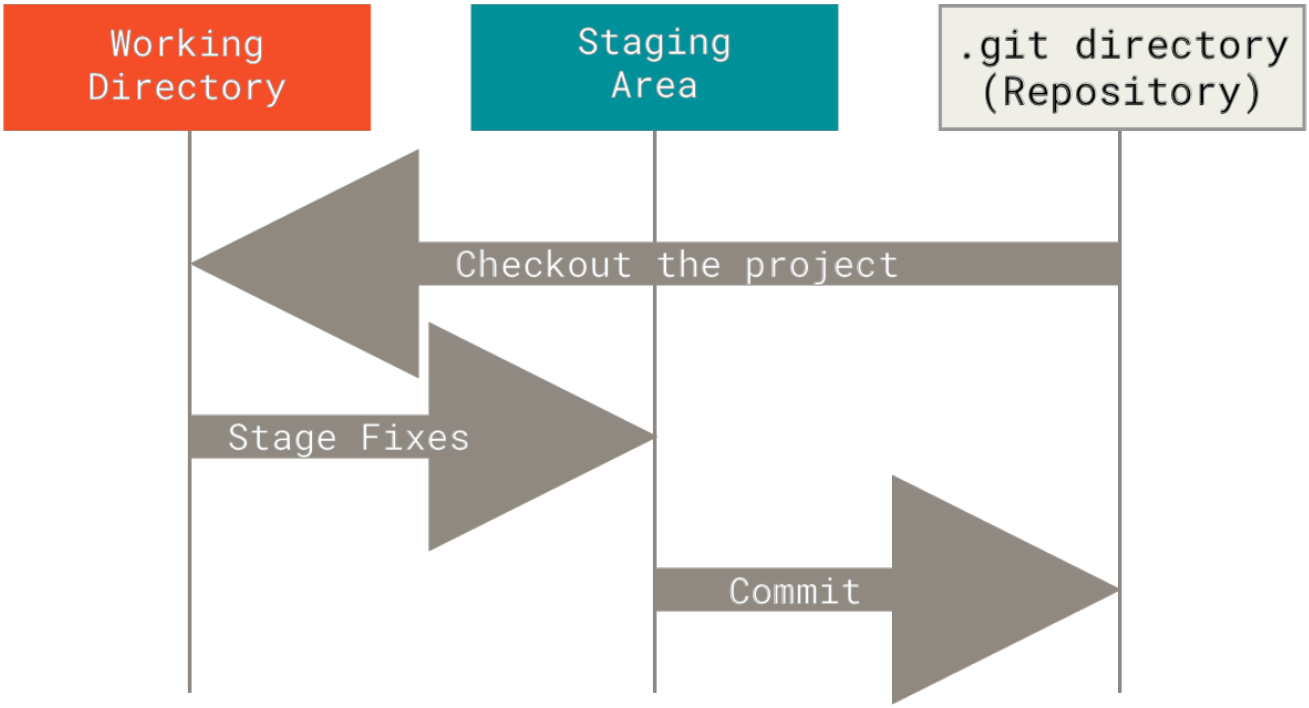


Figure 6. İş ağacı, quruluş sahəsi və Git qovluğu

İş ağacı, layihənin bir versiyasının tək bir yoxlanılmasıdır. Bu fayllar Git qovluğundakı sıxılmış verilənlər bazasından çıxarılır və istifadə və ya dəyişdirmək üçün diskə yerləşdirilir.

Quruluş sahəsi ümumiyyətlə Git qovluğunuzda olan sonrakı commit-inizə nə daxil olacağı barədə məlumat saxlayan bir sənəddir. Git dilində texniki adı “index”-dir, lakin “staging area” ifadəsi də eyni şəkildə işləyir.

Git qovluğu Git layihəniz üçün metadata və obyekt verilənlər bazasını saxladığı yerdir. Bu, Git’in ən vacib hissəsidir və başqa bir kompüterdən depo yerləşdirdiyiniz zaman *klon* olan şeydir.

Əsas Git iş axını aşağıdakı kimidir:

1. İş ağacınızdakı faylları dəyişdirirsiniz.
2. Seçimlə yalnız sonrakı öhdəliyinizin bir hissəsi olmağını istədiyiniz dəyişiklikləri mərhələli ş əkildə həyata keçirirsiniz, bu *yalnız* quruluş sahəsinə əlavə edir.

3. Sənədləri quruluş sahəsindəki kimi götürən və anlıq görüntünü Git qovluğunuzda daimi olaraq saxlayan bir öhdəlik kimi yerinə yetirirsiniz.

Bir faylın müəyyən bir versiyası Git qovluğunda varsa, bu *committed* hesab olunur. Dəyişdirilmiş və quruluş sahəsinə əlavə edilmişdirsə, bu *staged* hesab olunur. Yoxlanılandan bəri dəyişdirilmiş, lakin mərhələli olmamışdırsa, bu *modified* hesab olunur.

Git'in Əsasları-də bu vəziyyətlər haqqında daha çox məlumat əldə edəcəksiniz və onlardan necə yararlanmağı və ya mərhələli hissəni tamamilə keçə biləcəyinizi öyrənəcəksiniz.

Əmr Sətiri

Git'i istifadə etməyin bir çox yolu var. Original əmr-sətiri və müxtəlif özəlliklərə sahib olan çoxlu qrafikal istifadəçi interfeysinə sahib alətlər mövcuddur. Bu kitab üçün biz əmr sətirindən istifadə edəcəyik. Kimisi üçün əmr sətiri *bütün* Git əmrlərini işlədə biləcəyiniz tək yerdir — bir çox qrafiki istifadəçi interfeysləri sadəlik üçün Git-in funksionallığının alt çoxluğunu istifadə edir.

Əgər əmr sətirli versiyanın necə işlədildiyini bilirsinizsə, böyük ehtimalla siz həm də qrafikal istifadəçi interfeysi necə işlətməli olduğunuzu da anlayacaqsınız, ancaq bunun əksi çox doğru deyil.

Həmçinin sizin qrafiki proqramdan istifadəniz şəxsi zövq məsələsi olsa belə, əmr-sətiri alətləri *bütün* istifadəçilər üçün quraşdırılmış və əlçatan olacaqdır.

Beləliklə biz sizdən macOS'da Terminal-ı necə açmalı olduğunuzu və ya Windows'da PowerShell'i necə açmalı olduğunuzu bilməyinizi gözləyirik. Əgər bizim nə haqqında danışdığımızdan xəbəriniz yoxdursa, burada dayanıb, bu haqda sürətli araşdırma aparmalı və kitabda danışılan digər nümunələr və izahları izləməlisiniz.

Git'i Quraşdırmaq

Git'i istifadə etməyə başlamazdan qabaq öz komputerinizdə onu hazır etməlisiniz. Əgər artıq yükl ənmiş olsanız belə, sonuncu versiyaya yeniləmək daha yaxşı idea olacaqdır. Onu başqa quraşdırıcı vasitəsi ilə paket formasında ya da qaynaq kodlarını yükləyib özünüz kompayl edə bilərsiniz.

Bu kitab **2.8.0** versiyalı Git'dən istifadə edilərək yazılıb. Bizim istifadə etdiyimiz əmrlərin bir çoxu Git'in bir çox köhnə versiyalarında işləməlidir, köhnə versiyanı istifadə edirsinizsə, əmrlərin bəziləri işləməyə və ya fərqlilik göstərə bilər. Git köhnələri saxlamaq barəsində qabiliyyətli olduğuna görə, 2.8-dan sonrakı versiyalar yaxşı işləməlidir.

Linux üzərində Quraşdırma

Binary quraşdırıcı vasitəsi ilə sadə Git alətlərini Linux üzərində quraşdırmaq istəyirsinizsə, öz Linux distributorunuzla bərabər gələn paket menecment alətlərindən istifadə edə bilərsiniz. Əgər siz Fedora'dasınızsa (vəya CentOS yada RHEL kimi RPM əsaslı distributorla yaxın-əlaqəli başqası), `dnf`-dən istifadə edə bilərsiniz:

```
$ sudo dnf install git-all
```

Əgər siz Ubuntu kimi Debian əsaslı distributordasınızsa, `apt`-ni sınayın:

```
$ sudo apt install git-all
```

Daha çox seçimlər üçün, burada Git'in vebsaytında fərqli Unix distributorları üçün müxtəlif quraşdırma təlimatları mövcuddur: https://git-scm.com/download/linux.

macOS'da Quraşdırma

Git-i Mac üzərində quraşdırmaq üçün fərqli yollar mövcuddur. Yəqin ki, ən asanı Xcode əmrlər s ətiri alətlərini quraşdırmaqdır. Mavericks-də(10.9) və yuxarı versiyalarda ilk istifadə zamanı bunu terminalda `git` əmrini icra edərək edə bilərsiniz.

```
$ git --version
```

Əgər siz hələ də quraşdırmamısınızsa, o sizi quraşdırmağa yönləndirəcək.

Əgər yeni versiyanı istəyirsinizsə, binary quraşdırıcı vasitəsi ilə də onu quraşdıra bilərsiniz. macOS Git quraşdırıcısı Git vebsaytındadır və yükləmək üçün hazırdır. https://git-scm.com/download/mac.

Figure 7. Git macOS quraşdırıcısı.

Siz həm də bunu macOS quraşdırıcısı üçün GitHub-un bir parçası olaraq da quraşdıra bilərsiniz. Onların qrafiki istifadəçi intefeysli aləti həmçinin əmr sətirli alətləri yükləmək seçiminə də sahibdir. Siz bu aləti buradakı GitHub-un macOS üçün olan vebsaytından yükləyə bilərsiniz. https://desktop.github.com.

Windows'da Quraşdırma

Git-i Windowsda quraşdırmaq üçün də bir neçə yol mövcuddur

Ən rəsmi quruluş Git vebsaytında yüklənməyə hazırdır. Sadəcə bura https://git-scm.com/download/win gedin və yüklənmə avtomatik başlayacaqdır. Nəzərə alın ki, proyektin adı `Windows üçün Git` adlanır, hansı ki Git-in özündən ayrıdır; haqqında daha çox məlumat əldə etmək üçün bura gedinI https://gitforwindows.org.

Avtomatik quraşdırılma üçün siz Git Chocolatey paketini istifadə edə bilərsiniz. Nəzərə alın ki, Chocolatey paketini cəmiyyət saxlayır.

Giti quraşdırmağın digər asan yolu da GitHub Desktop-u quraşdırmaqdır. Quraşdırıcı Gitin həm qrafikal istifadəçi interfeyli versiyasına, həm də əmr sətirli versiyasına sahibdir. O həm də PowerShell ilə yaxşı işləyir və CRLF sazlamaları ilə güclü kimlik keşi quraşdırır.

Daha sonra bu şeylər haqqında daha çox məlumat alacağıq, hələlik istəyəcəyiniz şeylər olduğunu söyləmək kifayətdir. Onu buradan yükləyə bilərsiniz. GitHub Desktop vebsatı.

Qaynaq Kodlardan Quraşdırma

Bəzi insanlar Git'i qaynağından quraşdırmağı daha faydalı hesab edə bilər, ona görə ki siz ən yeni versiyanı əldə edirsiniz. Binary quraşdırıcılar nisbətən arxadan gəlirlər, Git son illərdə yetişdiyinə görə bu az fərq yaradır.

Əgər Git'i qaynağından quraşdırmaq istəyirsinizsə, Git'in asılı olduğu növbəti kitabxanalara sahib olmalısınız: autotools, curl, zlib, openssl, expat, və libiconv. Məsələn, əgər siz `dnf` olan (Fedora kimi) vəya `apt-get` (Debian əsaslı sistemlərdəki kimi) əməliyyat sistemindəsinizsə, aşağıdak Git binary-l ərini kompayl etmək və quraşdırmaq üçün lazım olan minimal aslılıqları aşağıdakı əmrlərdən birindən istifadə edərək quraşdıra bilərsiniz.

```
$ sudo dnf install dh-autoreconf curl-devel expat-devel gettext-devel \
  openssl-devel perl-devel zlib-devel
$ sudo apt-get install dh-autoreconf libcurl4-gnutls-dev libexpat1-dev \
  gettext libz-dev libssl-dev
```

Dokumentasiyanı müxtəlif formatlarda (doc, html, info) əlavə etmək üçün, bu əlavə asılılıqlar lazımdır (Qeyd: RHEL və CentOS, Scientific Linux kimi RHEL törəməsi istifadəçiləri `docbook2X` paketini buradan yükləməlidirlər. EPEL anbarını aktivləşdir

```
$ sudo dnf install asciidoc xmlto docbook2X
$ sudo apt-get install asciidoc xmlto docbook2x
```

Əgər siz Debian əsaslı distributordan istifadə edirsinizsə (Debian/Ubuntu/Ubuntu törəmələri), siz həm də `install-info` paketini yükləməlisiniz:

```
$ sudo apt-get install install-info
```

Əgər siz RPM əsaslı (Fedora/RHEL/RHEL törəmələri) distributor istifadə edirsinizsə, sizə həm də `getopt` paketi lazımdır. (hansı ki artıq Debian əsaslı distoya quraşdırılıb):

```
$ sudo dnf install getopt
$ sudo apt-get install getopt
```

Əlavə olaraq əgər siz Fedora/RHEL/RHEL istifadəçisinizsə, bunu etməlisiniz.

```
$ sudo ln -s /usr/bin/db2x_docbook2texi /usr/bin/docbook2x-texi
```

Bütün lazımi asılılıqlara sahib olduqdan sonra, müxtəlif yerlərdən axırıncı işarələnmiş tarball'ları gedib götürə bilərsiniz. Siz onu buradakı https://www.kernel.org/pub/software/scm/git kernel.org saytından vəya buradakı https://github.com/git/git/releases. GitHub saytından götürə bilərsiniz GitHub səhifəsindəki sonuncu versiya adətən daha təmiz olsa da, kernel.org səhifəsindəki imzalar vasitəsi ilə öz yükləmənizi təsdiqləyə bilərsiniz.

Bunlardan sonra, kompayl edin və quraşdırın:

```
$ tar -zxf git-2.0.0.tar.gz
$ cd git-2.0.0
$ make configure
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

Bunlar bittikdən sonra, siz Git'in yeniliklərini əldə etmək üçün Git'in özündən istifadə edə bil ərsiniz.

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

İlk Dəfə Git Quraşdırması

Sisteminizdə artıq Git olduğuna görə öz Git mühitinizi özəlləşdirmək üçünü bir neçə şey etmək ist əyəcəksiniz. Verilmiş komputerdə bunları sadəcə bir dəfə etməlisiniz; onlar yenilənmələrə baxmayaraq özlərini qoruyacaqlar. Siz həm də əmrləri yenidən işlətməklə onları istədiyiniz zaman dəyişdirə bilərsiniz.

Git, `git config` adlanan alət ilə yüklənir və bu Gitin bütün işləmə aspektlərini idarə edən konfiqurasiya dəyişənlərini almağa və təyin etməyə imkan verir.

Bu dəyişənlər üç fərqli yerdə saxlanıla bilərlər:

1. `/etc/gitconfig` faylı: Sistemdəki hər istifadəçiyə və onların sahib olduğu bütün anbarlara tətbiq olunan dəyərləri özündə saxlayır. Əgər `--system` seçimini `git config`-ə yazsanız, o xüsusi olaraq bu fayla oxuyub yazacaqdır. Sistem konfiqurasiya faylı olduğuna görə, siz onu dəyidirmək üçün administrator və ya superuser səlahiyyətlərinə sahib olmalısınız.
2. `~/.gitconfig` və ya `~/.config/git/config` faylı: Spesifik olaraq sizin istifadəçinizə uyğun olan də yərləri saxlayan fayldır. `--global` seçimini əlavə edərək, Git-in bu fayl üzərinə yazmasını və oxumasına yön verə bilərsiniz və sistem üzərində işlədiyiniz *bütün* anbarları əhatə edəcək.
3. Hansı anbarı işlətməyinizdən asılı olmayaraq Git qovluğundaki `config` faylı (`.git/config`) xüsusi olaraq həmin anbara aiddir. Siz Git-ə `--local` seçimini əlavə edərək bu fayldan oxumağa və yazmağa məcbur edə bilərsiniz, amma bu, əslində standart olaraq belədir. (Təəccüblü deyil ki, bu seçimin qaydasında işləməsi üçün Git anbarı içində bir yerdə olmağınız lazımdır.)

Hər səviyyənin dəyərləri, özündən əvvəlki səviyyənin dəyərlərinin üzərinə yazılır, beləliklə `.git/config`-dəki dəyərlər `/etc/gitconfig`-dəki dəyərləri əvəzləyir.

Windows sistemlərində Git `$HOME` qovluğunda olan (bir çoxu üçün `C:\Users\$USER`) `.gitconfig` faylını axtarır O həmçinin hələ də `/etc/gitconfig` axtarır çünki o MSys kökü ilə bağlantılıdır,

###It also still looks for `/etc/gitconfig`, although it's relative to the MSys root, which is wherever you decide to install Git on your Windows system when you run the installer.

Əgər siz Windows üçün Git-in 2.x və ya yuxarı versiyasını işlədirsinizsə, Windows XP'də `C:\Documents and Settings\All Users\Application Data\Git\config` adresində, Windows Vista və daha yenilərində isə `C:\ProgramData\Git\config` adresində sistem səviyyəli konfiqurasiya faylını tapa bilərsiniz. Konfiqurasiya faylı yalnız admin tərəfindən icra olunan `git config -f <file>` əmri ilə dəyişdirilə bilər.

Bütün sazlamalarınıza və onların hardan gəldiyinə aşağıdakı komandanı icra edərək baxa bil ərsiniz.

```
$ git config --list --show-origin
```

Sizin Kimliyiniz

Git-i quraşdırarkən birinci etməli olduğunuz şey öz istifadəçi adınızı və email adresinizi təyin etm əkdir. Bu vacibdir çünki, hər bir Git commit-i bu informasiya vasitəsiylə işləyir və siz aşağıdakıları yaratmağa başladığınız andan etibarən hər bir commitin içinə möhrlənir:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Təkrarlayaq, siz bunu yalnız `--global` seçimini işlədərkən bircə dəfə etməlisiniz, ona görə ki, ondan sonra Git həmişə nə etsəniz bu informasiyanı istifadə edəcək. Əgər xüsusi bir proyekt üçün ayrı bir ad və email işlətməli olsanız, həmin proyektin içində `--global` seçimini çağırmadan yuxarıdakı əmrləri icra edə bilərsiniz.

Qrafik interfeysli alətlərdən bir çoxu ilkin quraşdırma zamanı bunu etməyinizə kömək edəcək.

Sizin Redaktorunuz

Öz kimliyinizi təyin etdikdən sonra, Git'in sizdən mesaj yazmağınızı istəyərkən lazım olan default mətn redaktorunu sazlaya bilərsiniz.

Əgər sazlanmasa, Git sistemin default redaktorunu istifadə edir.

Əgər Emacs kimi fərqli mətn redaktoru istifadə etmək istəyirsinizsə, aşağıdakıları edə bilərsiniz:

```
$ git config --global core.editor emacs
```

Əgər Windows sistemində fərqli bir mətn redaktoru istifadə etmək istəyirsinizsə, onun icra olunan faylına tam yolu göstərməlisiniz. Redaktorunuzun necə * qablaşdırıldığından asılı olaraq bu fərqli ola bilər.

Məşhur bir proqramlaşdırma redaktoru olan Notepad ++ istifadə edərkən çox ehtimal ki, 32-bit versiyasından istifadə etmək istəyəcəksiniz, çünki 64-bit versiyasını yazarkən bütün plaginləri də stəklənmir. Əgər siz 32-bitli Windows sistemindəsinizsə və ya 64-bitlik bir sistemdə 64-bitlik bir redaktorunuz varsa, belə bir şey yazacaqsınız:

```
$ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe'
-multiInst -notabbar -nosession -noPlugin"
```

Vim, Emacs və Notepad++ mətn redaktorları Linux və macOS və ya Windows sistemi kimi Unix əsaslı sistemlərdə developerlər tərəfindən tez-tez istifadə olunan populyar mətn redaktorlarıdır. Əgər başqa bir redaktorunuzdan və ya 32-bitlik versiyadan istifadə edirsinizsə, git config core.editor əmrləri bölməsində Git ilə sevdiyiniz redaktorunuzu necə qurmağınızla bağlı xüsusi təlimatları tapa bil ərsiniz.

Əgər redaktorunuzu belə qurmursunuzsa, Git onu işə salmağa çalışdığı zaman h əqiqətən də çaşdırıcı vəziyyətə düşə bilərsiniz. Windows sistemindəki bir nümunə Git-in başlatdığı düzəliş zamanı vaxtından əvvəl bitmiş Git-i daxil edə bilər.

Parametrlərinizi Yoxlayın

Konfiqurasiya parametrlərinizi yoxlamaq istəyirsinizsə, Git'in həmin nöqtədə tapa biləcəyi bütün parametrləri sadalamaq üçün *git config --list* əmrindən istifadə edə bilərsiniz:

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Açarları bir dəfədən çox görə bilərsiniz, çünki Git eyni düyməni müxtəlif sənədlərdən (məsələn, `/etc/gitconfig` və `~/.gitconfig`) oxuyur. Bu vəziyyətdə Git gördüyü hər bir unikal açar üçün son d əyərdən istifadə edir. Git'in müəyyən bir açarının dəyərinin nə olduğunu `git config <key>`yazmaqla yoxlaya bilərsiniz:

```
$ git config user.name
John Doe
```

Git eyni konfiqurasiyalı dəyişən dəyərini birdən çox fayldan oxuya bildiyindən, bu dəyərlərdən biri üçün gözlənilməz bir dəyərin olması və bunun səbəbini bilməm əyiniz mümkündür. Buna bənzər hallarda, Git'i həmin dəyər üçün *origin*'ə soruşa bilərsiniz və bu dəyəri təyin edərkən hansı konfiqurasiya sənədinin söylədiyini sizə xəbər verəcəkdir:

```
$ git config --show-origin rerere.autoUpdate
file:/home/johndoe/.gitconfig   false
```

Kömək Almaq

Git'dən istifadə edərkən köməyə ehtiyacınız olarsa, istənilən Git əmrinə dair ətraflı təlimat səhif əsinə (manpage) keçid etmək üçün 3 eyni yol mövcuddur.

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

Məsələn, `git config` əmri üçün manpage yardımına əmrini icra edərək çata bilərsiniz.

```
$ git help config
```

Bu əmrlər yaxşıdır, çünki siz onlar istənilən yerdən, hətta oflayn olduqda belə istifadə edə bil ərsiniz. Əgər manpage-lər və bu kitab bəs etməzsə və sizin şəxsi yardıma ehtiyacınız olarsa, https://freenode.net adresindən tapa biləcəyiniz Freenode IRC serverindəki `#git` və ya `#github` kanallarını yoxlaya bilərsiniz Bu kanallar adətən Git haqqında bilgili və kömək etməyə açıq olan yüzlərlə insan tərəfindən dolu olurlar.

Əlavə olaraq, kömək üçün ətraflı yazılmış manpage-dən istifadə etmək istəməsəniz, Git əmrinin özəlliklərini xatırlamaq üçün daha yığcam çıxarış verən `-h` və ya `--help` seçimlərindən istifadə edə bilərsiniz:

```
$ git add -h
usage: git add [<options>] [--] <pathspec>...

    -n, --dry-run         dry run
    -v, --verbose         be verbose

    -i, --interactive     interactive picking
    -p, --patch           select hunks interactively
    -e, --edit            edit current diff and apply
    -f, --force           allow adding otherwise ignored files
    -u, --update          update tracked files
    --renormalize         renormalize EOL of tracked files (implies -u)
    -N, --intent-to-add   record only the fact that the path will be added later
    -A, --all             add changes from all tracked and untracked files
    --ignore-removal      ignore paths removed in the working tree (same as --no-all)
    --refresh             don't add, only refresh the index
    --ignore-errors       just skip files which cannot be added because of errors
    --ignore-missing      check if - even missing - files are ignored in dry run
    --chmod (+|-)x        override the executable bit of the listed files
```

Qısa Məzmun

Git'in nə olduğunu və əvvəllər istifadə etdiyiniz mərkəzləşdirilmiş versiya idarəetmə sistemlərind ən nə ilə fərqləndiyi haqqında əsas anlayışınız olmalıdır. Artıq sisteminizdə şəxsi kimliyinizlə qurulmuş işləyən bir Git versiyasına sahib olmalısınız. İndi bəzi Git əsaslarını öyrənməyin zamanıdır.

Git'in Əsasları

Git ilə davam edə bilmək üçün yalnız bir fəsil oxuya bilsəniz, o bu fəsildir. Bu fəsildə vaxtınızı Git ilə birlikdə keçirməyə sərf edəcəyiniz işlərin böyük əksəriyyətini yerinə yetirmək üçün lazım olan hər bir əsas əmri əhatə edir. Fəslin sonuna qədər bir deponu konfiqurasiya edə və işə sala, faylları izləməyə başlamağı və dayandırmağı, dəyişiklikləri səhnələşdirməyi və commit etməyi bacarmalısız. Bəzi faylları və fayl nümunələrini ignore etmək üçün Git'i necə quracağınızı, səhvləri necə tez və asanlıqla necə geri qaytaracağımızı, layihənizin tarixçəsinə necə baxacağınızı və commit'lər arasındakı dəyişiklikləri necə görəcəyinizi və remote depo'lardan necə push və pull edə biləcyinizi göstərəcəyik.

Git Deposunun Əldə Edilməsi

Siz adətən 2 yoldan biri ilə Git deposunu əldə edirsiniz bunlar aşağıdakılardır; 1. Hal-hazırda versiya nəzarəti altında olmayan bir lokal qovluğu götürə və beləliklə də Git deposuna çevirə bil ərsiniz. 2. Mövcud olan Git deposunu başqa bir yerdən klonlaşdıra bilərsiniz. Hər iki halda da Git deposu ilə işə hazırsınız.

Mövcud Bir Qovluqda Deponu İşə Salma

Hal-hazırda versiya nəzarəti altında olmayan bir layihə qovluğunuz varsa və onu Git ilə idarə etm əyə başlamaq istəyirsinizsə, əvvəlcə bu layihənin qovluğuna getməlisiniz.Əgər bunu heç etm əmisinizsə, işlədiyiniz sistemdən asılı olaraq biraz fərqli görünə bilər:

for Linux:

```
$ cd /home/user/my_project
```

for macOS:

```
$ cd /Users/user/my_project
```

for Windows:

```
$ cd C:/Users/user/my_project
```

və növü:

```
$ git init
```

Bu, bütün zəruri depo sənədlərinizi – Git depo skeletini saxlayan `.git` adlı yeni bir alt bölmə yaradır. Bu anda layihənizdə heç bir şey hələ izlənilməyib. (Yeni yaratdığınız `.git` qovluğunda tam olaraq hansı sənədlərin olduğu barədə daha çox məlumat əldə etmək üçün Git'in Daxili İşləri-a baxın)

Mövcud faylları (boş qovluqdan fərqli olaraq) idarə edən versiyaya başlamaq istəyirsinizsə, ilk olaraq, həmin faylları izləməyə başlamalısınız. Bunu baxmaq istədiyiniz faylları bir neçə git əmrini əlavə etməklə yerinə yetirə bilərsiniz. Bunu izləmək istədiyiniz faylları təyin edən bir neçə `git add` əmrləri ilə yerinə yetirə bilərsiniz:

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'Initial project version'
```

Bütün bu əmrlərin bir dəqiqədə nələr etdiyinin üzərindən keçəcəyik. Bu anda, sizin izlənilən fayllardan ibarət git deponuz və ilkin commit'iniz var.

Mövcud Deponu Klonlaşdırmaq

Əvvəlcədən mövcud olan git deposunun kopyasını əldə etmək istəyirsinizsə, məsələn hansısa proyektə töhfə verəcəksinizsə, ehtiyacınız olan əmr `git clone`-dur. Subversion kimi digər VNS sistemləri ilə tanış olsanız, əmrin "klon" olduğunu və "yoxlama" olmadığını görəcəksiniz. Bu vacib bir fərqdir ki - Git sadəcə işləyək olan bir kopyanı əldə etmək əvəzinə, serverdə olan bütün m əlumatların tam surətini alır. Siz `git clone` əmrini işlətdiyinizdə proyektin tarixindəki hər bir faylın hər bir versiyasını default olaraq yükləmiş olursunuz. Fakt budur ki, server disklәriniz pozulsa, adətən yaxın olan hər hansı klonu hər hansı müştəridə istifadə edib serveri klonlandığı v əziyyətinə geri qaytara bilərsiniz. (Ola bilər ki bəzi server-side-hook kimi şeyləri itirəsiniz lakin versiyalanmış bütün məlumatlar orada olacaq.- əlavə məlumat üçün Serverə Serverdə Git Əldə Etmək yüklənilməsinə baxın)

`git clone <url>` ilə bir depo klonlayın. Məsələn, `libgit2` adlı Git əlaqəli kitabxananı klonlaşdırmaq istəyirsinizsə, bunu edə bilərsiniz:

```
$ git clone https://github.com/libgit2/libgit2
```

`libgit2` adlı qovluq yaradılır, içərisindəki `.git` qovluğluğu işə salınır, həmin deponun bütün m əlumatları yüklənir və son olaraq qeyd olunmuş kopya yoxlanılır. Yenicə yaradılan yeni `libgit2` qovluğuna daxil olsanız, orada işləyən və istifadəyə hazır olan layihə sənədlərini görəcəksiniz. Deponu `libgit2`-dən başqa bir qovluğun içinə klonlaşdırmaq istəyirsinizsə, yeni bir qovluq adını əlavə bir arqument olaraq təyin edə bilərsiniz;

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

Bu əmr əvvəlki ilə eyni şeyi edir, lakin hədəf qovluğu `mylibgit` adlanır. Git istifadə edə biləcəyiniz bir sıra müxtəlif ötürmə protokollarına malikdir. Əvvəlki nümunədə `https://` protokolu istifadə olunur, ancaq SSH ötürmə protokolundan istifadə edən `git://` və ya `user@server:path/to/repo.git` görə bilərsiniz. Serverdə Git Əldə Etmək-də, Git depolarınıza və hər birinin üstünlük və əksiklikl ərinə daxil olmaq üçün qura biləcəyiniz bütün mümkün seçimləri təqdim edəcəkdir.

Depoda Dəyişikliklərin Qeyd Edilməsi

Bu nöqtədə, artıq lokal maşınızda düzgün şəkildə işləyən Git deposu olmalıdır hansı ki qarşınızdakı bütün sənədlərin *yoxlanılması* və *kopyalanması* ona daxildir. Tipik olaraq, layihə yazmaq ist ədiyiniz vəziyyətə çatdığı zaman dəyişiklik etməyi və həmçinin həmin dəyişikliklərin görüntülərini yerləşdirə bilmək imkanına malik olacaqsınız.

Unutmayın ki, işlək qovluqdakı hər bir fayl iki haldan birində ola bilər: *izlənilmiş* və ya *izlənilm əmiş*. İzlənmiş fayllar son görüntülərdə olan fayllardır; və həmin bu fayllar dəyişdirilə, də yişdirilməyə və ya səhnələşdirilə bilər. Bir sözlə, izlənən fayllar Git'in tanıdığı fayllardır.

Bəs işlənməmiş və ya yüklənməmiş fayllar nədir?- Bu fayllar işçi qovluğunuzdakı son görüntüdə və quruluş sahənizdə olmayan hər hansısa bir fayllardır. İlk depo klonlaşdırıldığı zaman, bütün sən ədləriniz izlənəcək və düzəldilməyəcək, çünki Git onları artıq yoxlayıb və siz heç bir şey düzəltm əmisiniz.

Faylları redaktə edərkən Git onları dəyişdirilmiş kimi görür, bunun səbəbi sonuncu əmrinizi icra etdikdən bəri onları dəyişmisiniz. Siz iş gördüyünüz zaman dəyişdirilmiş fayllar seçilərək mərhələ lənir və daha sonra mərhələlənmiş dəyişikliklər commit olunur və bu proses dövr şəkildə təkrar olunur.

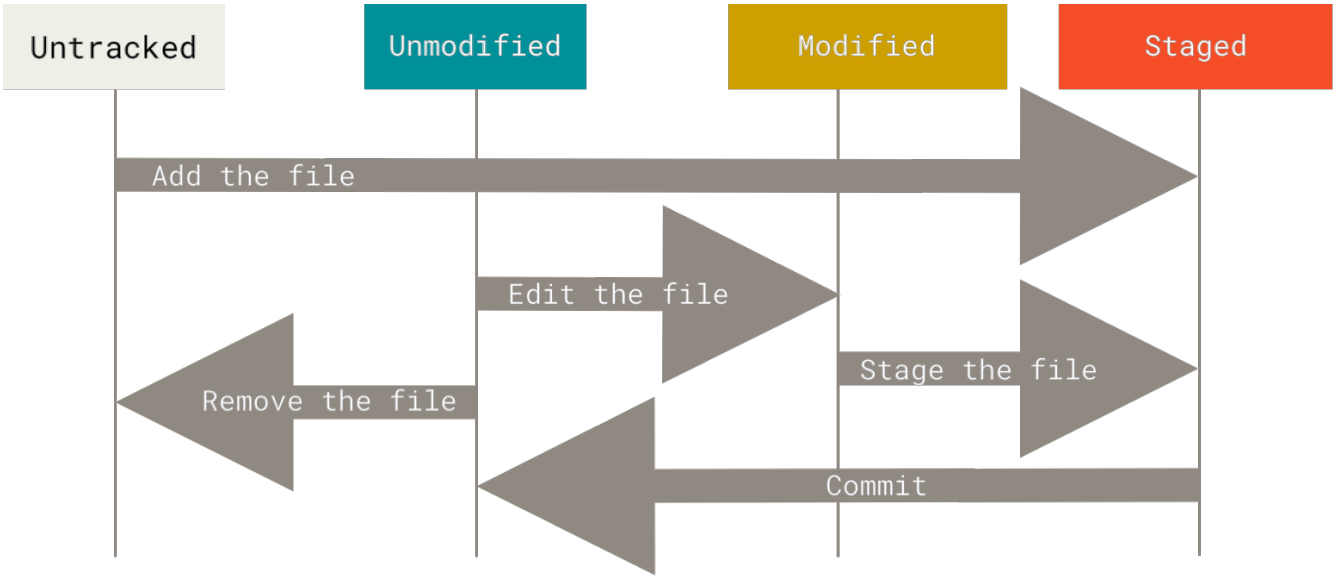


Figure 8. Fayllarınızın Statusunun Vaxtı

Fayllarınızın Vəziyyətinin Yoxlanılması

Faylların hansı vəziyyətdə və harda olduğunu müəyyənləşdirmək üçün istifadə olunan əsas alət `git status` əmridir. Bu əmri birbaşa klondan sonra işləsəniz, bunun kimi bir şey görməlisiniz:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Bu isə təmiz bir iş qovluğunun olması deməkdir; başqa sözlə izlənilən faylların heç biri d

əyişdirilmir. Git həmçinin əlavə edilməmiş faylları görmür və həmin fayllar burada siyahıya alınacaqdır. Sonda əmr sizə hansı branch'da olduğunuzu söyləyir və serverdəki eyni branch'dan ayrılmadığınızı bildirir. Hazırda da bu branch həmişə master funksiyasındadır. Git'də Branch branch'lar və arayışları ətraflı şəkildə araşdıracaq. İş funksiyası bundan ibarətdir.

Gəlin, layihənizə yeni bir fayl, sadə `README` faylı əlavə edək. Bu zaman əgər fayl əvvəllər yox idisə və `git status`-unu işlədirsinizsə, yüklənməmiş faylınızı belə görəcəksiniz:

```
$ echo 'My Project' > README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README

nothing added to commit but untracked files present (use "git add" to track)
```

Sizin hazırki vəziyyətinizdə "İzlənilməyən fayllar" başlığının altında yeni `README` faylınızın izlə nilmədiyini görə bilərsiniz. İzlənilməmiş fayllar dedikdə əsasən Git'in əvvəlki snapshot`da (commit) olmayan fayllar nəzərdə tutulur. Sən onu açıq şəkildə bildirməyənədək Git onu sənin commit snapshotuna daxil etməyə başlamayacaq.Təsadüfən yaradılan ikili faylları və başqa faylları hansı ki heç onları daxil etməyi düşünmürsünüz və beləliklə də daxil etməyə başlamırsınız. `README` daxil olmaqla başlamaq istəsəniz,bu zaman faylı izləməyə başlayırsınız.

Yeni Faylların Izlənməsi

Yeni bir faylı izləməyə başlamaq üçün `git add` əmrindən istifadə edirsiniz. `README` faylını izləməyə başlamaq üçün bunu işlədə bilərsiniz:

```
$ git add README
```

Status əmrinizi yenidən işləsəniz, `README` faylınızın izlənildiyini və mərtəbəli olaraq commit`l əndiyini görə bilərsiniz:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)

    new file:   README
```

Bunu da qeyd edə bilərsiniz ki,o mərtəbəlidir çünki `Dəyişikliklər commit olunub` başlığı altında verilir. Bu anda commit etsəniz `git add` əmrini işlətdiyiniz anda faylın versiyası növbəti tarixi

snapshotda olacaq. Xatırlaya bilərsiniz ki, əvvəllər `git init` işlətməyə başladığınız zaman `git add <files>` əmrindəki `faylları` yəni qovluğunuzdakı faylları izləməyə başladınız. `git add` əmri ya bir fayl ya da bir qovluğu adlandırır; əgər bu bir qovluqdursa, əmr həmin qovluqdakı bütün faylları rekursiv şəkildə əlavə edir.

Dəyişdirilmiş Faylların Quruluşu

Gəlin izlənilən bir faylı dəyişdirək. Əvvəllər izlənilən `CONTRIBUTING.md` adlı bir faylı dəyişdirsəniz və yenidən `git status` əmrinizi işə salsanız, bu kimi bir şey əldə edirsiniz:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

`CONTRIBUTING.md` faylı “Dəyişikliklər commit üçün mərhələnməyib” adlı bir bölmə altında görünür – bu isə izlənilən bir faylın iş qovluğunda dəyişdirildiyini, lakin hələ də mərhələləndirilməməsi deməkdir. Onu mərhələli etmək üçün çox funksiyalı əmr olan `git add` əmrindən istifadə olunur və bu əmrdən - yeni sənədləri izləməyə, faylları mərhələləndirməyə və həll edildiyi kimi birləşmə ziddiyyətli faylları qeyd etmək kimi digər işləri istifadə etmək üçün də istifadə edirsiniz. “Bu faylı layihəyə əlavə et” əvəzinə “bu məzmunu növbəti comittə əlavə et” kimi düşünmək daha faydalı ola bilər. Gəlin indi `CONTRIBUTING.md` faylını mərhələndirmək üçün `git add` əmrini və daha sonra `git status`-unu yenidən işə salaq:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Hər iki fayl mərhələləşdirilib və növbəti commit`inizə daxil olacaq. Bu anda bunu etməmişdən əvvəl `CONTRIBUTING.md`-də etmək istədiyiniz bir kiçik dəyişikliyi xatırlayırsınız. Onu yenidən açırsınız və bu dəyişikliyi edirsiniz və commit üçün hazırsınız. Bununla belə, gəlin `git status`-unu bir daha işə salaq:

```
$ vim CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Bu necə olur? İndi `CONTRIBUTING.md` həm mərhələli, həm də mərhələsiz olaraq siyahıya alınmışdır.Bəs bu necə mümkündür? Məlum olur ki, `git add` əmrini əlavə edərkən faylı mərhələlə rə ayırır.İndi commit etsəniz, görərsiniz ki, `CONTRIBUTING.md` versiyası, `git add` əmrini son dəfə işl ətdiyiniz kimidir, `git commit` işlətdiyiniz zaman işlək qovluqda göründüyüz faylın versiyası deyil. `git add` işlədikdən sonra bir fayl dəyişdirirsinizsə, faylın son versiyasını mərhələlərə ayırmaq üçün `git add`-ı yenidən işə salmalısınız:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Qısa Status

Gitin nəticə çıxarışı çox ətraflı həm də çox uzundur. Etdiyiniz dəyişikləri daha yığcam şəkildə görə bilməyiniz üçün gitin qısa işarəsi də mövcuddur. `git status -s` və ya `git status --short` əmrlərini işlətsəniz əmrdən daha sadə çıxarış əldə edəcəksiniz:

```
$ git status -s
 M README
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

İzlənməyən yeni sənədlər var `??` əlavə edilmiş yeni fayllarda `A`, dəyişdirilmiş fayllarda isə `M` və s var. Çıxışda iki sütun var ki bunlardan - sol tərəfdə olan sütun quruluş sahəsinin vəziyyətini, sağ t ərəfdə olan isə sütun işçi ağacının vəziyyətini göstərir. Məsələn, bu çıxışda, `README` faylı işçi qovluğunda dəyişdirilir, lakin hələ mərhələli deyil, `lib/simplegit.rb` faylı isə dəyişdirilir və mərhə lələrə ayrılır. `Rakefile` dəyişdirildi, mərhələlərə ayrıldı və sonra yenidən dəyişdirildi, buna görə h əm mərhələli, həm də sabit olmayan dəyişikliklər vardır.

Nəzərə Alınmayan Fayllar

Siz tez-tez Git'in avtomatik olaraq əlavə etməsini və izlənilməyənləri göstərməyə çalışan bir faylın olacağını görəcəksiniz. Ümumilikdə, avtomatik olaraq sistem tərəfindən yaradılan fayllar vardır ki, bunlara giriş faylları aid edilir. Belə hallarda, `.gitignore` kimi adlandırılan bir fayl siyahısı nümun ələri yarada bilərsiniz. Budur `.gitignore` faylına bir nümunə:

```
$ cat .gitignore
*.[oa]
*~
```

İlk sətir, Git`ə sonluğu `.o` və yaxud `.a` ilə bitən istənilən sənədləri yəni kodunuzu düzəltməyin m əhsulu ola biləcək obyekt və arxiv faylları nəzərə almamasını bildirir.İkinci sətir, Git`in müvəqqəti faylları qeyd etmək üçün Emacs kimi bir çox mətn redaktoru tərəfindən istifadə olunan bir tilde(`~`) ilə bitən bütün faylları nəzərə almadığını bildirir. Siz həmçinin log,tmp və ya pid qovluğunu; avtomatik olaraq yaradılan sənədləri və başqalarını da daxil edə bilərsiniz.Yeni depo üçün `.gitignore` faylını qurmaq ümumilikdə yaxşı fikirdir, buna görə də siz git deponuzdakı faylların birdən commit olunmasını istəmirsiniz.

`.gitignore` faylında nümunələr üçün qoya biləcəyiniz qaydalar bundan ibarətdir:

- Boş xəttlər və ya `#` başlayan sətirlər nəzərə alınmır.
- Standart qlob naxışları və bütün bunlar işçi ağacında tətbiq olunacaq.
- Rekursivliyə yol verməmək üçün naxışları irəli zolaqla (`/`) başlaya bilərsiniz.
- Bir qovluğu təyin etmək üçün nümunələri irəli bir xəttlə (`/`) bitirə bilərsiniz. *Nida (!) ilə başlayaraq bir nümunəni rədd edə bilərsiniz.

Qlob naxışlar gündəlik olaraq sadələşdirilən ifadələr kimidir. Bir ulduz (`*`) sıfır və ya daha çox simvola uyğun gəlir; `[abc]` mörtərizədəki istənilən simvola uyğundur (bu vəziyyətdə a,b və ya c); bir sual işarəsi (`?`) bir simvola uyğun gəlir; defislə ayrılmış simvolları əhatə edən mörtərizələr (`[0-9]`) aralarında olan hər hansı bir simvola uyğundur (bu vəziyyətdə 0 ilə 9 arasında) iç-içə qovluqları uyğunlaşdırmaq üçün 2 `**` ulduzdan da istifadə edə bilərsiniz; `a/**/z` `a/z`, `a/b/z`, `a/b/c/z` və s.

Budur başqa bir nümunə ``.gitignore` qovluğu:

```
# ignore all .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO
/TODO

# ignore all files in any directory named build
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories
doc/**/*.pdf
```

GitHub, layihəniz üçün bir başlanğıc nöqtəsi istəyirsinizsə https://github.com/github/gitignore-da sizin üçün onlarla layihə və dildə ``.gitignore` fayl nümunəl ərinin kifayət qədər əhatəli siyahısının saxlanıldığını görəcəksiniz..

Ən sadə halda yəni normal olaraq bütün depolarda olan `.gitignore` faylı deponuzun əsas hissəsində vardır. Bununla birlikdə alt qovluqlarda əlavə `.gitignore` sənədlərinin olması da mümkündür.Bu iç-içə `.gitignore` sənədlərind əki sənədlər yalnız yerləşdikləri qovluğun altındakı sənədlərə aiddir. Fayllar özünün aid olduğu qovluğun altında yerləşir. Linux deposunda 206 `.gitignore` faylı var.

Detallara baxmaq üçün `.gitignore` baxa bilərsiniz.

Mərhələli və Mərhələsiz Dəyişikliklərə Baxış

`git status` əmri sizin üçün çox qeyri-müəyyəndirsə - yalnız hansı sənədlərin dəyişdirildiyini deyil, nəyi dəyişdirdiyinizi bilmək istəyirsiniz - `git diff` əmrini istifadə edə bilərsiniz. Daha sonra `git diff`-u daha ətraflı açacağıq, ancaq bu iki suala cavab vermək üçün çox güman ki, istifadə edəc əksiniz: Nəsə dəyişmisiniz, amma hələ mərhələlərə ayrılmayıb? Və nə mərhələlərə ayrılmışdır ki,siz onları commit edəcəksiniz? `git status` bu suallara ümumiyyətlə fayl adlarını sadalayaraq cavab versə də, `git diff` sizə əlavə edilmiş və çıxarılan sətirləri göstərir - patch kimi.

Deyək ki, yenidən `README` faylını redaktə etdiniz və mərhələlərə ayırdınız və sonra `CONTRIBUTING.md` faylını mərhələlərə ayırmadan redaktə edin. `git status` əmrinizi işlədirsinizsə, yenidən belə bir şey görürsünüz:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Dəyişdirdiyiniz, lakin hələ mərhələlərə ayrılmamış olduğunu görmək üçün sadəcə `git diff` yazın, başqa heç bir arqument ehtiyac yoxdur:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

Bu əmr işçi qovluğunuzdakılar ilə quruluş sahənizdəkiləri müqayisə edir. Nəticə hələ də mərhələl ərə ayrılmamış dəyişiklikləri sizə bildirir.

Nəyi mərhələlərə ayırdığınızı görmək üçün `git diff --staged` əmrindən istifadə edə biləcəksiniz. Bu əmr mərhələli dəyişikliklərinizi son əmrinizlə müqayisə edir:

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

Qeyd etmək vacibdir ki, `git diff` özündə son commitdən bəri baş vermiş dəyişiklikləri göstərmir – yalnız hələ də mərhələlərə ayrılmayan dəyişikliklərdən başqa. Bütün dəyişikliklərinizi mərhələlərə ayırmısınızsa, `git diff` heç bir nəticə verməyəcək.

Başqa bir misal üçün, `CONTRIBUTING.md` faylını mərhələlərə ayırıb və sonra onu redaktə etsəniz,mə rhələli fayldakı dəyişiklikləri və sabit olmayan dəyişiklikləri görmək üçün `git diff` istifadə edə bil ərsiniz. Əgər bunu görürsünüzsə:

```
$ git add CONTRIBUTING.md
$ echo '# test line' >> CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

İndi nəyin hələ də mərhələlərə ayrılmadığını görə bilmək üçün `git diff`-dən istifadə edə bilərsiniz:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 643e24f..87f08c8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ at the
 ## Starter Projects

 See our [projects
list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).
+# test line
```

Və `git diff --cached` indiyənə qədər mərhələlərə ayırdıqlarınızı görə bilmək üçündür. (`--staged` və `--cached` sinonimdir.):

```
$ git diff --cached
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

Git Diff Xarici Alətdə

Kitabın qalan hissəsi ərzində `git diff` əmrini müxtəlif yollarla istifadə etməyə davam edəcəyik. Bu diff`lərə baxmağın başqa yollarıda vardır ki, əgər siz görüntü proqramı əvəzinə qrafik və ya xarici görüntülü proqrama üstünlük verəsəniz. `git diff` əvəzinə `git difftool` işlədirsinizsə, ortaya çıxan vimdiff və daha çox (kommersiya məhsulları daxil olmaqla) kimi proqramlarda bu diff`lərdən birini görə bilərsiniz. Sisteminizdə nə olduğunu görmək üçün `git difftool --tool-help` işlədin.

Dəyişikliklərinizin Commit'lənməsi

Artıq mərhələlərə ayrılmış sahəniz istədiyiniz şəkildə qurulduğundan dəyişikliklərinizi edə bil ərsiniz. Unutmayın ki, hələ də mərhələlərə ayılmamış bir şey –yəni yaratdığınız və dəyişdirdiyiniz hər hansı bir fayl onları redaktə etdiyinizdən bəri işləməmişdir - buna görə də bu commit`ə daxil olmayacaq. Diskinizdə dəyişdirilmiş fayllar kimi qalacaqlar. Bu vəziyyətdə, bildirək ki, `git status`-u son dəfə işlətdikdə hər şeyin səhnəyə mərhələlərə ayrıldığını gördünüz, buna görə dəyişiklikl ərinizi commitləməyə hazırsınız. Commit etməyin ən sadə yolu `git commit` yazmaqdır:

```
$ git commit
```

Doing so launches your editor of choice.

Bu, shell`nizin `EDITOR` mühitinin dəyişkənliyi ilə təyin olunur - adətən vim və ya emacs, buna baxmayaraq `git config --global core.editor` əmrindən istifadə edə rək istədiyiniz ilə konfiqurasiya edə bilərsiniz.Başlanğıc.

Redaktor aşağıdakı mətni göstərir (bu nümunə Vim ekranıdır):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#   new file:   README
#   modified:   CONTRIBUTING.md
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

Default commit mesajında şərh edilmiş `git status` əmrinin son çıxarışını və üstündə bir boş sətir olduğunu görürsünüz. Bu şərhləri silə və commit olunmuş mesajınızı yaza bilərsiniz və ya nə commit'lədiyinizi yadda saxlamağınız üçün onları tərk edə bilərsiniz.

Dəyişdirdiyiniz şeyi daha aydın xatırlatmaq üçün,`git commit`-ə `-v` seçimini verə bilərsiniz. Redaktorunuzdakı nə dəyişiklikləri commit'lədiyinizi dəqiqliyi ilə görə bilərsiniz.

Redaktordan çıxdıqda, Git commit mesajlı commit'inizi yaradır (şərhlərlə və fərq ləğv olunur).

Alternativ olaraq, commit mesajınızı `commit` əmrinə uyğun olaraq bir `-m` flag'dan sonra göstərərək yaza bilərsiniz:

```
$ git commit -m "Story 182: fix benchmarks for speed"
[master 463dc4f] Story 182: fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

Indi ilk commitinizi yaratdınız! Gördüyünüz kimi commit özü haqqında sizə çıxarışlar vermişdir;hansı branch'la siz commit etdiniz (`master`),nə qədər fayl dəyişdirilldiyini statiskalar commitdəki əlavə edilmiş və silinmiş xəttlər haqqındadır.

Siz hələ mərhələlərə ayrılmamış nə isə orada dəyişdirilmişdir; tarixinizə əlavə etmək üçün başqa bir commit edə bilərsiniz. Hər dəfə bir commit yerinə yetirdiyiniz zaman, geri qaytara biləcəyiniz və ya sonrakı ilə müqayisə edə biləcəyiniz layihənizin snapshotunuzu qeyd edirsiniz..

Mərhələləri Ayrılmış Sahəni Atlamaq

Sənətkarlıq commitləri istədiklərinizi dəqiq yerinə yetirmək üçün inanılmaz dərəcədə faydalı ola bilsə də, mərhələlərə ayırmaq sahəsi bəzən iş prosesinizdə lazım olduğundan biraz daha mürəkk əbdir. Mərhələlərə ayırma sahəsini atlamaq istəyirsinizsə, Git sadə bir qısa yol təqdim edir. `git command` əmrinə `-a` variantını əlavə edərək, commit etmədən əvvəl izlənilən hər bir faylı avtomatik olaraq mərhələ halına gətirir, `git add` hissəsini atlamağa imkan verir:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'Add new benchmarks'
[master 83e38c7] Add new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

Commit etməyinizdən əvvəl bu vəziyyətdə `CONTRIBUTING.md` faylına necə əlavə etməyiniz lazım olduğuna diqqət yetirin. Ona görə ki, `-a` flag'ı bütün dəyişdirilmiş faylları daxil edir. Bu əlverişlidir, amma diqqətli olun; bəzən bu flag istənməyən dəyişiklikləri daxil etməyinizə səbəb olacaqdır.

Faylların Silinməsi

Git'dən bir fayl çıxarmaq üçün onu izlənilmiş fayllarınızdan çıxartmalısınız (daha doğrusu, mərhələləyə ayrılmış sahənizdən çıxarın) və sonra commit etməlisiniz. `git rm` əmri bunu edir və h əmçinin növbəti dəfə ətrafınıza izlənməmiş fayl kimi görmədiyiniz üçün faylı iş qovluğunuzdan çıxarır.

Faylı sadəcə işçi qovluqdan çıxarırsınız, bu, çıxma `git status` çıxışınızdakı "Commit üçün mərhələlərə ayrılmamış dəyişikliklər" (yəni mərhələlərə ayrılmamış) altında göstərilir:

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    PROJECTS.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Sonra `git rm` ilə işləsəniz, bu, faylın silinməsini mərhələləndirir:

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    PROJECTS.md
```

Növbəti dəfə commit etdiyiniz zaman, fayl yox olacaq və artıq izlənilməyəcək. Fayl d əyişdirmisinizsə və ya əvvəlcədən mərhələlərə ayrılmış sahəyə əlavə etmisinizsə, `-f` seçimi ilə çıxarılmanı məcbur etməlisiniz. Bu, əvvəlcədən görünməmiş və Git'dən bərpa olunmayan m əlumatların təsadüfən çıxarılmasının qarşısını almaq üçün təhlükəsizlik xüsusiyyətidir.

Edə biləcəyiniz başqa faydalı bir şey, faylı iş ağacınızda saxlamaq, ancaq hazırlama sahənizdən çıxarmaqdır. Başqa sözlə, faylı sabit diskdə saxlamaq istəyə bilər, lakin artıq Git izləməməlidir. `.gitignore` faylınıza bir şey əlavə etməyi unutmusunuzsa və təsadüfən böyük bir giriş faylı və ya bir dəstə `.a` tərtib edilmiş fayllar kimi mərhələlərə ayırsanız bu xüsusilə faydalıdır. Bunu etmək üçün `--cached` seçimdən istifadə edin:

```
$ git rm --cached README
```

`git rm` əmrinə faylları, qovluqları və fayl-glob naxışlarını ötürə bilərsiniz. Bu kimi şeylər edə biləc əyinizi nəzərdə tutur:

```
$ git rm log/\*.log
```

`*` önündəki (`\`) qeyd edin. Bu, Git'in qabığındakı (shell) fayl adının genişlənməsinə əlavə olaraq, öz ad genişləndirməsini həyata keçirdiyinə görə lazımdır. Bu əmr `log/` qovluğunda `.log` genişlənm əsinə sahib olan bütün faylları silir. Və ya bu kimi bir şey edə bilərsiniz:

```
$ git rm \*~
```

Bu əmr adları `~` ilə bitən bütün faylları silir.

Daşınan Fayllar

Bir çox digər VNS sistemlərindən fərqli olaraq, Git fayl hərəkətini dəqiq izləmir. Git-də bir faylın adını dəyişsəniz, Git'də faylın adını dəyişdiyinizi bildirən heç bir metadata qeyd olunmur. Bununla belə Git, faktdan sonra bunu bilmək üçün olduqca ağıllıdır - bir az sonra isə fayl hərəkətini aşkarlamaqla məşğul olacağıq. Beləliklə, Git'in `mv` komandası olması bir az çaşdırıcıdır. Git'də bir faylın adını dəyişdirmək istəyirsinizsə, belə bir şey işlədə bilərsiniz:

```
$ git mv file_from file_to
```

və yaxşı işləyir. Əslində, bu kimi bir şey işlədib statusa baxsan, Git'in adını dəyişmiş bir fayl hesab etdiyini görərsiniz:

```
$ git mv README.md README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

Lakin, bu kimi bir şey işlətməyə bərabərdir:

```
$ mv README.md README
$ git rm README.md
$ git add README
```

Git bunun tamamilə adının dəyişdirildiyini göstərir, buna görə bir faylın bu şəkildə və ya mv əmri ilə dəyişdirilməsinin əhəmiyyəti yoxdur. Yeganə real fərq, `git mv`-nin üç əmrin əvəzinə bir əmrin olması - rahatlıq funksiyasıdır. Daha əhəmiyyətlisi, bir faylın adını dəyişmək üçün istədiyiniz hər hansı bir vasitədən istifadə edə bilərsiniz və commit etməməzdən əvvəl add/rm ünvanına müraciət edə bilərsiniz.

Commit Tarixçəsinə Baxış

Bir neçə əmr yaratdıqdan və ya mövcud bir commit tarixçəsi olan bir deponu klonlaşdırdıqdan sonra, yəqin ki, baş verənləri görmək üçün geri baxmaq istərdiniz. Bunu etmək üçün ən əsas və güclü vasitə `git log` əmridir.

Bu nümunələrdə `simplegit` adlı çox sadə bir layihədən istifadə olunur.

Layihəni əldə etmək üçün:

```
$ git clone https://github.com/schacon/simplegit-progit
```

Bu layihədə `git log` əmrini işlətdiyiniz zaman görünən belə bir nəticə əldə etməlisiniz:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    Remove unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    Initial commit
```

Default olaraq, heç bir arqument olmadan, `git log` həmin depoda commitlənən əmrləri tərs xronoloji qaydada sadalayır; yəni ən son tapşırıqlar ilk olaraq göstərilir. Gördüyünüz kimi, bu əmr hər bir commiti özünün SHA-1 çeki, müəllifin adı və e-poçtu, yazılmış tarixi və commit mesajı ilə sadalayır.

`git log` əmrində çox sayda və müxtəlif seçimlər istədiyinizi tam olaraq göstərmək üçün mövcuddur.

Daha faydalı seçimlərdən biri hər əməldə təqdim olunan fərqi (*patch* çıxışı) göstərən `-p` və ya `--patch`-dır. Yalnız son iki giriş göstərmək üçün `-2` istifadə kimi göstərilən girişlərinin sayını m əhdudlaşdıra bilərsiniz.

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.platform  =   Gem::Platform::RUBY
     s.name      =   "simplegit"
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com"
     s.summary   =   "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    Remove unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
     end

 end
-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end
```

Bu seçim hər girişin ardınca bir fərq ilə eyni məlumatı göstərir. Kodun yoxlanılması və ya bir əm əkdaşın əlavə etdiyi bir sıra müddətdə baş verənlərə tez baxmaq üçün çox faydalıdır. `git log` ilə bir sıra ümumiləşdirmə variantlarından da istifadə edə bilərsiniz.

Məsələn, hər bir əməl üçün bəzi qısaldılmış statistikanı görmək istəyirsinizsə, `--stat` seçimindən istifadə edə bilərsiniz:

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number

 Rakefile | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    Remove unnecessary test

 lib/simplegit.rb | 5 -----
 1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    Initial commit

 README           |  6 ++++++
 Rakefile         | 23 +++++++++++++++++++++
 lib/simplegit.rb | 25 +++++++++++++++++++++++
 3 files changed, 54 insertions(+)
```

`--stat` seçimi hər bir commit girişində dəyişdirilmiş faylların siyahısını, neçə faylın dəyişdirildiyini və həmin sənədlərdə neçə sətir əlavə olunduğunu və silindiyini görə bilərsiniz. Və bununla yanaşı məlumatın xülasəsini də sonunda qoyur.

Başqa bir həqiqətən faydalı seçim - `--pretty`-dır. Bu seçim log çıxışını standartdan başqa formatlara dəyişdirir. İstifadəniz üçün bir neçə əvvəlcədən qurulmuş seçim mövcuddur. Onlayn seçim hər bir commiti bir sətirdə yazdırır, bu çox sayda commitə baxdığınızda daha faydalı olacaqdır. Bundan əlavə, `qısa`, `dolğun` və `daha dolğun` variantlar çıxarışı təxminən eyni formatda göstərir, lakin müvafiq olaraq daha az və ya daha çox məlumatla:

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 Change version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 Remove unnecessary test
a11bef06a3f659402fe7563abf99ad00de2209e6 Initial commit
```

Ən maraqlı seçim, öz giriş-çıxış formatını təyin etməyə imkan verən `format`-dır. Bu, maşın analizi üçün çıxış hazırlayarkən xüsusilə faydalıdır - formatı dəqiq müəyyənləşdirdiyiniz üçün bilirsiniz ki, Git yeniləmələri ilə dəyişmir:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : Change version number
085bb3b - Scott Chacon, 6 years ago : Remove unnecessary test
a11bef0 - Scott Chacon, 6 years ago : Initial commit
```

Git log üçün faydalı seçimlər - Useful specifiers for `git log --pretty=format` , formatın qəbul etdiyi daha faydalı variantların bəzilərini sadalayır.

Table 1. Useful specifiers for `git log --pretty=format`

Specifier	Description of Output
`%H`	Commit hash
`%h`	Abbreviated commit hash
`%T`	Tree hash
`%t`	Abbreviated tree hash
`%P`	Parent hashes
`%p`	Abbreviated parent hashes
`%an`	Author name
`%ae`	Author email
`%ad`	Author date (format respects the --date=option)
`%ar`	Author date, relative
`%cn`	Committer name
`%ce`	Committer email
`%cd`	Committer date
`%cr`	Committer date, relative
`%s`	Subject

author və *committer* arasındakı fərqin nə olduğunu bilmək istəyirsiniz. Deməli müəllif əsəri əvvə lcə yazan şəxsdir, komissiyaçı isə əsəri son dəfə tətbiq etmiş şəxsdir. Beləliklə, bir layihəyə bir patch göndərsəniz və əsas üzvlərdən biri patch-ı tətbiq edərsə, bu halda ikiniz də kredit alırsınız - müəllif, komitənin üzvü də.

Bu fərqliliyi paylanmış Git-də bir az daha geniş şəkildə əhatə edəcəyik. `oneline` və `format` seçimləri `--graph` adlı başqa bir `log` seçimi ilə xüsusilə faydalıdır. Bu seçim filialınızı və birləşmə tarixinizi göstərən gözəl bir kiçik ASCII qrafikini əlavə edir:

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 Ignore errors from SIGCHLD on trap
*  5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Add method for getting the current branch
* | 30e367c Timeout code and tests
* | 5a09431 Add timeout protection to grit
* | e1193f8 Support for heads with slashes in them
|/
* d6016bc Require time for xmlschema
*  11d191e Merge branch 'defunkt' into local
```

Növbəti fəsildə budaqlamaq və birləşmə yolu ilə olduqda bu cür çıxış daha maraqlı olacaq.

Bunlar günlük çıxarmaq üçün yalnız sadə çıxış formatı `git log`-dur və orada daha çox şeylər var. Common options to `git log` bu günə qədər əhatə etdiyimiz seçimlər, həmçinin faydalı ola biləcək bəzi digər ümumi formatlaşdırma seçimlərini və log əmrinin nəticələrini necə dəyişdirdiyini sadalayır.

Table 2. Common options to `git log`

Option	Description
`-p`	Show the patch introduced with each commit.
`--stat`	Show statistics for files modified in each commit.
`--shortstat`	Display only the changed/insertions/deletions line from the --stat command.
`--name-only`	Show the list of files modified after the commit information.
`--name-status`	Show the list of files affected with added/modified/deleted information as well.
`--abbrev-commit`	Show only the first few characters of the SHA-1 checksum instead of all 40.
`--relative-date`	Display the date in a relative format (for example, “2 weeks ago”) instead of using the full date format.
`--graph`	Display an ASCII graph of the branch and merge history beside the log output.
`--pretty`	Show commits in an alternate format. Option values include oneline, short, full, fuller, and format (where you specify your own format).
`--oneline`	Shorthand for `--pretty=oneline --abbrev-commit` used together.

Giriş Çıxışın Məhdudlaşdırılması

Çıxış formatlama seçimlərinə əlavə olaraq, `git log` bir sıra faydalı məhdudlaşdırma seçimlərini də tələb edir; yəni, yalnız alt hissəni göstərməyinizə imkan verən seçimlər. Artıq belə bir seçim görmüsünüz - yalnız son iki nəticəni əks etdirən `-2` seçimi. Əslində edə bilərsiniz `-<n>`, burada `n` son commit `n` işləndiyini göstərmək üçün istənilən tamdır. Əslində, sizin tez-tez istifadə etm əyinizin ehtimalı azdır, çünki standart borularla Git bütün pager vasitəsi ilə çıxır, bir anda giriş çıxışının yalnız bir səhifəsini görürsünüz. Bununla birlikdə, `--since` və `--until` kimi seçimlər çox faydalıdır. Məsələn, bu əmr son iki həftədə edilən əmrlərin siyahısını alır:

```
$ git log --since=2.weeks
```

Bu əmr çox sayda formatla işləyir - "2008-01-15" və ya "2 il 1 gün 3 dəqiqə əvvəl" kimi nisbi bir tarix təyin edə bilərsiniz.

Həmçinin bəzi axtarış meyarlarına uyğun gələnləri siyahıya silə bilərsiniz. `--author` seçimi, müə yyən bir müəllif üzərində filter etməyə imkan verir və `--grep` seçimi mesaj commitində açar sözlər axtarmağa imkan verir.

`--author` və `--grep` axtarış meyarlarının birdən çoxunu göstərə bilərsiniz, bu da commitin nəticəsini (çıxarışını) hər hansı bir `--author` və nümunələrin hər hansı birinə uyğun gəlməsini məhdudlaşdıracaq; Bununla birlikdə, uyğunluq seçimi əlavə, bütün uyğun modellərə uyğun gələnləri `--grep` məhdudlaşdırır.

Digər həqiqətən faydalı bir filtr, bir simli və yalnız bu sətirdə baş verənlərin sayını dəyişdirən ə mrləri göstərən `-S` seçimidir (ümumiyyətlə Git-in “pickaxe” seçimi adlanır). Məsələn, müəyyən bir funksiyaya istinad əlavə edən və ya çıxarmış son commiti tapmaq istəyirsinizsə, çağıra edə bil ərsiniz:

```
$ git log -S function_name
```

`git log` bir filtr kimi keçmək üçün faydalı bir seçim yoludur. Dəqiqləşdirilmiş şəkildə bir qovluq və ya fayl adını göstərsəniz, log çıxarışının commitlənməsi ilə həmin fayllara bir dəyişiklik təqdim etməyi məhdudlaşdıra bilərsiniz. Bu həmişə son seçimdir və ümumiyyətlə yolları seçimlərdən ayırmaq üçün ikiqat tire (`--`) əvvəl olur. Options to limit the output of `git log` çıxışını (çıxarışını) məhdudlaşdırmaq üçün Seçimlərdə bu və arayışınız(rəy) üçün bir neçə digər ümumi variantları sadalayacağıq.

Table 3. Options to limit the output of `git log`

Option	Description
`-<n>`	Show only the last n commits
`--since`, `--after`	Limit the commits to those made after the specified date.
`--until`, `--before`	Limit the commits to those made before the specified date.
`--author`	Only show commits in which the author entry matches the specified string.
`--committer`	Only show commits in which the committer entry matches the specified string.
`--grep`	Only show commits with a commit message containing the string
`-S`	Only show commits adding or removing code matching the string

Məsələn, Git mənbə kodu tarixində dəyişən test sənədlərinin Junio Hamano tərəfindən oktyabr 2008-ci ildə commitləndiyini və birləşdirilmədiyini görmək istəyirsinizsə, bu kimi bir şey işlədə bil ərsiniz:

```
$ git log --pretty="%h - %s" --author='Junio C Hamano' --since="2008-10-01" \
   --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attributes are in use
acd3b9e - Enhance hold_lock_file_for_{update,append}() API
f563754 - demonstrate breakage of detached checkout with symbolic link HEAD
d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths
51a94af - Fix "checkout --track -b newbranch" on detached HEAD
b0ad11e - pull: allow "git pull origin $something:$current_branch" into an unborn
branch
```

Git mənbə kodu tarixində təxminən 40.000 commitdən bu əmr bu meyarlara uyğun 6-nı göstərir.

Birləşmə Commit Görüntüsünün Qarşısını Alır

Deponuzda istifadə olunan iş axınından asılı olaraq, giriş tarixinizdəki commitl ərin əhəmiyyətli bir faizinin yəni adətən çox məlumat verməyən commitlərin sadəcə birləşməsi mümkündür. Birləşmə ekranının giriş tarixinizi ləkələməsini qarşısını almaq üçün, günlük log seçimini əlavə edin `--no-merges`.

Ləğv Edilən İşlər (Geri qaytarılan)

İstənilən mərhələdə hansısa əməliyyatı və yaxud bir şeyi geri qaytarmaq istəyə bilərsiniz. Burada etdiyiniz dəyişiklikləri geri qaytarmaq üçün bir neçə əsas vasitəni nəzərdən keçirəcəyik. Diqqətli olun, çünki bu boşluqların bəzisini geri qaytara bilməzsiniz.Git'də səhv etdiyiniz zaman hər hansısa işinizi itirə biləcəyiniz sahələrdən biridir.

Ümumi geri qaytarılmaların biri çoxu erkən commit etdiyinizdə və bəlkə də bəzi faylları əlavə etməyi unutduğunuzda və ya commit mesajınızı qarışdırdığınız zaman baş verir. Bu commiti yenidən yerinə yetirmək istəyirsinizsə, unutduğunuz əlavə dəyişiklikləri düzəldin, onları mərhələ lərə ayırın və yenidən `--amend` seçimini istifadə edərək commitləyin:

```
$ git commit --amend
```

Bu əmr mərhələli sahənizi alır və commit üçün istifadə edir. Sonuncu commitinizdən bəri heç bir dəyişiklik etməmisiniz (məsələn, əvvəlki commitinizdən dərhal sonra bu əmri işə salırsınız), sonra sizin snapshotunuz tam olaraq eyni görünəcəkdir və bütün dəyişikliklər sizin commit mesajınızdır.

Eyni commit mesajı redaktoru işə düşür, lakin artıq əvvəlki commitlərinizin mesajı var. Mesajı h əmişəki kimi eyni şəkildə düzəldə bilərsiniz, ancaq o əvvəlki commitlərinizin üstünə yazlacaqdır.

Məsələn, commitləsəniz və commitə əlavə etmək istədiyiniz bir fayla dəyişiklik etməyi unutduğunuzu başa düşsəniz, belə bir şey edə bilərsiniz:

```
$ git commit -m 'Initial commit'
$ git add forgotten_file
$ git commit --amend
```

Bir tək commitlə başa vurursunuz - ikinci commit birincinin nəticələrini əvəz edir.

Sonuncu commitinizi dəyişdirərək, onu köhnə commiti itələyən və yerinə yeni t əkmilləşdirilmiş commitlə əvəz etməyi başa düşmək vacibdir. Effektiv olaraq, ə vvəlki əməl heç vaxt olmayıb və depozit tarixinizdə görünməyəcək.

Dəyişdirmə əmsalının açıq dəyəri, "Faylı əlavə etməyi unutmusunuz" və ya "Darn, sonuncu commitə yazmağı düzəlt" şəklindəki commit mesajları ilə depozit tarixç ənizi ləkələmədən, son commitinizə kiçik düzəlişlər etməkdir.

Mərhələli Bir Faylın Mərhələlərə Ayrılmaması

Növbəti iki bölmə sizə mərhələlərə ayrılmış sahənizi və iş qovluq dəyişikliklərinizlə necə işl ədiyinizi nümayiş etdirir. Gözəl tərəfi odur ki, həmin iki sahənin vəziyyətini müəyyənləşdirmək üçün istifadə etdiyiniz əmr onlara dəyişiklikləri necə geri qaytarmağınızı da xatırladır. Məsələn, deyək ki, iki fayl dəyişdirmisiniz və bunları iki ayrı dəyişiklik kimi commit etmək istəyirsiniz, amma təsadüfən `git add *` yazın və ikisini də düzəldin. İkisindən birini necə bağlamaq olar?

`git status` əmri sizə xatırladır:

```
$ git add *
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
    modified:   CONTRIBUTING.md
```

"Dəyişikliklər commitlənir" mətninin altından, və mərhələlərə ayrılmadığı üçün `git reset HEAD <file>...`-dan istifadə edildiyi deyilir. Beləliklə, `CONTRIBUTING.md` faylını açmaq üçün bu tövsiyədən istifadə edək:

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M   CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Komanda bir az qəribədir, amma işləyir. `CONTRIBUTING.md` faylı dəyişdirilir, lakin bir daha dayanmır.

Doğrudur, `git reset` təhlükəli bir əmr ola bilər, xüsusilə də `--hard` flag-ı təmin edirsinizsə. Bununla birlikdə, yuxarıda təsvir olunan ssenaridə, işçi qovluqdakı fayla toxunulmur, buna görə nisbətən təhlükəsizdir.

İndi bu sehrli çağırış, `git reset` əmri haqqında bilməyimiz üçün lazım olan bütün şeylərdir. Yenid ən qurmağıç `reset`-i necə edəcəyimizi və digər maraqlı məlumatları Reset Demystified'də necə düz əltməli olduğumuz haqqında daha çox məlumat verəcəyik.

Dəyişdirilmiş Faylın Dəyişdirilməməsi

`CONTRIBUTING.md` faylındakı dəyişiklikləri saxlamaq istəmədiyinizi başa düşdüyünüz zaman nə olacaq? Onu necə asanlıqla düzəliş edə bilərsiniz - sonuncu dəfə commit etdikdə (və ya əvvəlcə klonlaşdırıldığınızda və ya iş dəftərinizə daxil olduqda) göründüyü vəziyyətə qaytara bilərsiniz? Xoşbəxtlikdən, `git status`-u da bunu necə edəcəyinizi söyləyir. Son nümunə çıxışında, tənzimlə nməmiş sahə belə görünür:

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Etdiyiniz dəyişiklikləri necə ləğv edəcəyinizi açıq şəkildə izah edir. Gəlin deyilənləri edək:

```
$ git checkout -- CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

Dəyişikliklərin geri qaytarıldığını görə bilərsiniz.

Başa düşmək üçün vacibdir ki, `git checkout -- <file>` təhlükəli bir əmrdir. Bu zaman fayla etdiyiniz hər hansı bir yerli dəyişiklik yox ola bilər. Git sadəcə bu faylı ən son commitlənmiş versiya ilə əvəz edir. Bu saxlanmamış yerli dəyişiklikləri tamamilə bilməyincəyə qədər bu əmrdən heç vaxt istifadə etməyin.

Həmin fayla etdiyiniz dəyişiklikləri saxlamaq istəyirsinizsə, lakin hələ də onu aradan qaldırmaq lazımdırsa, GGit'də Branch etməyə davam edəcəyik; ümumiyyətlə isə bunlar daha yaxşı yoldur.

Unutmayın, Git-də commitlənən hər şey demək olar ki, həmişə bərpa edilə bilər. Hətta silinmiş branch-da olan və ya `--amend` commitlə yazılmış commitlər bərpa edilə bilər (Data Recovery m əlumatların bərpası üçün məlumatların bərpasına baxın).

Uzaqdan İşləmək

Hər hansı bir Git layihəsində əməkdaşlıq edə bilmək üçün uzaqdakı depolarınızı necə idarə edəc əyinizi bilməlisiniz. Uzaqdan yerləşdirilmiş depolar İnternetdə və ya şəbəkədə bir yerdə yerlə şdirilən layihənizin versiyalarıdır. Onlardan bir neçəsinə sahib ola bilərsiniz, bunların hər biri ümumiyyətlə read-only və ya sizin üçün read/write olur. Başqaları ilə əməkdaşlıq etmək, bu uzaq məsafəli depoların idarə edilməsini və iş bölüşmək lazım olduqda məlumatları onlara və onlardan push və pull etməyi özündə əks etdirir. Uzaq depoları idarə etmək və uzaq depoları necə əlavə etməyi, artıq yararsız olanları uzaqlaşdırmağı, müxtəlif uzaq filialları idarə etməyi və onları izlə nilən və ya izlənilməyəni müəyyənləşdirməyi və daha çoxunu əhatə edir.

Bu hissədə bu uzaqdan idarəetmə bacarıqlarından bəzilərini əhatə edəcəyik.

Uzaqdakı depolar lokal maşınınızda ola bilər.

"Remote" bir depo ilə işləməlisinizsə, əslində tamamilə mümkündür. Belə bir uzaq depo ilə işləmək, hər hansı digər uzaqdan idarə olunan depoda olduğu kimi, bütün standart fetching, pushing və pulling əməliyyatları daxildir.

Uzaqdan Göstərişləriniz

Konfiqurasiya etdiyiniz uzaq serverləri görmək üçün `git remote` əmrini işlədə bilərsiniz. Bu göst ərdiyiniz hər remote handle'ın qısa adlarını sadalayır.Deponuzu klonlaşdırmısınızsa, ən azından `origin`-i görməlisiniz - bu sizin klonlaşmış Git serverinizə verilən default addır:

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

Həmçinin, uzaqdan oxuyarkən və yazarkən istifadə ediləcək qısa ad üçün Git'in saxladığı URL'ləri sizə göstərən `-v` ilə göstərə bilərsiniz:

```
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
```

Birdən daha çox remote qurmusunuzsa, komanda bunların hamısını sadalayır.

```
$ cd grit
$ git remote -v
bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)
cho45     https://github.com/cho45/grit (fetch)
cho45     https://github.com/cho45/grit (push)
defunkt   https://github.com/defunkt/grit (fetch)
defunkt   https://github.com/defunkt/grit (push)
koke      git://github.com/koke/grit.git (fetch)
koke      git://github.com/koke/grit.git (push)
origin    git@github.com:mojombo/grit.git (fetch)
origin    git@github.com:mojombo/grit.git (push)
```

Bu, istifadəçilərdən hər hansı birindən çox asanlıqla pull contributions edə biləcəyimiz deməkdir. Əlavə olaraq bunlardan birini və ya daha çoxunu pull etməyə icazə verə bilərik, baxmayaraq ki, burada qeyd edə bilmərik.

Bu remote'lərin müxtəlif protokollar istifadə etdiyinə diqqət yetirin; bu barədə daha çox məlumatı bir Serverdə Git Əldə Etmək-də əhatə edəcəyik.

Uzaqdan Depolara Əlavə Edilməsi

`git clone` əmrinin sizin üçün uzaqdan `origin` necə əlavə etdiyini bir neçə nümayiş etdik və qeyd etdik. Yeni bir uzaqdan idarə etmə necə əlavə etmək olar. Asanlıqla istinad edə biləcəyiniz qısa ad kimi yeni bir uzaq Git deposuna əlavə etmək üçün `git remote add <shortname> <url>` əlavə edin:

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
pb  https://github.com/paulboone/ticgit (fetch)
pb  https://github.com/paulboone/ticgit (push)
```

İndi bütün URL əvəzinə əmr sətirindəki `pb` sətirindən istifadə edə bilərsiniz. Məsələn, Paulun bütün məlumatlarını götürmək istəyirsinizsə, ancaq hələ deponuzda yoxdursa, `git fetch pb` işlədə bil ərsiniz:

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

Paulun `master` branch'ı indi `pb/master` olaraq lokal olaraq əlçatandır - onu branch'lardan birinə birləşdirə bilərsiniz və ya yoxlamaq istəsəniz lokal bir branch'ı yoxlaya bilərsiniz. Branch'ların nə dən ibarət olduğunu və onlardan daha ətraflı necə istifadə edəcəyimizi Git'də Branch-dan öyrənəc əyik.

Uzaqdan Fetching və Pulling

Gördüyü kimi, remote layihələrinizdən məlumat əldə etmək üçün işləyə bilərsiniz:

```
$ git fetch <remote>
```

Komanda o uzaq layihəyə çıxır və sizdə olmayan o remote layihədən olan bütün məlumatları çıxarır, yəni pull edir. Bunu etdikdən sonra, istənilən vaxt birləşdirə və ya yoxlaya biləcəyiniz uzaqdan bütün branch'lara istinadlar olmalıdır.

Deponu klonlaşdırsanız, əmr avtomatik olaraq uzaqdan yerləşdirilən depoları "origin" adı altında əlavə edir. Beləliklə, `git fetch origin` onu klonlaşdırdığınızdan (və ya sonuncu götürdüyünüzdən) sonra bu serverə sövq edilmiş hər hansı yeni bir işi gətirir. Qeyd etmək vacibdir ki, `git fetch` əmri yalnız məlumatları lokal depoya yükləyir - onu avtomatik olaraq hər hansı bir işlə birləşdirmir və ya hazırda olan işinizi dəyişdirmir. Hazır olanda manual olaraq işinizə birləşdirməlisiniz.

Cari branch'ınız uzaq bir branch'ı izləmək üçün qurulmuşdursa,(daha çox məlumat üçün sonrakı hissəyə və Git'də Branch-a bax), avtomatik olaraq götürmək üçün `git pull` əmrindən istifadə edə bilərsiniz və sonra uzaq branch-ı cari branch'ınıza birləşdirməlisiniz. Bu sizin üçün daha asan və

ya daha rahat bir iş axını ola bilər; Varsayılan olaraq, `git clone` əmri avtomatik olaraq local `master` branch'nızı klonlaşdırdığınız serverdə uzaqdan `master` branch'ını (və ya standart budaq adlanır) izləmək üçün qurur. `git pull` ümumiyyətlə əvvəlcədən klonlanmış serverdən məlumat alır və avtomatik olaraq hazırda işlədiyiniz koda daxil olmağa çalışır.

Git versiyasının 2.27-dən başlayaraq, `pull.rebase` dəyişəninin təyin edilməməsi halında `git pull` xəbərdarlıq verəcəkdir. Dəyişən təyin edənə qədər Git sizə xəb ərdarlıq edəcəkdir.

Git'in standart davranışını istəyirsinizsə (mümkünsə sürətli irəli göndərin, başqa bir birləşmə yarat): `git config --global pull.rebase "false"`

Pulling edərkən yenidən yazmaq istəsəniz: `git config --global pull.rebase "true"`

Uzaqdan Pushing etmək

Layihənizi bölüşmək istədiyiniz bir nöqtədə olduqda, onu yuxarıdan push etməlisiniz. Bunun əmri sadədir: `git push <remote> <branch>`. Əgər `master` branch-nızı `origin`serverinə push etmək ist əyirsinizsə (yenidən klonlaşdırmaq bu adların hər ikisini avtomatik olaraq sizin üçün yaradır), bu zaman serverə etdiyiniz hər hansı bir commit-i push etmək üçün işləyə bilərsiniz:

```
$ git push origin master
```

Bu əmr yalnız yazılı girişi olan bir serverdən klonlanmış və bu vaxt heç kim push etmədikdə işl əyir. Eyni anda başqası ilə klonlaşsanız və onlar yuxarıya doğru push etsələr və sonra yuxarıya doğru push etsən, təkanlarınız rədd ediləcəkdir. Əvvəlcə işlərini götürməli və təkan verməyinizə qədər bunu özünüzə daxil etməlisiniz. Uzaq serverlərə necə push barədə daha ətraflı məlumat üçün Git'də Branch baxın.

Uzaqdan Yoxlama

Müəyyən bir remote haqqında daha çox məlumat görmək istəyirsinizsə, `git remote show <remote>` komandasını istifadə edə bilərsiniz. Bu əmri `origin` kimi müəyyən bir qısa ad ilə işləsəniz, bu kimi bir şey əldə edirsiniz:

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master                               tracked
    dev-branch                           tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

Uzaqdakı depo üçün URL, eləcə də izləmə branch'ı məlumatlarını sadalayır. Komanda sizə köm əkliklə deyir ki, `master` branch'ında olsanız və `git pull` işlədirsinizsə, bütün uzaq istinadları aldıqdan sonra avtomatik olaraq uzaqdakı `master` branch'ında birləşəcəkdir. Ayrıca, endirdiyi bütün uzaq istinadları sadalayır.

Bu qarşılaşa biləcəyiniz sadə bir nümunədir. Git'i daha çox istifadə edərkən, `git remote show`-dan daha çox məlumat görə bilərsiniz:

```
$ git remote show origin
* remote origin
  URL: https://github.com/my-org/complex-project
  Fetch URL: https://github.com/my-org/complex-project
  Push  URL: https://github.com/my-org/complex-project
  HEAD branch: master
  Remote branches:
    master                           tracked
    dev-branch                       tracked
    markdown-strip                   tracked
    issue-43                         new (next fetch will store in remotes/origin)
    issue-45                         new (next fetch will store in remotes/origin)
    refs/remotes/origin/issue-11     stale (use 'git remote prune' to remove)
  Local branches configured for 'git pull':
    dev-branch merges with remote dev-branch
    master     merges with remote master
  Local refs configured for 'git push':
    dev-branch                     pushes to dev-branch                     (up to
date)
    markdown-strip                 pushes to markdown-strip                 (up to
date)
    master                         pushes to master                         (up to
date)
```

Bu əmr müəyyən branch'larda olarkən `git push` işlədikdə hansı branch'ın avtomatik push edildiyini göstərir. Ayrıca, `git pull` hələ serverinizdə olmayan uzaq branch'ları, serverdən hansısa uzaq branch'ları çıxartdığınızı və çalışdığınız zaman avtomatik olaraq uzaqdan izləmə branch'ı ilə

birləşdirə bilən çoxsaylı lokal branch'ları göstərir.

Uzaqdan Adların Dəyişdirilməsi və Çıxarılması

Remote qısa adını dəyişdirmək üçün `git remote rename` funksiyasını işlədə bilərsiniz. Məsələn, `pb` adını `paul` olaraq dəyişdirmək istəyirsinizsə, `git remote rename` ilə bunu edə bilərsiniz:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

Bu, bütün uzaqdan izləyən branch adlarınızı da dəyişdirdiyini qeyd etmək lazımdır. Əvvəllər `pb/master` də istinad edilən şey indi `paul/master`-ə istinad ediləcəkdir.

Bir səbəbdən bir uzaqdan silmək istəsəniz - serveri köçürdünüz və ya artıq müəyyən bir güzgüdən istifadə etmirsiniz, ya da bəlkə bir dəfə töhfə verən başqa bir töhfə vermirsə, `git remote remove` və ya `git remote rm` istifadə edə bilərsiniz:

```
$ git remote remove paul
$ git remote
origin
```

Uzaqdan bir istinadı bu şəkildə sildikdən sonra, uzaqdan izlənən bütün branch'lar və bu uzaqdan əlaqəli konfiqurasiya parametrləri də silinir.

Etiketləmə

Əksər VNS'lər kimi, Git, depo tarixçəsindəki müəyyən nöqtələri önəmli olaraq etiketləyə bilir. Tipik olaraq, insanlar bu funksionallıqdan istifadə nöqtələrini qeyd etmək üçün istifadə edirlər (`v1.0`,` v2.0` və s.). Bu bölmədə mövcud etiketlərin necə siyahıya alınmasını, etiketləri necə yaratmaq və silməyi və fərqli etiket növlərinin nə olduğunu öyrənəcəksiniz.

Etiketlərinizi Listləyin

Git-də mövcud etiketlərin siyahısı sadədir. Yalnız `git tag` yazın (əlavə olaraq `-l` və ya `--list` ilə):

```
$ git tag
v1.0
v2.0
```

Bu əmr etiketləri əlifba sırası ilə sıralayır; göstərilən qaydanın heç bir əsası yoxdur.

Ayrıca müəyyən bir xüsusi nümunəyə uyğun etiketlər axtara bilərsiniz. Məsələn Git mənbə repo, 500-dən çox etiketi ehtiva edir. Yalnız 1.8.5 seriyasına baxmaq istəyirsinizsə, bunu işlədə bilərsiniz:

```
$ git tag -l "v1.8.5*"
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

Etiket kartı siyahısına qoymaq üçün `-l` *və ya* `--list` *seçimi tələb olunur*

Etiketlərin yalnız bütün siyahısını istəsəniz, `git tag` əmrini istifadə edərək bir siyahı istəməyinizi ehtimal edir və təqdim edir; bu vəziyyətdə `-l` və ya `--list` istifadəsi istəyə bağlıdır.

Lakin, etiket adlarına uyğun bir işaret naxışını təqdim edirsinizsə, *-l* və ya - *List* istifadəsi məcburidir.

Etiket Yaratmaq

Git iki növ etiket dəstəkləyir: *yüngül* və *əlavə*.

Yüngül etiket dəyişməyən bir budağa bənzəyir - yalnız müəyyən bir öhdəliyi göstərir.

Qeyd olunan etiketlər Git verilənlər bazasında tam obyekt kimi saxlanılır. Çeki qiymətləndirdilər; etiketçi adını, e-poçtunu və tarixini ehtiva edir; etiketləmə mesajı var; və GNU Privacy Guard (GPG) ilə imzalanıb təsdiqlənə bilər. Bütün bu məlumatlara sahib olmağınız üçün ümumiyyətlə annotasiya etiketləri yaratmağınız tövsiyə olunur; ancaq müvəqqəti etiket istəsəniz və ya nədənsə digər məlumatı saxlamaq istəmirsinizsə yüngül etiketlər də mövcuddur.

Əlavə Etiketlər

Git-də əlavə etiket yaratmaq çox sadədir. Ən asan yol, `tag` əmrini işlədərkən `-a` göstərməkdir:

```
$ git tag -a v1.4 -m "my version 1.4"
$ git tag
v0.1
v1.3
v1.4
```

`-m` etiket ilə birlikdə saxlanan bir etiket mesajını göstərir. Əlavə etiket üçün bir mesaj göstərm əmisinizsə, Git redaktorunuzu işə salır ki, daxil edə bilərsiniz.

Etiket məlumatlarını `git show` əmrindən istifadə edərək etiketlənmiş öhdəliklə birlikdə görə bil ərsiniz:

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number
```

Bu etiket məlumatını, öhdəliyin etiketlənmə tarixini və öhdəlik məlumatını göstərmədən əvvəl qeyd mesajını göstərir.

Yüngül Etiketlər

Etiketləyin başqa bir yolu yüngül bir etiketdir. Bu, əsasən bir sənəddə saxlanılan tapşırıq çeki mə bləğidir - başqa heç bir məlumat saxlanılmır. Yüngül bir etiket yaratmaq üçün `-a`,` -s` və ya `-m` variantlarından heç birini təmin etmədən etiket adını təqdim etmə bəs edər:

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

Bu dəfə etiketdə `git show` işlədirsinizsə, əlavə etiket məlumatlarını görmürsünüz. Komanda sadəcə öhdəliyi göstərir:

```
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number
```

Daha Sonra Etiketləmə

Keçmişləri köçürdükdən sonra qeyd edə bilərsiniz. Təqdim etdiyiniz tarix belə görünərsə:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 Create write support
0d52aaab4479697da7686c15f77a3d64d9165190 One more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc Add commit function
4682c3261057305bdd616e23b64b0857d832627b Add todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a Create write support
9fceb02d0ae598e95dc970b74767f19372d61af8 Update rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc Commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a Update readme
```

İndi güman ki, layihəni v1.2-də etiketləməyi unutmusunuz ki, bu da “Yeniləmə rakefile” commitində idi. Faktdan sonra əlavə edə bilərsiniz. Bu əməli etiketləmək üçün əmr sonunda tapşırıq çeki məbləğini (və ya onun bir hissəsini) göstərin:

```
$ git tag -a v1.2 9fceb02
```

Commit etiketlədiyinizi görə bilərsiniz:

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    Update rakefile
...
```

Etiketləri Paylaşmaq

Varsayılan olaraq, `git push` əmri etiketləri uzaq serverlərə ötürmür. Etiketləri yaratdıqdan sonra açıq şəkildə ortaq bir serverə push olacaqsınız. Bu proses uzaq filialları bölüşmək üçün `git push origin <tagname>` işlədə bilərsiniz.

```
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.5 -> v1.5
```

Push istədiyiniz çox etiket varsa, `git push` əmrinə `--tags` seçimini də istifadə edə bilərsiniz. Bu, bütün etiketlərinizi artıq olmayan uzaq serverə köçürəcəkdir.

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.4 -> v1.4
 * [new tag]         v1.4-lw -> v1.4-lw
```

İndi başqası deponuzu klonlaşdırdıqda və ya pull edəndə bütün etiketlərinizi də alacaqsınız.

`git push` *hər iki etiketi push edir*

`git push <remote> --tags` həm yüngül, həm də əlavə etiketlər basacaq. Hal-hazırda yalnız yüngül etiketləri basmaq üçün bir seçim yoxdur, ancaq `git push <remote> --follow-tags` istifadə etsəniz, yalnız əlavə etiketlər uzaqdan push ediləcək.

Etiketləri Silmək

Yerli depolarınızdakı etiketi silmək üçün `git tag -d <tagname>` istifadə edə bilərsiniz. Məsələn, yuxarıdakı yüngül etiketimizi silə bilərik:

```
$ git tag -d v1.4-lw
Deleted tag 'v1.4-lw' (was e7d5add)
```

Qeyd edək ki, bu etiketi heç bir uzaq serverdən silmir. Bir etiketi uzaq bir serverdən silmək üçün iki ümumi variant var.

Birinci variant `git push <remote> :refs/tags/<tagname>`-dır:

```
$ git push origin :refs/tags/v1.4-lw
To /git@github.com:schacon/simplegit.git
 - [deleted]         v1.4-lw
```

Yuxarıda göstərilənləri şərh etməyin yolu, nöqtəni kolon uzaq məsafəli etiket adına sövq edilərək

effektiv şəkildə silməkdən əvvəl oxumaqdır.

Uzaq bir etiketi silməyin ikinci (və daha asan) yolu:

```
$ git push origin --delete <tagname>
```

Etiketlərin Yoxlanılması

Etiket işarələdiyi sənədlərin versiyalarına baxmaq istəyirsinizsə, etiketinizə `git checkout` edə bil ərsiniz, baxmayaraq ki, bu, depolarınızı bəzi pis yan təsirləri olan "detached HEAD" vəziyyətinə qoyur:

```
$ git checkout v2.0.0
Note: checking out 'v2.0.0'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch>

HEAD is now at 99ada87... Merge pull request #89 from schacon/appendix-final

$ git checkout 2.0-beta-0.1
Previous HEAD position was 99ada87... Merge pull request #89 from schacon/appendix-
final
HEAD is now at df3f601... Add atlas.json and cover image
```

"detached HEAD" vəziyyətində, dəyişiklik edirsinizsə və sonra öhdəlik yaradırsınızsa, etiket eyni qalacaq, ancaq yeni öhdəliyiniz heç bir branch-a aid olmayacaq və dəqiq bir işləmə hash istisna olmaqla əlçatmaz olacaq. Beləliklə, dəyişiklik etmək lazımdırsa - məsələn, köhnə bir versiyada bir səhv düzəldirsinizsə - ümumiyyətlə bir filial yaratmaq istəyəcəksiniz:

```
$ git checkout -b version2 v2.0.0
Switched to a new branch 'version2'
```

Bunu etsəniz və öhdəlik götürsəniz, `version2` filialınız `v2.0.0` etiketinizdən bir qədər fərqli olacaq, çünki yeni dəyişikliklərinizlə irəliləyəcək, buna görə diqqətli olun.

Git Alias'lar

Növbəti fəsilə keçmədən əvvəl Git təcrübənizi daha sadə, asan və daha yaxşı tanış edə biləcək bir xüsusiyyəti təqdim etmək istəyirik: alias'lar. Aydınlıq gətirmək üçün deməliyik ki, biz bunu kitabın

başqa yerlərindən istifadə etməyəcəyik, ancaq hər hansı bir müntəzəmliklə Git istifadə etməyə davam etsəniz, aliases bu barədə bilməli olduğunuz bir şeydir.

Əgər Git əmrinizi qismən yazsanız, o avtomatik olaraq tamamlanmayacaqdır. Git əmrlərinin hər birinin e*-ntire mətnini yazmaq istəmirsinizsə, asanlıqla `git config` istifadə edərək hər komanda üçün bir alias qura bilərsiniz. (git commands, config Burada qurmaq istəyəcəyiniz bir neçə nümunə:

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

Bu o deməkdir ki, məsələn, `git commit` yazmaq əvəzinə sadəcə `git ci` yazmaq lazım olacaq. Əgər Git istifadə etməyə davam etsəniz, digər komandalardan da tez-tez istifadə edəcəksiniz; yeni alias'lar yaratmaqdan çəkinməyin.

Bu texnika mövcud olduğunu düşündüyünüz əmrləri yaratmaqda da çox faydalı ola bilər. Məsələn, faylı pozmaqla qarşılaşdığınız məqsədəuyğun problemi düzəltmək üçün Git-ə öz qeyri-sabit alias'nızı əlavə edə bilərsiniz:

```
$ git config --global alias.unstage 'reset HEAD --'
```

Bu, aşağıdakı iki əmri ekvivalent edir:

```
$ git unstage fileA
$ git reset HEAD -- fileA
```

Burada bir az daha aydın görünür. Aşağıakı kimi `last` əmrini əlavə etmək də adi haldır:

```
$ git config --global alias.last 'log -1 HEAD'
```

Siz bu yolla sonuncu əməliyyatı rahatlıqla görə bilərsiniz:

```
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
Author: Josh Goebel <dreamer3@example.com>
Date:   Tue Aug 26 19:48:51 2008 +0800

    Test for current head

    Signed-off-by: Scott Chacon <schacon@example.com>
```

Gördüyünüz kimi, Git sadəcə yeni əmr alias'la əvəz olunur. Bununla birlikdə, siz Git alt əmri deyil, xarici bir əmr işlətmək istəyə bilərsiniz. Bu vəziyyətdə əmrini `!` işarəsi ilə başlamalısınız. Git

anbarı ilə işləyəndə öz alətlərinizi yazsanız bu daha faydalı olacaqdır. `gitk` əmrini işə salmaq üçün `git visual`-ı digər adı ilə nümayiş etdirə bilərik:

```
$ git config --global alias.visual '!gitk'
```

Qısa Məzmun

Bu nöqtədə, bütün əsas lokal Git əməliyyatlarını edə bilərsiniz - bir depo yaratmaq və ya klonlaşdırmaq, dəyişikliklər etmək, bu dəyişiklikləri commit etmək və həyata keçirmək və deponun bütün dəyişiklik tarixçəsini izləmək. Sonra Git'in qatil xüsusiyyətini əhatə edəcəyik: branching modeli.

Git'də Branch

Demək olar ki, hər bir VNS'nin bir növ branching dəstəyi var. Branching əsas inkişaf xəttindən uzaqlaşmağınız və bu ana xəttlə qarışmadan işinizi davam etdirməyiniz deməkdir. Bir çox VNS al ətində bu, bir qədər bahalı bir prosesdir və tez-tez mənbə kodu qovluğunun yeni bir nüsxəsini yaratmağınızı tələb edir, bu da böyük layihələr üçün çox vaxt apara bilər.

Bəzi insanlar Git'in branching modelini "qatil xüsusiyyəti" olaraq adlandırırlar və bu, Git'i VNS c əmiyyətində fərqləndirir. Niyə bu qədər xüsusidir? Git branch'larının yolu inanılmaz dərəcədə yüngüldür və branching əməliyyatlarını anında edir və branch'lar arasında ümumiyyətlə eyni sür ətlə irəliləyir. Bir çox digər VNS'lərdən fərqli olaraq Git, gündə bir neçə dəfə dəfələrlə branch və birləşən iş axınlarını təşviq edir. Bu xüsusiyyəti anlamaq və mənimsəmək sizə güclü və bənzərsiz bir vasitə verir və inkişaf yolunuzu tamamilə dəyişə bilər.

Nutshell'də Branch'lar

Git-in branching yolunu həqiqətən başa düşmək üçün geri addım atmalı və Git-in məlumatlarını necə saxladığını araşdırmalıyıq.

Başlanğıc fəsildən xatırladığınız kimi, Git məlumatları bir sıra dəyişikliklər və ya fərqlər kimi saxlamır, əksinə *snapshotlar* kimi saxlayır.

Commit etdiyiniz zaman Git, səhnələşdirdiyiniz məzmunun snapshotu olan bir commit obyekti saxlayır. Bu obyekt eyni zamanda müəllifin adını və e-poçt adresini, yazdığınız mesajı və bu commit-i yerinə yetirmədən əvvəl gələn (törədicisi və ya valideynləri) göstəriş və göstərişləri göst ərmişdir: ilkin commit üçün sıfır valideynlər, normal bir commit üçün bir valideyn və iki və ya daha çox branch-ın birləşməsi nəticəsində əmələ gələn çoxlu valideynlər.

Bunu görüntüləmək üçün üç fayldan ibarət bir qovluğunun olduğunu fərz edək və hamısını səhnə ləşdirib commit götürdünüz. Faylların qurulması hər biri üçün bir yoxlama cədvəlini tərtib edir (Başlanğıcbəhs etdiyimiz SHA-1 hash), faylın həmin versiyasını Git depolarında saxlayır (Git kimi *blobs* onlara aiddir) istifadə edərək sahəsinə nəzarət hissəsini əlavə edir:

```
$ git add README test.rb LICENSE
$ git commit -m 'Initial commit'
```

`git commit` əmrini yerinə yetirdikdə, Git hər alt bölməni yoxlayır (bu vəziyyətdə yalnız root layihə qovluğu) və onları Git deposunda ağac obyekti kimi saxlayır. Git sonra metadata və root layihə ağacına göstərici olan bir obyekt yaradır ki, lazım olduqda həmin anı yenidən yarada bilsin.

Git deposunda artıq beş obyekt var: üç *blobs* (hər üç fayldan birinin məzmununu təmsil edən), qovluğun məzmununu sadalayan bir *tree* və hansı fayl adlarının blobs kimi saxlanıldığını və o root ağacına və göstərilən bütün metadata göstərici ilə bir *commit*.

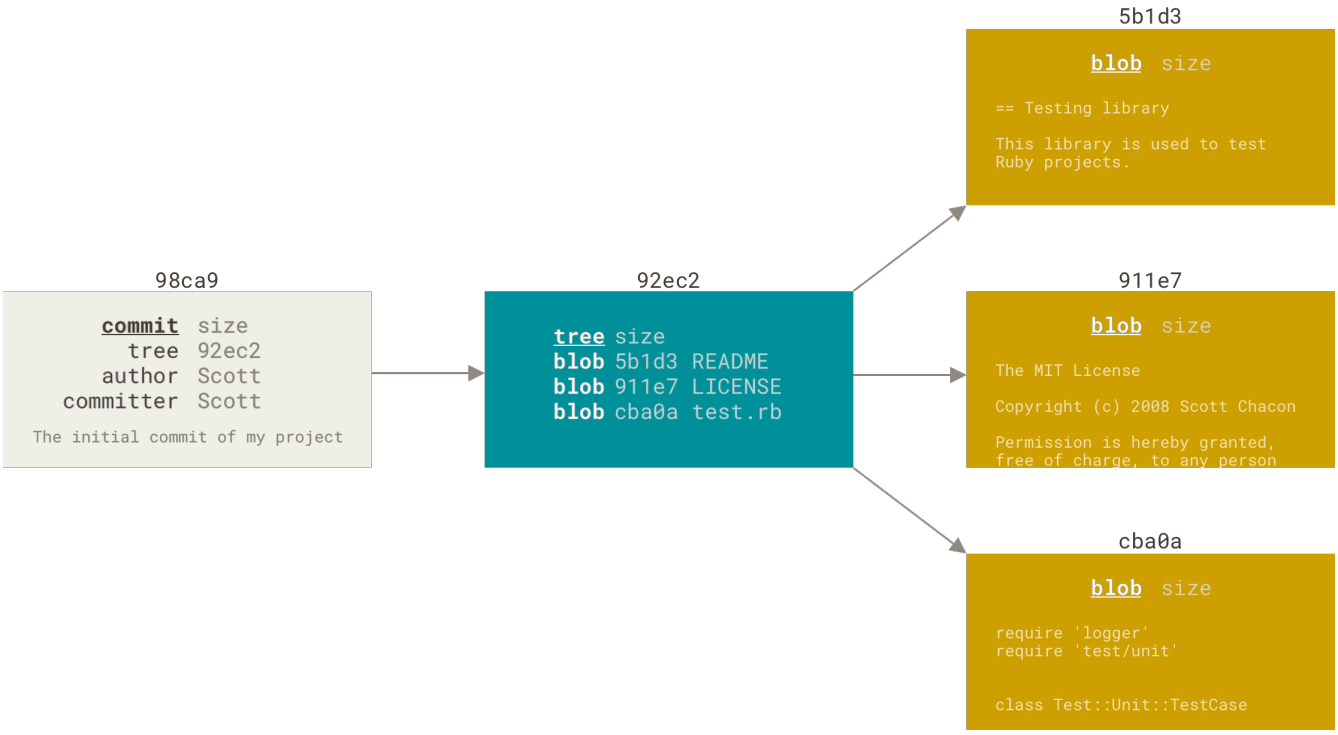


Figure 9. Commit və Onun Ağacı

Bəzi dəyişikliklər etsəniz və yenidən commit etsəniz, növbəti əmr əvvəlcədən gələn commit göst əricisini saxlayır.

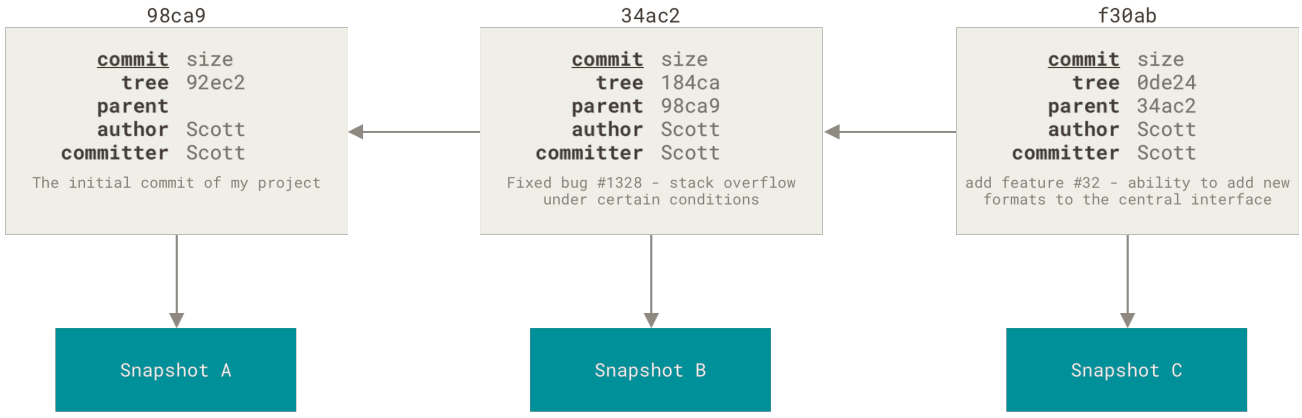


Figure 10. Commit-lər və Onun Valideynləri

Git-dəki bir branch, onun commit-ləri arasındakı yüngül daşınan göstəricilərindən biridir. Git-də standart branch adı `master`-dir. Commit etməyə başladığınız zaman etdiyiniz son commit-ə işarə edən bir `master` branch verilir. Hər dəfə commit etdiyiniz zaman `master` branch-ı göstəricisi avtomatik olaraq irəliləyir.

“mater” branch-ı Git-də xüsusi branch deyildir. Tamamilə hər hansı digər branch kimidir. Təxminən hər bir deponun bir səbəbi, `git init` əmrinin onu standart olaraq yaratması və çoxlarının onu dəyişdirmək üçün narahat etməməsidir.

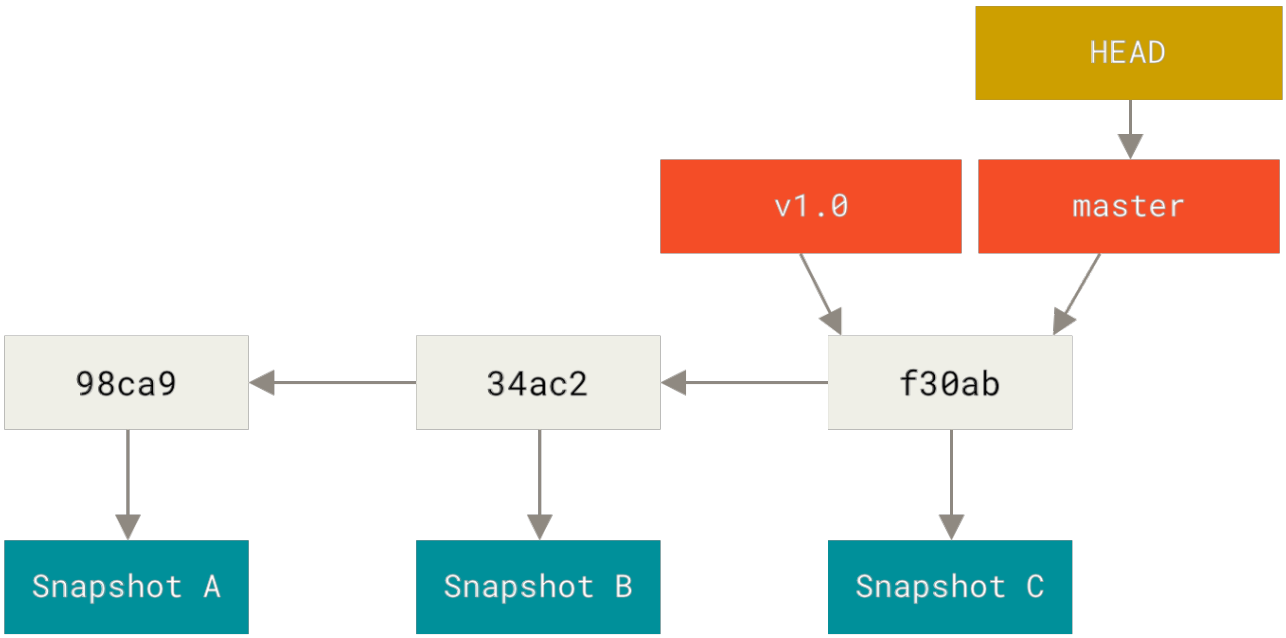


Figure 11. Branch və Onun Commit Tarixi

Təzə Branch Yaratmaq

Yeni bir branch yaratdıqda nə baş verir? Yaxşı, bunu etmək hərəkət etməyiniz üçün yeni bir göst ərici yaradır. Deyək ki, `testing` adlı yeni bir branch yaratmaq istəyirsiniz. Bunu `git branch` əmri ilə edirsiniz:

```
$ git branch testing
```

Bu, hazırda olduğunuz eyni commit yeni bir göstərici yaradır.

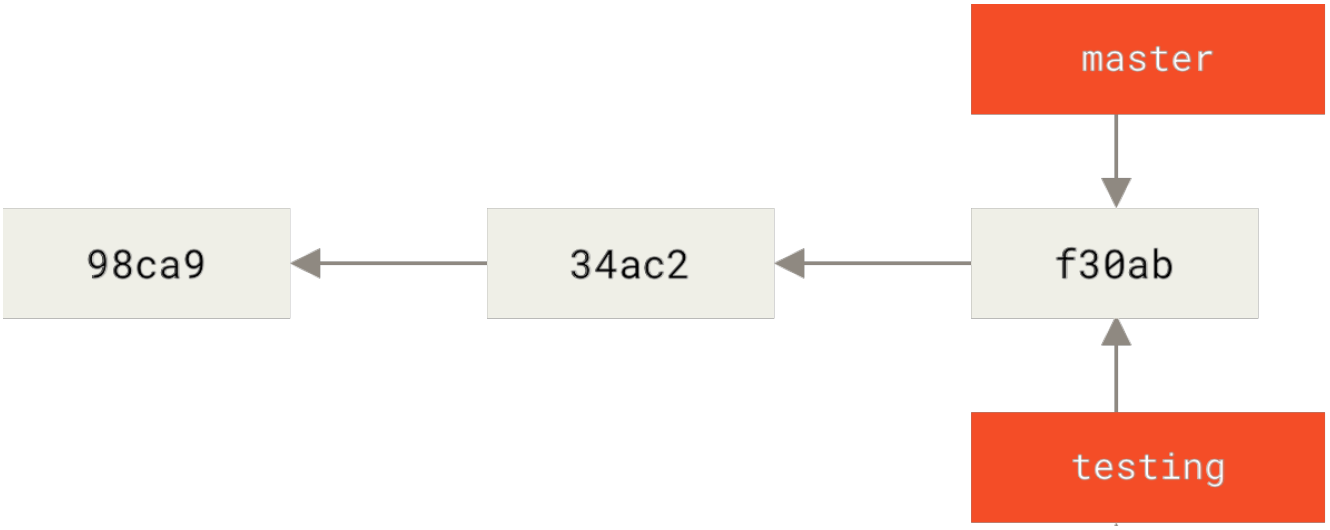


Figure 12. Eyni seriyaya commit edən iki branch

Git hazırda hansı branch-ın üzərində olduğunu necə bilir? `HEAD` adlı xüsusi bir göstərici saxlayır. Qeyd edək ki, Subversion və ya CVS kimi istifadə oluna biləcəyiniz digər VNS-lərdəki `HEAD` anlayışından çox fərqlidir. Git-də bu, hazırda olduğunuz lokal branch-a işarədir. Bu vəziyyətdə yenə də `master`-in üstündəsiniz. `git branch` əmri yalnız yeni bir branch yaratdı - o branch-a keçm

ədi.

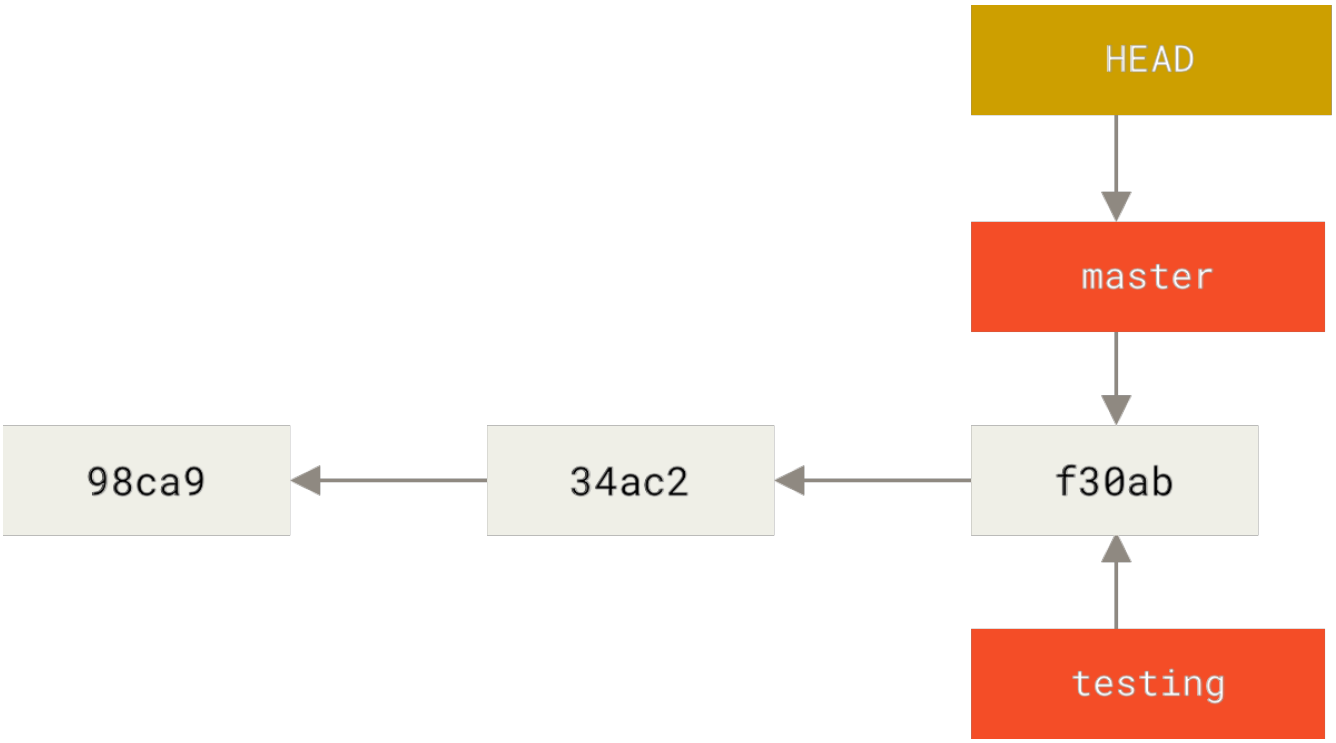


Figure 13. Branch-a HEAD göstəricisi

Şöbə nöqtələrinin göstərildiyi yerləri göstərən sadə `git log` əmrini işlədərək bunu asanlıqla görə bilərsiniz. Bu seçim `--decorate` adlanır.

```
$ git log --oneline --decorate
f30ab (HEAD -> master, testing) Add feature #32 - ability to add new formats to the
central interface
34ac2 Fix bug #1328 - stack overflow under certain conditions
98ca9 Initial commit
```

`f30ab` əməlinin yanında orada olan `master` və `testing` branch-larını görə bilərsiniz.

Switching Branches

Mövcud bir branch-a keçmək üçün `git checkout` əmrini yerinə yetirirsiniz. Yeni `testing` branch-a keçək:

```
$ git checkout testing
```

Bu, `testing` şöbəsinə işarə etmək üçün `HEAD`-ı hərəkətə gətirir.

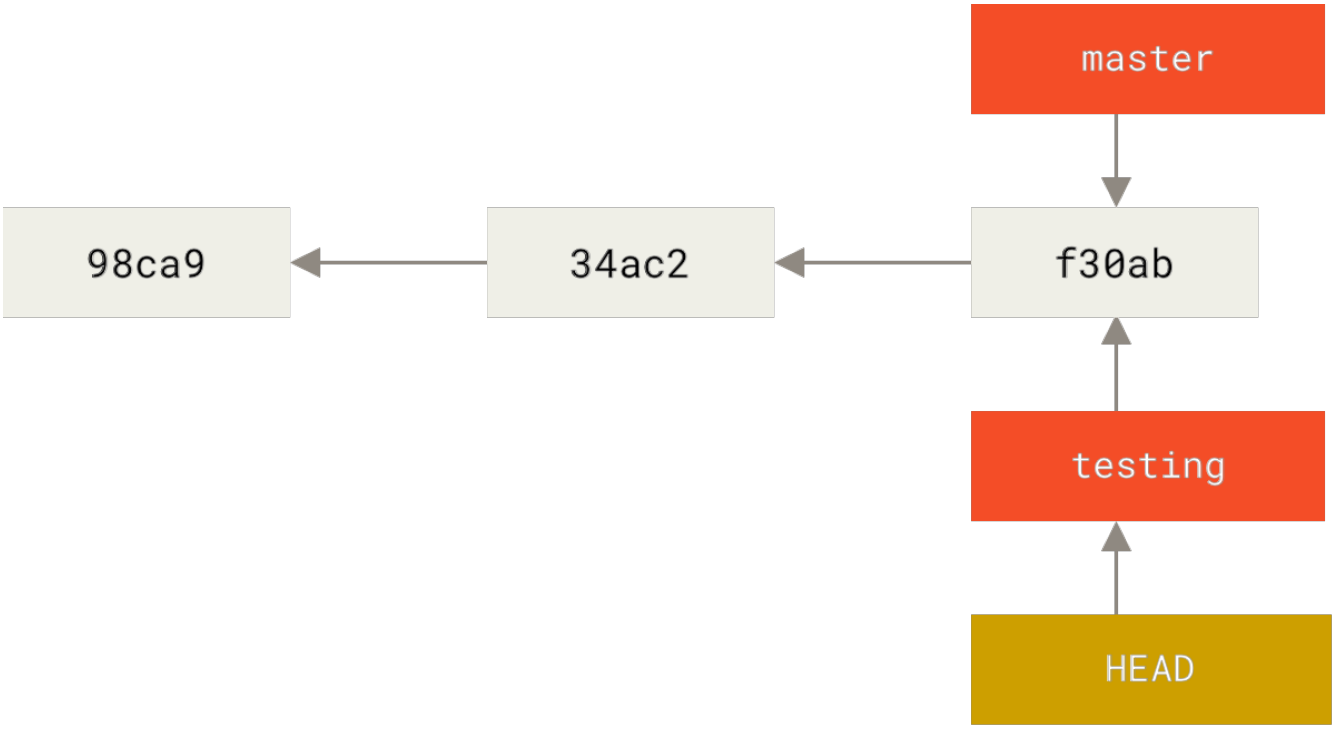


Figure 14. HEAD mövcud branch-ı işarə edir

Bunun əhəmiyyəti nədir? Yaxşı, başqa bir commit edək:

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

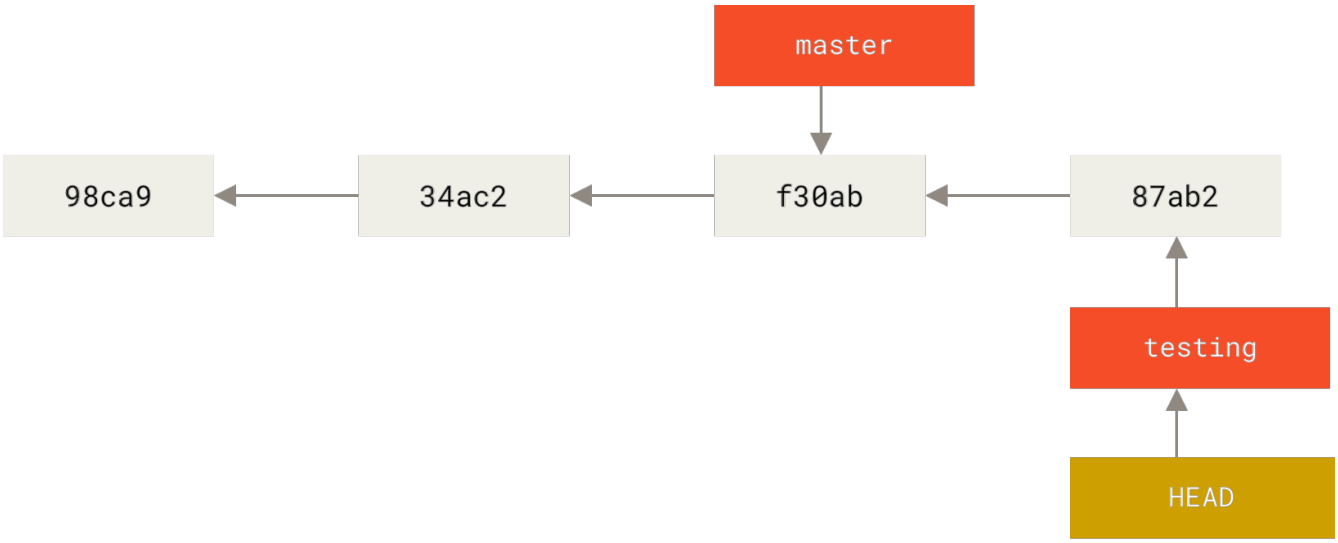


Figure 15. HEAD branch-ı bir commit götürüldükdə irəliləyir

Bu, maraqlıdır, çünki indi `testing` branch-nız irəliləmişdir, ancaq `master` branch-nız branch-ları d əyişmək üçün `git checkout` zamanı işlədiyiniz vəzifəni göstərir. Gəlin yenidən `master` branch-a qayıdaq:

```
$ git checkout master
```

`git log` həmişə _bütün branch-ları göstərmir

Hal-hazırda `git log` işlətməli olsaydınız, yeni yaradılan `testing` branch-nın hara getdiyi barədə düşünə bilərsiniz, çünki çıxışda görünməyəcəkdir.

Branch yoxa çıxmadı; Git sadəcə bu branch-la maraqlandığınızı bilmir və sizə nə istədiyini göstərməyə çalışır. Başqa sözlə, default olaraq, `git log`, yalnız yoxladığınız branch-ın altındakı törəmə tarixçəsini göstərəcəkdir.

İstədiyiniz branch üçün törəmə tarixçəsini göstərmək üçün onu dəqiq göstərm əlisiniz: `git log testing`. Bütün branch-ları göstərmək üçün `git log` əmrinizə `--all` əlavə edin.

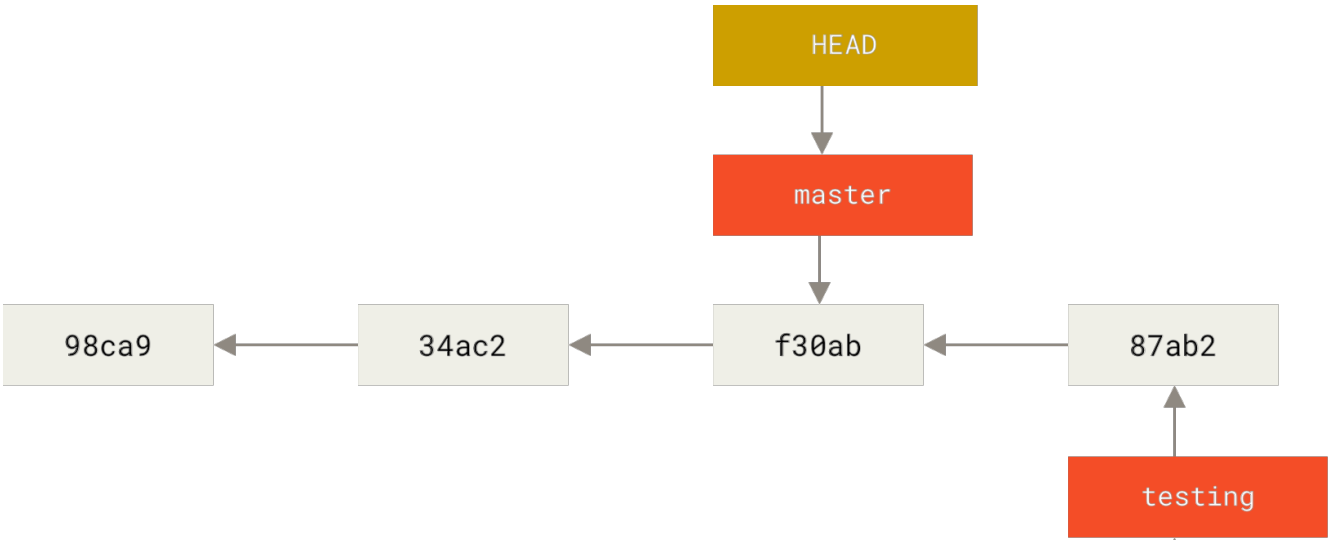


Figure 16. Siz çıxış edəndə HEAD hərəkət edir

Bu əmr iki şeyi etdi. HEAD göstəricisini `master` branch-a göstərmək üçün geri qoydu və işçi qovluğunuzdakı faylları `master` işarələdiyi şəklə geri çevirdi. Bu da bu anda etdiyiniz dəyişikliklərin layihənin köhnə versiyasından ayrılacağını bildirir. Fərqli bir istiqamətə gedə bilmək üçün `testing` şöbənizdə gördüyünüz işləri geri qaytarır.

Branch-ları dəyişdirmək işçi qovluğunuzdakı faylları dəyişdirir

Git-də filialları dəyişdirdiyiniz zaman işlədiyiniz qovluqdakı faylların dəyişəcəyini nəzərə almaq vacibdir. Köhnə bir branch-a keçsəniz, işlədiyiniz qovluq bu branch-da sonuncu dəfə etdiyiniz kimi görünəcəkdir. Git təmiz bir şəkildə edə bilmirsə, ümumiyyətlə keçməyə imkan verməz.

Yeniən bir neçə dəyişiklik və commit edək:

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

İndi layihənizin tarixi fərqləndi (bax Fərqli tarix). Bir branch yaratdınız və işə keçdiniz, bir az iş gördünüz, sonra yenidən əsas branch-a keçdiniz və başqa işlər gördünüz. Bu dəyişikliklərin hər ikisi ayrı-ayrı branch-da ayrılır: hazır olduqda filiallar arasında geri və irəli keçə bilərsiniz. Və siz bunların hamısını sadə `branch`, `checkout`və `commit` əmrləri ilə etdiniz.

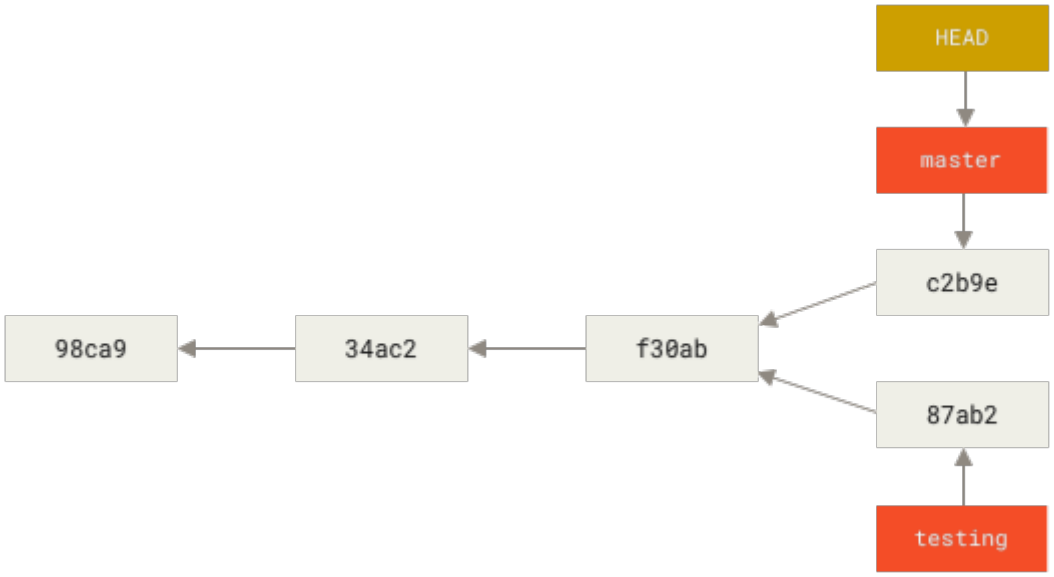


Figure 17. Fərqli tarix

Bunu `git log` əmri ilə də asanlıqla görə bilərsiniz. `git log --oneline --decorate --graph --all` işl ədirsinizsə,branch-larınızın harada olduğunu və tarixinizin necə ayrıldığını göstərən əmrlərinizin tarixini çap edəcəkdir.

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) Made other changes
| * 87ab2 (testing) Made a change
|/
* f30ab Add feature #32 - ability to add new formats to the central interface
* 34ac2 Fix bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

Git-dəki bir branch, əslində, işarələdiyi 40 simvol SHA-1 çeki olan sadə bir fayl olduğundan, branch-lar yaratmaq və məhv etmək asan və ucuzdur. Yeni bir filial yaratmaq, 41 baytı bir fayla yazmaq q ədər sürətli və sadədir (40 simvol və yeni bir xətt).

Bu, ən köhnə VNS alətlər şöbəsinin, layihənin bütün sənədlərinin ikinci qovluğa kopyalanmasını özündə cəmləşdirən yoldan kəskin şəkildə fərqlidir. Layihənin ölçüsündən asılı olaraq bu bir neçə saniyə və ya hətta dəqiqə çəkə bilər, halbuki Git-də proses həmişə ani olur. Ayrıca, valideynlərimizi etdiyimiz zaman qeyd etdiyimiz üçün birləşmə üçün uyğun bir birləşmə bazası tapmaq avtomatik olaraq bizim üçün edilir və ümumiyyətlə bunu etmək çox asandır. Bu xüsusiyyətlər inkişaf etdiriciləri tez-tez filial yaratmağa və istifadə etməyə həvəsləndirməyə kömək edir.

Görək niyə belə etməlisiniz.

Yeni bir branch yaratmaq və eyni zamanda ona keçid

Yeni bir branch yaratmaq və eyni zamanda yeni branch-a keçmək istəmək tipikdir - bu bir əməliyyatla with `git checkout -b <newbranchname>` ilə edilə bilər.

Sadə Branching və Birləşdirmə

Real dünyada istifadə edə biləcəyiniz bir iş axını ilə branching və birləşmənin sadə bir nümunə sinə keçək. Bu addımları izləyəcəksiniz:

1. Bir veb saytda bir az iş gör.
2. Çalışdığınız yeni bir istifadəçi hekayəsi üçün bir branch yarat.
3. O branch-da bəzi işlər gör.

Bu mərhələdə başqa bir məsələnin kritik olduğuna dair bir bildiriş alacaqsınız və düzəliş lazım olacaqdır. Siz aşağıdakıları edəcəksiniz:

1. Production branch-ı seçin.
2. Düzəliş əlavə etmək üçün bir branch yaradın.
3. Test edildikdən sonra, düzəldici branch-ı birləşdirin və production-a push edin.
4. Orijinal istifadəçi hekayənizə qayıdın və işə davam edin.

Sadə Branching

Əvvəlcə deyək ki, layihəniz üzərində işləyirsiniz və artıq `master` branch-ında bir neçə iş görmüsünüz.

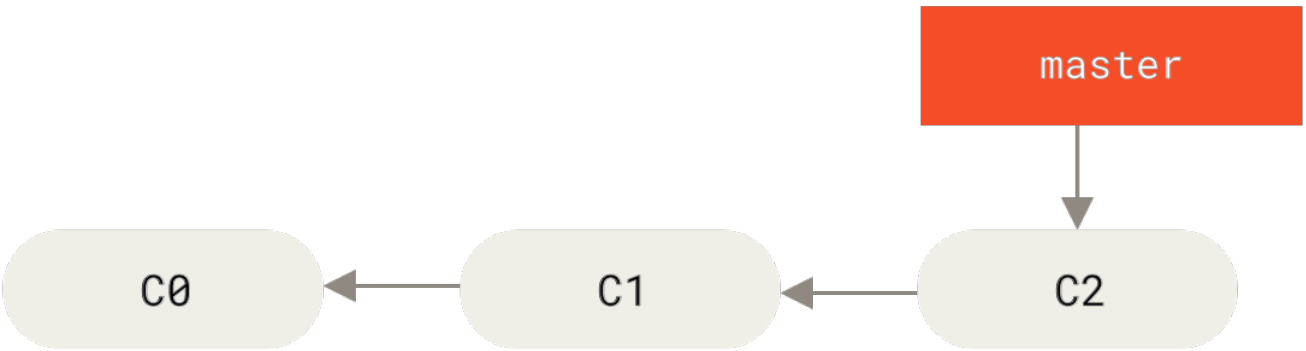


Figure 18. Sadə commit tarixi

Şirkətinizin istifadə etdiyi hər hansı bir izləmə sistemində 53-cü sayda çalışacağınıza qərar verdiniz. Yeni bir branch yaratmaq və eyni zamanda ona keçmək üçün `-b` keçidi ilə `git checkout` əmrini işlətmək olar:

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

Bu ssenaridir:

```
$ git branch iss53
$ git checkout iss53
```

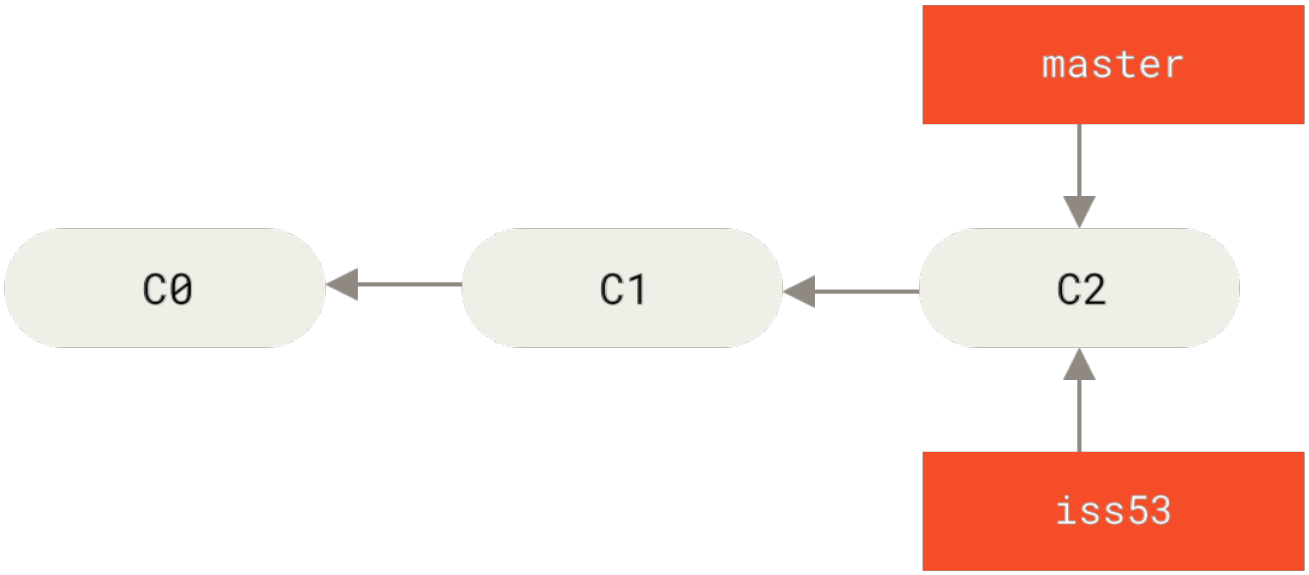


Figure 19. Yeni bir branch göstəricisi yaratmaq

Veb saytınızda işləyirsiniz və bəzi commit-lər verirsiniz. Bunu etmək, `iss53` branch-ını irəli aparır, çünki siz onu yoxlamısınız (yəni `HEAD` ona işarə edir):

```
$ vim index.html
$ git commit -a -m 'Create new footer [issue 53]'
```

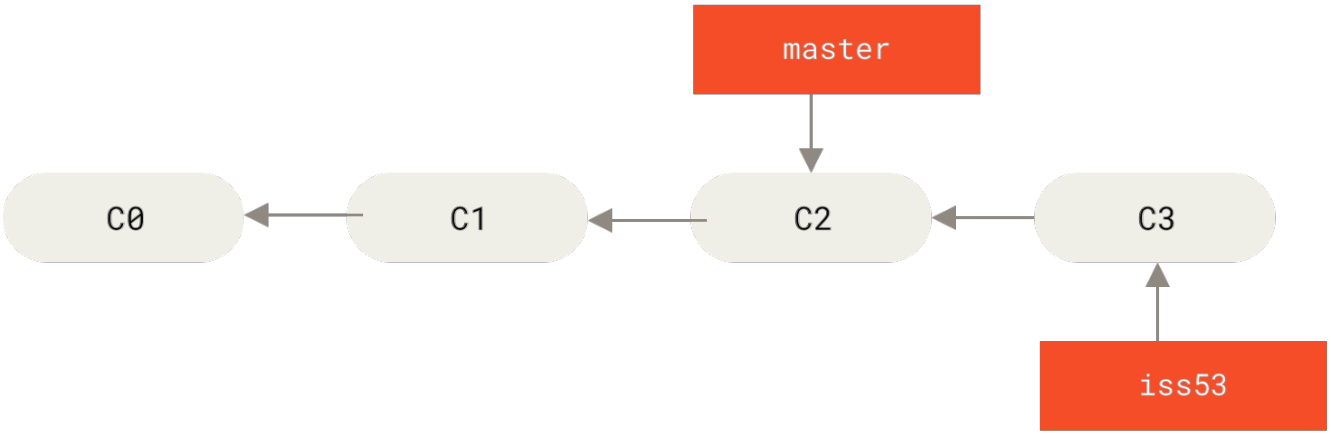


Figure 20. `iss53` branch-ı işinizlə irəlilədi

İndi veb saytında bir problem olduğuna dair zəng alırsınız və dərhal onu düzəltməlisiniz. Git ilə düzəlişlərinizi `iss53` dəyişiklikləri ilə birlikdə düzəltməyə ehtiyac yoxdur və düzəlişlərinizi production-da olanlara tətbiq etməzdən əvvəl bu dəyişiklikləri geri qaytarmaq üçün əziyyət çəkm əyinizə ehtiyac yoxdur. Sadəcə `master` branch-ına qayıtmaq lazımdır.

Ancaq bunu etməzdən əvvəl qeyd edin ki, əgər işlədiyiniz qovluq və ya quruluş sahəniz yoxlanılan branch-la konfilikt təşkil etmirsə, Git branch-ları dəyişməyə imkan vermir. Branch-ları dəyişdirə

rkən təmiz bir iş vəziyyətinə sahib olmağınız yaxşıdır. Daha sonra Stashing və Təmizləmə-də bəhs edəcəyimiz (yəni stashing və commit etmək) yolları var. Hələlik, bütün dəyişikliklərinizi etdiyinizi düşünün və yenidən `master` branch-ınıza qayıda bilərsiniz:

```
$ git checkout master
Switched to branch 'master'
```

Bu anda layihə işlədiyiniz qovluq #53 nömrəli problem üzərində işləməyə başlamamışdan əvvəlki yoldur və fikrinizi düzəlişə cəmləşdirə bilərsiniz. Bu yadda saxlamağınız vacib olan bir məqamdır: branch-ları dəyişdirdiyiniz zaman Git, işçi qovluğunuzu bu branch-da sonuncu dəfə etdiyiniz kimi görmək üçün sıfırlayır. Bu işləmə kopyasının branch-ının son commit-ə bənzədiyinə əmin olmaq üçün faylları əlavə edir, çıxarır və avtomatik olaraq dəyişdirir.

Sonra düzəltmək üçün düzəltmə nöqtəniz var. Gəlin tamamlanana qədər işləyəcək bir `hotfix` branch-ı yaradaq:

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'Fix broken email address'
[hotfix 1fb7853] Fix broken email address
 1 file changed, 2 insertions(+)
```

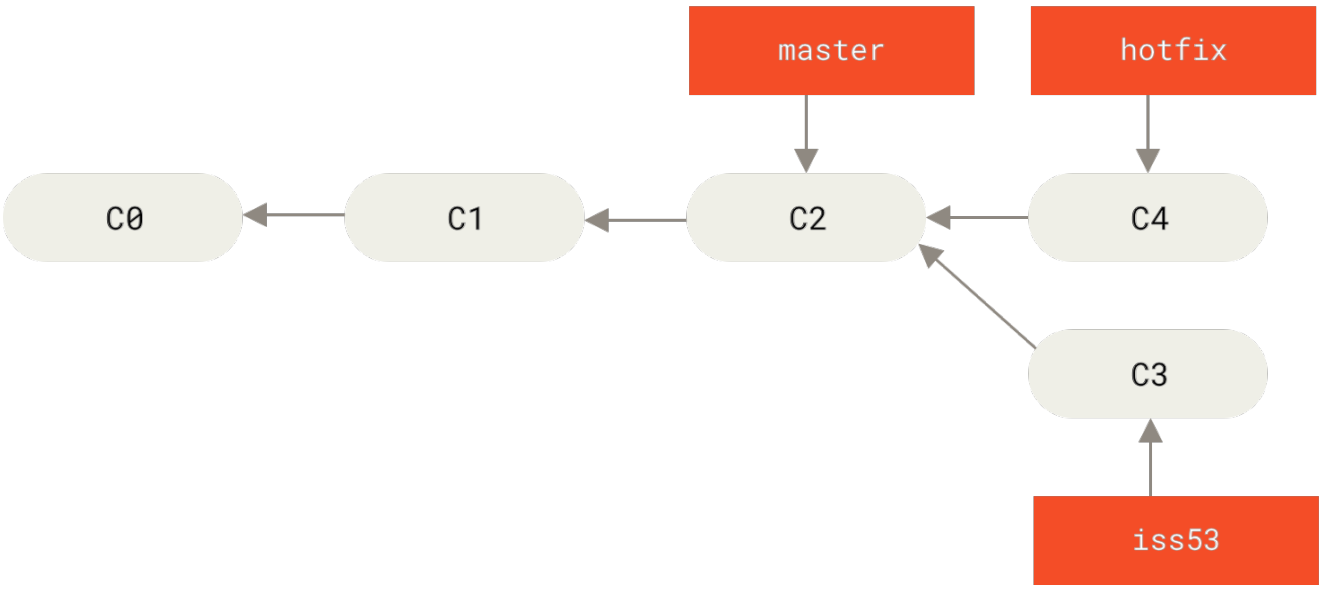


Figure 21. `master` əsasında Hotfix branch-ı

Testlərinizi işə sala bilərsiniz,hansı düzəlişi istədiyinizdən əmin olun və nəhayət `hotfix` branch-ını istehsal etmək üçün yenidən `master` branch-ınıza birləşdirin.

Siz bunu `git merge` əmri ilə edin:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

Bu birləşmədə “fast-forward” ifadəsini görəcəksiniz. Birləşdirdiyiniz `hotfix` branch-ı ilə göstərilən `C4`commit-i, üzərində olduğunuz `C2` commit-inin qabağında olduğundan Git sadəcə göstəricini ir əli aparır. Başqa bir şəkildə ifadə etmək üçün ilk commit-in tarixini izləməklə əldə edilə bilən bir commit-lə birləşməyə çalışdığınız zaman, Git göstərici irəli apararaq işləri sadələşdirir, çünki birlikdə birləşmək üçün ayrı-ayrılıqda işlər görülmür - buna deyilir “fast-forward.”

Dəyişiklik artıq `master` branch-nın işarə etdiyi commit-in snapshotudur və düzəlişi yerləşdirə bil ərsiniz.

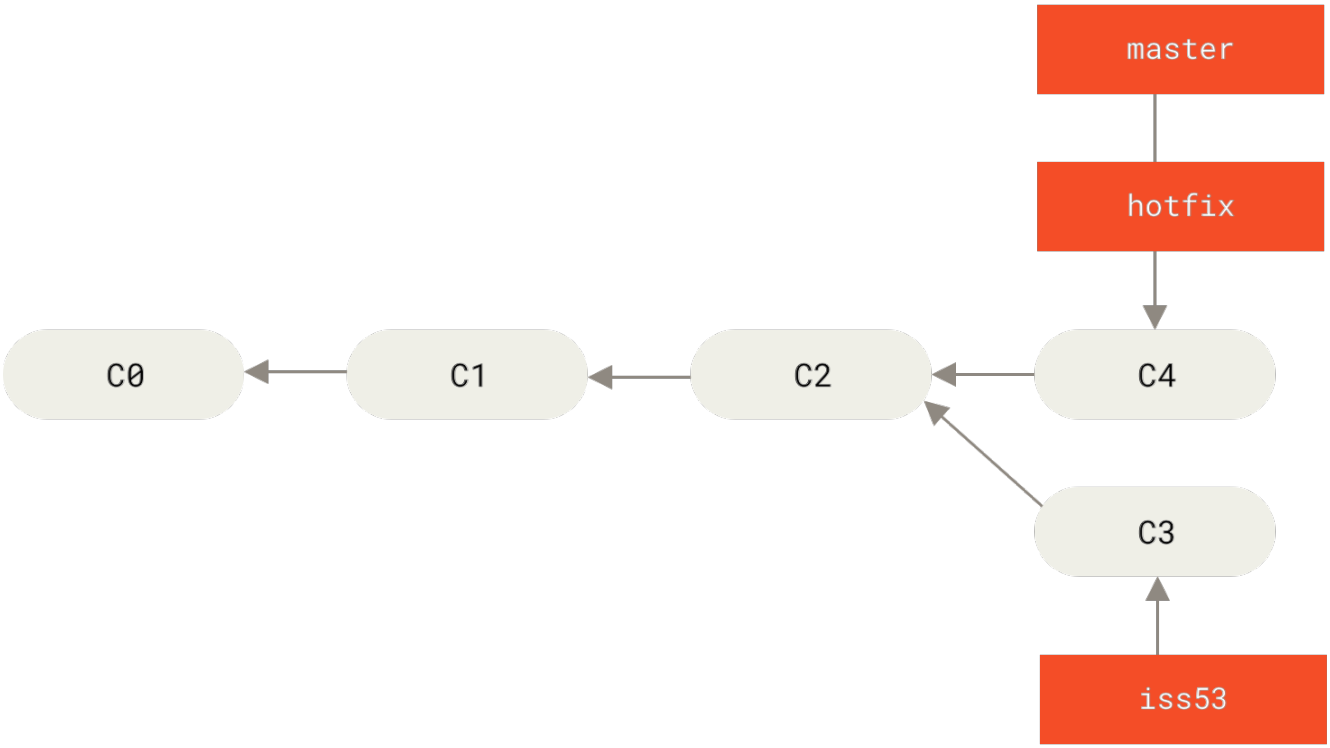


Figure 22. `master` sürətli şəkildə `hotfix`-ə yönləndirilir

Super əhəmiyyətli düzəlişiniz yerləşdirildikdən sonra müdaxilə etməmişdən əvvəl etdiyiniz işə geri dönməyə hazırsınız.

Ancaq əvvəlcə `hotfix` branch-nı silirsiniz, çünki artıq ehtiyacınız yoxdur - `master` branch-ı eyni yeri göstərir.

Onu `-d` seçimini `git branch` əmrinə əlavə edərək silə bilərsiniz:

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

İndi #53 nömrəli məsələ ilə əlaqədar işinizdə olan şöbəyə qayıda bilərsiniz və üzərində işləməyə davam edə bilərsiniz.

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'Finish the new footer [issue 53]'
[iss53 ad82d7a] Finish the new footer [issue 53]
1 file changed, 1 insertion(+)
```

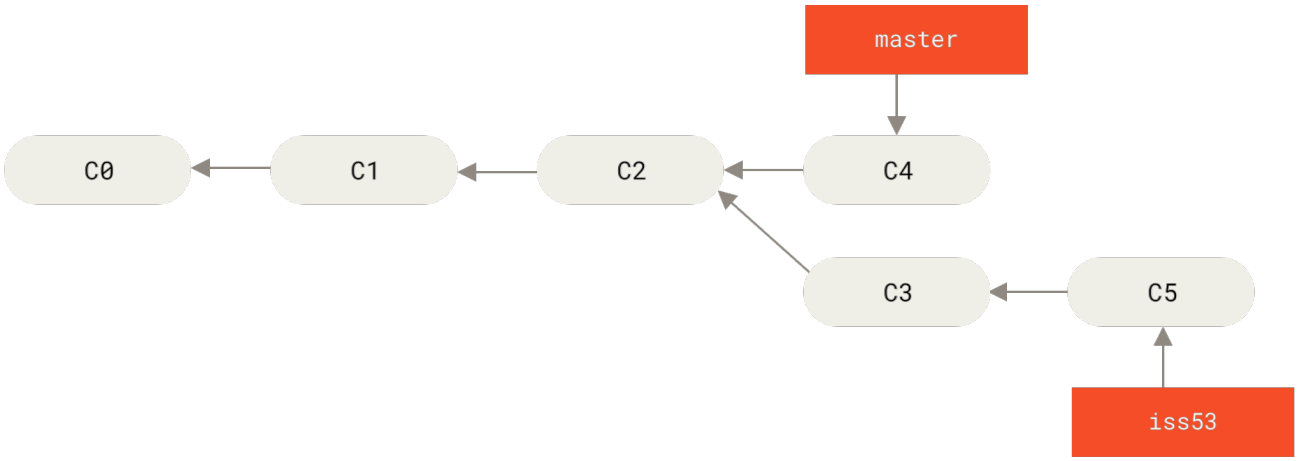


Figure 23. `iss53` üzərində işləməyə dəvam etmək

Burada qeyd etmək lazımdır ki, `hotfix` branch-da gördüyünüz işlər `iss53` branch-dakı sənədlərdə yoxdur. Əgər onu pull etmək lazımdırsa, `git merge master`-i işə salmaqla `master` branch-nızı `iss53` branch-a birləşdirə bilərsiniz və ya `iss53` branch-nı daha sonra `master` halına gətirməyə qərar ver ənə qədər bu dəyişikliklərin inteqrasiyasını gözləyə bilərsiniz.

Sadə Birləşdirmə

Təqdim etdiyiniz #53 nömrəli işiniz tamamlandı və `master` branch-nıza birləşdirilməyə artıq hazırdır. Bunu etmək üçün, `iss53` branch-nızı əvvəllər `hotfix` branch-nı birləşdirdiyiniz kimi `master` -ə birləşdirəcəksiniz. Yalnız etməli olduğunuz branch-ı yoxlamaq və sonra `git merge` əmrini icra etməkdir:

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

Bu əvvəl düzəltdiyiniz `hotfix` birləşməsindən bir az fərqli görünür. Bu vəziyyətdə inkişaf tarixiniz

bəzi köhnə nöqtələrdən ayrılacaqdır. Gördüyünüz branch-dakı commit birləşdiyiniz branch-ın birbaşa ancestor-u olmadığından Git bəzi işlər görməlidir. Bu vəziyyətdə Git branch ucları və ikisinin ortaq ancestor-u ilə göstərilən iki snapshot-u istifadə edərək sadə üç tərəfli birləşmə edir.

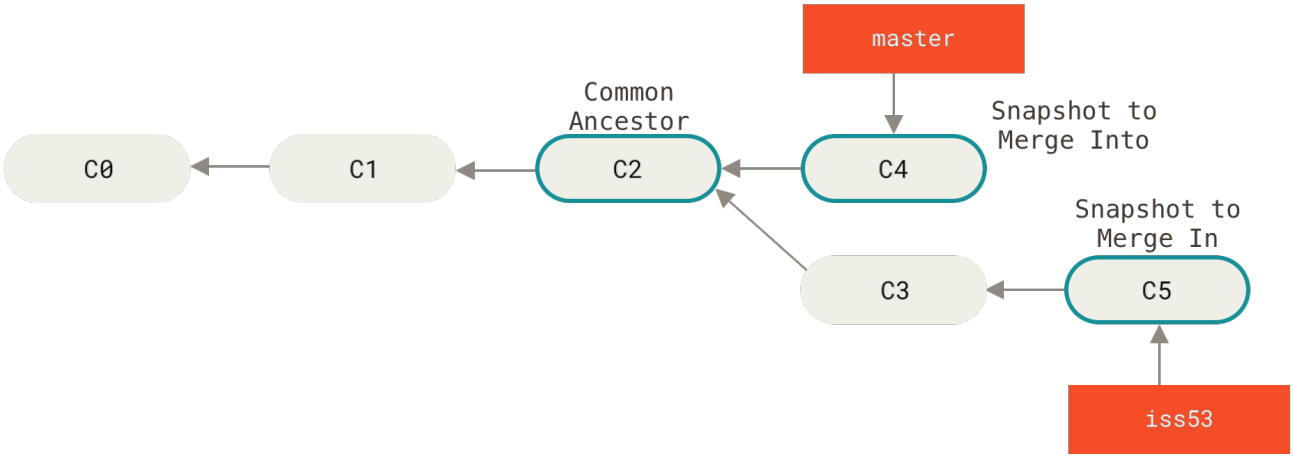


Figure 24. Tipik birləşmədə istifadə olunan üç snapshot

Branch göstərişini irəli sürmək əvəzinə, Git bu üç tərəfli birləşmənin nəticəsi olan yeni bir görüntü yaradır və avtomatik olaraq ona işarə edən yeni bir commit yaradır. Bu birləşmə commit-i adlandırılır və birdən çox ancestor-un olması ilə xüsusi olur.

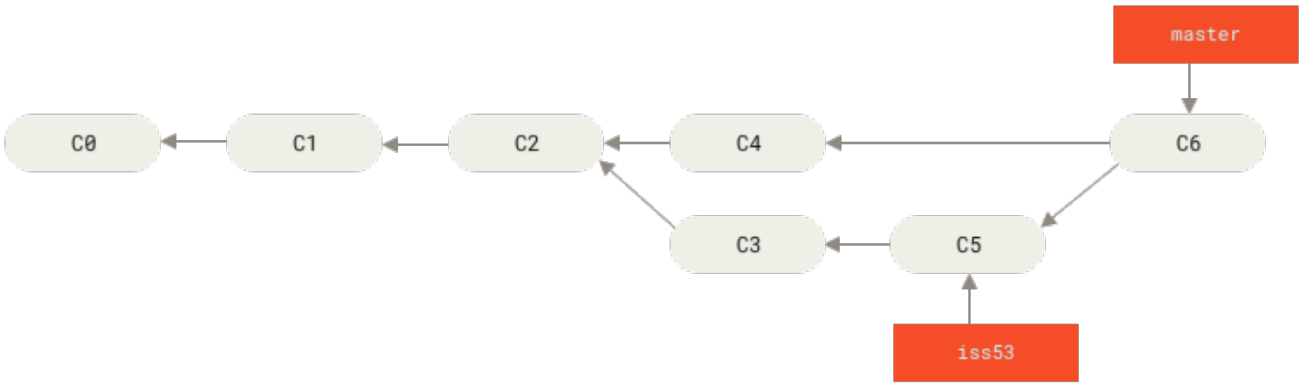


Figure 25. Birləşmə commit-i

İndi işiniz birləşdirildikdən sonra *iss53* branch-ına ehtiyacınız yoxdur. Məsələni izləmə sisteminizdə bağlaya və branch-ı silə bilərsiniz:

```
$ git branch -d iss53
```

Əsas Birləşmə Konfiliktləri

Bəzən bu proses əngəlsiz getmir. Birləşdirdiyiniz iki branch-da eyni faylın eyni hissəsini fərqli ş

əkildə dəyişsəniz, Git onları təmiz birləşdirə bilməyəcək. Əgər #53 nömrəli məsələ ilə bağlı düz əlişiniz `hotfix` branch-ı kimi bir faylın eyni hissəsini dəyişdirsə, bu kimi bir şeyə bənzər birləşmə konflikti alacaqsınız:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git avtomatik olaraq yeni birləşmə commit-i yaratmadı. Və konflikti həll edərkən prosesi dayandırdı. Birləşmə konfliktindən sonra hər hansı bir nöqtədə hansı sənədlərin açılmadığını görmək istəyirsinizsə, `git status`-u istifadə edə bilərsiniz:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Konfiliktləri birləşdirən və həll edilməmiş hər hansı bir şey əlaqələndirilməmiş kimi verilmişdir. Git, konfliktləri olan fayllara standart konflikt həll etmə işarələrini əlavə edir, onları manual açа və bu konfiliktləri həll edə bilərsiniz. Faylınızda bənzər bir hissə var:

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

Bu `HEAD`dakı versiya (sizin `master` branch-nız, çünki birləşmə əmrinizi icra edərkən yoxladığınız şey bu idi) blokun yuxarı hissəsidir (`=======`-dən yuxarıdakı hər şey), `iss53` branch-nızdakı versiya alt hissədəki hər şeyə bənzəyir.

Konflikti həll etmək üçün ya bir tərəfi,ya da digərini seçməlisiniz və ya özünüzü birləşdirməlisiniz.

Məsələn, bütün bloku aşağıdakı ilə əvəz etməklə bu konflikti həll edə bilərsiniz:

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

Bu həll yolunuz hər hissəsində bir az var və `<<<<<<<`, `=======` və `>>>>>>>` sətirləri tamamilə çıxarıldı. Hər bir konflikt faylda bu bölmələrin hər birini həll etdikdən sonra, həll olunduğunu qeyd etmək üçün hər bir fayl üzərində `git add` edin. Faylın səhnələşdirilməsi Git-də həll olunduğu kimi qeyd olunur.

Bu problemləri həll etmək üçün bir qrafik vasitədən istifadə etmək istəyirsinizsə, uyğun vizual birləşmə vasitəsini işə salan və qarşıdurmalardan keçən `git mergetool` işlədə bilərsiniz:

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge
p4merge araxis bc3 codecompare vimdiff emerge
Merging:
index.html

Normal merge conflict for 'index.html':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

Standartdan başqa bir birləşmə vasitəsini istifadə etmək istəyirsinizsə (əmr Mac-də işlədiyi üçün Git bu vəziyyətdə `opendiff` seçdi), aşağıdakı alətlərdən birini istifadə etdikdən sonra yuxarıda sadalanan bütün dəstəklənən alətləri görə bilərsiniz. İstədiyiniz vasitənin adını yazın.

Çətin birləşmə konfliktlərini həll etmək üçün daha inkişaf etmiş vasitələrə ehtiyacınız varsa, İnkişaf etmiş Birləşmə bölməsində daha çox məlumat veririk.

Birləşmə alətindən çıxdıqdan sonra Git birləşmənin uğurlu olub olmadığını soruşur. Skriptin olduğunu söyləsəniz, sənədin sizin üçün həll olunduğunu qeyd etmək üçün mərhələləndirir. Bütün konfliktlərin həll olunduğunu təsdiqləmək üçün yenidən `git status ` tətbiq edə bilərsiniz:

```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   index.html
```

Bundan məmnun olsanız və konflikt ilə əlaqəli hər şeyin səhnəyə qoyulduğunu təsdiqləsəniz, birl əşmə commitini yekunlaşdırmaq üçün `git commit` yazın. Varsayılan mesaj bu şəkildə olacaqdır:

```
Merge branch 'iss53'

Conflicts:
    index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#   .git/MERGE_HEAD
# and try again.


# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#   modified:   index.html
#
```

Gələcəkdə bu birləşməyə baxan başqalarına faydalı olacağını düşünürsənsə, birləşmə məsələsini necə həll etdiyiniz barədə təfərrüatları olan bu mesajı dəyişdirə bilərsiniz və ya bunlar aydın deyilsə, etdiyiniz dəyişiklikləri niyə etdiyinizi izah edə bilərsiniz.

Branch İdarəedilməsi

İndi bəzi branch-ları yaratdınız, birləşdirdiniz və sildiniz. Hər zaman branch-lardan istifadə etmə yə başladığınız zaman yararlanacağınız bəzi branch idarəetmə vasitələrinə baxaq.

`git branch` əmri branch-ları yaratmaq və silməkdən daha çox şey edir. Heç bir argument olmadan işləsəniz, cari branch-larınızın sadə bir siyahısını alırsınız:

```
$ git branch
  iss53
* master
  testing
```

`master` branch-nı əvvəlcədən təyin edən `*` simvoluna diqqət yetirin: bu, hazırda yoxladığınız branch-ı göstərir (yəni,`HEAD`-ın göstərdiyi branch-ı). Bu o deməkdir ki, bu anda commit etsəniz, `master` branch yeni işinizlə irəliləyəcəkdir. Hər bir branch-dakı son commit-i görmək üçün `git branch -v` işlədə bilərsiniz:

```
$ git branch -v
  iss53   93b412c Fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 Add scott to the author list in the readme
```

Faydalı `--merged` və `--no-merged` seçimlər bu siyahını mövcud olduğunuz branch-ı filtrləyə bilər. Hansı branch-ların artıq olduğunuz branch-a birləşdirildiyini görmək üçün `git branch --merged` işlədə bilərsiniz:

```
$ git branch --merged
  iss53
* master
```

Daha əvvəl `iss53`-də birləşdiyiniz üçün siyahınızda görürsünüz. Bu siyahıdakı branch-ları önlə rində "*" olmadan, ümumiyyətlə, `git branch -d` ilə silmək yaxşıdır; onsuz da işlərini başqa bir branch-a birləşdirmisiniz, buna görə heç nə itirməyəcəksiniz.

Hələ birləşdirmədiyiniz işlək olan bütün branch-ları görmək üçün `git branch --no-merged` işləyə bilərsiniz:

```
$ git branch --no-merged
  testing
```

Bu, digər branch-nızı göstərir. Çünki hələ birləşdirilməmiş bir iş var, onu `git branch -d` ilə silmək istədikdə uğursuz olacaq:

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

Həqiqətən branch-ı silmək və bu işi itirmək istəyirsinizsə, faydalı mesajda göstərildiyi kimi onu `-D` ilə məcbur edə bilərsiniz.

Yuxarıda təsvir edilən `--merged` və `--no-merged` variantları, bir arqument olaraq bir commit və ya branch adı verilmirsə, müvafiq olaraq sizin *cari* branch-ınıza birl əşdirildiyini və ya birləşdirilmədiyini göstərir.

Həmişə olduğu kimi bu branch-ı əvvəlcədən yoxlamadan başqa bir branch-a münasibətdə birləşmə vəziyyəti haqqında soruşmaq üçün əlavə bir arqument t əqdim edə bilərsiniz. `master` branch-a birləşməyən nədir?

```
$ git checkout testing
$ git branch --no-merged master
  topicA
  featureB
```

Branching İş Axınları

İndi branch və aşağı birləşmə əsaslarını öyrəndiyimizə görə, onlarla nə edə bilərsən və ya etməlis ən? Bu bölmədə, bu yüngül branch-ın mümkün olduğu bəzi ümumi iş axınlarını əhatə edəcəyik, buna görə onları öz inkişaf dövrünüzə daxil etmək istəməyinizə qərar verə bilərsiniz.

Uzun Müddət İşləyən Branch-lar

Git sadə üç tərəfli birləşmə istifadə etdiyi üçün uzun bir müddət ərzində bir branch-dan digərinə bir neçə dəfə birləşmək ümumiyyətlə asandır. Bu, həmişə açıq olan və inkişaf dövrünüzün müxt əlif mərhələləri üçün istifadə etdiyiniz bir neçə branch-a sahib ola biləcəksiniz deməkdir; müntəz əm olaraq bəzilərindən digərlərinə birləşə bilərsiniz.

Bir çox Git developerində bu yanaşmanı əhatə edən bir iş axını var, məsələn, yalnız `master` branch-da tamamilə sabit bir koda sahib olmaq - bəlkə də yalnız buraxılmış və ya buraxılacaq kod. Onların sabitlik yoxlamaq üçün işlədikləri və ya sabitliyi test etmək üçün `develop` və ya `next` adlı başqa bir paralel branch-ı var - bu mütləq həmişə sabit olmur, lakin sabit vəziyyətə gəldikdə onu `master`-ə birləşdirmək olar. Hazır olduqları zaman mövzu branch-ları (qısa müddətli branch-lar, əvvəlki `iss53` branch-ı kimi) hazırlamaq üçün istifadə etdikləri, bütün testləri keçdiklərini və səhvləri tanımadıqlarını təmin etmək üçün istifadə olunur.

Əslində, etdiyiniz tapşırığın cərgəsinə doğru hərəkət edən göstəricilərdən bəhs edirik. Sabit branch-lar törətmə tarixinizdəki xəttdən daha aşağı, bleeding-edge branch-lar isə tarixdən daha uzaqdır.

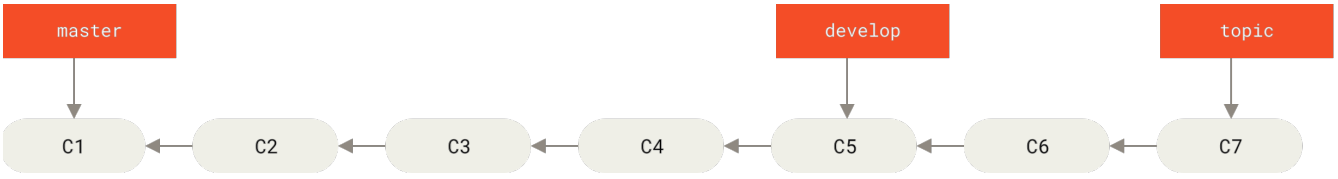


Figure 26. Proqressiv-sabitlik branch-nın xətti görünüşü

Tamamilə sınaqdan keçirildikdə dəstləri məzunları daha stabil bir silosa verdikləri iş silosları kimi düşünmək ümumiyyətlə daha asandır.

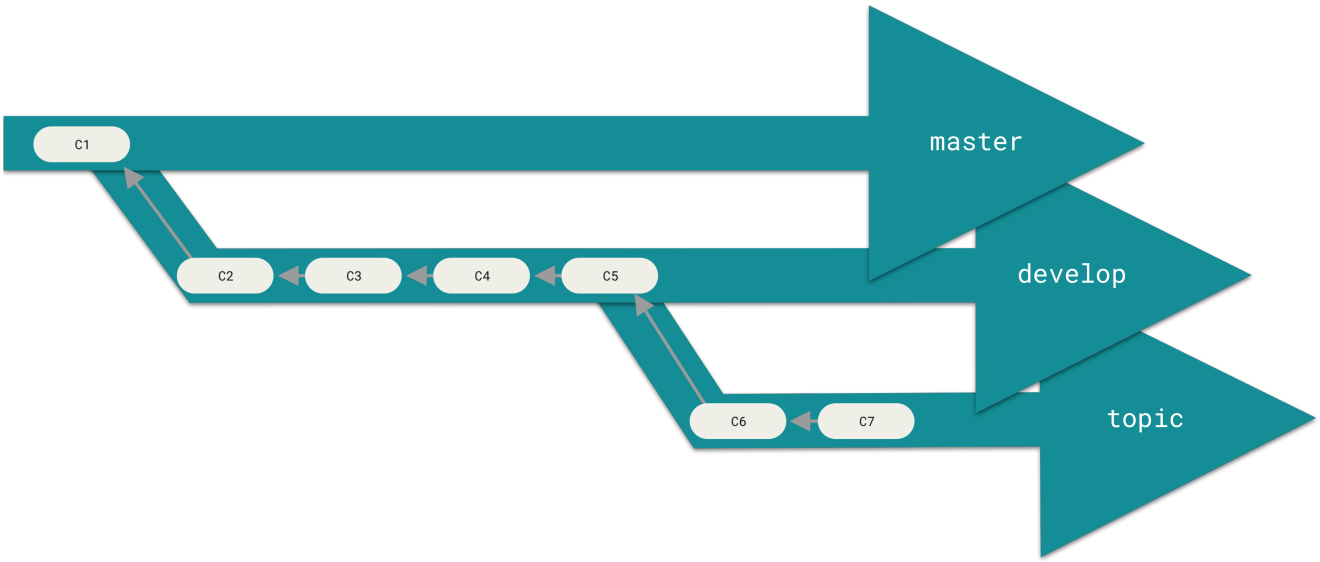


Figure 27. Proqressiv-sabitlik branch-a bir "silo" görünüşü

Bunu bir neçə səviyyəli sabitlik üçün davam etdirə bilərsiniz. Bəzi daha böyük layihələrdə, `next` və ya `master` branch-a girməyə hazır ola bilməyən branch-ı birləşdirən `proposed` və ya `pu` (təklif olunan yeniləmələr) branch-ı vardır. İdeya budur ki, branch-larınızın sabitliyin müxtəlif səviyyələrind ədir; daha sabit bir səviyyəyə çatdıqda, yuxarıdakı branch-a birləşdirilirlər. Yenə də çoxsaylı uzun branch-lara sahib olmaq lazım deyil, ancaq çox böyük və ya mürəkkəb layihələrlə məşğul olduğunuzda çox vaxt faydalıdır.

Mövzu Branch-ları

Mövzu branch-ları istənilən ölçüdə layihələrdə faydalıdır. Bir mövzu branch-ı müəyyən bir xüsusiyyət və ya əlaqəli bir iş üçün yaratdığınız və istifadə etdiyiniz qısa ömürlü bir branch-dır. Bu, əvvəllər bir VNS ilə heç görmədiyiniz bir şeydir, çünki branch yaratmaq və birləşdirmək ümumiyyətlə çox bahadır. Ancaq Git-də gündə bir neçə dəfə branch yaratmaq, üzərində işləmək, birləşdirmək və silmək çox yaygındır.

Bunu son hissədə yaratdığınız `iss53` və `hotfix` branch-ları gördünüz. Onlara bir neçə commit-lər verdiniz və əsas branch-ınıza birləşdirildikdən sonra birbaşa sildisiniz. Bu üsul sizə tez və tamamilə kontekstə keçməyə imkan verir - işiniz siloslara ayrıldığı üçün bu şöbədəki bütün d əyişikliklərin bu mövzu ilə əlaqəli olması, kodun baxılması zamanı nələrin baş verdiyini və bu kimi şeyləri görmək asandır. Dəyişiklikləri dəqiqələr, günlər və ya aylar ərzində saxlaya və hazır olduqları müddətdə, yaradılan və ya işlədikləri qaydadan asılı olmayaraq birləşdirə bilərsiniz.

Bəzi işlərin nümunəsini nəzərdən keçirin (`master`-da),bir məsələ üçün branching (`iss91`),eyni işin başqa bir yolunu sınamaq üçün ikinci branch üçün bir az üzərində işləyirik (`iss91v2`),sonra `master` branch-na qayıdıb bir müddət orada işlədib, sonra yaxşı bir fikir olduğuna əmin olmadığınız bir işi görmək üçün oraya branch edin (*dumbidea* branch-ı).

Sizin commit tarixiniz belə görünəcək:

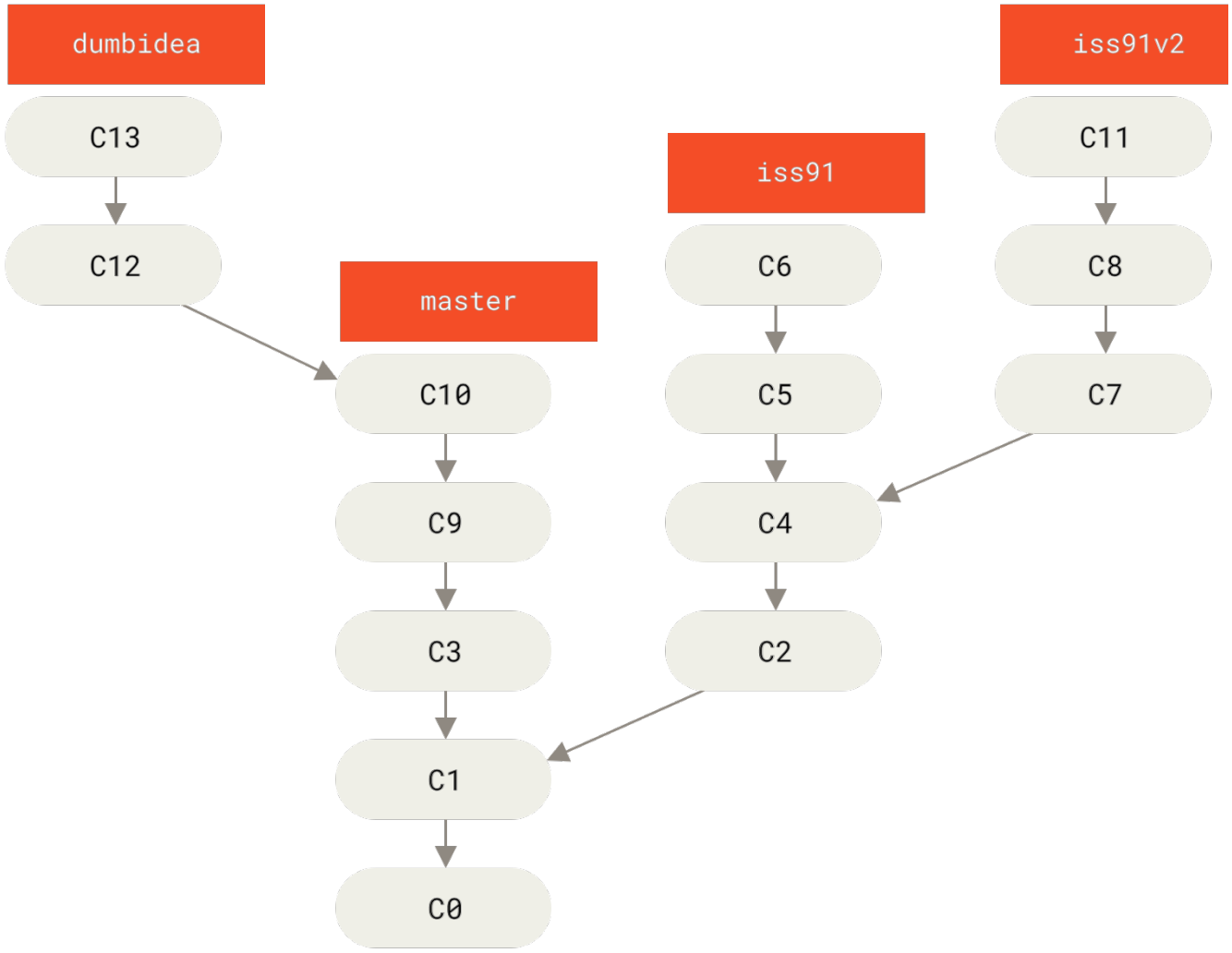


Figure 28. Bir neçə mövzulu branch-lar

İndi deyək ki, məsələnizin ən yaxşı ikinci həll yolu kimi qərar verdiniz (`iss91v2`);və iş yoldaşlarınıza `dumbidea` branch-nı göstərdiniz və genius olduğu ortaya çıxır. Orijinal `iss91` filialını ata bilərsiniz (`C5` və `C6` commit-lərini itirir) və digər ikisində birləşdirə bilərsiniz.

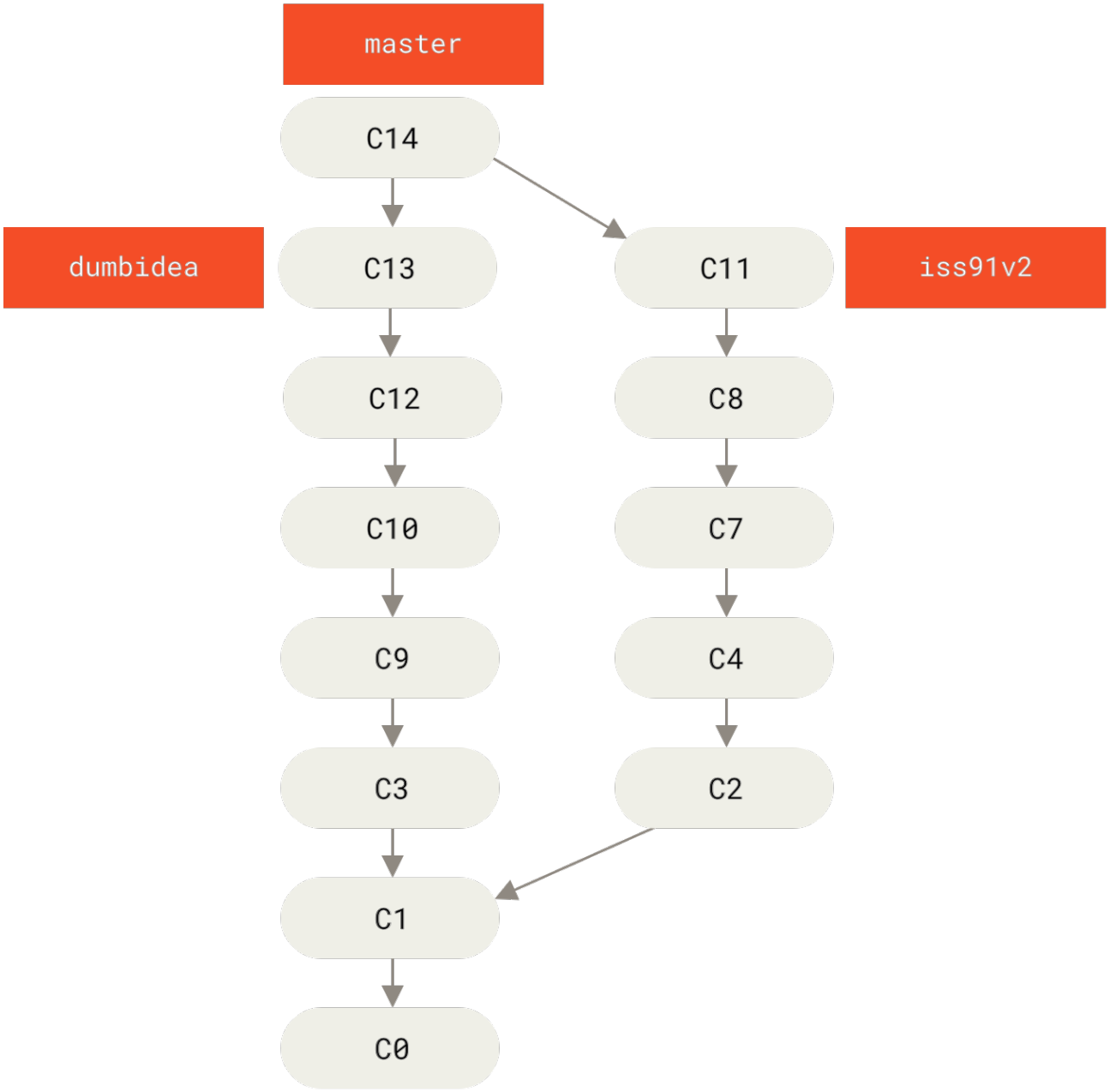


Figure 29. `dumbidea` *və* `iss91v2`*-ı birləşdikdən sonra tarix*

Git layihəniz üçün mümkün olan müxtəlif iş axını barədə Paylanmış Git-də daha ətraflı məlumat verəcəyik, buna görə növbəti layihənizin hansı branching sxemindən istifadə edəcəyinə qərar verməzdən əvvəl əmin olun.

Bütün bunları etdiyiniz zaman bu branch-ların tamamilə local olduğunu xatırlamaq vacibdir. Branching və birləşmə halında hər şey yalnız Git depolarınızda edilir - serverlə heç bir əlaqə yoxdur.

Uzaq Branch'lar

Uzaqdan verilən arayışlar, filiallar, etiketlər və sair də daxil olmaqla, uzaq depolarınızdakı istinadlar (işarələr). Əlavə məlumat üçün uzaq filiallar üçün `git ls-remote <remote>`və ya `git remote show <remote>` ilə açıq arayışların tam siyahısını əldə edə bilərsiniz. Buna baxmayaraq, daha yaygın bir yol, uzaqdan idarə olunan branch-lardan faydalanmaqdır.

Uzaqdan izləyən branch-la, uzaq branch-ların vəziyyətinə istinadlardır. Onlar hərəkət edə bilməy əcəyiniz yerli istinadlardır; Hər hansı bir şəbəkə rabitəsi etdikdə Git onları sizin üçün uzaq depo v əziyyətini dəqiq şəkildə təmsil etdiyinə əmin olmaq üçün hərəkət etdirir. Uzaqdakı depolarınızdakı filialların son dəfə bağlandığınızı xatırlatmaq üçün onları bookmark kimi düşünün.

Uzaqdan izləyən branch adları `<remote>/<branch>` şəklini alır. Məsələn, uzaqdan `origin` uzaqdakı `master` branch-nın nə ilə əlaqə qurduğunu görməyini istəsəniz, `origin.master` branch-nı yoxlayırsınız. Bir tərəfdaşla bir problem üzərində işləyirdinizsə və onlar `iss53` branch-nı push etdilərsə, local `iss53` branch-nız ola bilər, ancaq serverdəki branch uzaqdan izləmə branch-ı `origin/iss53` ilə təmsil ediləcəkdir .

Bu bir az qarışıq ola bilər, buna görə bir nümunəyə baxaq. Deyək ki, şəbəkənizdə Git serveriniz var `git.ourcompany.com`. Bundan klonlasanız, Git-in `clone` əmrini avtomatik olaraq sizin üçün `origin` adlandırır, bütün məlumatları aşağı salır, `master` branch-nın olduğu yerə bir göstərici yaradır və local olaraq `origin/master` adlandırır. Git, local `master` branch-ı ilə eyni yerdə başlayaraq local `master` branch-nı verir, buna görə işləmək üçün bir şeyiniz var.

“origin” xüsusi deyil

Nə branch adı “master”, nə də “origin” Git’də xüsusi bir məna daşımır. Geniş istifadə edilməsinin yeganə səbəbi olan `git init` işlədikdə “master” başlanğıc branch üçün standart ad olsa da,`git clone` işlədərkən “origin” uzaqdan verilən addır. Bunun əvəzinə `git clone -o booyah` işlədirsinizsə, standart uzaq filialınız olaraq `booyah/master` olacaqdır.

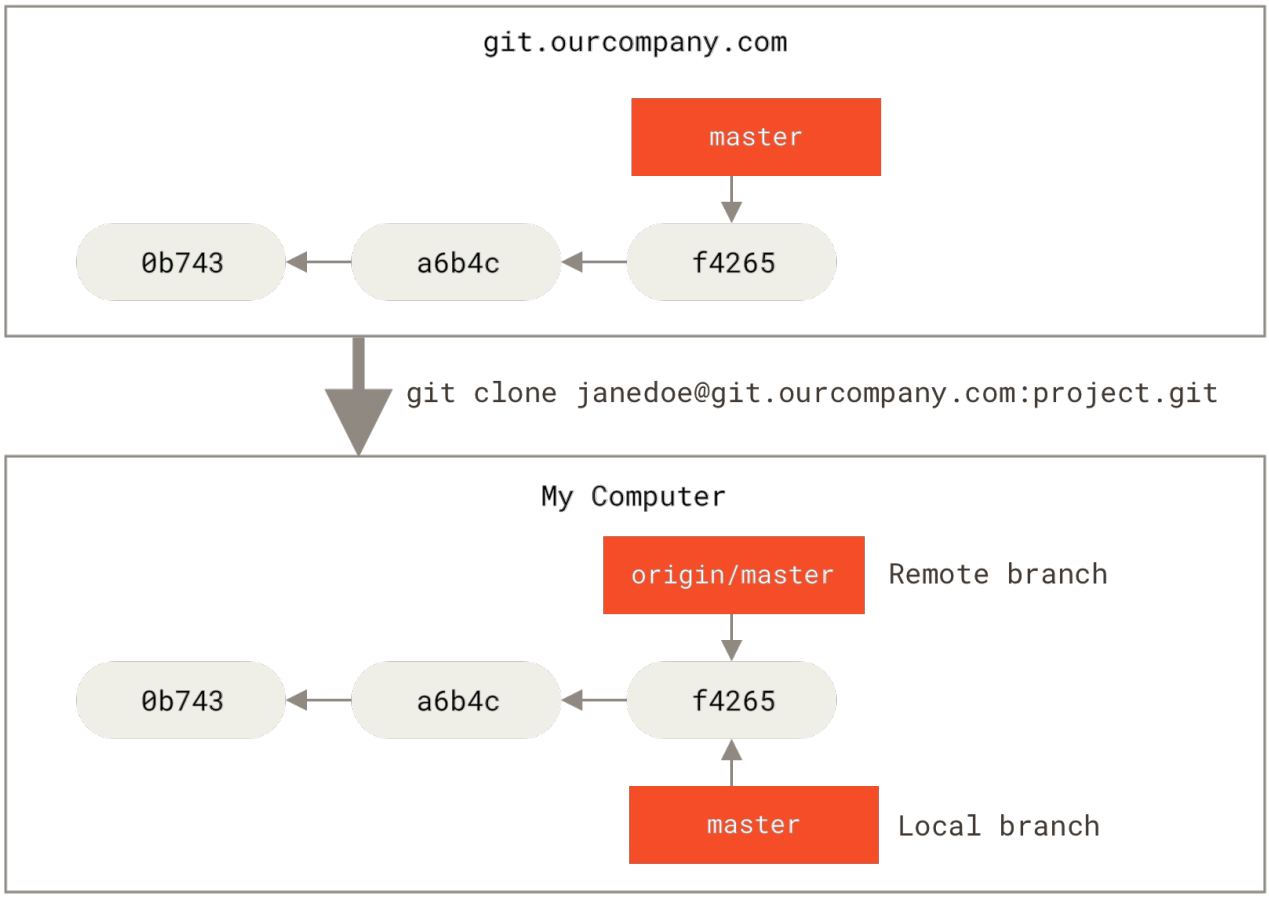


Figure 30. Klonlamadan sonra server və local depo

Local `master` branch-nızda bir az iş görsəniz və bu vaxt başqa birisi `git.ourcompany.com`a push etsə və `master branch-nı yeniləyirsə, onda tarixləriniz fərqli şəkildə irəliləyəcəkdir. Ayrıca, `origin` serverinizlə əlaqədən kənarda qaldığınız müddətcə, `origin/master` markeriniz hərəkət etmir.

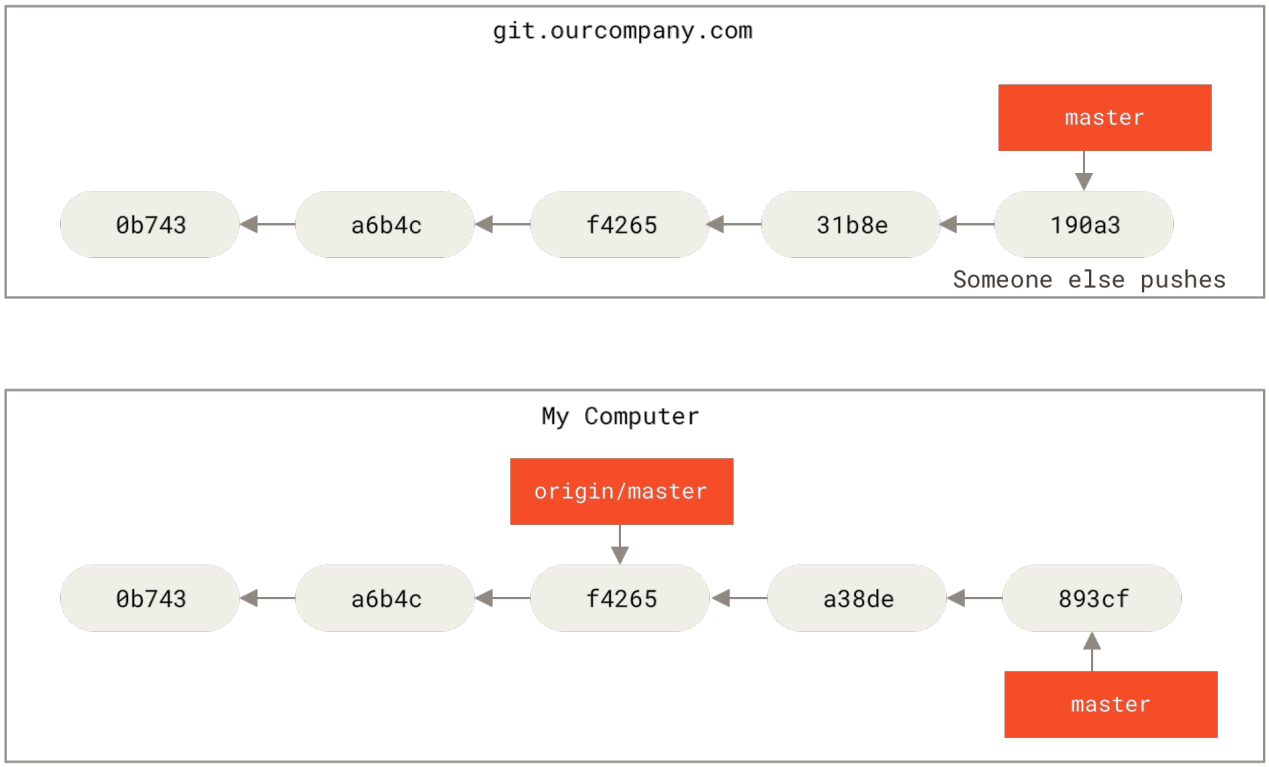


Figure 31. Local və uzaqdan iş bir-birindən ayrıla bilər

Uzaqdan işinizi sinxronlaşdırmaq üçün `git fetch <remote>` əmrini işlədirsiniz (bizim vəziyyə timizdə `git fetch origin`). Bu əmr "origin" hansı server olduğunu (bu vəziyyətdə bu `git.ourcompany.com`) axtarır, olmayan hər hansı bir məlumat alır və local mənbənizi yeniləyir `origin/master`i yeni, daha müasir vəziyyətə gətirir.

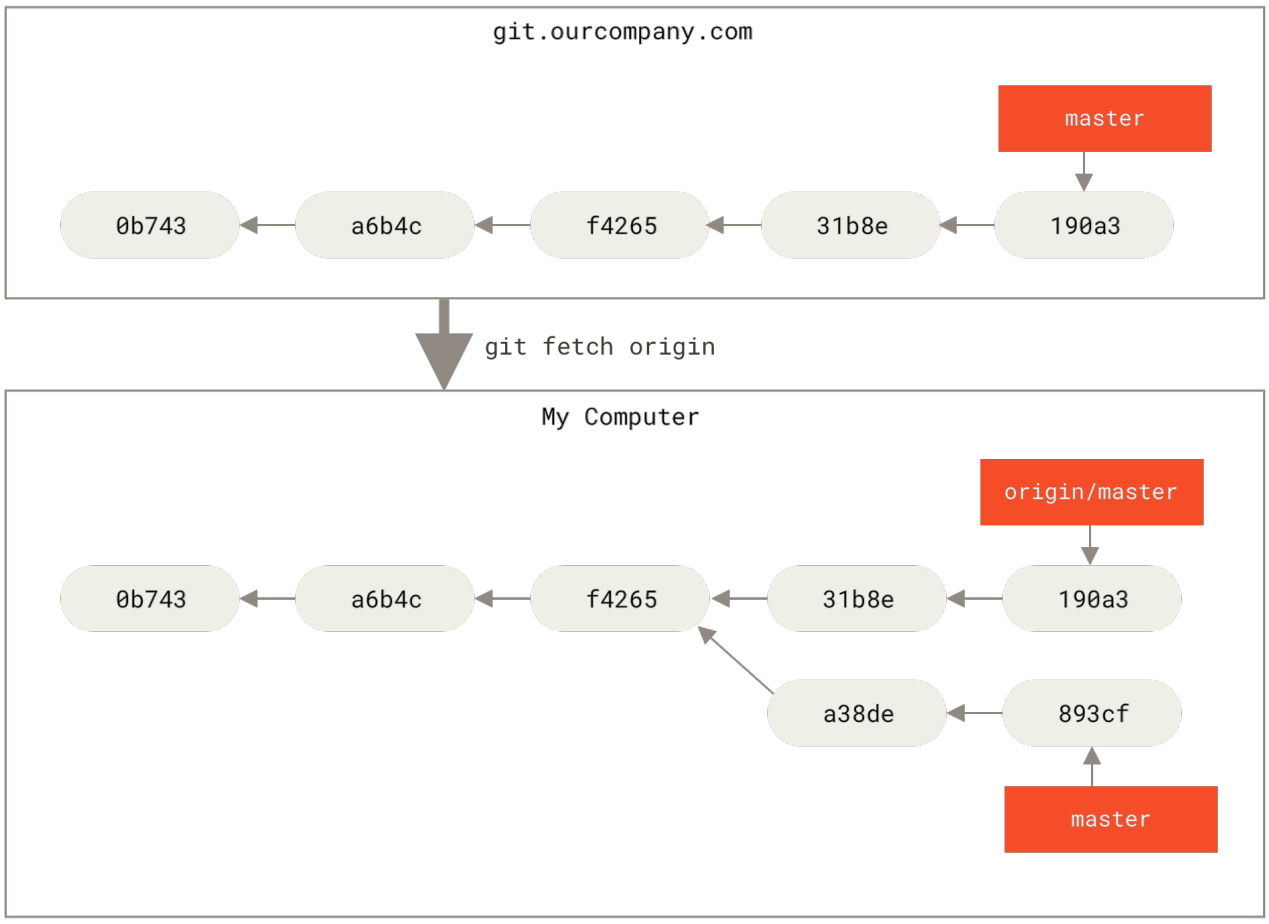
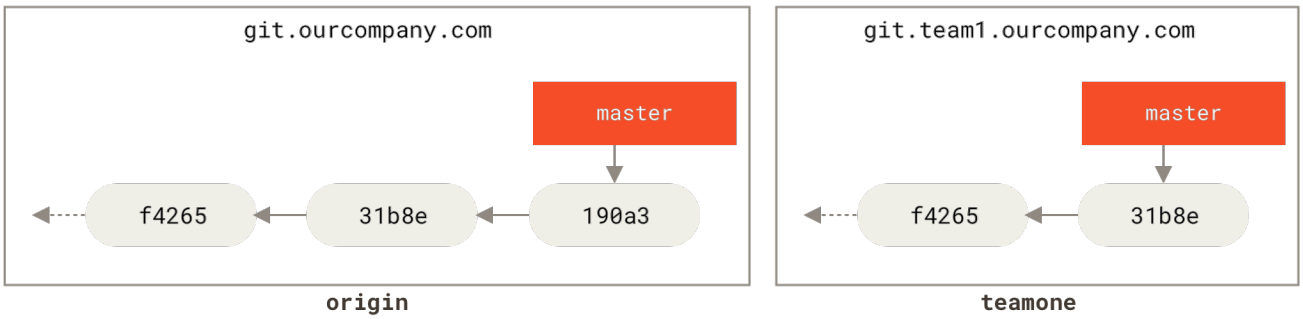


Figure 32. `git fetch` uzaqdan izləyən branch-ları yeniləyir

Fərz edək ki, çoxsaylı uzaq serverlərin olduğunu və bu uzaq layihələr üçün hansı uzaq filialların göründüyünü göstərmək üçün yalnız sprint komandalarınızdan birinin inkişafı üçün istifadə olunan başqa bir daxili Git serveriniz var. Bu server `git.team1.ourcompany.com`-dadır. Bunu Git'in Əsasları bəhs etdiyimiz kimi `git remote add` əmrini işlədərək hazırladığınız layihəyə yeni bir uzaq istinad kimi əlavə edə bilərsiniz. Bütün bu URL üçün qısa ad olacaq bu uzaq "teamone" adlandırın.




```
git remote add teamone git://git.team1.ourcompany.com
```

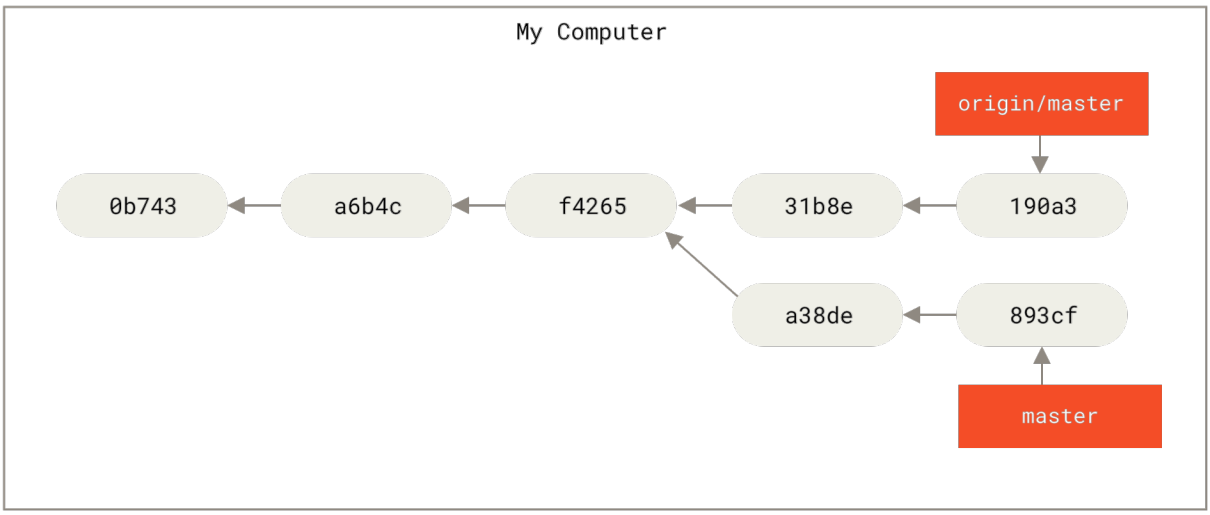


Figure 33. Uzaqdan başqa serverin əlavə edilməsi

İndi uzaq olan `teamone` serverində hələ sizdə olmayan şeyləri əldə etmək üçün `git fetch teamone` işlədə bilərsiniz. Çünki bu server sizin `origin` serverinizdə hazırda verilənlərin alt hissəsinə malikdir. Git heç bir məlumat əldə etmir, lakin `teamone` nin `master` branch-ı kimi götürdüyü tapşırığa işarə etmək üçün `teamone/master` adlı uzaqdan izləmə branch-ı təyin edir.

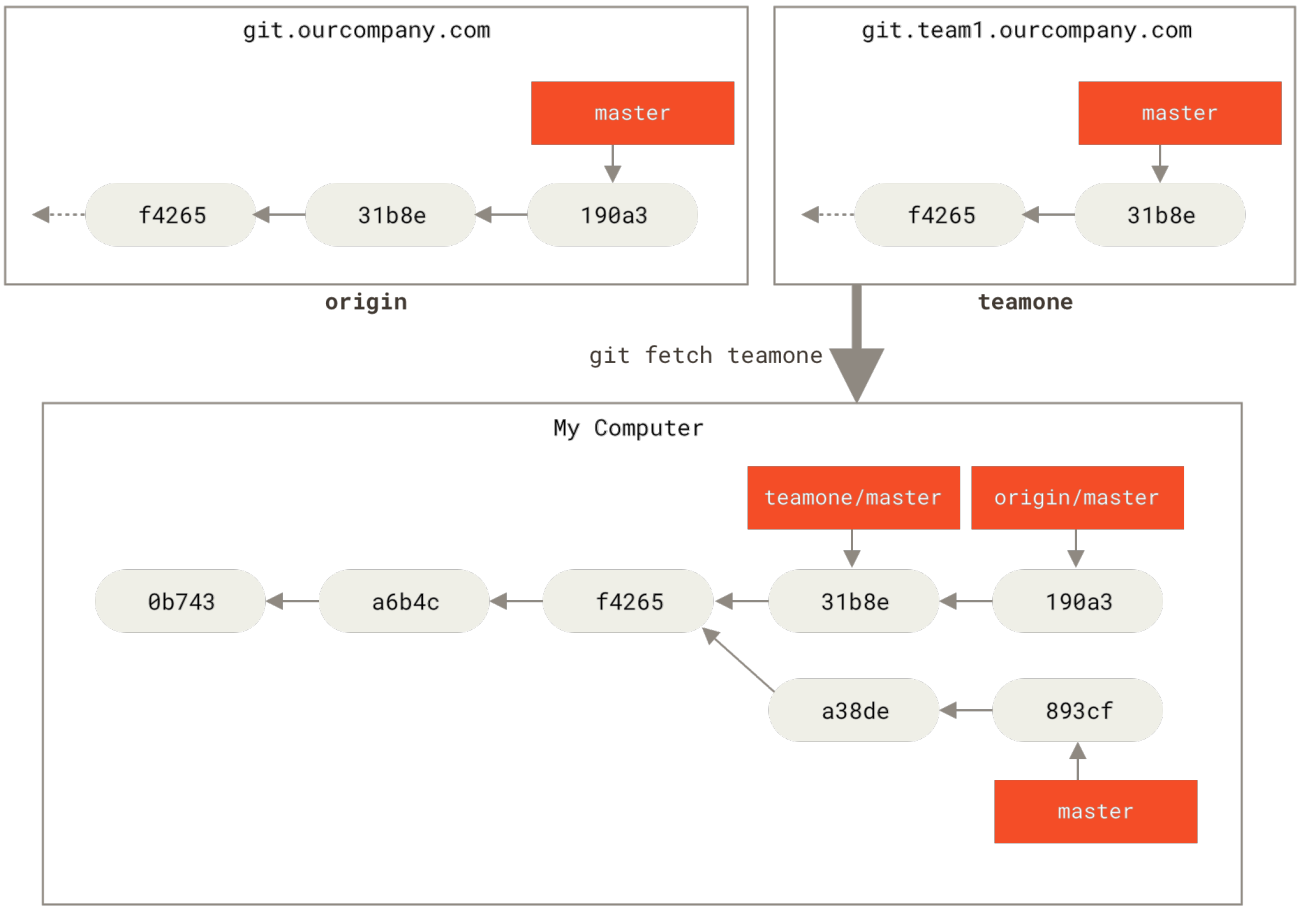


Figure 34. `teamone/master` *üçün uzaqdan izləmə filialı*

Pushing (İtələmə)

Bir filialı dünya ilə bölüşmək istədikdə, onu yazmaq imkanı olan uzaq bir yerə itələməlisiniz. Local branch-larınız avtomatik olaraq yazdığınız məsafələrə sinxronizasiya olunmur - bölüşmək ist ədiyiniz branch-ları açıq şəkildə göstərməlisiniz. Beləliklə, bölüşmək istəmədiyiniz iş üçün private branch-lardan istifadə edə bilərsiniz və yalnız əməkdaşlıq etmək istədiyiniz topic branchl-larını push edə bilərsiniz.

Başqaları ilə işləmək istədiyiniz `serverfix` adlı bir branch-nız varsa, ilk branch-nızı push etdiyiniz kimi push edə bilərsiniz. `git push <remote> <branch>` əmrini işlət:

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

Bu bir az qısa yoldur. Git avtomatik olaraq `serverfix` branch adını `refs/heads/serverfix:refs/heads/serverfix` olaraq genişləndirir. Bu o deməkdir ki, “Take my `serverfix` local branch and push it to update the remote’s `serverfix` branch.” `refs/heads/` hissəsini

detallı Git'in Daxili İşləri üzərində keçəcəyik, ümumiyyətlə onu bağlıya da bilərsiniz. Eyni şeyi `git push origin serverfix:serverfix`-də edə bilərsiniz — "Take my serverfix and make it the remote's serverfix." Local bir branch-ı fərqli adlandırılan uzaq bir brancha-a push etmək üçün bu formatdan istifadə edə bilərsiniz. Uzaqdan `serverfix` adlandırılmasını istəməsəniz, local `serverfix` branch-nı uzaq layihədəki `awesomebranch` branch-a push etmək üçün `git push origin serverfix:awesomebranch` işlətmək olar.

Şifrənizi hər dəfə yazmayın

Üzərinə push etmək üçün bir HTTPS URL istifadə edirsinizsə, Git server istifadəçi adınızı və identifikasiyanız üçün şifrənizi istəyəcək. Varsayılan olaraq, bu m əlumat üçün terminalda ad və şifrənizi soruşacaq, buna görə server təkan verə biləcəyinizi söyləyə bilər.

Hər dəfə push etdiyiniz zaman onu yazmaq istəmirsinizsə, "credential cache" qura bilərsiniz. Ən sadə, onu `git config --global credential.helper cache` ilə asanlıqla qurub bir neçə dəqiqə ərzində yaddaşda saxlamaqdır.

Daha çox məlumat üçün Etibarlı Yaddaş-ə bax.

Növbəti dəfə əməkdaşlarınızdan biri serverdən aldıqda `serverfix`-ın server versiyasının uzaq branch-ı `origin/serverfix` altında olduğu barədə bir məlumat əldə edəcəklər:

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

Diqqətə çatdırmaq lazımdır ki, yeni uzaqdan idarə olunan branch-ları çıxaran zaman avtomatik olaraq onların local, düzəliş olunan kopyaları olmur. Başqa sözlə, bu vəziyyətdə yeni bir `serverfix` branch-nız yoxdur - yalnız dəyişdirə bilmədiyiniz bir `origin/serverfix` göstərici var.

Bu işi cari işləyən branch-ınıza birləşdirmək üçün `git merge origin/serverfix` işlədə bilərsiniz. Çalışa biləcəyiniz öz `serverfix` branch-nızı istəyirsinizsə, onu uzaqdan izləyən branch-nızdan əsas götürə bilərsiniz:

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

Bu, `origin/serverfix` olduğu yerdə başlaya biləcəyiniz local bir branch verir.

İzləmə Branch-ları

Local bir branch-ı uzaqdan izləyən bir branch-dan yoxlamaq avtomatik olaraq "tracking branch"

adlandırılan bir şey yaradır (və izlədiyi branch “upstream branch”'adlanır). İzləmə branch-larınız uzaq bir branch-la birbaşa əlaqəsi olan local branch-lardır. Bir izləmə branch-ında olsanız və `git pull` yazsanız, Git hansı serverdən alınacağını və hansı branch-a qoşulacağını avtomatik olaraq bilir.

Bir depo klonlaşdırdıqda, ümumiyyətlə avtomatik olaraq `origin/master` izləyən bir `master` branch-ı yaradır. Ancaq istəsəniz digər izləmə branch-larını - digər uzaqdan branch-ları izləyənləri və ya `master` branch-nı izləməyənləri qura bilərsiniz. Gördüyünüz nümunə sadə işdir, `git checkout -b <branch> <remote>/<branch>`. Bu Git `--track` təqdim etdiyi kifayət qədər ümumi bir əməliyyatdır:

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

Əslində, bu o qədər yaygındır ki, hətta bu qısa yol üçün qısa yol var. Əgər yoxlamağa çalışdığınız branch-ın adı yoxdursa (a) yalnız bir uzaqdan bir adla (b) tam uyğun gəlirsə, Git sizin üçün bir izl əmə branch-ı yaradacaq:

```
$ git checkout serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

Uzaq branch-dan fərqli bir adla local bir branch qurmaq üçün ilk versiyanı fərqli bir local branch adı ilə asanlıqla istifadə edə bilərsiniz:

```
$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch serverfix from origin.
Switched to a new branch 'sf'
```

İndi `sf` local branch-nız avtomatik olaraq`origin/serverfix`-dən pull olacaq. Artıq local bir branch-nız varsa və onu yeni pull down etdiyiniz uzaq bir branch-a qurmaq və ya izlədiyiniz yuxarı branch-ı dəyişdirmək istəyirsinizsə, `-u` və ya `--set-upstream-to` istifadə edə bilərsiniz. İstədiyiniz zaman açıq şəkildə təyin etmək üçün `git branch` seçimini seçin.

```
$ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
```

Yuxarıdakı shorthand

Bir izləmə branch-ı qurulduğunda onun yuxarı hissəsini `@{upstream}` və ya `@{u}` shorthand-ləri ilə istinad edə bilərsiniz. Beləliklə, `master` branch-ında olsanız və `origin/master` izləyirsinizsə,`git merge origin/master` yerinə `git merge @{u}` istifadə edə bilərsiniz.

Hansı izləmə branch-larını qurduğunuzu görmək istəyirsinizsə, `-vv` seçimini `git branch-a` a istifadə

edə bilərsiniz. Bu, hər branch-ın izlədiyi və local branch-ın qabaqda, arxada və ya hər ikisində olduğu kimi daha çox məlumatla local branch-larınızı siyahıya alacaq.

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] Add forgotten brackets
  master    1ae2a45 [origin/master] Deploy index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] This should do it
  testing   5ea463a Try something new
```

Beləliklə, görürük ki, `iss53` branch-mız `origin/iss53`-i izləyir və iki addım qabaqdadır, yəni yerli olaraq serverə sövq edilməyən iki tapşırıq var. Bir də görə bilərik ki, bu günə qədər `master` branch-mız `origin/master`-i izləyir. Sonradan görürük ki, `serverfix` branch-mız `teamone` serverimizdəki `server-fix-good` branch-nı izləyir və üç qabaqda və bir arxadadır.Bu o deməkdir ki, hələ birləşm ədiyimiz serverdə bir commit var və üçü push etmədyimizi local olaraq həyata keçirir. Nəhayət, `testing` branch-mızın heç bir branch-ı izləmədiyini görə bilərik.

Bu nömrələr yalnız hər bir serverdən sonuncu dəfə alındıqdan bəri qeyd etmək vacibdir. Bu əmr serverlərə çatmır, local olaraq bu serverlərdən nə saxladığını izah edir. Tamamilə əvvəlcədən və arxadan nömrələrin yenilənməsini istəyirsinizsə, bu işə başlamazdan əvvəl bütün uzaqdan m əlumatları gətirməlisiniz. Siz bunu belə edə biləsiniz:

```
$ git fetch --all; git branch -vv
```

Pulling (Çəkmək)

`git fetch` əmri hələ olmayan serverdəki bütün dəyişiklikləri götürsə də, işləyən qovluğunuzu ümumiyyətlə dəyişdirməyəcəkdir. Sadəcə sizin üçün məlumatları əldə edəcək və özünüzü birl əşdirməyə imkan verəcəkdir. Bununla birlikdə, `git pull` deyilən bir əmr var və əksər hallarda d ərhal `git merge` ilə izlənilən `git fetch` deməkdir. Son hissədə göstərildiyi kimi bir izləmə branch-ı varsa, onu açıq şəkildə təyin etməklə və ya `clone` və ya `checkout` əmrləri ilə yaradaraq, `git pull` server və branch-nıza baxacaq. Cari branch-ı izləyib, həmin serverdən alacaq və sonra uzaq branch-da birləşməyə çalıcaqsınız. Ümumiyyətlə `fetch` və `merge` əmrlərini istifadə etmək daha yaxşıdır, çünki `git pull` əmri çox vaxt qarışıq ola bilər.

Uzaq Branch-ların Silinməsi

Uzaq bir branch-la bitdiyinizi düşünün - sizin və əməkdaşlarınızın bir xüsusiyyət ilə tamamlandığını və uzaqdan `master` branch-nıza birləşdirin (və ya sabit bir kod xəttiniz olan hər hansı bir şöbə). `git push`-da `--delete` seçimini istifadə edərək uzaqdan bir branch-ı silmək olar.

Serverdən `serverfix` branch-nı silmək istəyirsinizsə, aşağıdakı əmrləri izləyə bilərsiniz

```
$ git push origin --delete serverfix
To https://github.com/schacon/simplegit
 - [deleted]         serverfix
```

Əsas olaraq bütün bunlar göstəricini serverdən silməkdir. Git server ümumiyyətlə məlumatları bir garbage collection işləməyincə bir müddət saxlayacaq, buna görə təsadüfən silinibsə, tez-tez bərpa etmək asandır.

Rebasing

Git'də dəyişiklikləri bir budaqdan digərinə birləşdirməyin iki əsas yolu var: `merge` və `rebase`. Bu bölmədə rebase-in nə olduğunu, bunu necə edəcəyinizi, niyə olduqca gözəl bir vasitədir və hansı hallarda istifadə etmək istəmədiyinizi öyrənəcəksiniz.

Sadə Rebase

Əvvəlki bir nümunəyə qayıtsanız Sadə Birləşdirmə, işinizi bölüşdürdüyünüzü və iki fərqli branch-da qərar verdiyinizi görə bilərsiniz.

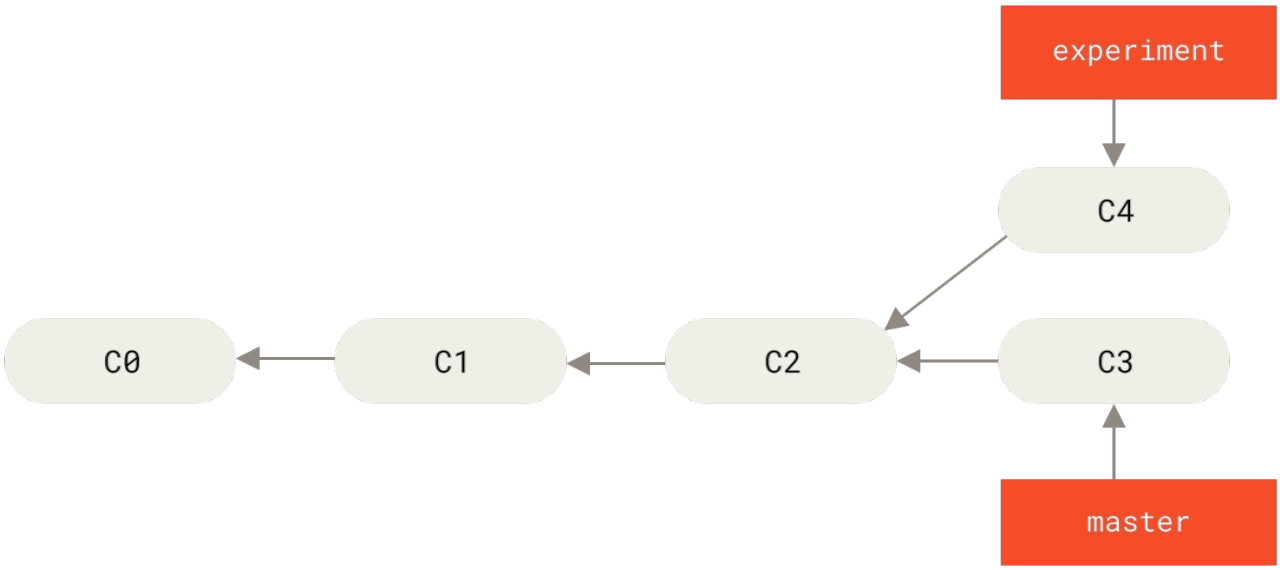


Figure 35. Sadə fikri ayrılığı

Branch-ları birləşdirməyin ən asan yolu, yuxarıda bəhs etdiyimiz kimi, `merge` əmridir. İki ən son branch snapshotu (`C3` və` C4`) və ikisinin ən son ortaq ancestor-u (`C2`) arasında üç tərəfli birləşmə həyata keçirir, yeni bir snapshot yaradır (və commit edir).

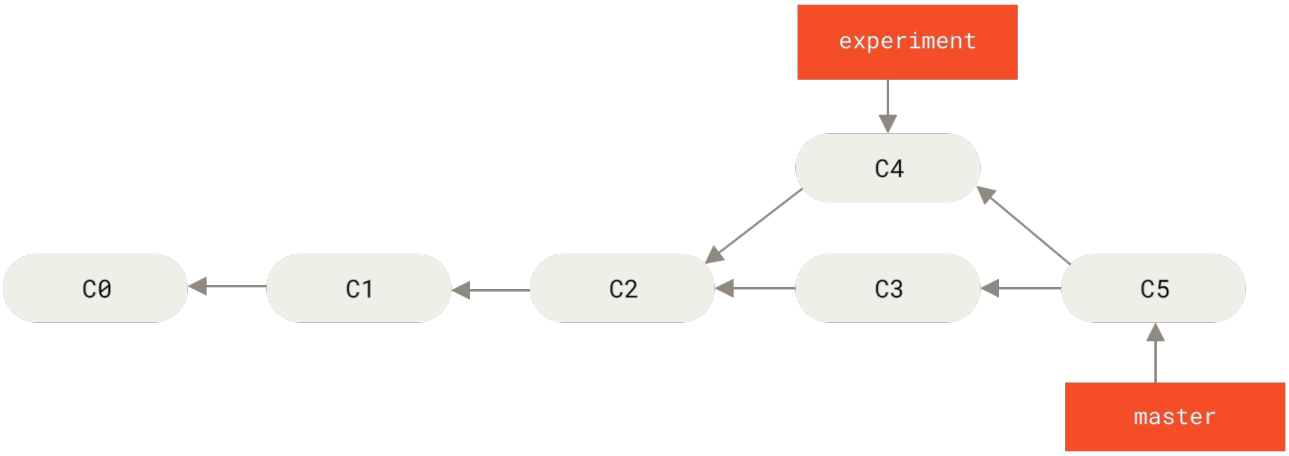


Figure 36. Ayrılmış iş tarixini inteqrasiya etmək üçün birləştirmək

Ancaq başqa bir yol var: `C4`-də təqdim edilən dəyişikliyin patch-ını götürüb `C3`-ün üstünə yenidən t ətbiq edə bilərsiniz. Git-də buna *rebasing* deyilir. `rebase` əmri ilə bir branch-da edilmiş bütün d əyişiklikləri götürüb başqa bir branch-da təkrarlaya bilərsiniz.

Bu misal üçün `experiment` branch-ını yoxlayıb, `master` branch-a aşağıdakı kimi qaytarın:

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

Bu əməliyyat iki branch-ın ortaq ancestor-una (birinə davam etdiyiniz və birinə rebasing etdiyiniz) gedərək işləyir,olduğunuz branch-ın hər bir commit-i tərəfindən təqdim olunan fərqləri əldə edər ək, həmin fərqləri müvəqqəti sənədlərdə saxlayaraq işləyir, hazırkı branch-ı yenidən düzəltdiyiniz branch-la eyni commit-ə bərpa edin və nəhayət hər dəyişikliyi öz növbəsində tətbiq edin.

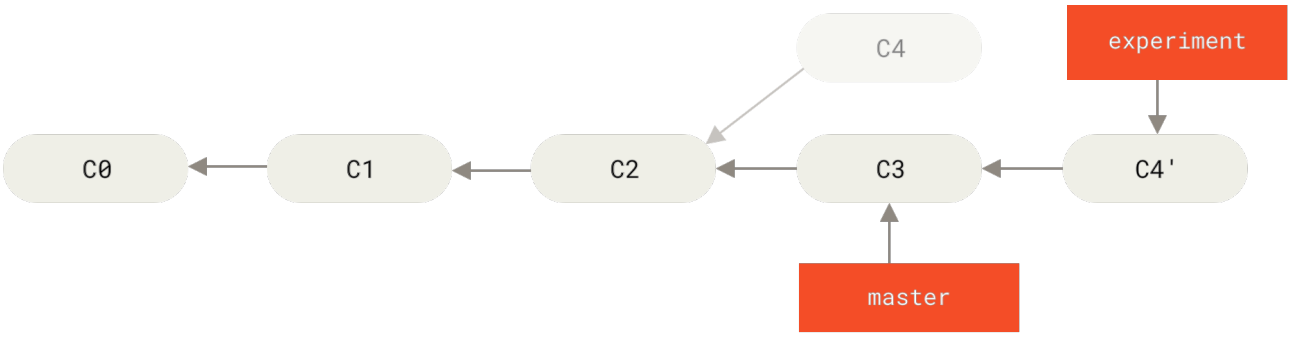


Figure 37. `C4`də `C3`-ə edilən dəyişikliyin rebasing edilməsi

Bu nöqtədə, `master` branch-a qayıda və sürətli bir şəkildə birləşə bilərsiniz.

```
$ git checkout master
$ git merge experiment
```

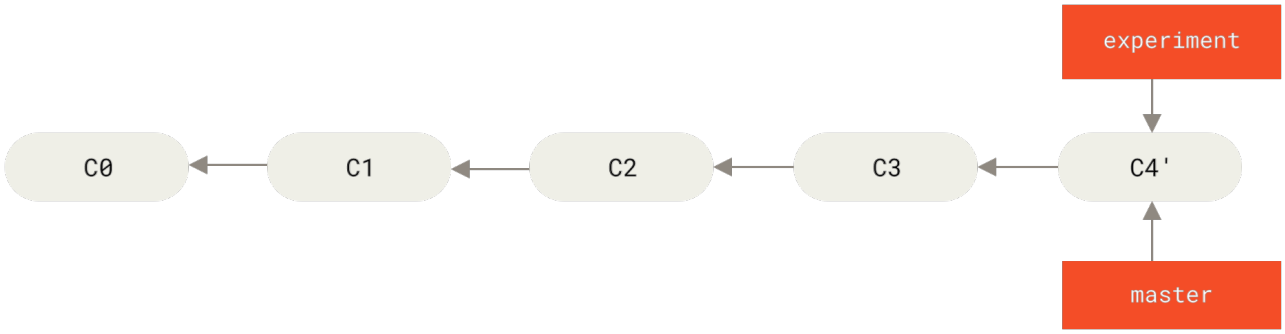


Figure 38. Sürətli irəliləyən `master` branch-ı

İndi `C4`-ün işarələdiyi snapshot <the merge example-də `C5` ilə işarələnmiş snapshotla eynidir. İnteqrasiyanın son məhsulu arasında heç bir fərq yoxdur, lakin rebasing daha təmiz bir tarix yaradır. Rebase edilmiş bir branch-ın log-larını araşdırsanız, xətti bir tarixə bənzəyir: görünür ki, bütün işlər əvvəlcə paralel olaraq baş versə də, bütün işlərin ardıcıllıqla baş verdiyi anlaşılır.

Tez-tez verdiyiniz tapşırıqların uzaq bir branch-a təmiz tətbiq olunmasını təmin etmək üçün bunu edəcəksiniz - töhfə verməyə çalışdığınız, ancaq qorumadığınız bir layihədə. Bu vəziyyətdə işinizi bir branch-da edər və sonra patch-larını əsas layihəyə təqdim etməyə hazır olduğunuz zaman işinizi `origin/master`-ə ya dəyişdirərdiniz. Bu yolla, qoruyucu heç bir inteqrasiya işi görməməlidir - sadəcə irəli və ya təmiz tətbiq olunmalıdır.

Diqqət yetirin ki, başa çatdığınız son commit-in göstərildiyi snapshot, istər bir düzəltmə əmrinin sonuncusu olsun, istərsə birləşmədən sonra edilən son birləşmə, eyni Rebasing dəyişikliklərini bir iş xəttindən digərinə təqdim edilmiş qaydada dəyişdirir, birləşmə isə son nöqtələri götürərək onları birləşdirir.

Daha Maraqlı Rebase-lər

Ayrıca, rebase hədəf branch-ında başqa bir şeydə yenidən istifadə edə bilərsiniz. Məsələn, Başqa bir mövzu branch-ından bir mövzu branch-a olan bir tarix kimi bir tarix çəkin. Layihəinizə bəzi server tərəfi funksionallıq əlavə etmək üçün bir mövzu branch-nı (`server`) əlavə etdiniz və commit yaratdınız. Sonra müştəri tərəfində dəyişiklik etmək üçün (`client`) branch-ı düzəldin və bir neçə d əfə commit edin. Nəhayət, yenidən server branch-ınıza qayıtdınız və daha bir neçə commit etdiniz.

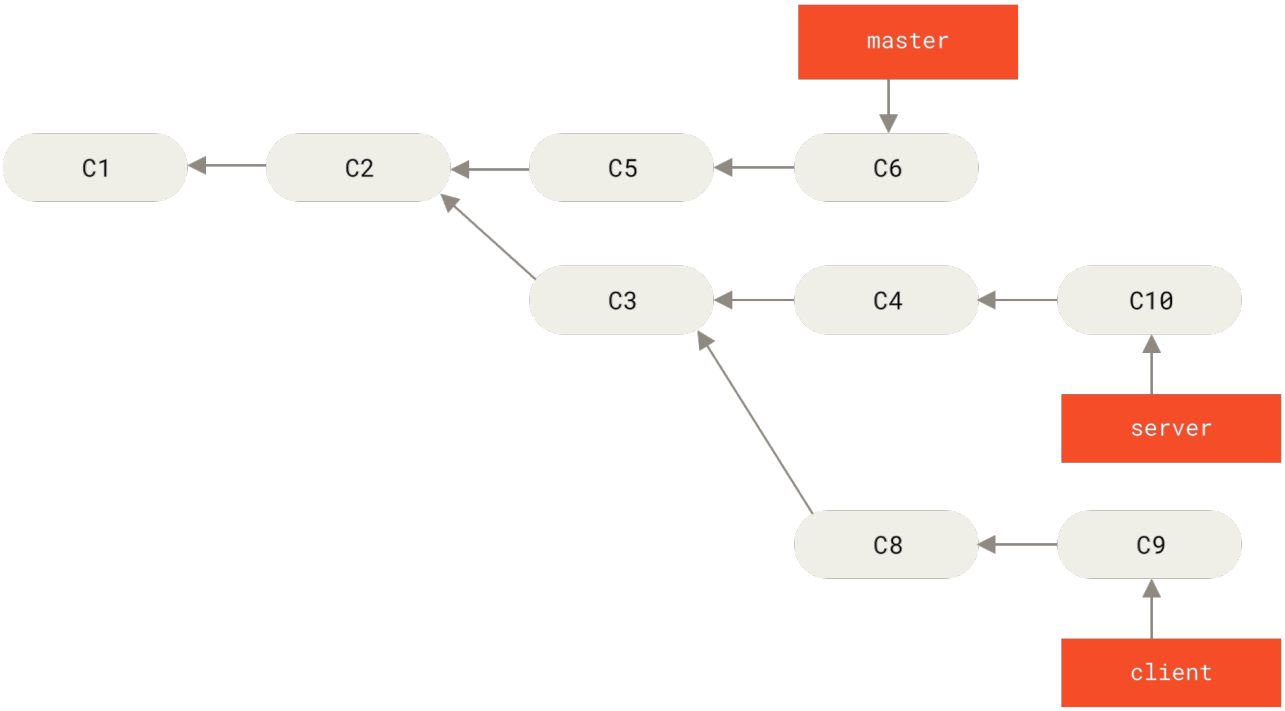


Figure 39. Başqa bir mövzu branch-ından bir mövzu branch-a olan bir tarix

Tutaq ki, bir müştəri tərəfindəki dəyişiklikləri sərbəst buraxmaq üçün ana xəttinizə birləşdirmək istədiyinizə qərar verirsiniz, ancaq daha sonra sınaqdan keçirilməyincə server tərəfindəki d əyişiklikləri dayandırmaq istəyərsiniz. `server`-də (` C8` və `C9`) olmayan `client`-dəki dəyişiklikləri götürüb `git rebase`-in `--onto` seçimini istifadə edərək `master` branch-ında təkrarlaya bilərsiniz:

```
$ git rebase --onto master server client
```

Bu, əsasən ‘` client' branch-nı götürün, `server` branch-ından ayrıldığından patch-ları müəyyənl əşdirin və bu patch-ları `client` branch-ında birbaşa `master` branch-ından kənarda qurulmuş kimi t əkrarlayın . ” Biraz mürəkkəb olsa da, amma nəticə olduqca gözəldir.

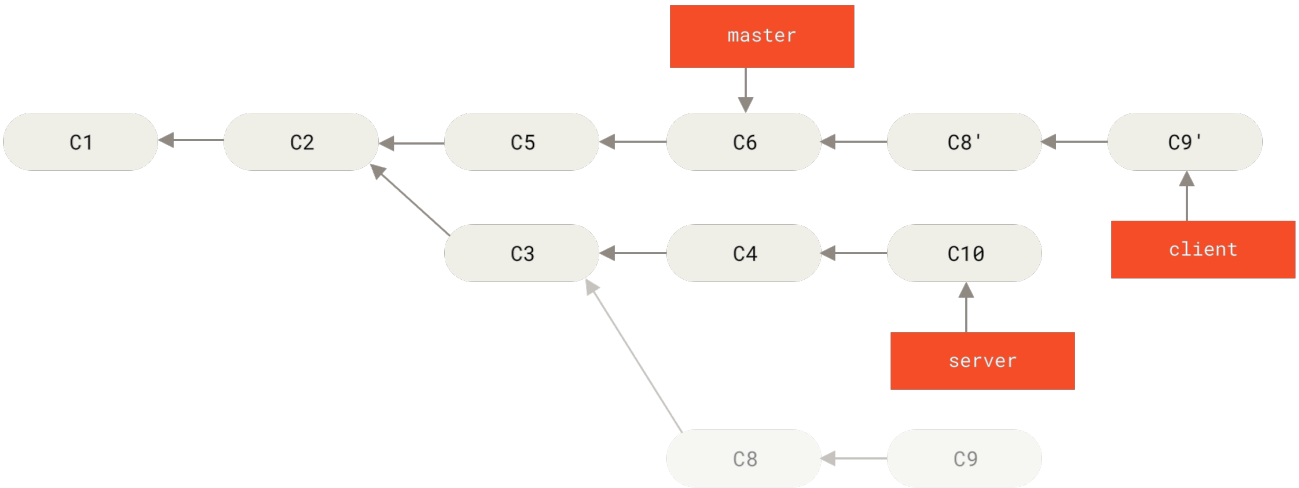


Figure 40. Bir mövzu branch-nı başqa bir mövzu branch-ından rebasing etmək

İndi `master` branch-ınızı sürətlə irəliləyə apara bilərsiniz (Client branch-ındakı dəyişiklikləri daxil etmək üçün `master` branch-ınızı sürətli yönləndirin-ə bax):

```
$ git checkout master
$ git merge client
```

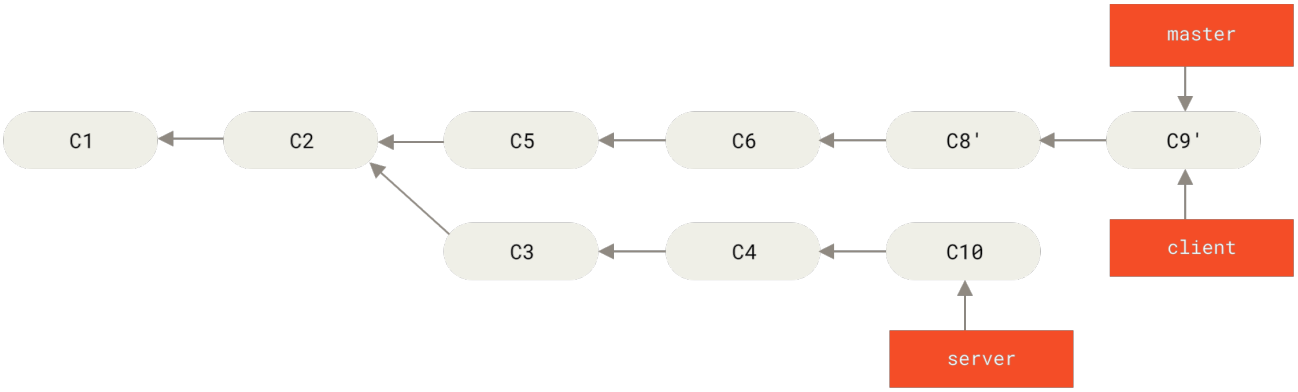


Figure 41. Client branch-ındakı dəyişiklikləri daxil etmək üçün `master` branch-ınızı sürətli yönləndirin

Deyək ki, həm də server branch-nızı pull etməyə qərar verdiniz. `git rebase <basebranch> <topicbranch>`-ı işə salmaqdan əvvəl `server` branch-ını `master` branch -a rebase edə bilərsiniz. Mövzu branch-nı yoxlayır (bu vəziyyətdə, `server`) və əsas branch-a (`master`) qaytarır.

```
$ git rebase master server
```

Bu, `server` işinizi `master` işinizin üstündə, Server branch-nızı `master` branch-nızın üstünə rebasing etmək-da göstərildiyi kimi təkrarlayır.

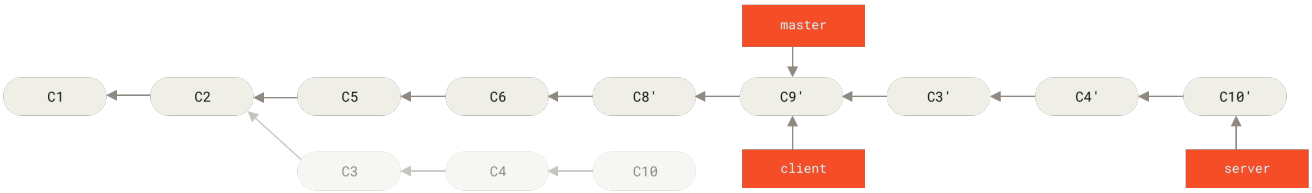


Figure 42. Server branch-nızı `master` branch-nızın üstünə rebasing etmək

Sonra əsas branch-ı (`master`) sürətlə irəli sürə bilərsiniz:

```
$ git checkout master
$ git merge server
```

Bütün işlər inteqrasiya olunduğuna görə, `client` və `server` branch-larını silə bilərsiniz, bu da bütün bu proses üçün tarixinizi Son commit tarixi kimi qoyur:

```
$ git branch -d client
$ git branch -d server
```

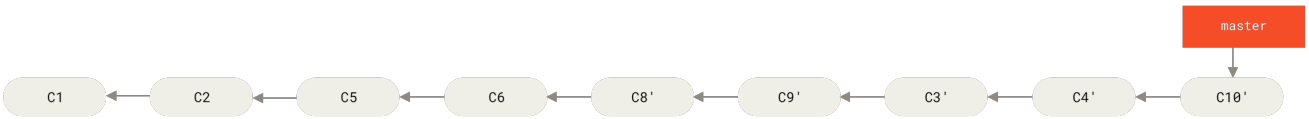


Figure 43. Son commit tarixi

Rebasing-in Təhlükələri

Ahh, lakin çatışmazlıqlar olmadan rebasing-in zövqü olmaz,hansı ki onları bir sətirdə yekunlaşdırmaq olar.

Depolarınızdan kənarda mövcud olan və insanların üzərində işləyə bildikləri commitləri rebase etməyin.

Bu qaydaya əməl etsəniz, yaxşı olacaqsınız. Bunu etməsəniz, insanlar sizə nifrət edəcəklər və dostlarınız və ailəniz tərəfindən rüsvay olacaqsınız.

Əşyaları rebase etdiyinizdə mövcud commit-lərdən imtina edirsiniz və oxşar, lakin fərqli olanları yaradırsınız. Əgər commit-ləri bir yerə push etsəniz və başqaları onları aşağı pull edib üzərində işləyirsə, sonra `git rebase` ilə yenidən yazıb yenidən push etsəniz, iş yoldaşların işlərini yenidən birləşdirməyə məcbur olacaqlar və işləriniz qarışıq olacaq. Ona görə işlərini özünüzə qaytarmağa çalışın.

Öublic etdiyiniz işin rebasing olunanda problem yarada biləcəyinə dair bir nümunəyə baxaq. Tutaq ki, mərkəzi bir serverdən klonlaşdırırsınız və daha sonra bir az iş görürsünüz.

Commit tarixçəniz belə görünür:

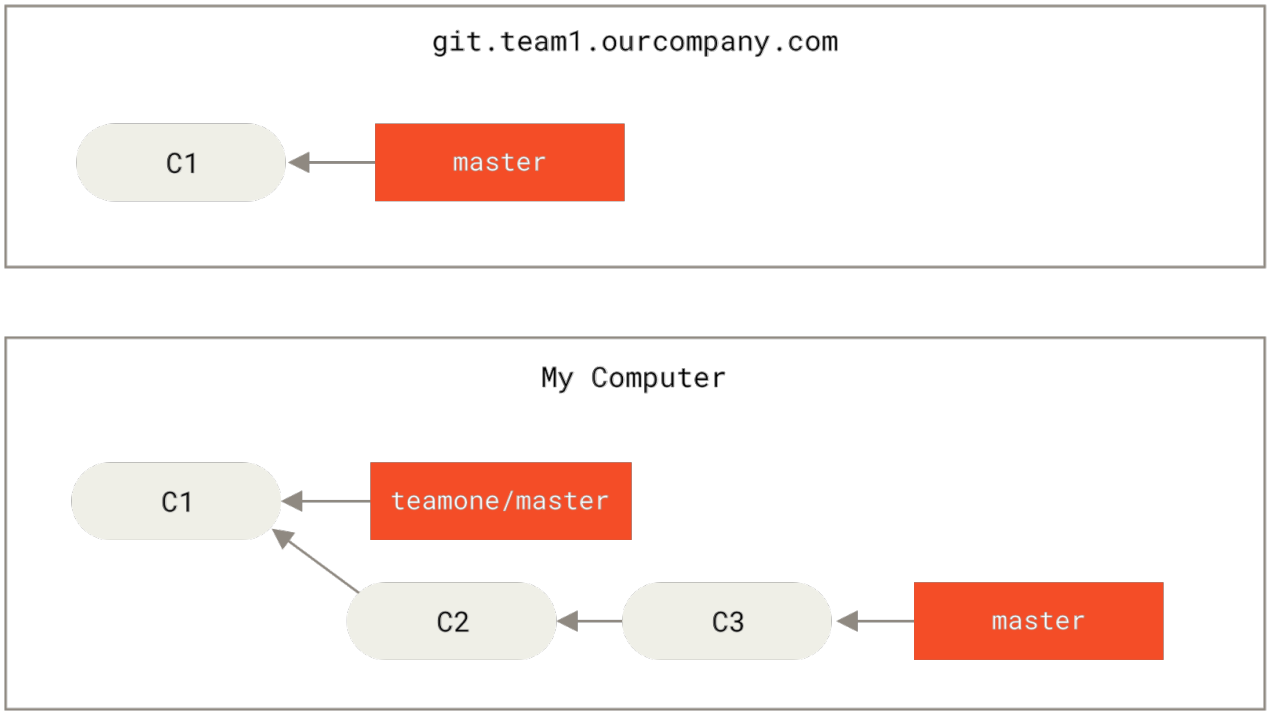


Figure 44. Deponu klonlaşdırın və üzərində bir az iş aparın

İndi başqası birləşməyi əhatə edən daha çox iş görür və mərkəzi serverə işləyir. Siz götürün və

yeni uzaq branch-ı işinizə birləşdirərək tarixinizi bu kimi bir hala gətirin:

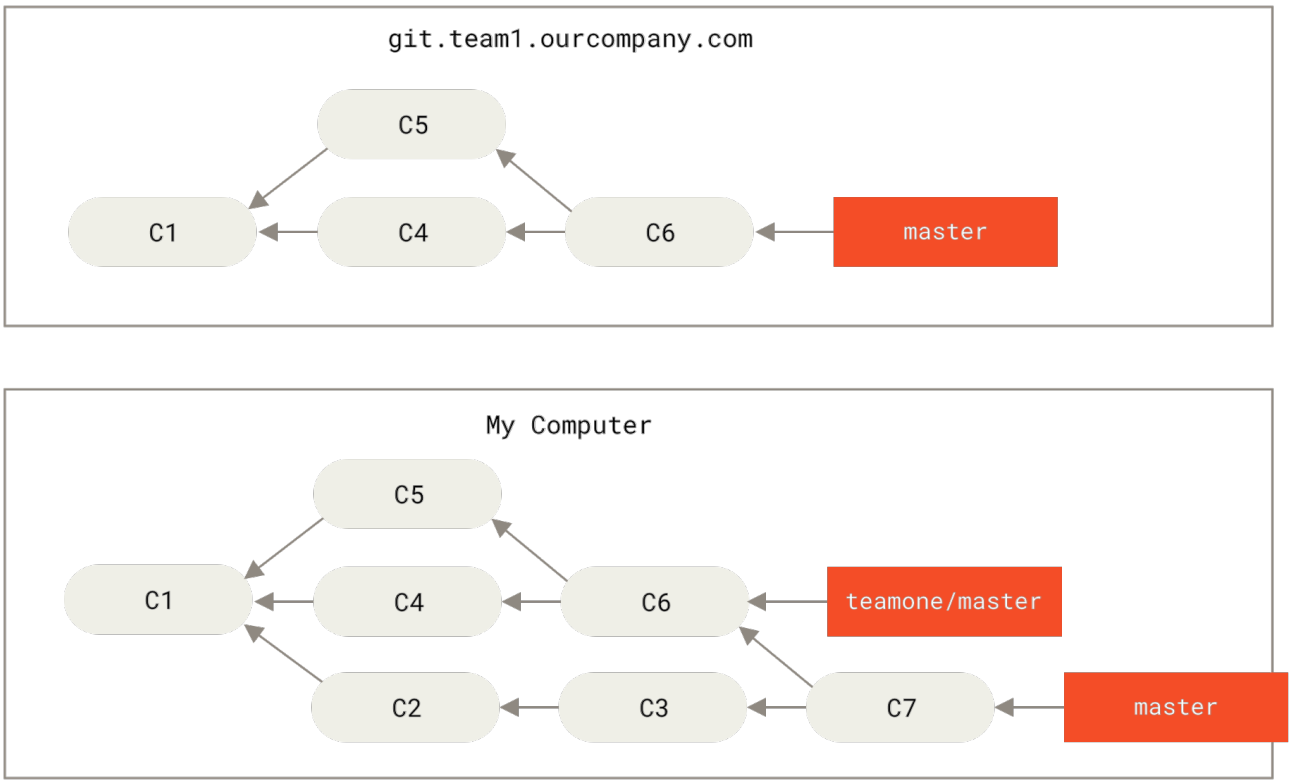


Figure 45. Daha çox commit fetch edin və onları işinizə birləşdirin

Sonra birləşən işi push edən şəxs geri qayıtmağı və işlərini dəyişdirməyi qərara alır; serverdəki tarixi yenidən yazmaq üçün `git push --force` tətbiq edirlər. Daha sonra yeni commit-ləri gətirərək həmin serverdən alırsınız.

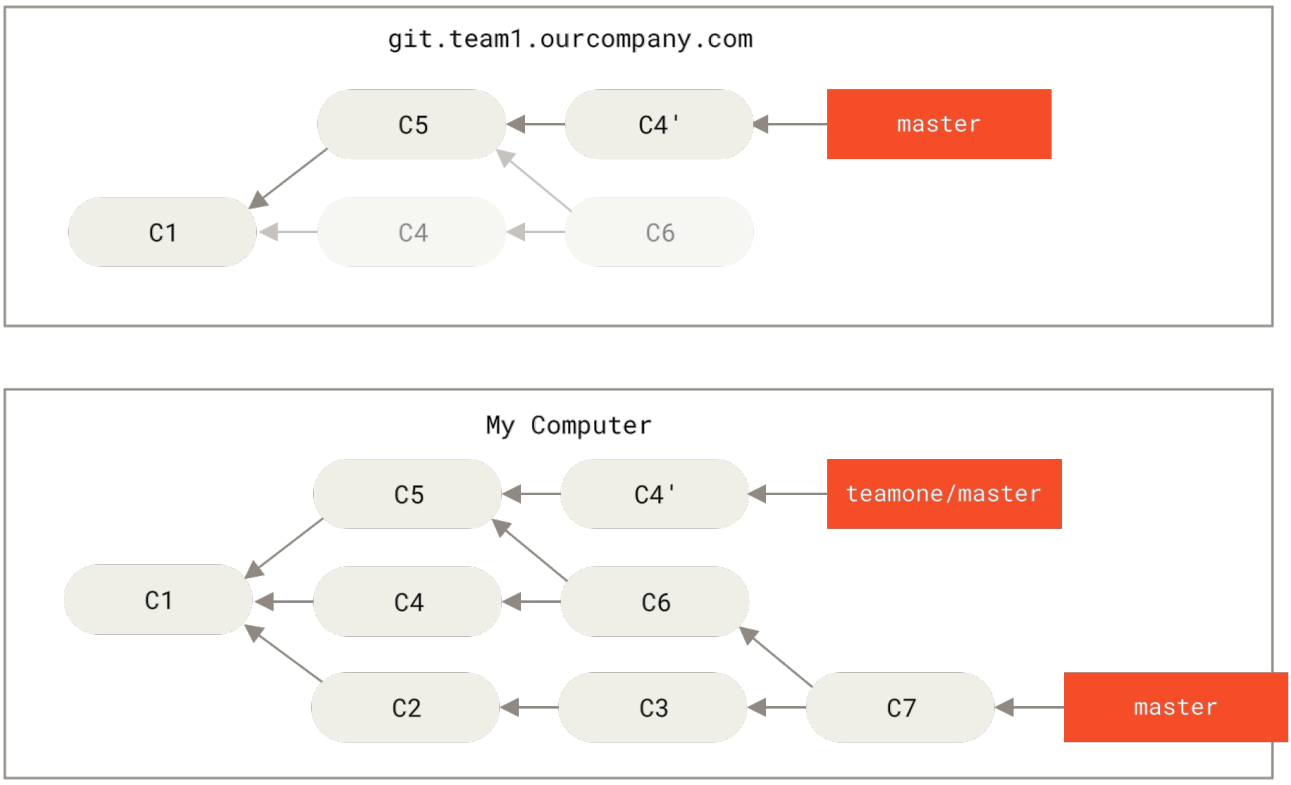


Figure 46. Kimsə işinizə əsaslanaraq verdiyiniz commit-ləri tərk edərək, rebase edilmiş commit-ləri push edir

İndi ikiniz də turşu içindəsiniz. Bir `git pull` etsəniz, tarixin hər iki xəttini özündə birləşdirən birl əşmə əməliyyatı yaradacaqsınız və depo bu cür görünəcək:

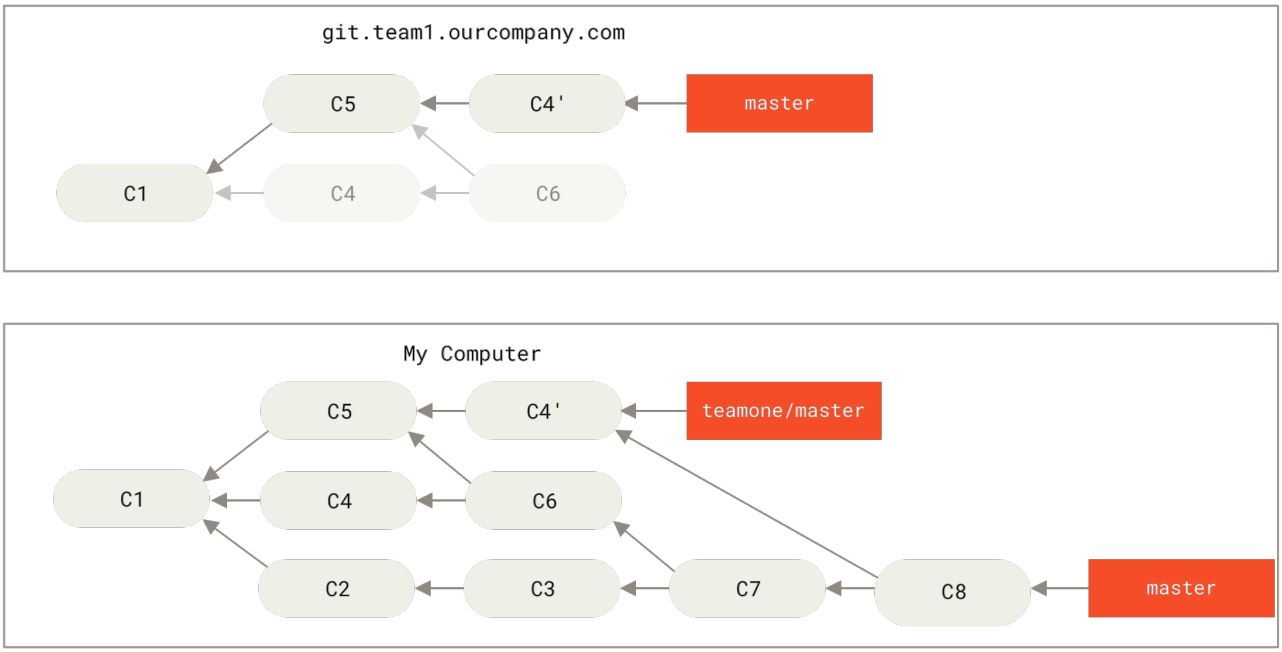


Figure 47. Eyni işdə yenidən yeni birləşmə commit-nə qoşulursunuz

Tarixiniz bu kimi görünəndə `git log` işlədirsinizsə, qarışıqlıq yaradan eyni müəllif, tarix və mesajı olan iki commit görürsünüz. Bundan əlavə, bu tarixi yenidən serverə push etsəniz, insanları qarışdıra biləcək bütün əvəz edilmiş sənədləri mərkəzi serverə yenidən təqdim etmiş olacaqsınız.

Digər developerin `C4` və `C6`-ların tarixdə olmasını istəmədiyini güman etmək olduqca təhlükəsizdir; buna görə ilk növbədə yenidən çap etdilər.

Rebase etdiyiniz zaman yenidən yazın

Əgər belə bir vəziyyətdə özünüzü taparsanız, Git sizə kömək edə biləcək daha bir sehrə sahibdir. Əgər komandanızdakı kimsə işə əsaslandığınız işin üzərində yazılan dəyişiklikləri push edirsə, problem sizin kim olduğunuzu və yenidən yazdıqlarını anlamaqdır.

Məlum olub ki, SHA-1 yoxlama cədvəlinə əlavə olaraq, Git yalnız commit ilə təqdim olunan patch-a əsaslanan bir çek məbləğini də hesablayır. Buna “patch-id” deyilir.

Yenidən yazılmış işi pull doün etsəniz və tərəfdaşınızdan aldığınız yeni işin üstünə yazsanız, Git tez-tez misilsiz sizin nəyi başa düşdüyünüzü yeni branch-ın üstünə tətbiq edə bilər.

Məsələn, əvvəlki ssenaridə, əgər Kimsə işinizə əsaslanaraq verdiyiniz commit-ləri tərk edərək, rebase edilmiş commit-ləri push edir olduğumuz zaman birləşmə yerinə `git rebase teamone/master` işlədiriksə, Git:

- Branch-ımız üçün hansı işin özünəməxsus olduğunu müəyyənləşdirin (C2, C3, C4, C6, C7)
- Hansıların birləşmədiyini təyin edin (C2, C3, C4)
- Hədəf branch-ına yenidən yazılmayanları təyin edin (C4 C4 'ilə eyni patch olduğundan)
- Bu commit-ləri `teamone/master` başına tətbiq edin

Beləliklə, Eyni işdə yenidən yeni birləşmə commit-nə qoşulursunuz-də gördüyümüz nəticənin əv əzinə daha çox Force-pushed rebase işə yenidən başlayın kimi bir şeylə nəticələnərdik.

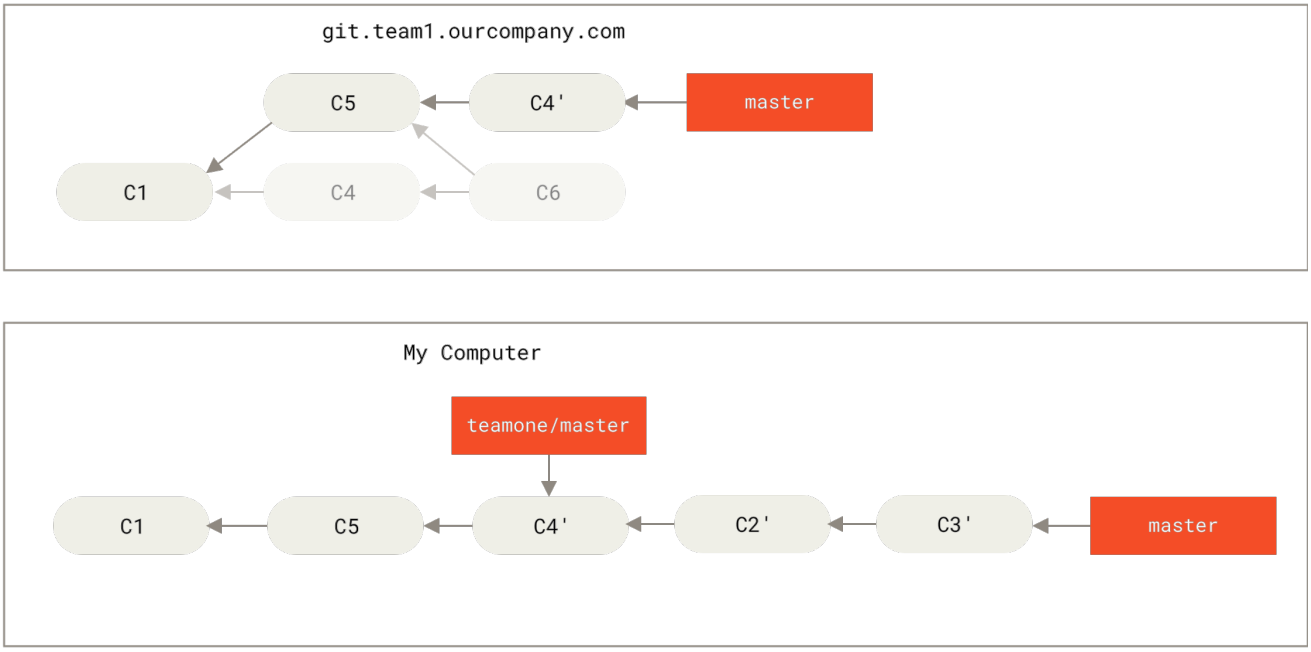


Figure 48. Force-pushed rebase işə yenidən başlayın

Bu yalnız ortağınızın hazırladığı `C4` və `C4'` demək olar ki, eyni bir patch olduqda işləyir. Əks tə qdirdə, yenidən yükləmə bunun bir dublikat olduğunu söyləyə bilməyəcək və başqa bir C4-ə bənz

ər bir yamaq əlavə edəcəkdir (ehtimal ki, təmiz tətbiq olunmayacaq, çünki dəyişikliklər heç olmasa orada olacaq).

Normal bir `git pull` yerinə `git pull --rebase` işlətməklə bunu asanlaşdıra bilərsiniz. Və ya bu v əziyyətdə `git rebase teamone/master` ardınca `git fetch` ilə manual edə bilərsiniz.

`git pull` istifadə edirsinizsə və `--rebase` standart etmək istəyirsinizsə, `git config --global pull.rebase true` kimi bir şey ilə `pull.rebase` konfiqurasiya dəyərini təyin edə bilərsiniz.

Heç vaxt öz kompüterinizi tərk etməyən commit-ləri yenidən yerinə yetirsəniz, yaxşı olacaqsınız. Əgər push etdiyiniz commit-ləri geri qaytarsanız, lakin başqa heç kim əsas götürməmişsə, h əmçinin yaxşı olacaqsınız. Əgər siz əvvəlcədən ictimailəşdirilmiş commit-ləri geri qaytarsanız və insanlar bu commit-ləri əsas götürsələr, o zaman bəzi narahat problemlərə və komanda yoldaşlarınızın qınağına düşə bilərsiniz.

Əgər siz və ya tərəfdaş bir anda zəruri hesab edirsinizsə,hər kəsin `git pull --rebase` işlətdiyini bildiyinizdən əmin olun ki, bu da bir az daha sadə olur.

Rebase vs Birləşdirmək

İndi siz rebasing və birləşdirmə hərəkətlərini gördünüz, hansının daha yaxşı olduğunu düşünə bil ərsiniz. Buna cavab verməzdən əvvəl bir az geri çəkilib tarixin nə demək olduğunu danışaq.

Bu baxımdan bir məqam odur ki, depozit tarixçənizin tarixçəsi əslində **baş verənlərin qeydidir**. Tarixi bir sənəddir, öz dəyərindədir və dəyişdirilməməlidir. Bu baxımdan, commit tarixinin d əyişdirilməsi demək olar ki, küfrdür; əslində nəyi ötürdüyünüz barədə *lying* danışırsınız. Birləşm əyin qarışıq sıra seriyası varsa nə olacaq? Bu belə oldu və depoları bunu sonrakı nəsillər üçün saxlamalıdır.

Qarşı nöqtə, commit tarixinin olmasıdır, yəni **layihənizin necə edildiyi haqqında hekayədir**. Bir kitabın ilk layihəsini yayımlamazsınız və proqramınızı necə qoruyacağınıza dair təlimat diqqətlə redaktəyə etməlisiniz. Bu, hekayəni gələcək oxuculara ən yaxşı şəkildə izah etmək üçün `rebase` və `filter-branch` kimi vasitələrdən istifadə edən düşərgədir.

İndi gələk birləşməyin və ya rebasing etməyin daha yaxşı olması sualına: ümid edirik bunun asan olmadığını görərsiniz. Git güclü bir vasitədir və tarixinizlə birlikdə çox şey etməyə imkan verir, lakin hər komanda və hər layihə fərqlidir.

Artıq hər ikisinin necə işlədiyini bildiyinizdən, hansınızın vəziyyətiniz üçün ən uyğun olduğunu q ərar verməyiniz sizə bağlıdır.

Ümumiyyətlə, hər iki dünyanın ən yaxşısını qazanmağın yolu hekayənizi təmizləmək üçün onları təkzib etmədən əvvəl etdiyiniz, lakin hələ paylaşmadığınız yerli dəyişiklikləri geri qaytarmaqdır.

Qısa Məzmun

Git'də əsas branching və birləşmə barədə məlumat verdik. Yeni branch'lar yaratmaq və yeni branch-lara keçid etmək, branch'lar arasında keçid etmək və lokal branch'ları birləşdirmək üçün rahat hiss etməlisiniz. Ayrıca branch'larınızı paylaşılan bir serverə push edərək, başqaları ilə paylaşılan branch'larda işləyərək və bölmələrinizi bölüşmədən əvvəl bərpa edərək bölüşməyi

bacarmalısınız. Bundan sonra, öz Git deposu yerləşdirmə serverinizi idarə etmək üçün lazım olanları nəzərdən keçirəcəyik.

Server'də Git

Bu nöqtədə Git'i istifadə edəcəyiniz gündəlik vəzifələrin çoxunu edə bilməlisiniz. Bununla birlikdə, Git'də hər hansı bir əməkdaşlıq etmək üçün remote bir Git deposuna sahib olmalısınız. Dəyişiklikl əri texniki cəhətdən push edə və fərdi depolardakı dəyişiklikləri pull edə bilsəniz də, diqqətli olmadığınız halda üzərində işlədiklərini kifayət qədər asanlıqla qarışdıra biləcəyiniz üçün bunu etmək məsləhət görülmür. Bundan əlavə, kompüteriniz oflayn olsa belə, əməkdaşlarınızın depoya daxil ola bilməsini istəyirsiniz - daha etibarlı ümumi bir deponun olması çox vaxt faydalıdır. Bu sə bəbdən kimsə ilə əməkdaşlıq etmək üçün üstünlük verilən metod, hər ikinizin daxil ola biləcəyiniz bir aralıq depo qurmaq və buradan pull və push etməkdir.

Bir Git serverini idarə etmək olduqca sadədir. Əvvəlcə serverinizin hansı protokolları dəstəkləm əsini istədiyinizi seçin. Bu fəslin birinci hissəsi mövcud protokolları və hər birinin müsbət və mənfi tərəflərini əhatə edəcəkdir. Növbəti hissələrdə bu protokollardan istifadə edərək bəzi tipik quraşdırma işləri və serverinizin onlarla necə işləməsini izah edəcəyik. Son olaraq kodunuzu başqasının serverində yerləşdirməyinizə qarşı çıxmasanız və öz serverinizi qurmaq və saxlamaq çətinliyindən keçmək istəmirsinizsə, bir neçə yerləşdirilmiş seçimdən keçəcəyik.

Öz serverinizi idarə etməklə maraqlanmırsınızsa, yerləşdirilmiş bir hesab qurmaq üçün bəzi variantları görmək üçün bölmənin son hissəsinə keçib paylanmış mənbə nəzarəti mühitində işin müxtəlif müsbət və mənfi tərəflərini müzakirə etdiyimiz növbəti hissəyə keçə bilərsiniz.

Remote bir depo ümumiyyətlə bir işləmə qovluğu olmayan bir Git deposu olan bir *bare deposu*-dur. Depo yalnız bir işləmə nöqtəsi olaraq istifadə olunduğundan, bir snapshot-u diskdə yoxlamaq üçün heç bir səbəb yoxdur; yalnız Git məlumatlarıdır. Ən sadə dildə desək, bare bir depo, layihənizin `.git` qovluğunun məzmunudur və başqa heç nə deyildir.

Protokollar

Git məlumat ötürmək üçün dörd fərqli protokoldan istifadə edə bilər: Local, HTTP, Secure Shell (SSH) və Git. Burada bunların nə olduğunu və hansı əsas şərtlərdə istifadə etməyinizi (və ya istəm ədiyinizi) müzakirə edəcəyik.

Local Protokol

Ən əsası uzaq depo ilə eyni hostdakı başqa bir qovluqda olan *Local protocol* -dur. Bu, komandanızdakı hər kəsin bir NFS quraşdırılmış bir fayl sisteminə çıxışı olduqda və ya hamının eyni kompüterə girməsi ehtimalı az olduqda istifadə olunur. Sonuncu ideal olmazdı, çünki bütün kod depo instansiyaları eyni kompüterdə yerləşəcək və fəlakətli itkini daha çox edə bilər.

Birgə quraşdırılmış bir fayl sisteminiz varsa, local bir sənəd əsaslı depo ilə klonlaşa, pus və pull edə bilərsiniz. Bu kimi bir depo klonlaşdırmaq və ya mövcud bir layihəyə uzaqdan bir əlavə etmək üçün URL olaraq depo path-dan istifadə edin. Məsələn, local bir depo klonlaşdırmaq üçün belə bir şey işlədə bilərsiniz:

```
$ git clone /srv/git/project.git
```

Or you can do this:

```
$ git clone file:///srv/git/project.git
```

URL'in əvvəlində `file://`-nı dəqiq göstərsəniz, Git bir qədər fərqli işləyər. Yalnız path-ı göstərsəniz, Git sərt keçidlərdən istifadə etməyə və ya lazım olan sənədləri birbaşa kopyalamağa çalışar. `file://` yazsanız, Git ümumiyyətlə çox az səmərəli olan bir şəbəkə üzərindən məlumat ötürmək üçün istifadə etdiyi prosesləri başladar. `file://` prefiksinin göstərilməsinin əsas səbəbi, başqa bir VNS və ya buna bənzər bir şeyin idxalından sonra deponun lazımsız istinadlar və ya obyektlər xaric təmiz kopyasını istəməyinizdir.(Baxım işləri üçün Git'in Daxili İşləri-a baxın).

Burada normal yoldan istifadə edəcəyik, çünki bunu etmək demək olar ki, həmişə daha sürətli olur.

Mövcud Git layihəsinə local bir depo əlavə etmək üçün belə bir şey işlədə bilərsiniz:

```
$ git remote add local_proj /srv/git/project.git
```

Daha sonra yeni bir uzaq adınızı `local_proj` vasitəsilə uzaqdan push və pull edə bilərsiniz.

Üstünlüklər

Fayl əsaslı depoların üstünlükləri sadə olduqları və mövcud fayl icazələrindən və şəbəkə girişl ərindən istifadə etmələridir. Bütün qrupunuzun daxil olduğu ortaq bir fayl sisteminiz varsa, depo qurmaq çox asandır. Çılpaq depo nüsxəsini hər kəsin paylaşdığı bir yerə yapışdırırsınız və hər hansı digər paylaşılan qovluq üçün olduğu kimi oxumaq/yazma icazələrini təyin edirsiniz. Bunun üçün Serverdə Git Əldə Etmək-də çılpaq bir depo kopyasını necə ixrac edəcəyimizi müzakirə edəc əyik.

Başqasının işlədiyi depo-dan tez bir zamanda iş aparmaq üçün bu da əlverişli bir seçimdir. Bir iş yoldaşınızla eyni layihə üzərində işləsəniz və bir şeyin yoxlanılmasını istəsəniz, `git pull /home/john/project` kimi bir əmr işlətmək uzaq bir serverə pushing edib və sonradan fetching etm əkdən daha asandır.

Çatızmazlıqları

Bu metodun mənfi cəhətləri budur ki, paylaşılan girişin qurulmasının və birdən çox yerdən əsas ş əbəkə girişinin ümumiyyətlə daha çətin olmasıdır. Evdə olduğunuz zaman dizüstü kompüterinizd ən push etmək istəsəniz, şəbəkə əsaslı girişlə müqayisədə çətin və yavaş ola bilən uzaq disk quraşdırmalısınız.

Bir növ ortaq bir montajdan istifadə edirsinizsə, bu mütləq ən sürətli seçim olmadığını qeyd etmək vacibdir. Local bir depo yalnız məlumatlara sürətli çıxışınız olduqda sürətli olur. NFS-də bir depo eyni serverdəki SSH üzərindəki depozitdən daha yavaş olur, bu da Git-in hər sistemdəki yerli diskl əri işə salmasına imkan verir.

Nəhayət, bu protokol depo-nu təsadüfi zərərdən qoruyur. Hər bir istifadəçinin "uzaqdan" qovluğuna tam shell çıxışı var və onların daxili Git fayllarını dəyişdirməsinə və ya silinməsinə və depo-nun korlanmasına mane olan heç bir şey yoxdur.

HTTP Protokolları

Git iki fərqli rejimi istifadə edərək HTTP ilə əlaqə qura bilər. Git 1.6.6-dan əvvəl bunun sadə və ümumiyyətlə sadəcə oxumaq üçün edə biləcəyi bir yolu var idi. 1.6.6 versiyasında Git-in SSH üz ərində olduğu kimi bir şəkildə məlumat ötürmə danışıqlarını daha ağıllıca edə bilməsi ilə əlaqəli yeni, daha asan və daha ağıllı bir protokol təqdim edildi. Son bir neçə ildə bu yeni HTTP protokolu istifadəçi və ünsiyyət qurma qaydaları haqqında daha sadə və ağıllı olduğu üçün çox məşhur oldu. Daha yeni versiyaya ümumiyyətlə *Smart* HTTP protokolu ve daha köhnə olana *Dumb* HTTP denir. Əvvəlcə daha yeni Smart HTTP protokolunu incəliyəcəyik.

Smart HTTP

Smart HTTP, SSH və ya Git protokollarına bənzər şəkildə işləyir, lakin standart HTTPS portları üz ərində işləyir və müxtəlif HTTP identifikasiya mexanizmlərindən istifadə edə bilir. Bu o deməkdir ki, SSH keys-i quraşdırmaq yerinə istifadəçi adı/parol identifikasiyası kimi şeylərdən istifadə edə biləcəyiniz üçün istifadəçi üçün SSH kimi bir şeydən daha asan olduğu anlamına gəlir.

Yəqin ki, Git istifadə etmək üçün ən populyar bir yol halına gəlmişdir, çünki həm anonim olaraq `git://` protokolu kimi xidmət edəcək şəkildə istifadə edilə bilər, həm də SSH protokolu kimi identifikasiya və şifrələmə ilə ötürülə bilər. Bu şeylər üçün fərqli URL-lər qurmaq əvəzinə indi hər ikisi üçün tək bir URL istifadə edə bilərsiniz. Əgər push etməyə cəhd etsəniz və depo doğrulamasını (adətən olmalıdır) tələb edirsə, server istifadəçi adı və şifrə tələb edə bilər. Oxuma girişi üçün də eyni şey keçərlidir.

Əslində, GitHub kimi xidmətlər üçün depo-nu onlayn olaraq görüntüləmək üçün istifadə etdiyiniz URL (məsələn, https://github.com/schacon/simplegit) klonlaşdırmaq üçün istifadə edə biləcəyiniz URL-dir və əgər daxil ola bilirsinizsə, push over edin.

Dumb HTTP

Server Git HTTP ağıllı xidməti ilə cavab vermirsə, Git müştəri daha sadə *Dumb* HTTP protokoluna geri dönməyə çalışacaq. Dumb protokolu çılpaq Git depolarının veb serverdən normal fayllar kimi təqdim olunmasını gözləyir. Dumb HTTP-nin gözəlliyi onu qurmağın sadəliyidir. Əsasən, HTTP sə nəd root-nuzun altına çılpaq Git depoziti qoymalı və konkret bir `post-update` hook qurmağınız kifayətdir(Git Hook'ları-a baxın). Bu zaman depo qoyduğunuz veb serverə daxil ola bilən hər kəs deponuzu klonlaya bilər. HTTP üzərindəki depo yerinə oxunuşa icazə vermək üçün belə bir şey edin:

```
$ cd /var/www/htdocs/
$ git clone --bare /path/to/git_project gitproject.git
$ cd gitproject.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

Bu qədər. Default olaraq Git ilə birlikdə gələn `post-update` hook, HTTP-nin alınması və klonlanmasının düzgün qurulması üçün müvafiq əmr (`git update-server-info`) işlədir. Bu əmr bu deponu (SSH-dən çox) push etdikdə yerinə yetirilir; sonra digər insanlar da klonlaya bilər:

```
$ git clone https://example.com/gitproject.git
```

Bu vəziyyətdə, Apache tənzimləmələri üçün ümumi olan `/var/www/htdocs` yolunu istifadə edirik, ancaq hər hansı bir statik veb serverdən istifadə edə bilərsiniz - bu path-a sadəcə çılpaq depoları qoyun. Git məlumatları əsas statik sənədlər kimi təqdim olunur (xidmətin necə aparıldığı barədə ətraflı məlumat üçün Git'in Daxili İşləri-a baxın).

Ümumiyyətlə, Smart HTTP serverini oxumaq/yazmaq və ya sadəcə Dumb şəkildə sadəcə oxunan kimi əlçatan faylları seçməyi seçərdiniz. İki xidmətin qarışığını işlətmək nadirdir.

Üstünlüklər

HTTP protokolunun Smart versiyasının üstünlüklərini cəmləşdirəcəyik.

Hər cür giriş üçün vahid bir URL-yə sahib olmağın və serverinin yalnız identifikasiya lazım olduqda istəməsinin olması sadəliyi son istifadəçi üçün işləri çox asanlaşdırır. Bir istifadəçi adı və şifrə ilə identifikasiya edə bilməyiniz SSH-a da böyük bir üstünlükdür, çünki istifadəçilər local SSH key-lərini yarada və açıq key-ləri serverə yükləmək lazım deyil. Daha az inkişaf etmiş istifadəçilər və ya SSH-nin daha az yayıldığı sistemlərdəki istifadəçilər üçün bu əsas üstünlükdür. Həm də SSH protokoluna bənzər çox sürətli və səmərəli bir protokoldur.

Ayrıca depolarınıza HTTPS üzərində yalnız oxuya bilərsiniz, yəni məzmun ötürməsini şifrələyə bil ərsiniz; və ya müştərilərin xüsusi imzalı SSL sertifikatlarından istifadə etmələri üçün bu qədər ir əliyə gedə bilərsiniz.

Başqa bir xoş bir şey, HTTP və HTTPS bu qədər istifadə olunan protokollardır ki, korporativ firewal-lar tez-tez portları üzərindən trafikə icazə vermək üçün qurulur.

Çatızmazlıqları

HTTPS üzərindən Git bəzi serverlərdə SSH ilə müqayisədə bir az daha çətin ola bilər. Bundan başqa, digər protokolların Git məzmununa xidmət üçün Smart HTTP-ə görə üstünlüyü çox azdır.

Kimliği təsdiq edilmiş pushing üçün HTTP istifadə edirsinizsə, kimlik məlumatlarınızı təmin etmək bəzən SSH üzərindəki key-lərədən istifadə etməkdən daha mürəkkəbdir. Bununla birlikdə, bu, MacOS-da Keychain access-i və Windows-da Credential Manager kimi olduqca ağrısız hala gətirmək üçün istifadə edə biləcəyiniz bir neçə credential caching vasitəsi var. Sisteminizdə təhlükəsiz HTTP şifrələməsini necə qurulacağını görmək üçün Etibarlı Yaddaş oxuyun.

SSH Protokolu

Self-hosting SSH üzərində olanda Git üçün ümumi transport protokolu. Bunun səbəbi, serverlərə SSH girişi artıq əksər yerlərdə qurulmuşdur - əgər olmursa, bunu etmək asandır. SSH eyni zamanda təsdiq edilmiş bir şəbəkə protokoludur və local olduğundan ümumiyyətlə qurmaq və istifadə etmək asandır.

Bir Git depozitini SSH üzərində klonlaşdırmaq üçün `ssh:// URL-i kimi göstərə bilərsiniz:

```
$ git clone ssh://[user@]server/project.git
```

Və ya SSH protokolu üçün daha qısa scp kimi sintaksisdən istifadə edə bilərsiniz:

```
$ git clone [user@]server:project.git
```

Yuxarıdakı hər iki halda əlavə istifadəçi adını göstərməmisinizsə, Git hazırda daxil olduğunuz istifadəçini qəbul edəcəkdir.

Üstünlükləri

SSH istifadə üstünlükləri çoxdur. Birincisi, SSH qurmaq nisbətən asandır - SSH daemonları adi haldır, bir çox şəbəkə rəhbərləri onlarla təcrübəyə malikdirlər və bir çox OS paylamaları onlarla qurulur və ya onları idarə etmək üçün vasitələr var. Sonra SSH üzərindən giriş etibarlıdır - bütün məlumat ötürülməsi şifrələnmiş və təsdiq edilmişdir. Sonuncu, HTTPS, Git və Local protokollar kimi SSH səmərəlidir, ötürülmədən əvvəl məlumatları mümkün qədər yığcam edir.

Çatızmazlıqları

SSH-in mənfi tərəfi, Git depozitinizə anonim girişi dəstəkləməməsidir. SSH istifadə edirsinizsə, insanların çoxu yalnız oxumaq qabiliyyətində olsa da SSH-nin maşınlarınıza SSH girişi əldə edə bilər, bu da insanların araşdırmaq üçün depozitlərinizi klonlaşdırmaq istədikləri açıq mənbəli layihələri SSH üçün əlverişli etmir. Bunu yalnız korporativ şəbəkənizdə istifadə edirsinizsə, SSH ilə əlaqəli olduğunuz yeganə protokol ola bilər. Layihələrinizə anonim oxunuşlu giriş imkanı vermək və həmçinin SSH-dən istifadə etmək istəyirsinizsə, SSH-ı təkan verməyiniz üçün başqalarından almaq üçün başqa bir şey qurmalı olacaqsınız.

Git Protokolu

Nəhayət, Git protokolumuz var. Bu Git ilə birliktə gələn xüsusi bir daemon-dur; SSH protokoluna b ənzər bir xidmət təqdim edən xüsusi bir portda (9418) dinləyir, lakin tamamilə təsdiqlənməsi yoxdur. Git protokolu üzərində depo xidmətinin göstərilməsi üçün bir `git-daemon-export-ok` faylı yaratmalısınız - daemon bu sənəd olmadan depoya xidmət göstərməyəcək - ancaq bundan başqa t əhlükəsizlik yoxdur. Hər kəsin klonlaşması üçün Git depoziti mövcuddur, ya da yoxdur. Bu o dem əkdir ki, ümumiyyətlə bu protokola push etmək olmur. Push access-i aktivləşdirə bilərsiniz, ancaq identifikasiyanın olmaması halında, İnternetdə layihənizin URL-ini tapan hər kəs bu layihəyə push edə bilər. Bunun nadir olduğunu söyləmək kifayətdir.

Üstünlükləri

Git protokolu tez-tez mövcud olan ən sürətli şəbəkə ötürmə protokoludur. Bir ictimai layihə üçün bir çox trafikə xidmət edirsinizsə və ya oxumaq üçün istifadəçi identifikasiyasını tələb etməyən çox böyük bir layihəyə xidmət edirsinizsə, ehtimal ki, proyektinizə xidmət etmək üçün Git daemon qurmaq istəyə bilərsiniz. Şifrələmə və autentifikasiya olmadan SSH protokolu ilə eyni məlumat ötürmə mexanizmini istifadə edir.

Çatışmazlıqları

Git protokolunun mənfi tərəfi identifikasiyanın olmamasıdır. Git protokolunun layihənizə yeganə giriş olması ümumiyyətlə arzuolunmazdır. Ümumiyyətlə, push (yazma) girişi olan və hər kəsin yalnız oxumaq üçün istifadə etdiyi `git://` istifadə edən bir neçə developer üçün SSH və ya HTTPS ilə uyğunlaşdıracaqsınız. Yəqin ki, qurmaq üçün ən çətin protokoldur. `xinetd` və ya `systemd` konfiqurasiyasını və ya bənzərliyini tələb edən öz daemini işlətməlidir, bu da həmişə parkda gə zmək deyil. Həm də korporativ firewall-ların həmişə icazə verdiyi standart bir port olmayan 9418 portuna firewall girişi tələb olunur. Böyük korporativ firewall-ların arxasında bu qaranlıq port ümumiyyətlə bloklanır.

Serverdə Git Əldə Etmək

İndi biz bu protokolları öz serverində işlədən Git serveri qoşulmasını tamamlayacağıq.

Burada biz, bu servisləri macOS və Windows serverlərdə də işləməsi mümkün olan Linux bazalı serverdə adi, sadələşdirilmiş quraşdırılmaya lazım olan addımları və əmrləri nümayiş etdirəcəyik. Əsasən, öz infrastrukturunla server istehsalı tərtib etmək əməliyyat sistemi vasitələrində və ya təhlükəsizlik ölçül ərində fərqlər yaradır, lakin ümid edirik ki, bu sizə qarışıqlığın nədə olduğu haqda ümumi ideya verəcək.

Qaydasıyla ilk olaraq hər hansı bir Git server tərtib etmək üçün mövcud olan bir deponu yeni bir depoya (içində işlək qovluq olmayan) ixrac etməlisiniz. Bu ümumilikdə çox asandır. Əvvəlcə öz deponuzu yeni boş depo yaratmaq məqsədilə klonlamaq üçün `--bare` seçimi ilə klonlama əmrini işə salırsınız. Konvensiyaya göre boş depo qovluğunun adı `.git` sonluğu ilə bitir, məsələn:

```
$ git clone --bare my_project my_project.git
Cloning into bare repository 'my_project.git'...
done.
```

İndi sizin `my_project.git` qovluğunuzda Git qovluğu datanızın kopyası olmalıdır.

Bu təxminən belə bir şeyə bərabərdir:

```
$ cp -Rf my_project/.git my_project.git
```

Konfiqurasiya faylında bəzi xırda dəyişikliklər ola bilər, lakin sizin məqsədinizdə bu təxminən eynidir. O, Git deposunu işlək qovluq olmadan özü götürür və yalnız özünə aid xüsusi qovluq yaradır.

Serverə Boş Depo Daxil Edilməsi

Artıq sizin boş deponuzun kopyası var və sadəcə etməli olduğunuz onu serverə qoyub protokolları tərtib etməkdir. Fərz edək ki, siz SSH girişi olan `git.example.com` adlanan server tərtib etmisiniz və bütün Git depolarınızı /srv/git qovluğunda saxlamaq istəyirsiniz. `/srv/git`-in serverdə mövcud

olduğunu fərz etsək, siz boş deponuzu aşağıdakı kimi kopyalayaraq yeni depo yarada bilərsiniz:

```
$ scp -r my_project.git user@git.example.com:/srv/git
```

Bu zaman, `/srv/git` qovluğuna SSH bazalı oxuma girişi olan digər istifadəçilər bunu işlədərək sizin deponuzu klonlaya bilər:

```
$ git clone user@git.example.com:/srv/git/my_project.git
```

Əgər SSH-lərin istifadəçisi serverdədirsə və `/srv/git/my_project.git` qovluğuna girişi icazəsi varsa, onların avtomatik olaraq push girişi də olacaqdır.

Əgər siz `--shared` seçimindən istifadə edib `git init` əmrini işlətsəniz, Git birmənalı şəkildə avtomatik olaraq depoya icazələr yazılması üçün qrup əlavə edəcək. Qeyd edək ki, bu əmri qoşduğunuz zaman prosesdəki heç bir əmri və ya işi tələf etməyəcəksiniz.

```
$ ssh user@git.example.com
$ cd /srv/git/my_project.git
$ git init --bare --shared
```

Gördüyünüz kimi, Git deposu almaq, boş versiyasını yaratmaq və SSH girişi olan əməkdaşlarınız və sizin üçün serverə yerləşdirmək olduqca asandır. Artıq siz eyni proyektdə əməkdaşlıq etməyə hazırsınız.

Onu da qeyd etmək zəruridir ki, bir çox insanın giriş edə biləcəyi faydalı Git serveri tərtib etmək üçün bu ehtiyacınız olan hər şeydir - sadəcə SSH'lı hesabları serverə daxil edin və boş deponu bütün istifadəçilərin oxuma və yazma girişi olan yerə yapışdırın. Başqa heç nəyə ehtiyac yoxdur, artıq hazırsınız.

Digər bölmələrdə artıq daha mürəkkəb quraşdırmalarda genişləndirmələri görəcəksiniz. Bu müzakirəyə hər istifadəçi üçün hesab açmaya ehtiyac duyulmaması, depolara publik oxuma icazəsi verilir, veb UI və digərlərini tərtib etmək daxil edilir. Eyni zamanda yadda saxlamaq lazımdır ki, bir çox insanla özəl layihədə əməkdaşlıq etmək üçün sadəcə SSH serveri və boş depoya *ehtiyacınız* vardır.

Kiçik Quraşdırmalar

Əgər siz balaca qrupsunuzsa və ya az developer olan təşkilatınızda Git sınamaq istəyirsinizsə, bu sizin üçün çox asan ola bilər. Git serverin ən mürəkkəb aspektlərindən biri də istifadəçi idarəetmə əsidir. Əgər bəzi depoların istifadəçilərin bir hissəsi üçün yalnız oxunan, digər hissəsi üçün həm yazılan, həm oxunan olmasını istəyirsinizsə bunu quraşdırmaq daha çətin ola bilər.

SSH Girişi

Əgər sizin bütün developerlərinizin SSH girişi olan serveriniz varsa, bu ilk deponuzu quraşdırmaq üçün ən asan yerdir, çünki burada etməli olduğunuz heç nə yoxdur (son bölmədə göstərdiyimiz

kimi). Əgər siz depolarınızda daha çox girişə nəzarət tipli icazələr olmasını istəyirsinizsə, siz onu öz serverinizin əməliyyat sisteminizin normal fayl sistemindəki icazələrlə həll edə bilərsiniz.

Əgər siz öz depolarınızı hər hesabın yazma icazəsi olmayan komandanızın serverində yerləşdirmək istəyirsinizsə, onlar üçün SSH sistemi tərtib etməlisiniz. Fərz etsək ki, sizin bunu edəcək serveriniz var, deməli quraşdırılmış SSH serveriniz var və bu şəkildə serverə daxil olmağınız mümkündür.

Komandanızdakı hər kəsə giriş təmin etməyin bir neçə yolu var. Birincisi odur ki, hər kəsə hesab açmalısınız ki, bu da tam olaraq çox çətin bir yoldur. Ola bilər ki siz `adduser` (və ya mümkün `useradd` alternativi) işlətmək istəməyəsiniz və hər yeni istifadəçi üçün müvəqqəti parol yaratmalı olasınız.

İkinci metod mexanizmdə tək bir `git` istifadəçi yaratmaq, hər bir yazı icazəsi olan istifadəçidən SSH public key'ni sizə göndərməsini istəmək və həmin key'i yeni git hesabındakı `~/.ssh/authorized_keys` faylına əlavə etməkdir. Həmin anda hər kəs git hesabına bu mexanizmlə giriş edə bilər. Bu heç bir şəkildə tapşırıqların yerinə yetirilməsinə təsir etmir, yəni sizin əlaqə qurduğunuz SSH istifadəçisi sizin qeyd etdiyiniz tapşırıqlara mane olmur.

Bunu etməyin digər yolu LDAP serverindən və ya artıq qurduğunuz digər mərkəzləşdirilmiş mənbələrdən identifikasiya edilmiş SSH serverin olmasıdır. Hər bir istifadəçinin mexanizmə shell girişi olduğu müddətcə fikirləşdiyiniz bütün SSH identifikasiya mexanizmləri işləyə bilər.

Sizin öz SSH Public Key'nizi yaratmaq

Bir çox Git serverləri SSH public keysdən istifadə edərək identifikasiya edirlər. Əvvəlcə public key yaratmaq üçün hər bir istifadəçi əgər yoxdursa, birini yaratmalıdır. Bu prosess bütün əməliyyat sistemlərinə bənzərdir. Əvvəlcə, özünüzdə key olmadığına əmin olmaq üçün yoxlamaq lazımdır. Standart olaraq istifadəçinin SSH keysi həmin istifadəçinin `~/.ssh` qovluğunda saxlanılır. Siz həmin qovluğa gedərək və contentləri sıralayaraq artıq açarınızın olub olmadığını yoxlaya bilərsiniz.

```
$ cd ~/.ssh
$ ls
authorized_keys2  id_dsa       known_hosts
config            id_dsa.pub
```

Siz adı `id_dsa` və ya `id_rsa`olan bir neçə fayl və sonu `.pub` ilə bitən eyni fayl axtarırsınız. `.pub` faylı sizin public keynizdir, digər fayl isə ona uyğun private keynizdir. Əgər sizdə bu fayllar (və ya hətta `.ssh` qovluğu belə) yoxdursa, siz onu Git for Windowsla gələn Linux/MacOs sistemlərində SSH paketi ilə təmin olunan `ssh-keygen` proqramında yarada bilərsiniz.

```
$ ssh-keygen -o
Generating public/private rsa key pair.
Enter file in which to save the key (/home/schacon/.ssh/id_rsa):
Created directory '/home/schacon/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/schacon/.ssh/id_rsa.
Your public key has been saved in /home/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3 schacon@mylaptop.local
```

Əvvəlcə o açarı harada yadda saxlamaq istədiyinizi təsdiqləyir (`.ssh/id_rsa`), daha sonra iki dəfə parol tələb edir və əgər açar istifadə edərkən parol yazmaq istəmirsinizsə boş saxlaya bilərsiniz. Lakin, açar istifadə edirsinizsə `-o` seçimi əlavə etdiyinizdən əmin olun; bu private key'i standart formatdan fərqli olaraq güc tətbiqi parol qırmalarına qarşı daha davamlıdır. Siz həmçinin `ssh-agent` vasitəsilə hər dəfə parol daxil etmənin qarşısını ala bilərsiniz. İndi hər bir istifadəçi public key-ni sizə və ya Git serverini idarə edən kimdirsə, (fərz edək ki, siz public keys tələb edən SSH server sistemindən istifadə edirsiniz) ona göndərməlidir. Onların etməli olduqları sadəcə `.pub` fayl contentini kopyalayıb mail atmaqdır. Public keys görünüşü belədir:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

Çoxsaylı əməliyyat sistemlərində SSH açarları yaratmaqda daha dərin məlumay üçüb SSH açarlarının GitHub bələdçisinə baxın https://help.github.com/articles/generating-ssh-keys.

Server qurmaq

Gəlin server tərəfində SSH girişini quraraq gəzək. Bu nümunədə istifadəçilərinizin identifikasiyası üçün `authorized_keys` metodundan istifadə edəcəksiniz. Ubuntu kimi standart Linux distributorunda işlədiyinizi güman edirik.

Burada təsvir edilənlərin yaxşı bir hissəsi, manual kopyalama və public key-ləri quraşdırmaq yerinə `ssh-copy-id` əmrindən istifadə etməklə avtomatlaşdırıla bilər.

Birincisi, bir istifadəçi hesabı və bu istifadəçi üçün *.ssh* qovluğu yaradırsınız.

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh && chmod 700 .ssh
$ touch .ssh/authorized_keys && chmod 600 .ssh/authorized_keys
```

Bundan sonra, `git` istifadəçisi üçün `authorized_keys` faylına bəzi developer SSH public key-lərini əlavə etməlisiniz. Güman edək ki, bəzi etibarlı public key-ləriniz var və onları müvəqqəti sənədlə rdə saxlamısınız. Yenə public key-lər bu kimi bir şeyə bənzəyir:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

Onları sadəcə `.ssh` qovluğundakı `git` istifadəçinin `authorized_keys` faylına əlavə edirsiniz:

```
$ cat /tmp/id_rsa.john.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.josie.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.jessica.pub >> ~/.ssh/authorized_keys
```

İndi, `--bare` seçimi ilə `git init` əmrindən istifadə edərək boş bir depo qura bilərsiniz:

```
$ cd /srv/git
$ mkdir project.git
$ cd project.git
$ git init --bare
Initialized empty Git repository in /srv/git/project.git/
```

Sonra, John, Josie və ya Jessica, layihənin ilk versiyasını uzaqdan əlavə edərək branch-a push edər ək həmin depo içərisinə push edə bilər. Diqqət yetirin ki, kimsə bir layihə əlavə etmək istədikdə maşına shell verməli və çılpaq bir depo yaratmalıdır. `git` istifadəçi və depo qurduğunuz serverin hostname olaraq `gitserver`-dən istifadə edək. Bunu daxili işlədirsinizsə və bu serverə işarə etmək üçün `gitserver` üçün DNS qurursunuzsa, əmrləri olduğu kimi istifadə edə bilərsiniz (`myproject` iç ərisində faylları olan bir layihə olduğuna inanaraq):

```
# on John's computer
$ cd myproject
$ git init
$ git add .
$ git commit -m 'Initial commit'
$ git remote add origin git@gitserver:/srv/git/project.git
$ git push origin master
```

Bu zaman digərləri onu klonlaşdıra bilər və dəyişiklikləri asanlıqla geri qaytara bilər:

```
$ git clone git@gitserver:/srv/git/project.git
$ cd project
$ vim README
$ git commit -am 'Fix for README file'
$ git push origin master
```

Bu metodla bir ovuc developer üçün Git serverini oxumaq/yazmaq üçün tez bir zamanda əldə edə bilərsiniz.

Qeyd etməlisiniz ki, hazırda bu istifadəçilərin hamısı serverə girib `git` istifadəçisi kimi bir shell ə ldə edə bilərlər. Bunu məhdudlaşdırmaq istəyirsinizsə, shell-in `/etc/passwd` faylındakı başqa bir şeyə dəyişdirməlisiniz.

Git ilə əlaqəli `git-shell` adlı məhdud bir alət vasitəsi ilə `git` istifadəçi hesabını yalnız Git ilə əlaqəli fəaliyyətlə asanlıqla məhdudlaşdıra bilərsiniz. Bunu `git` istifadəçi hesabının giriş shell-i kimi qursanız, o zaman bu hesab serverinizə normal shell çıxışı əldə edə bilməz. Bunun üçün `bash` və `csh` əvəzinə `git-shell` daxil edin. Bunu etmək üçün əvvəlcə `git-shell` əmrinin tam path adını onsuz da olmadıqda `/etc/shells`-ə əlavə etməlisiniz:

```
$ cat /etc/shells   # see if git-shell is already in there. If not...
$ which git-shell   # make sure git-shell is installed on your system.
$ sudo -e /etc/shells  # and add the path to git-shell from last command
```

İndi isə `chsh <username> -s <shell>` istifadə edərək istifadəçi üçün shell düzəldə bilərsiniz:

```
$ sudo chsh git -s $(which git-shell)
```

İndi `git` istifadəçisi hələ də SSH bağlantısından Git depolarını push və pull etmık üçün istifadə edə bilər, ancaq maşın üzərində shell qoya bilmir. Əgər cəhd etsəniz, giriş üçün rədd cavabı görəc əksiniz:

```
$ ssh git@gitserver
fatal: Interactive git shell is not enabled.
hint: ~/git-shell-commands should exist and have read and execute access.
Connection to gitserver closed.
```

Bu nöqtədə, istifadəçilər hələ də git serverin çata biləcəyi hər hansı bir hosta daxil olmaq üçün SSH portunu istifadə edə bilərlər. Bunun qarşısını almaq istəyirsinizsə, `authorized_keys` faylını düzəldə bilərsiniz və məhdudlaşdırmaq istədiyiniz hər key-ə aşağıdakı seçimləri göndərə bilərsiniz:

```
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty
```

Nəticə belə görünəcəkdir:

```
$ cat ~/.ssh/authorized_keys
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4LojG6rs6h
PB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4kYjh6541N
YsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9EzSdfd8AcC
IicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myivO7TCUSBd
LQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPqdAv8JggJ
ICUvax2T9va5 gsg-keypair

no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQDEwENNMomTboYI+LJieaAY16qiXiH3wuvENhBG...
```

İndi Git şəbəkə əmrləri hələ də yaxşı işləyəcək, lakin istifadəçilər bir shell əldə edə bilməyəcəklər. Çıxışda deyildiyi kimi `git-shell` əmrini bir az tənzimləyən `git` istifadəçi ev qovluğunda bir qovluq da qura bilərsiniz. Məsələn, serverin qəbul edəcəyi Git əmrlərini məhdudlaşdıra bilərsiniz və ya istifadəçilərin belə bir şəkildə SSH etməyə çalışdıqlarını görən mesajı düzəldə bilərsiniz. Shell-in f ərdiləşdirilməsi haqqında daha çox məlumat üçün `git help shell`-i işə salın.

Git Daemon

Daha sonra biz “Git” protokolundan istifadə edərək daemon xidməti göstərən depolar quracağıq. Bu sizin Git datanıza girmək üçün sürətli təsdiqlənməmiş yoldur. Yadda saxlayın ki, bu təsdiqlə nməmiş servis olduğu üçün bu protokol üzərindən xidmət etdiyiniz hər şey onun şəbəkəsində publikdir. Əgər siz bunu təhlükəsizlik divarınızdan kənarda edirsinizsə, onda bu mütləq dünyaya görünən proyektlər üçün istifadə olunmalıdır. Lakin əgər bu serveri təhlükəsizlik divarınızdan iç əridə edirsinizsə, onu bir çox read-only girişi olan insanlar və kompyuterlər (davamlı inteqrasiya və ya tikinti servisi) üçün SSH key istifadə etmək istəməyəndə istifadə edə bilərsiniz. Hər şəkildə Git protokolu quraşdırılması çox asandır. Sadəcə olaraq siz bu commandı demonized mannerdə qoşmalısınız:

```
$ git daemon --reuseaddr --base-path=/srv/git/ /srv/git/
```

`--base-path` seçimi insanlara bütün path'i dəqiqləşdirmədən proyektləri klonlamağa imkan verir və sonda path Git daemon'a ixrac edilməli depoları harada axtarmalı olduğunu deyərkən, `--reuseaddr` seçimi isə serverin köhnə əlaqənin vaxtının bitməsini gözləmədən yenilənməsinə icazə verir. Əgər siz təhlükəsizlik divarı işlədirsinizsə, siz həmçinin qoşduğunuz qutudakı 9418 port'unda dəlik açmalısınız.

Siz bu prosesi işlətdiyiniz əməliyyat sistemindən asılı olaraq bir neçə yolla daemonize edə bil ərsiniz.

`systemd` müasir Linux distribyutorlarında ən ümumi sistem olduğu üçün siz onu bu məqsədlə istifadə edə bilərsiniz. Asanlıqla faylı aşağıdakı məzmunla `/etc/systemd/system/git-daemon.service` daxil edin.

```
[Unit]
Description=Start Git Daemon

[Service]
ExecStart=/usr/bin/git daemon --reuseaddr --base-path=/srv/git/ /srv/git/

Restart=always
RestartSec=500ms

StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=git-daemon

User=git
Group=git

[Install]
WantedBy=multi-user.target
```

Fikir versəniz burada Git daemon həm istifadəçi, həm də qrup üçün `git` ilə başlayır. Onu sizə lazım olduğu kimi dəyişdirin və əmin olun ki, təmin olunan istifadəçi (provided user) sistemdə mövcuddur. Eyni zamanda, `/usr/bin/git` də yerləşən Git binary'ni yoxlayın və əgər ehtiyac varsa path'i dəyişdirin.

Son olaraq, ön yükləmədəki servisi (service on boot) avtomatik başlatmaq üçün `systemctl enable git-daemon`-u və ayrılıqda servisi başlatmaq və dayandırmaq üçün `systemctl start git-daemon` və `systemctl stop git-daemon`-u qoşacaqsınız.

Digər sistemlərdə siz həmçinin sizin `sysvinit` sisteminizdəki `xinetd` sənədini (script) və ya başqa - aldığınız əmr demonized olduğu və ya baxıldığı müddətcə - istifadə edə bilərsiniz.

Daha sonra, Git'ə hansı depoların Git server bazasına təsdiqlənməmiş girişinə icazə verildiyini deməlisiniz. Bunu hər depoda `git-daemon-export-ok` adlı fayl yaradaraq edə bilərsiniz.

```
$ cd /path/to/project.git
$ touch git-daemon-export-ok
```

Həmin faylın mövcud olması Git'ə bu proyektin təsdiqlənmədən xidmət edilməsinin mümkün olduğunu deyir.

Smart HTTP

İndi SSH vasitəsilə autentifikasiya edilmiş giriş və `git://` vasitəsilə təsdiqlənməmiş bir giriş əldə etdik, eyni zamanda hər ikisini eyni anda edə biləcək bir protokol da var. Smart HTTP qurmaq, sadəcə serverdə `git-http-backend` adlı Git ilə təmin olunan CGI skriptini təmin etməkdir. Bu CGI `git fetch` və ya `git push` tərəfindən göndərilən path və başlıqları bir HTTP URL-ə oxuyacaq və müşt ərinin HTTP ilə əlaqə qura biləcəyini müəyyən edəcək(bu 1.6.6 versiyasından bəri hər hansı bir müştəri üçün doğrudur). CGI müştərinin ağıllı olduğunu görsə, onunla ağıllı əlaqə quracaq; əks t əqdirdə lal davranışa geri dönəcəkdir.(buna görə köhnə müştərilərlə oxunuş geriyə uyğundur).

Gəlin çox təməl və sadə quruluşdan danışaq. Bunu Apache ilə CGI server olaraq quracağıq. Apache quruluşunuz yoxdursa, bunu Linux qutusunda bu kimi bir şeylə edə bilərsiniz:

```
$ sudo apt-get install apache2 apache2-utils
$ a2enmod cgi alias env
```

Bu da lazımi şəkildə işləməsi üçün lazım olan `mod_cgi`, `mod_alias` və `mod_env` modullarına imkan verir.

Ayrıca `/srv/git` qovluqlarının Unix istifadəçi qrupunu `www-data` olaraq təyin etməlisiniz, beləliklə veb serveriniz depoları oxuya və yaza bilərsiniz, çünki CGI skriptini işlədən Apache nümunəsi (default olaraq) həmin istifadəçi kimi çalışır:

```
$ chgrp -R www-data /srv/git
```

Bundan sonra veb serverinizin `/git` yoluna girən bir şey üçün işçi olaraq `git-http-backend` işlətmək üçün Apache konfiqurasiyasına bəzi şeylər əlavə etməliyik.

```
SetEnv GIT_PROJECT_ROOT /srv/git
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAlias /git/ /usr/lib/git-core/git-http-backend/
```

Əgər siz `GIT_HTTP_EXPORT_ALL` mühit dəyişkənliyini kənara qoyursunuzsa, Git yalnız Git demonunda olduğu kimi təsdiqlənməmiş müştərilərə `git-daemon-export-ok` faylı olan depoları təmin edəcəkdir.

Nəhayət, Apache-yə `git-http-backend` sorğularına icazə verməsini və yazıların müəyyən bir şəkildə doğrulanmasını, bəlkə də bu kimi bir Auth bloku ilə etməsini istəməlisiniz:

```
<Files "git-http-backend">
    AuthType Basic
    AuthName "Git Access"
    AuthUserFile /srv/git/.htpasswd
    Require expr !(%{QUERY_STRING} -strmatch '*service=git-receive-pack*' ||
%{REQUEST_URI} =~ m#/git-receive-pack$#)
    Require valid-user
</Files>
```

Bütün etibarlı istifadəçilərin şifrələrini ehtiva edən bir “.htpasswd` faylını yaratmağı tələb edəc əkdir. ``schacon” istifadəçisini fayla əlavə etmək nümunəsi belədir:

```
$ htpasswd -c /srv/git/.htpasswd schacon
```

Apache identifikasiyası istifadəçilərinə sahib olmaq üçün bir çox yol var, onlardan hər hansı birini seçib uygulamalısınız. Bu, gələ biləcəyimiz ən sadə nümunədir. Bütün məlumatların şifrələnməsi üçün bunu SSL üzərində qurmaq istəyəcəksiniz.

Apache konfiqurasiya xüsusiyyətlərinin rabbit hole-undan çox uzaqlaşmaq istəmirik, çünki fərqli bir serverdən istifadə edə və ya fərqli identifikasiyaya ehtiyacınız ola bilər. Fikir budur ki, Git `git-http-backend` adlı bir CGI ilə birlikdə HTTP üzərindən məlumat göndərmək və qəbul etmək üçün bütün danışıqları aparacaqdır. Heç bir identifikasiyanı özü həyata keçirmir, ancaq onu çağıran veb serverin qatında asanlıqla idarə oluna bilər. Bunu hər hansı bir CGI bacarıqlı bir veb serveri ilə edə bilərsiniz, ona görə ən yaxşı bildiyiniz biri ilə dəvam edin.

Apache-də identifikasiyanı konfiqurasiya etmək haqqında daha çox məlumat üçün Apache sənədlərini buradan yoxlaya bilərsiniz: https://httpd.apache.org/docs/current/howto/auth.html

GitWeb

Artıq sizin proyektinizə oxuma/yazma və sadəcə-oxuma girişiniz var və siz sadə veb bazasında görüntüləyə bilərsiniz. Git Gitweb adlanan və bunun üçün istifadə olunan GGI skriplə gəlir.

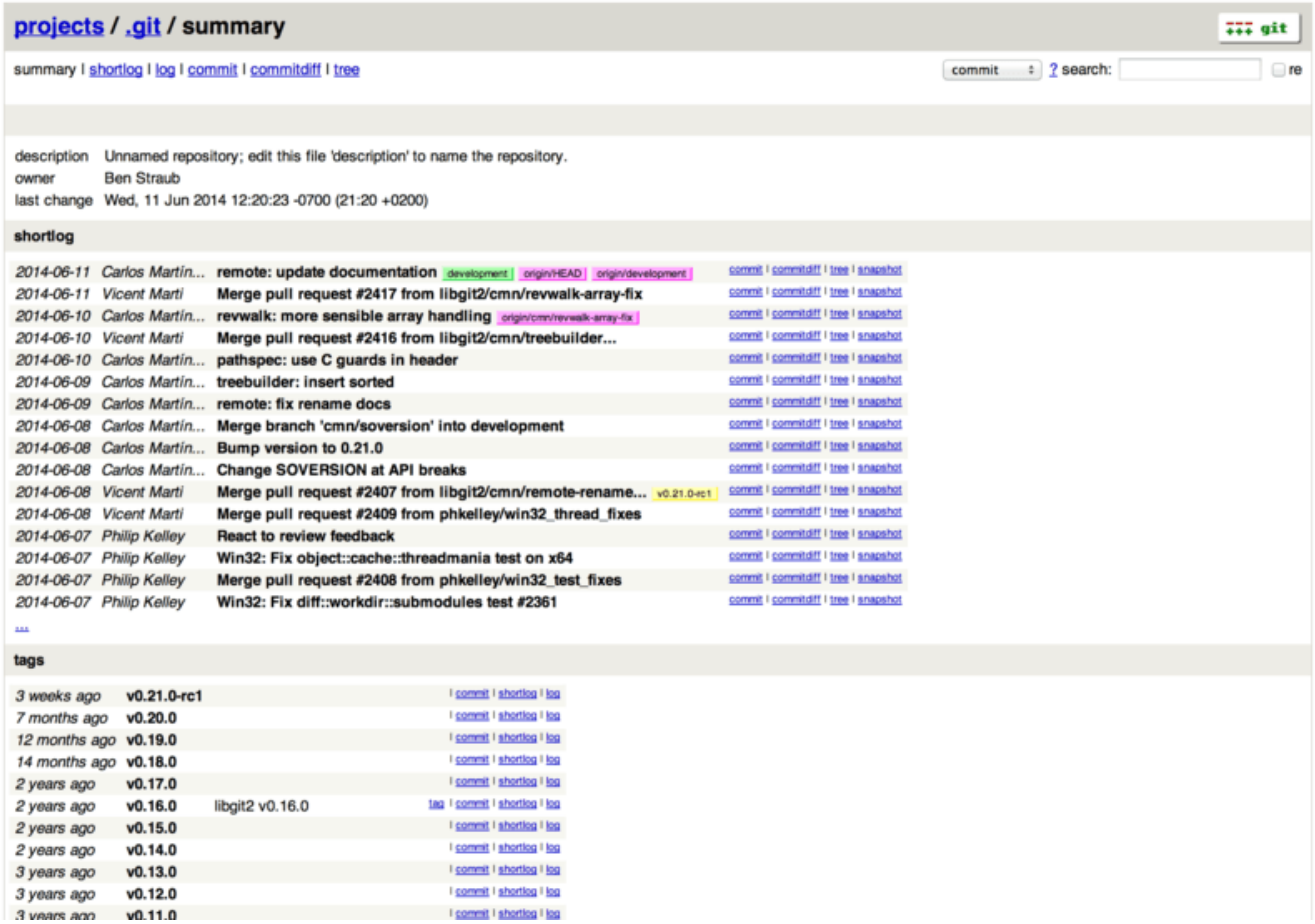


Figure 49. GitWebin veb bazalı istifadəçi interfeysi.

Əgər siz GitWebin proyektinizdə necə görünəcəyini görmək istəyirsinizsə, Git sizin siteminizdə `lighttpd` və ya `webrick` kimi yüngül veb serveri varsa müvəqqəti instasiyanı yandırmaq əmri ilə gə ləcək. Linux mexanizmlərində `lighttpd` çox vaxt quraşdırılmış olur, yəni siz proyekt qovluğunuza `git instaweb` yazaraq onu işə sala bilərsiniz. Lakin, Mac işlədirsinizsə Leopard Rubdən əvvəl quraşdırılmış olduğundan `webrick` sizə ən uyğun seçim olacaq. `Instaweb`-i lighttpd olmadan başlatmaq istəyirsinizsə onu `--httpd` seçimilə işə sala bilərsiniz.

```
$ git instaweb --httpd=webrick
[2009-02-21 10:02:21] INFO  WEBrick 1.3.1
[2009-02-21 10:02:21] INFO  ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```

O port 1234-də HTTPD serverini işə salır və daha sonra həmin səhifədə açılan veb browseri avtomatik işə salır. İşinizi bitirib serveri söndürmək istəyəndə isə eyni əmri `--stop` seçimi ilə edə bilərsiniz.

```
$ git instaweb --httpd=webrick --stop
```

Əgər siz öz komandanız və ya idarə etdiyiniz açıq mənbəli proyektlər üçün veb interfeys qoşmaq istəyirsinizsə, öz normal veb serverinizin dəstəklədiyi CGI skriptini qoşmalısınız. Bəzi Linux distribyutorlarının `apt` və ya `dnf` ilə quraşdırılabilən gitweb paketi var və siz ilk onları yoxlaya bil ərsiniz. Biz manual və sürətli şəkildə Gitwebi işə salacağıq. İlkin olaraq sizə Gitweb ilə gələn Git m ənbə kodunu əldə etmək və uyğum CGI skripti yaratmaq lazım olacaq:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/srv/git" prefix=/usr gitweb
    SUBDIR gitweb
    SUBDIR ../
make[2]: `GIT-VERSION-FILE' is up to date.
    GEN gitweb.cgi
    GEN static/gitweb.js
$ sudo cp -Rf gitweb /var/www/
```

Yadda saxlayın ki, `GITWEB_PROJECTROOT`-da Git deposunu tapmaq əmri dəyişkəndir. İndi siz VirtualHost'a əlavə edə biləcəyiniz Apache istifadəsi üçün CGI skripti yaratmaq lazımdır:

```
<VirtualHost *:80>
    ServerName gitserver
    DocumentRoot /var/www/gitweb
    <Directory /var/www/gitweb>
        Options +ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
        AllowOverride All
        order allow,deny
        Allow from all
        AddHandler cgi-script cgi
        DirectoryIndex gitweb.cgi
    </Directory>
</VirtualHost>
```

Təkrar qeyd etməliyik ki, GitWeb hər hansı bir CGI və ya Perl dəstəkli veb serverlə işə salına bilər; lakin siz başqa bir şey istifadə etmək istəsəniz onu da quraşdırmaq çətin olmayacaqdır. Bu halda siz `http://gitserver/`-a daxil olaraq onlayn depolarınıza baxa bilərsiniz.

GitLab

GitWeb olduqca asan quruluşdadır. Əgər siz daha müasir və tam xüsusiyyətli Git server axtarırsınızsa, onun yerinə quraşdıra biləcəyiniz başqa open source solutions var. Gitlab onların ən məşhurlarından olduğu kimi biz onun quraşdırılmasını mühafizə edə və misal kimi göstərə bilərik. Bu GitWeb seçimindən bir az daha mürəkkəbdir və daha çox təminat tələb edir, lakin tam xüsusiyyətli seçimdir.

Quraşdırılma

GitLab məlumat bazalı veb-tətbiqdir, buna görə də onun quraşdırılması digər Git serverlərə görə daha dolaşıqdır. Yaxşı ki, bu prosess tam sənədli və dəstəklənəndir.

GitLab quraşdırılmasını izləmək üçün bir neçə metod vardır. Bir şeyi tez işlətmək istəyirsinizsə, https://bitnami.com/stack/gitlab-dan bir klik quraşdırıcını və ya virtual mexanizm təsvirini yükləyə və konfiqurasiyanı xüsusi çevrənizə uyğun dəyişə bilərsiniz. Bir incə toxunuşla Bitnami login ekranını daxil edir (alt+→ yazaraq daxil edilə bilir); bu sizə GitLab üçün quraşdırılmış ip ünvan,

standart istifadəçi adı və parolu göstərir.

Figure 50. Virtual mexanizm Bitnami Gitlabın giriş ekranı

Başqa hər şey üçün, https://gitlab.com/gitlab-org/gitlab-ce/tree/master-da yerləşən GitLab Community Editiondakı rəhbəri izləyə bilərsiniz. Orada siz Chef reseptlərindən istifadə edərək GitLab quraşdırılmasına yardım, Digital Oceanda virtual ekran və RPM və DEB paketlərini (betadakı bu yazılar kimi) tapa bilərsiniz. Orada həmçinin, standart olmayan əməliyyat sisteml ərində GitLab qoşmağın "qeyri-rəsmi" yollarını, tam manual quraşdırmaları və digər mövzuları tapa bilərsiniz.

Idarəetmə

GitLab'ın idarəetmə interfeysi veb üzərindən daxil olunandır. Sadəcə brauzerinizi GitLab'ın quraşdırıldığı IP adressə və ya host adına işarələyin və admin istifadəçi kimi daxil olun. Standart istifadəçi adı `admin@local.host` və standart parol `5iveL!fe` (hansı ki daxil olduğunuz anda dəyişə bil ərsiniz) olacaq. Daxil olduqda menyunun sağ üst tərəfindəki "Admin area" ikonuna klikləyin.

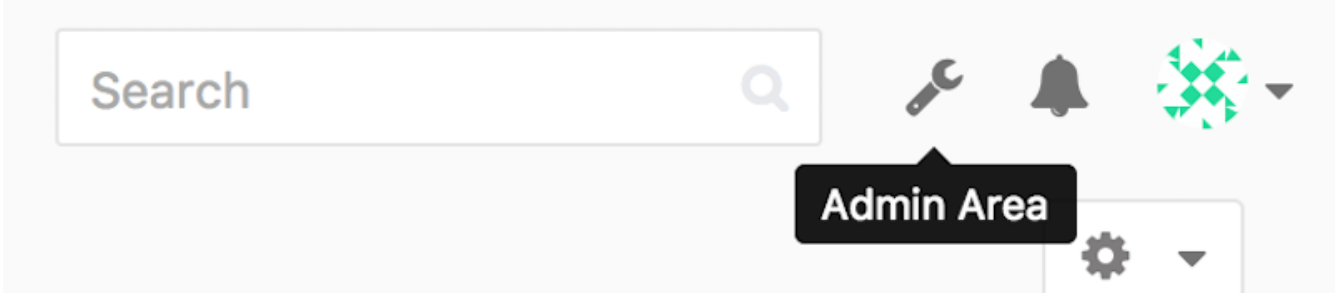


Figure 51. GitLab menyusundakı "Admin area" ikonu.

İstifadəçilər

GitLab hesablarında istifadəçilər insanlarla yazışanlardır. İstifadəçi hesablarında çox mürəkkəblik yoxdur; əsasən onlar əsas dataya yığılmış şəxsi informasiyalar toplusudur. Hər bir istifadəçi hesabı

həmin istifadəçiyə aid olan proyektlər qrupunun **namespace**'i (ad boşluğu) ilə gəlir. Əgər istifadəçi jane'in proyekt adlı proyekti olarsa, onda həmin proyektin url'i belə görünür: http://server/jane/project

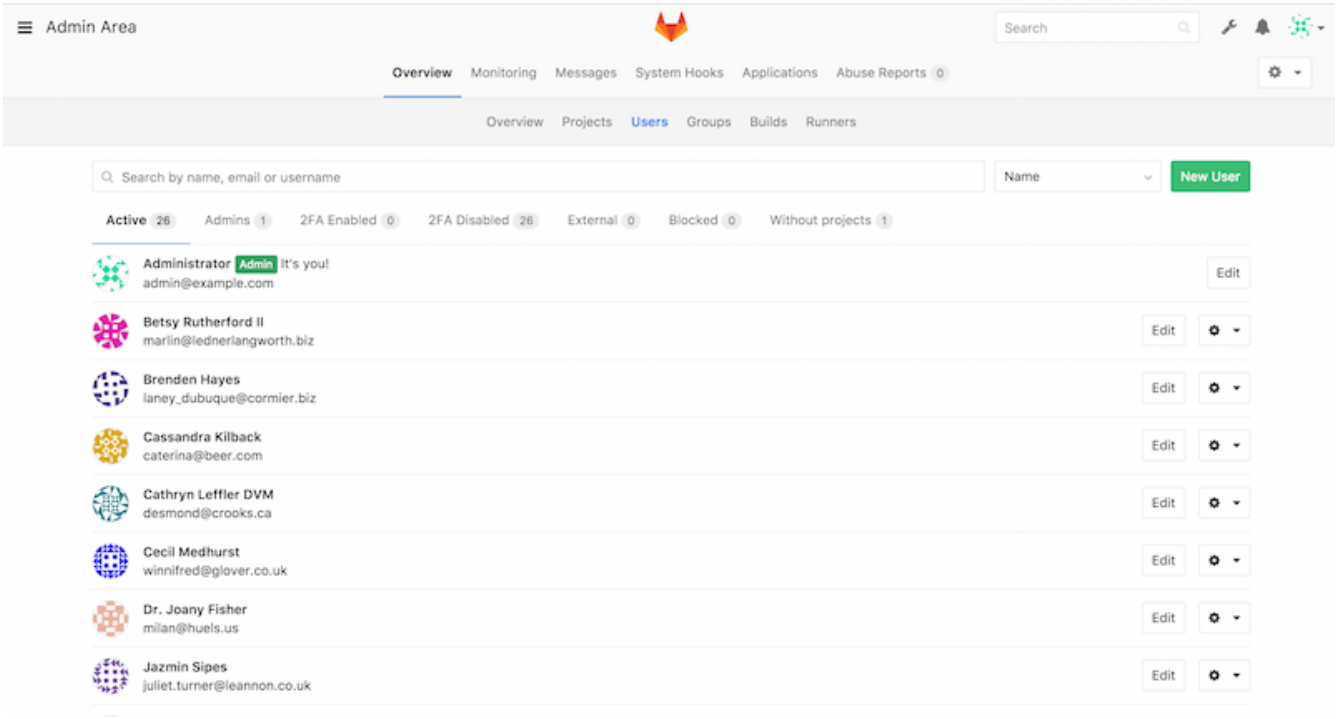


Figure 52. GitLab istifadəçisinin idarəetmə ekranı.

İstifadəçini iki yolla silmək olar. "Blocking" istifadəçini GitLab'a girişdən məhrum edir, lakin həmin istifadəçinin ad boşluğundakı bütün datası qorunur və həmin istifadəçinin mail adresi ilə verdiyi bütün commit-lər həmin istifadəçinin profilinə linklənir.

Digər tərəfdən isə, istifadəçini "Destroying" (yox etmək) onu faylsistemdən və databasedən tam olaraq silir. Onların ad boşluğundakı bütün data və proyektlər silinir, həmçinin onlara məxsus bütün qruplar aradan qaldırılır. Bu çox nadir hallarda istifadə edilir və daha qalıcı və dağıdıcı addımdır.

Qruplar

GitLab qrupu istifadəçilərin proyektlərə necə daxil olduğunun datasını əhatə edən proyektlər assambleyasıdır. Hər qrupun -istifadəçilərdə olduğu kimi- proyekt ad boşluğu mövcuddur, yəni qrup məşqinin proyekt materialları olarsa, onlar belə görünür: `http://server/training/materials`.

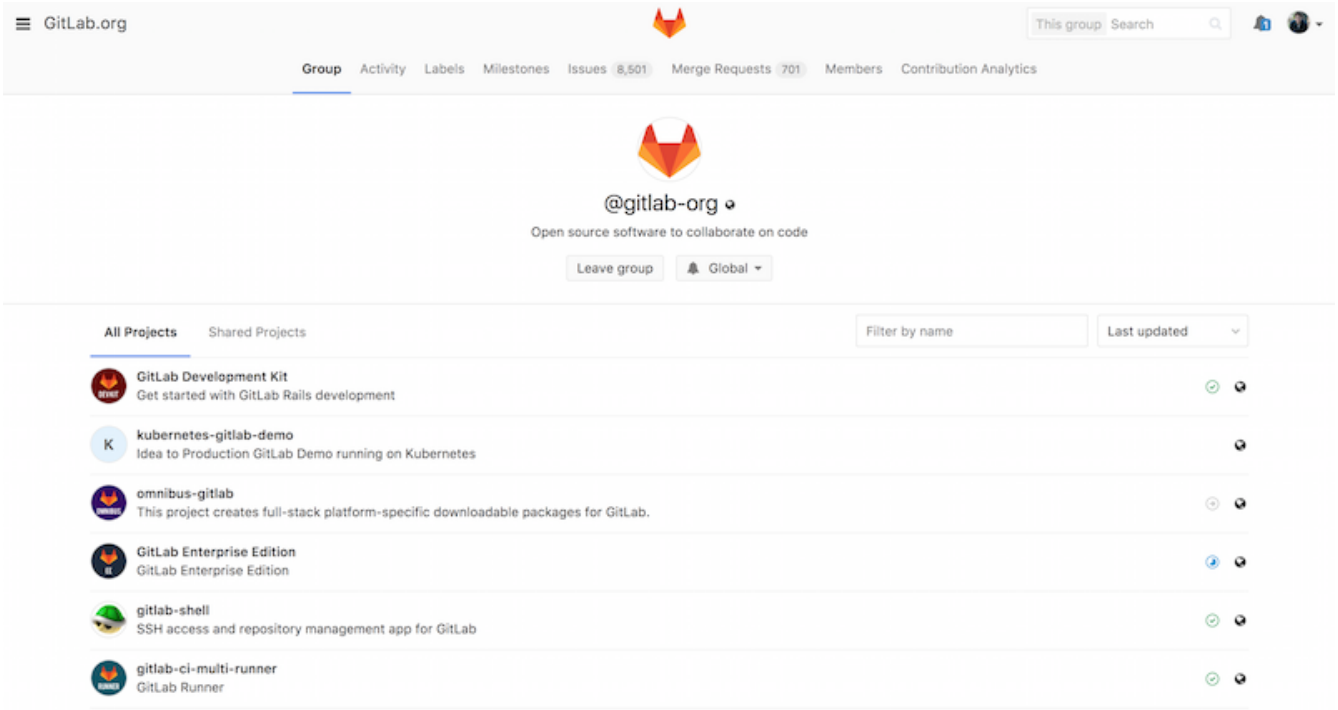


Figure 53. GitLab qrup idarəetmə ekranı.

Hər bir qrup həmin qrupun icazələri və proyektləri bir neçə istifadəçiyə bağlıdır. Bu “Guest”-dən (sadəcə problemlər və söhbətlər) “Owner”-ə (bütün qrupu, üzvlərini və proyektlərini idarəetmə) q ədər hissəni əhatə edir. Burada sadalanan çoxsaylı proyekt tipləri var, lakin GitLab’ın idarəetmə ekranında yardımçı linki mövcuddur.

Proyektlər

GitLab proyekti tək bir Git deposuna uyğunlaşır. Hər proyekt bir ad boşluğuna, istifadəçiyə və ya qrupa aiddir. Əgər proyekt istifadəçiyə aiddirsə, proyektin sahibi proyektə girişi olan hər kəsə birbaşa nəzarət edə bilir; əgər proyekt qrupa məxsusdursa, istifadəçi səviyyə icazəsi də həmçinin təsir edə bilir.

Hər proyektin depolara və proyektin səhifələrinə oxuma girişi olanlara nəzarət edilə bilən görünmə dərəcəsi var. Əgər proyekt *Private*-dirsə proyektin sahibi xüsusi istifadəçilərə geniş daxilolma icazəsi verməlidir. *Internal* proyekt daxil olan hər istifadəçiyə görünəndir, *Public* proyekt isə hər kəsə görünəndir. Qeyd edin ki, bu UI veb proyekti üçün hər iki `git fetch` girişinə nəzarət edir.

Hooklar

GitLab proyekt və sistem səviyyəsində hookları da daxil edir. Bunların hər ikisi üçün GitLab serveri müvafiq hadisələr baş verəndə JSON təsvirilə HTTP POST yerinə yetirir. Bu sizin Git depolarınızı və GitLab instansiyasınızı inkişaf avtomatlaşdırmanızın qalan hissəsinə bağlamaq üçün əla bir yoldur, məsələn, CI serverləri, söhbət otaqları və ya yerləşdirmə qurğuları.

Əsas İStifadə

GitLab ilə etmək istənilən ilk şey yeni proyekt yaratmaqdır. Bu alətlər panelindəki “+” ikonuna basmaqla yerinə yetirilir. Sizdən proyektin adı, hansı ad boşluğuna aid olduğu və görünmə səviyy əsinin nə olduğu soruşulacaqdır. Burada qeyd edilənlərin çoxu daimi olmur və tənzimlənmələr

bölməsindən təkrar dəyişdirilə bilər. “Create Project”-ə kliklәyin və budur, artıq hazırdır.

Proyekt hazır olduqdan sonra siz onu böyük ehtimalla lokal Git deposuna bağlayacaqsınız. Hər proyekt HTTPS və ya SSH üzərindən daxil olunandır və hər ikisi Git yayımını konfiqurasiya etmək üçün istifadə oluna bilər. URL’lar proyektin ana səhifəsində yuxarıda görünürlər. Bu əmr yayımlanan ərazidə mövcud yerli depo üçün `gitlab` adlı remote yaradacaq:

```
$ git remote add gitlab https://server/namespace/project.git
```

Deponun sizdə local kopyası yoxdursa, onda siz sadə şəkildə belə edə bilərsiniz:

```
$ git clone https://server/namespace/project.git
```

Veb UI deponun özündə işlək görünmələri təmin edir. Hər proyektin ana səhifəsi ən son hərəkətl əri göstərir və üst tərəfdəki linklər sizə proyektin faylları və tapşırıqlarını göstərir.

Birlikdə İşləmək

GitLab proyektində birgə işləməyin ən rahat yolu digər istifadəçiyə Git deposuna birbaşa push access verməkdir. Proyektin tənzimləmələr hissəsində “Members” hissəsində proyektə istifadəçi əlavə edə bilərsiniz və yeni istifadəçinin giriş səviyyəsini əlaqələndirə bilərsiniz (fərqli giriş sə viyyələri Qruplar'da müzakirə edilir). İstifadəçi developer və ya yuxarı səviyyə giriş verilərsə, c əzasız birbaşa depo olan branch-lar ilə commit-lər verə bilər.

Əməkdaşlıq etməyin digər yollarından biri də istəkləri birləşdirməkdir. Bu özəllik hər bir istifad əçiyə nəzarətli şəkildə proyektə dəstək verməyə imkan verir. Birbaşa girişi olan istifadəçilər branch yarada, ona commit-lər verə və master və ya başqa branch-larda birləşdirilmiş istəklər yarada bilərlər. Depoya push icazəsi olmayan istifadəçilər onu “fork” (öz kopyasını yaratmaq) edə bilər, həmin kopyaya push tapşırıqları verə bilər və əsas proyektdən geriyə forkdan istəklər aça bilərlər. Bu model sahibinə etibar etibarsız istifadəçilərin dəstəyi ilə depoya nəyin nə zaman g əldiyini nəzarət altına almağa imkan verir.

Birləşdirilmiş istəklər və problemlər Gitlabdakı uzunmüddətli diskussiyanın əsas hissələrindəndir. Hər birləşmə istəyi təklif olunan dəyişikliyin(hansı ki, yüngül kod yığımını tələb edir), eləcə də ümumi müzakirə mövzusunun xətti olaraq müzakirə edilməsinə imkan verir. Onların hər ikisi istifadəçilər tərəfindən təyin oluna və mərhələlərə bölünə bilər.

Bu hissə ən əsas GitLabın Git ilə əlaqəli xüsusiyyətlərinə yönəldilmişdir, lakin yekun proyekt olaraq o sizə wiki və sistem vasitələri ilə komandanızla birgə işləməyə imkan verir. GitLabin başqa bir üstünlüyü də odur ki, server quraşdırılıb işləyirsə, sizin nadir hallarda konfiqurasiya faylını tweakləməyə və ya SSH serveri ilə giriş etməyə ehtiyacınız olacaq (əksər idarəetmə və ümumi istifadə browserin interfeysində başa çatdırılır).

Üçüncü Tərəf Seçimləri

Öz Git serverinizi qurmaqla bağlı bütün işlərdən keçmək istəmirsinizsə, Git layihələrinizi xarici xüsusi bir hosting saytında yerləşdirmək üçün bir neçə seçiminiz var. Bunu etmək bir sıra

üstünlüklər təqdim edir: bir hosting saytı ümumiyyətlə tez qurulur və layihələrə başlamaq asandır və heç bir serverə qulluq və ya monitorinq iştirak etmir. Öz serverinizi daxili qurduğunuz və işl ədiyiniz təqdirdə yenə də açıq mənbə kodu üçün açıq bir hosting saytından istifadə etmək istəyə bilərsiniz - ümumiyyətlə açıq mənbə cəmiyyətinin sizə kömək etməsi deməkdir və kömək etməsi daha asan olacaqdır.

Bu günlərdə fərqli üstünlükləri və mənfi cəhətləri ilə seçmək üçün çox sayda hosting seçiminiz var.

Son siyahıları görmək üçün əsas Git wiki-də GitHosting səhifəsini https://git.wiki.kernel.org/index.php/GitHosting ünvanına baxın.

GitHub-dan istifadə edərək GitHub-da ətraflı məlumat verəcəyik, çünki orada ən böyük Git hostdur və hər halda bu layihə ilə qarşılıqlı əlaqə qurmağınız lazım ola bilər, lakin öz Git serverinizi qurmaq istəmədiyiniz təqdirdə daha çox seçim var.

Qısa Məzmun

Başqaları ilə əməkdaşlıq edə bilmək və ya işinizi paylaşmaq üçün remote bir Git deposunu işə salmaq üçün bir neçə seçiminiz var.

Öz serverinizi işlətmək sizə çox nəzarət verir və serverinizi öz firewall içərisində işləməyinizə imkan verir, lakin belə bir server ümumiyyətlə qurmaq və saxlamaq üçün kifayət qədər vaxt tələb edir. Verilərinizi hosted bir serverə yerləşdirirsinizsə, qurmaq və qorumaq asandır; bununla birlikdə kodunuzu başqasının serverlərində saxlamağınız lazımdır və bəzi təşkilatlar buna icazə vermir.

Hansı həllin və ya həll birləşməsinin sizin və təşkilatınız üçün uyğun olduğunu müəyyənləşdirmək kifayət qədər sadə olmalıdır.

Paylanmış Git

Artıq bütün developer'lərin kodlarını paylaşması üçün mərkəz nöqtəsi olaraq qurulmuş remote bir Git deposuna sahib olduğunuzdan və lokal iş axınındakı əsas Git əmrləri ilə tanış olduğunuza görə bəzi paylanmış iş axınlarından necə istifadə edəcəyinizə baxacaqsınız.

Bu fəsildə paylayıcı və inteqrasiyaedici olaraq paylanmış bir mühitdə Git ilə necə işləməli olduğunuzu görəcəksiniz. Yəni bir layihəyə uğurla kod əlavə etməyi və onu və layihə qoruyucusunu mümkün qədər asanlaşdırmağı və bir sıra developer'lərlə layihəni necə uğurla davam etdirməyi öyrənəcəksiniz.

Distribyutorluq İş Axınları

Centralized Version Control Systems (CVCS)'dən fərqli olaraq distribyutor xarakterli Git sizə developerlərin proyektlərdə əməkdaşlığı zamanı daha çevik olmağa imkan verir. Mərkəzl əşdirilmiş sistemlərdə hər developer mərkəz nöqtə ilə az ya da çox eyni işləyir. Gitdə hər developer potensial olaraq düyün və mərkəz nöqtədir; hər bir developer həm digər depolara kod dəstəyi verə bilər, həm də başqalarının işləyə biləcəyi və dəstək verə biləcəyi public depo saxlaya bilər. Bu, sizin layihəniz və ya komandanız üçün çox sayda workflow imkanlarını təqdim edir, buna görə də bu rahatlıqdan istifadə edən bir neçə ümumi paradiqma əhatə edəcəyik. Hər dizaynın güclü və mümkün zəif tərəflərini keçəcəyik; bu halda istifadə etmək üçün tək birini seçə bilər və ya bir neç əsinin xüsusiyyətlərini qarışdırıb uyğunlaşdıra bilərsiniz.

Mərkəzləşdirilmiş İş Axınları

Mərkəzləşdirilmiş sistemlərdə sadəcə bir əməkdaşlıq modeli mövcuddur - mərkəzləşdirilmiş iş workflow. Bir mərkəzi nöqtə ya da depo, kodu qəbul edə bilir və o zaman hər kəs öz işini onunla sinxronizasiya edir. Bir sıra developerlər düyünlərdir, yəni o mərkəzin istehlakçılarıdırlar və mə rkəzləşdirilmiş yer ilə sinxronizasiya edirlər.

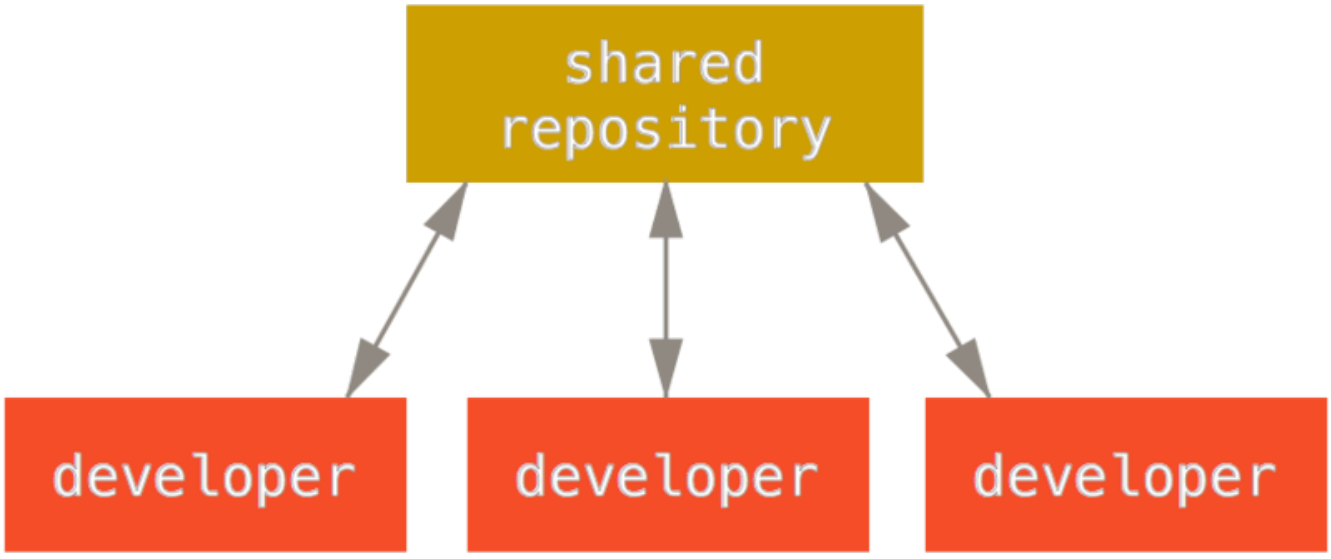


Figure 54. Mərkəzləşdirilmiş İş Axınları

Bu o deməkdir ki, iki developer mərkəzdən klonlanırsa və hər ikisi də dəyişiklik edirsə, dəyişiklikl ərini geri göndərən ilk tərtibatçı bunu heç bir problem olmadan edə bilər. İkinci developer d

əyişiklikləri qeyd etməzdən əvvəl birincinin işində birləşməlidir ki, ilk developerin dəyişikliklərini təkrar yazmasın. Bu anlayış Git-də Subversion (və ya hər hansı CVCS) olduğu kimi doğrudur və bu model Git'də də əla işləyir.

Şirkətinizdə və ya komandanızda mərkəzləşdirilmiş bir workflow ilə onsuz da rahatsınızsa, Git ilə bu iş istifadəsini asanlıqla davam etdirə bilərsiniz. Sadəcə bir depo qurun və komandanızdakı hər kəsə push imkanı verin; bu zaman Git istifadəçilərin bir-birinin üstünə təkrar yazmasına imkan vermir.

Fərz edək ki, John və Jessica eyni anda işə başlayır. John dəyişiklikini bitirib serverə yükləyir. Sonra Jessica dəyişikliklərini yükləməyə çalışır, lakin server onları rədd edir. Ona sürətli olmayan irəli dəyişiklikləri etməyə çalışdığını və qoşulub birləşməyincə edə bilməyəcəyi deyildi. Bu workflow bir çox insanı cəlb edir, çünki o bir çoxunun tanıdığı və rahat olduğu bir paradiqmadır.

Bu həm də kiçik komandalarla məhdudlaşmır. Git-in branch modeli yüzlərlə developerin eyni vaxtda onlarla branch vasitəsilə bir proyekt üzərində uğurla çalışmasını mümkün edir.

İnteqrasiya-Menecer İş Axınları

Git birdən çox uzaq depolarınıza sahib olmağa imkan verdiyi üçün, hər bir developerin öz şəxsi depolarına yazmaq və hər kəsin girişlərini oxumaq üçün bir workflow əldə etməsi mümkündür. Bu ssenariyə tez-tez "official" layihəsini təmsil edən bir kanonik bir depo daxildir. O proyektə dəst ək vermək üçün proyektin öz ictimai klonunu yaradırsınız və dəyişikliklərinizi ona istiqamətl əndirirsiniz. Sonra dəyişikliklərinizi çəkmək üçün əsas layihənin qoruyucusuna sorğu göndərə bil ərsiniz. Ardından saylayıcı, depo saxlayan yerinizi məsafədən əlavə edə bilər, dəyişikliklərinizi yerli olaraq sınayır, onları branch-lara birləşdirə və geri depolarına push edə bilər. Proses aşağıdakı kimi işləyir (İnteqrasiya-menecer İş Axınları-a bax):

1. Layihə qoruyucusu public depolarına push edir.
2. Bir dəstəkçi həmin deponu klonlayır və dəyişikliklər edir.
3. Dəstəkçi öz public kopyasını daxil edir.
4. Dəstəkçi, düzəlişlərin alınmasını xahiş edən bir e-poçt göndərir.
5. Təminatçı dəstəkçinin depolarını uzaqdan əlavə edir və yerli olaraq birləşdirir.
6. Təminatçı birləşmiş dəyişiklikləri əsas depoya daxil edir.

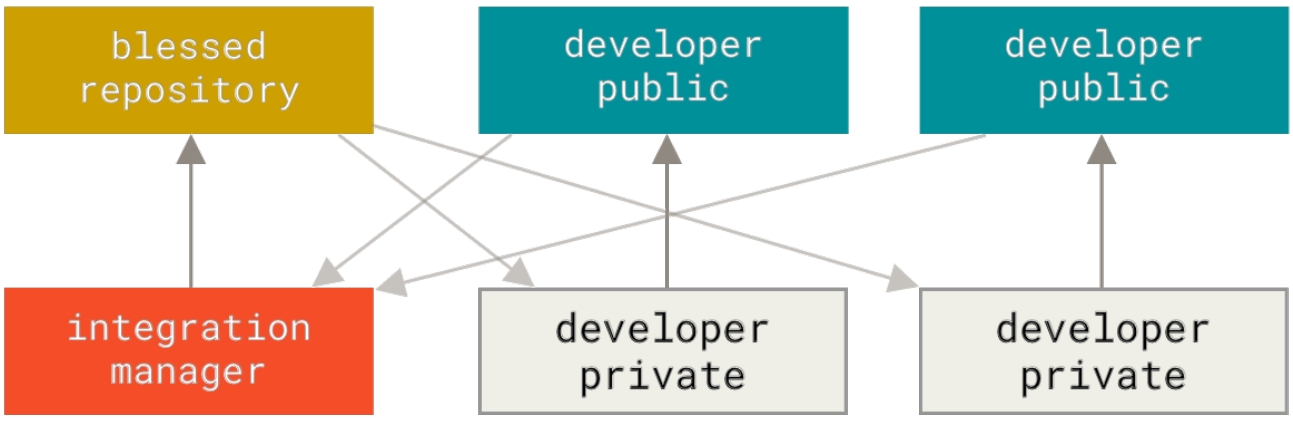


Figure 55. İnteqrasiya-menecer İş Axınları

Bu GitHub və ya GitLab kimi hub əsaslı alətlər proyekti forking etmək və dəyişikliklərinizi hamının görə bilməsi üçün fork etmək asan olduğundan çox istifadə olunanlardandır. Bu yanaşmanın əsas üstünlüklərindən biri odur ki, siz işləməyə davam edərkən əsas depo təminatçısı hər an də yişikliklərinizi daxil edə bilər.

İştirakçılar layihənin dəyişikliklərini daxil etməsini gözləməli deyil - yəni, hər tərəf öz sürətiylə işl əyə bilər.

Diktator və Leytenantların İş Axınları

Bu, çoxsaylı depozit bir workflow variantıdır. O, ümumiyyətlə yüzlərlə tərəfdaş ilə birgə nəhəng layihələr tərəfindən istifadə olunur və onun məşhur nümunələrindən biri Linux kernelidir. Müxt əlif inteqrasiya menecerləri depoların müəyyən hissələrinə cavabdehdirlər; onlara leytenantlar deyilir. Bütün leytenantların xeyirxah diktator kimi tanınan bir inteqrasiya meneceri var. Xeyirxah diktator öz direktoriyalarından bütün əməkdaşların çəkməli olduğu bir istinad depolarına gönd ərir. Proses belə işləyir (Xeyirxah diktator İş Axınları-a bax):

1. Daimi tərtibatçılar mövzu branch-ları üzərində işləyir və işlərini `master`-in üstünə qoyurlar. `master` branch diktatorun göndərdiyi istinad anbarıdır.
2. Leytenantlar, developerlərin mövzu şöbələrini öz `master` branch-larına birləşdirirlər.
3. Diktator leytenantların `master` branch-larını diktatorun üst branch-na birləşdirir.
4. Son olaraq, diktator o `master` bu branch-ı arayış depozitinə göndərir ki, digər tərtibatçılar bunun üzərində yenidən yaza bilsinlər.

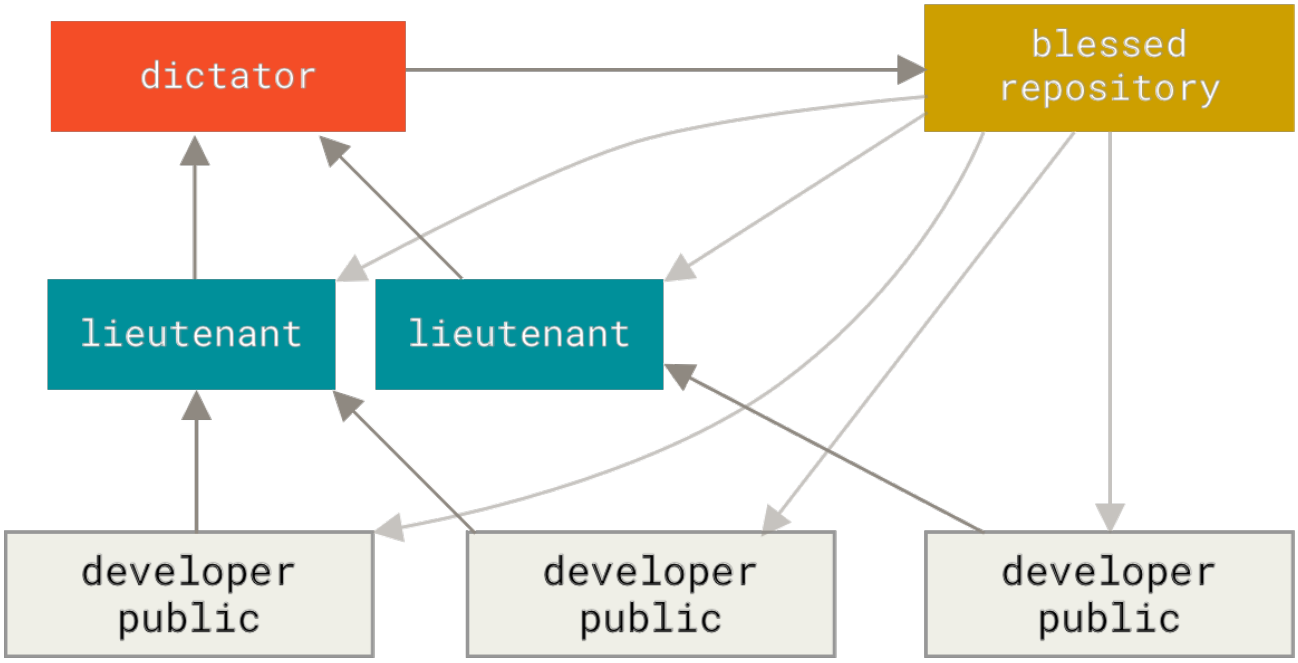


Figure 56. Xeyirxah diktator İş Axınları

Bu cür woekflow çox da işlək deyil, lakin çox böyük layihələrdə və ya yüksək iyerarxik mühitlərdə faydalı ola bilər. Bu layihə rəhbərinə (diktatora) işin çox hissəsini həvalə etməyə və onları birl əşdirmədən əvvəl çox nöqtədə böyük kod toplamağa imkan verir.

Mənbə Kodu Branch'larının İdarə Nümunələri

Martin Fowler "Mənbə kodu filiallarını idarə etmək üçün nümunələr" kitabçası hazırlamışdır. Bu təlimat bütün ümumi Git workflowlarını əhatə edir və onlardan necə istifadə edəcəyinizi izah edir. Orada həmçinin yüksək və aşağı inteqrasiya tezliklərini müqayisə edən bir bölmə də var.

https://martinfowler.com/articles/branching-patterns.html

İş Axınlarının Qısa Məzmunu

Bəzi Git kimi paylanmış bir sistemlə mümkün olan bir çox istifadə olunan workflowlar vardır, ancaq bir çox dəyişikliyin xüsusi real dünya workflowunuza uyğun olmasının mümkün olduğunu görə bilərsiniz. İndi (ümid edirik ki) hansı iş axını birləşməsinin sizin üçün işləyə biləcəyini müə yyənləşdirə bildiyiniz üçün, müxtəlif axınları təşkil edən əsas rolları necə yerinə yetirəcəyinizə dair daha bir neçə misal göstərəcəyik. Növbəti hissədə bir layihəyə dəstək vermək üçün bir neçə ümumi nümunə haqqında məlumat əldə edəcəksiniz.

Layihəyə Töhfə vermək

Bir layihəyə necə töhfə verəcəyinizi izah etməyin əsas çətinliyi bunu necə etmək barədə olan çoxsaylı dəyişikliklərdir. Git çox çevik olduğu üçün insanlar bir çox cəhətdən birlikdə işləyə bilər və necə töhfə verəcəyinizi izah etmək problemlidir - hər layihə bir az fərqlidir. İştirak edən bəzi d əyişənlər aktiv töhfə sayı, seçilmiş iş axını, commit girişiniz və bəlkə də xarici töhfə metodudur.

Birinci dəyişən aktiv iştirakçı sayı - bu layihəyə neçə istifadəçi kodu töhfə edir və neçə dəfə? Bir çox hallarda, gündə bir neçə layihə üçün iki və ya üç developer və ya daha az hərəkətsiz layihə üçün daha az developer lazım olacaq. Daha böyük şirkətlər və ya layihələr üçün developer-lərin sayı minlərlə ola bilər, hər gün yüzlərlə və ya minlərlə commit gəlir. Bu vacibdir, çünki daha çox developerlə kodunuzun təmiz tətbiq olunduğuna və ya asanlıqla birləşdirilə biləcəyinə əmin olmaq üçün daha çox problem qarşılaşırsınız. Təqdim etdiyiniz dəyişikliklər işlədiyiniz müddətdə birl əşdirilmiş və ya dəyişikliklərinizin təsdiqlənməsini və tətbiq olunmasını gözlədiyi zaman köhn əlmiş və ya ciddi şəkildə pozulmuş ola bilər. Kodunuzu həmişə aktual və commit-lərinizi necə keç ərli saxlaya bilərsiniz?

Növbəti dəyişən, layihə üçün istifadə olunan iş axınlarıdır. Hər bir developerin əsas kod xəttinə b ərabər yazılı girişi olması mərkəzləşdirilib? Layihədə bütün patch-ları yoxlayan bir qoruyucu və ya inteqrasiya meneceri varmı? Bütün patch-lar araşdırılıb təsdiqlənibmi? Bu müddətdə iştirak edirsiniz? Leytenant bir sistem var və ilk növbədə işinizi onlara təqdim etməlisiniz?

Növbəti dəyişən commit girişinizdir. Bir layihəyə töhfə vermək üçün tələb olunan iş axını, layihəyə yazılı girişiniz varsa, etmədiyinizdən daha çox fərqlidir. Yazı icazəniz yoxdursa, layihə töhfə olunan işləri necə qəbul etməyi üstün tutur? Hətta bir politikası varmı? Bir anda nə qədər işə töhfə verirsiniz? Nə qədər tez-tez iştirak edirsiniz?

Bütün bu suallar bir layihəyə necə təsirli bir şəkildə töhfə verdiyinizə və hansı iş axınının sizin üçün tərcih edilib istifadə edə biləcəyinizə təsir edə bilər. Bunların hər birinin aspektlərini sadə v əziyyətdən daha mürəkkəbinə keçərək bir sıra istifadə hallarında əhatə edəcəyik; praktikada

ehtiyac duyduğunuz xüsusi iş axınlarını bu nümunələrdən qura bilməlisiniz.

Commit Guidelines

Xüsusi istifadə hallarına baxmağa başlamazdan əvvəl commit mesajları haqqında qısa bir qeyd ed ək. Commit yaratmaq və buna bağlı qalmaq üçün yaxşı guideline-a sahib olmaq Git və başqları ilə işləməyi daha çox asanlaşdırır. Git layihəsi, patch-lar göndərilməsini təmin edən bir sıra yaxşı tövsiyələri özündə cəmləşdirən bir sənəd təqdim edir - onu Git mənbə kodundan `Documentation/SubmittingPatches` sənədində oxuya bilərsiniz.

Birincisi, təqdimatlarınızda boşluq xətaları olmamalıdır. Git bunu yoxlamaq üçün asan bir yol t əqdim edir - commit etməzdən əvvəl, boşluq xətalarını müəyyənləşdirən və onları sizin üçün siyahıya salan `git diff --check` əmrini tətbiq edin.

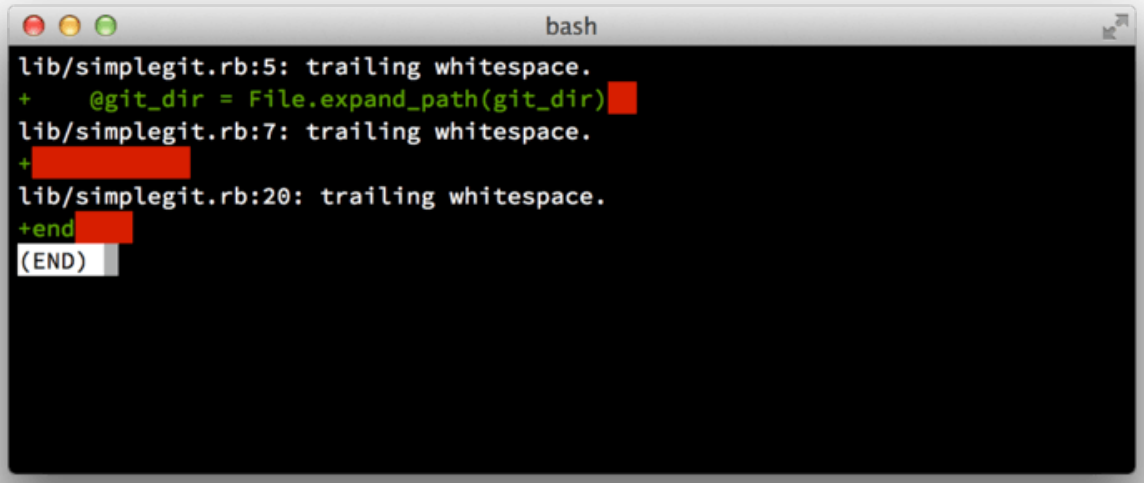


Figure 57. `git diff --check` nəticəsi

Commit etmədən əvvəl bu əmri işlədirsinizsə, digər developerləri qıcıqlandıra biləcək boşluq məs ələlərini etməyiniz barədə edib etməycəyinizi anlaya bilərsiniz.

Sonrasında, hər bir commiti məntiqi olaraq ayrı bir dəyişiklik etməyə çalışın. Əgər edə bilsəniz, d əyişikliklərinizi həzm oluna bləcək hala çevirməyə çalışın - bütün həftə sonu üçün beş fərqli mövzu kodlamayın və sonra hamısını bazar ertəsi bir massiv commit kimi təqdim edin. Həftə sonu commit etməsəniz belə, bazar ertəsi günü işinizi hər mövzu üçün ən azı bircommit etmək üçün hazırlıq sahəsindən istifadə edin və hər commit üçün faydalı bir mesaj əlavə edin. Bəzi dəyişiklikl ər eyni faylı dəyişdirirsə, qismən sənədləşdirmək üçün `git add --patch` istifadə edin(Ətraflı detallar Interaktiv Səhnələşdirmə-da əhatə olunub). Branch-ın ucundakı layihə snapshotu bir və ya beş dəyişiklik etməyinizlə eynidir. Buna görə də, dəyişikliklər bir anda əlavə olunduğundan developerlərin işini asanlaşdırmaq üçün dəyişikliklərinizi nəzərdən keçirməyə çalışın.

Bu yanaşma daha sonra ehtiyacınız olduqda dəyişikliklərdən birini çıxarmağı və ya geri qaytarmağı da asanlaşdırır. Tarixi Yenidən Yazmaq tarixin yenidən yazılması və interaktiv şəkildə qurulması üçün bir sıra faydalı Git tövsiyələrini təsvir edir - bu vasitələrdən istifadə edərək başqasına göndərməzdən əvvəl təmiz və başa düşülən bir tarixi hazırlamaqda istifadə edin.

Unudulmamalı olan son şey commit mesajıdır. Keyfiyyətli commit mesajlar yaratmaq vərdişinə yiyələnmək Git ilə işləməyi asanlaşdırır. Bir qayda olaraq, mesajlarınız təxminən 50 işarədən çox olmayan bir dəyişikliyi qısa təsvir edən, boş bir xətt izləyən və daha ətraflı izahat verilən bir xətt ilə başlamalıdır. Git layihəsi tələb edir ki, daha ətraflı izahat dəyişiklik üçün motivasiyanızı daxil etsin və həyata keçirilməsini əvvəlki davranışla müqayisə etsin - bu əməl etmək üçün yaxşı bir t əlimatdır. Commit mesajınızı əmr qutusuna yazın: "Fix bug","Fixed bug" və ya "Fixes bug." deyil. Budur, Tim Pope tərəfindən yazılmış original məqalə:

```
Capitalized, short (50 chars or less) summary

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so.  In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body.  The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.

Write your commit message in the imperative: "Fix bug" and not "Fixed bug"
or "Fixes bug."  This convention matches up with commit messages generated
by commands like git merge and git revert.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, followed by a
  single space, with blank lines in between, but conventions vary here

- Use a hanging indent
```

Bütün commit mesajlarınız bu modeli təqib edərsə, sizin və əməkdaşlıq etdiyiniz developer-lər üçün işlər çox asan olacaqdır. Git layihəsi yaxşı formatlanmış commit mesajlarına malikdir - qəş əng formatlı bir layihə commit-i tarixinin necə olduğunu görmək üçün orada `git log --no-merges` işlətməyə çalışın.

Dediyimiz kimi edin, etdiyimiz kimi yox.

Qısaca demək lazımdırsa, bu kitabdakı nümunələrin çoxunda bunun kimi yaxşı işlənmiş commit mesajları yoxdur; bunun yerinə `git commit` üçün `-m` seçimi istifadə edirik.

Bir sözlə, etdiyimiz kimi deyil, dediyimiz kimi edin.

Private Kiçik Komanda

Qarşınıza gələ biləcəyiniz ən sadə quraşdırma, bir və ya iki developeri olan xüsusi bir layihədir. Bu kontekstdə “private,”, qapalı mənbə anlamına gəlir - xarici dünya üçün əlçatmazdır. Siz və digər developerlərinizin hamısının depoya girmə imkanı var.

Bu mühitdə Subversion və ya başqa bir mərkəzləşdirilmiş sistem istifadə edərkən edə biləcəyinizə bənzər bir iş axını izləyə bilərsiniz.

Siz hələ də oflayn işləmə və olduqca sadə branch-lara ayırma və birləşmə kimi şeylərin üstünlükl ərini əldə edirsiniz, amma iş axını çox oxşar ola bilər; Əsas fərq, birləşmələrin törədildiyi vaxt serverdə deyil, müştəri tərəfində baş verməsidir. İki developerin ortaq bir depo ilə birlikdə işə başladıqda nə görünə biləcəyinə baxaq. İlk developer John depolarını klonlaşdırır, dəyişiklik edir və local olaraq yerinə yetirir. Protokol mesajları bu nümunələrdə bir qədər qısaldılması üçün `...` ilə əvəz edilmişdir.

```
# John's Machine
$ git clone john@githost:simplegit.git
Cloning into 'simplegit'...
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'Remove invalid default value'
[master 738ee87] Remove invalid default value
 1 files changed, 1 insertions(+), 1 deletions(-)
```

İkinci developer Jessica da eyni şeyi edir - depoları klonlaşdırır və dəyişiklik edir:

```
# Jessica's Machine
$ git clone jessica@githost:simplegit.git
Cloning into 'simplegit'...
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'Add reset task'
[master fbff5bc] Add reset task
 1 files changed, 1 insertions(+), 0 deletions(-)
```

İndi Jessica öz işini yalnız yaxşı işləyən serverə push edir:

```
# Jessica's Machine
$ git push origin master
...
To jessica@githost:simplegit.git
   1edee6b..fbff5bc  master -> master
```

Yuxarıdakı çıxışın son sətri push əməliyyatından faydalı bir geri göndərmə mesajını göstərir. Əsas format <oldref>..<newref> fromref → toref` şəklindədir. Burada `oldref` köhnə referans mənasına gəlir, `newref` yeni referans mənasına gəlir, `fromref` push edilən local referansdır və `toref` yenilənən uzaq referansın adıdır. Müzakirələrdə aşağıdakı kimi oxşar nəticəni görəcəksən, buna görə məna haqqında əsas təsəvvürə sahib olmaq, depoların müxtəlif vəziyyətlərini anlamağa kömək edəc əkdir. Daha çox detallı məlumat üçün git-push-a baxın.

Bu nümunə ilə davam edək, qısa müddətdən sonra John bəzi dəyişikliklər edir, onları local depolarına commit edir və eyni serverə push etməyə çalışır:

```
# John's Machine
$ git push origin master
To john@githost:simplegit.git
 ! [rejected]        master -> master (non-fast forward)
error: failed to push some refs to 'john@githost:simplegit.git'
```

Bu vəziyyətdə, John'un push etməsi Jessica'nın dəyişikliklərini daha əvvəl itələməsinə görə uğursuz olur. Subversiya anlamaq xüsusilə vacibdir, çünki iki developerin eyni faylı düzəltm ədiyini görəcəksiniz. Müxtəlif fayllar Git ilə düzəldilibsə Subversion avtomatik olaraq serverdə belə bir birləşmə əmələ gətirsə də, local əməliyyatları birləşdirməlisiniz. Başqa sözlə, John əvvəlcə Jessica'nın yuxarıdakı dəyişikliklərini götürməli və push etməyə icazə verilməzdən əvvəl onları local depolarına birləşdirməlidir.

İlk addım olaraq John Jessica'nın işini fetch edir (bu yalnız *fetches* Jessica'nın yuxarı işidir, hələ onu John'un işinə birləşdirmir):

```
$ git fetch origin
...
From john@githost:simplegit
 + 049d078...fbff5bc master     -> origin/master
```

Bu halda John'un local deposu bu kimi bir şeyə bənzəyəcəkdir:

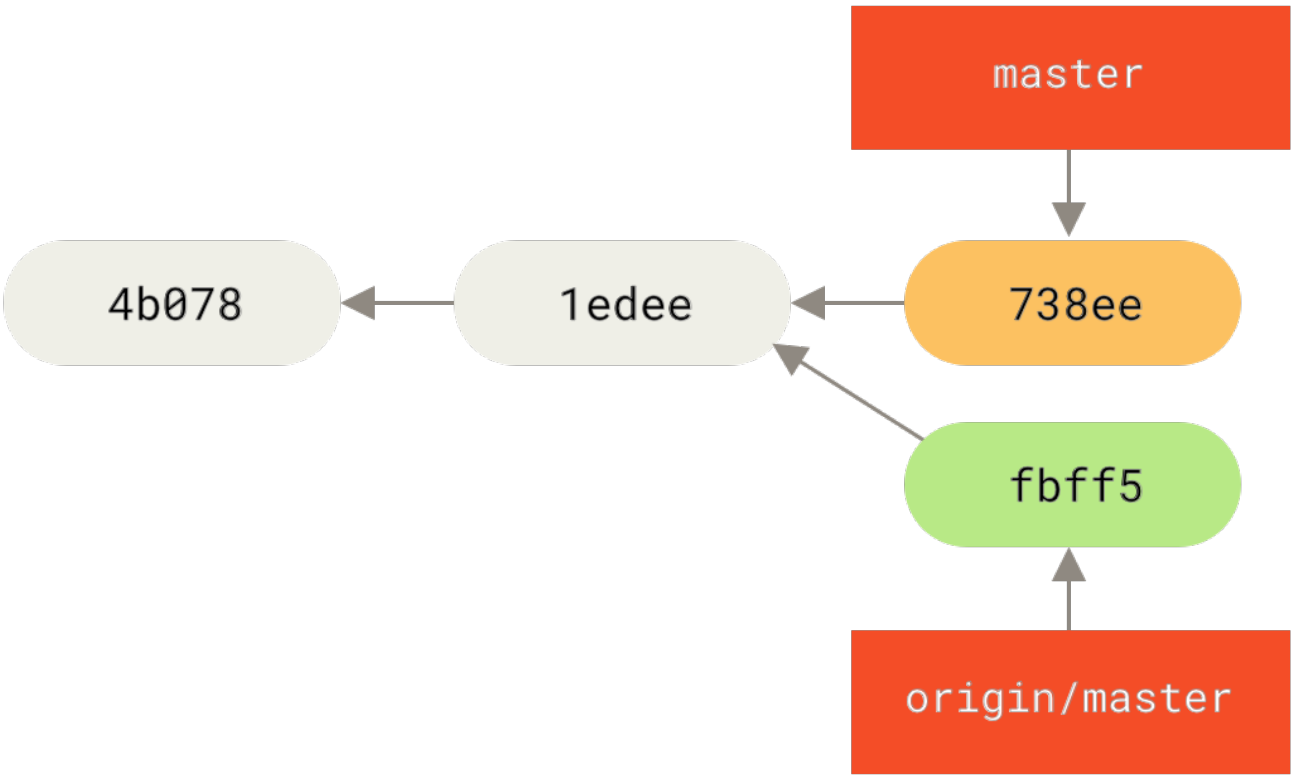


Figure 58. John'un tarix ayrılığı

İndi John Jessica'nın öz local işinə apardığı işləri birləşdirə bilər:

```
$ git merge origin/master
Merge made by the 'recursive' strategy.
 TODO |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Yerli birləşmə rəvan getdikcə Johnun yenilənmiş tarixi indi belə görünəcək:

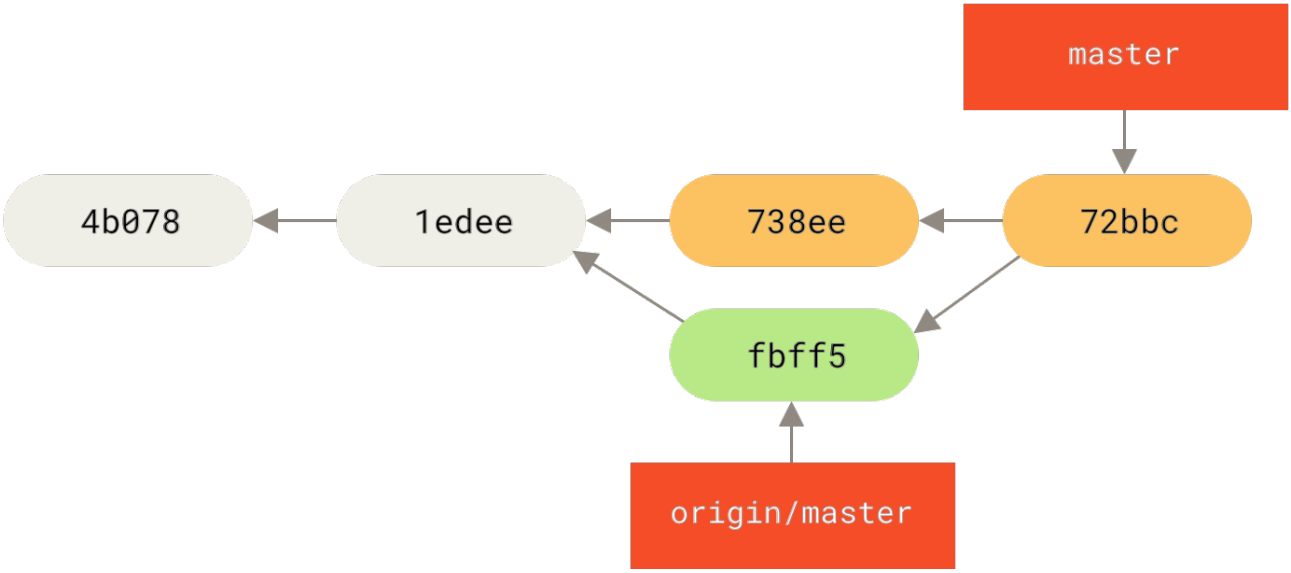


Figure 59. John'un deposu `origin/master` *birləşdirdikdən sonra*

Bu anda John Jessica'nın heç bir işinin onun hər hansı birinə təsir etmədiyindən əmin olmaq üçün bu yeni kodu sınamaq istəyə bilər və hər şeyin yaxşı göründüyü müddətcə nəhayət yeni birl əşdirilmiş işi serverə push edə bilər:

```
$ git push origin master
...
To john@githost:simplegit.git
   fbff5bc..72bbc59  master -> master
```

Sonda John'un commit tarixi belə görünəcək:

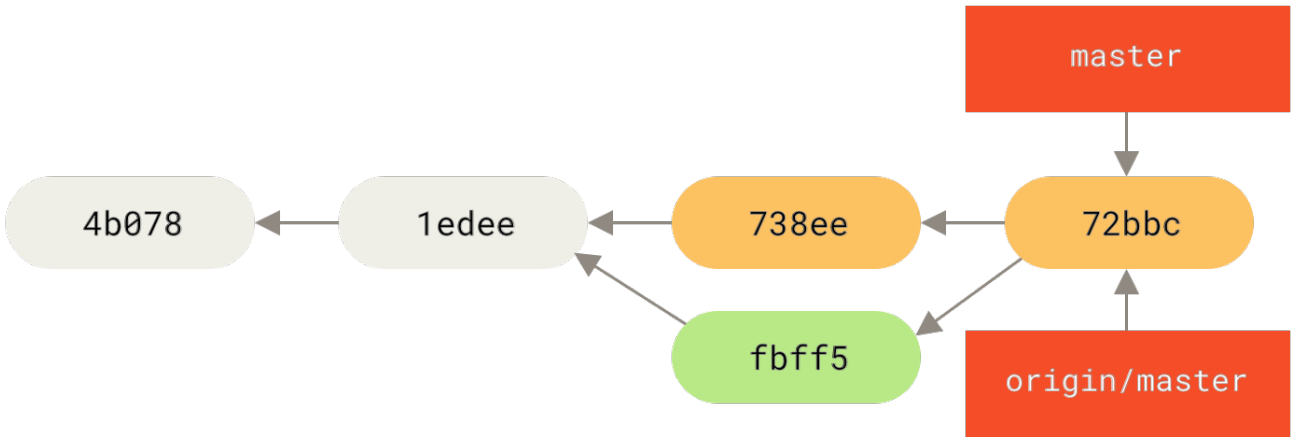


Figure 60. `origin` serverə pushing etdikdən sonra John'un tarixi

Bu vaxt, Jessica `issue54` adlı yeni bir mövzu branch-ı yaratdı və bu branch üçün üç commit yaratdı. O, John'un dəyişikliklərini fetch etmədi, buna görə də commit tarixi bu kimi görünür:

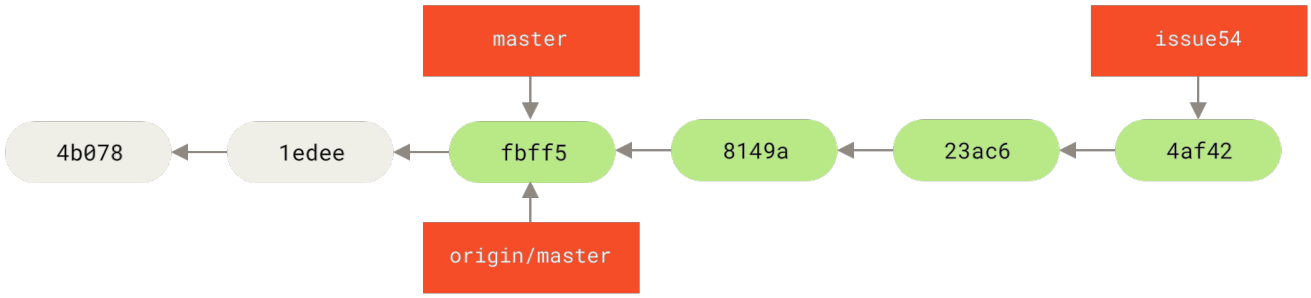


Figure 61. Jessica'nın mövzu branch-ı

Birdən, Jessica John'un serverə yeni bir iş sövq etdiyini və ona bir nəzər salmaq istədiyini öyrənir və buna görə hələ də olmayan serverdən bütün yeni məzmunu ala bilir:

```
# Jessica's Machine
$ git fetch origin
...
From jessica@githost:simplegit
   fbff5bc..72bbc59  master     -> origin/master
```

Bu vaxt John-un pushed up etdiyi işi pulls down edir. Jessica'nın tarixi indi belə görünür:

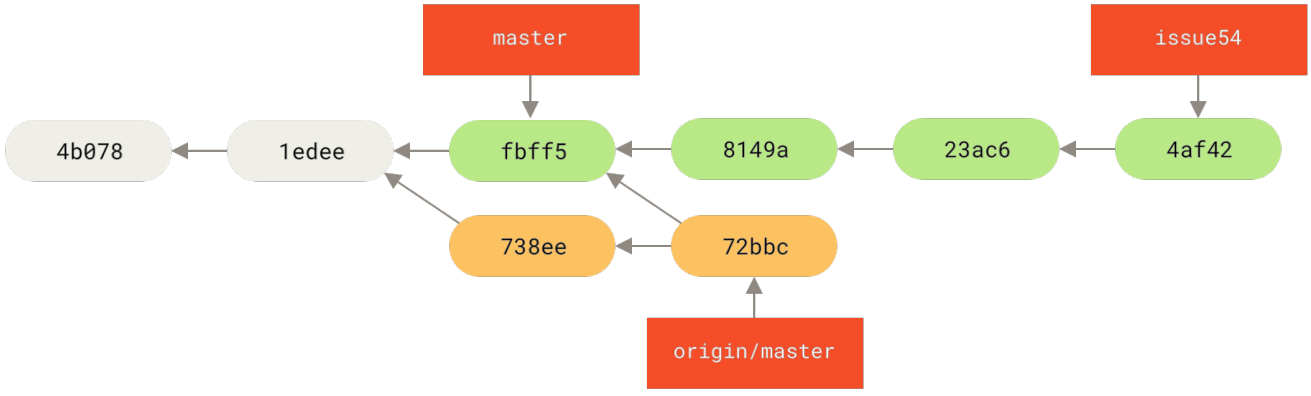


Figure 62. John dəyişikliklərini aldıqdan sonra Jessica'nın tarixi

Jessica onun mövzu branch-nın hazır olduğunu düşünür, ancaq John-un aldığı işin hansı hissəsini işinə birləşdirməli olduğunu bilmək istəyir. Bunu tapmaq üçün `git log` işlədir:

```
$ git log --no-merges issue54..origin/master
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

   Remove invalid default value
```

`issue54..origin/master` sintaksis Git-dən yalnız sonuncu branch-da olan commit-ləri göstərməyi tələb edən bir log filtridir (bu vəziyyətdə `origin/master`) bunlar ilk branch-da yoxdur (bu vəziyyətdə `issue54`). Daha ətraflı bu sintaksisi Commit Aralıqları-da əldə edə bilərsiniz.

Yuxarıdakı çıxışdan, John'un Jessica'nın local işinə birləşməməsini təmin etdiyi bir commit-in olduğunu görə bilərik. `origin/master` ilə birləşirsə, bu, local işini dəyişdirəcək yeganə commit-dir.

İndi Jessica mövzu işini özünün `master` branch-a birləşdirə bilər, John'un işini (`origin/master`) `master` branch-a birləşdirə bilər və sonra yenidən serverə push edə bilər.

Birincisi (bütün işlərini "problem54" mövzu branch-ı üzərində işləmiş), Jessica bütün bu işə inteqrasiya etməyə hazırlaşaraq yenidən `master` branch-a keçir:

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

Jessica əvvəlcə `origin/master` və ya `issue54` branch-nı birləşdirə bilər — hər ikisi upstream olduğu üçün sıra heç bir əhəmiyyət kəsb etmir. Son snapshot hansı seçimi seçməsindən asılı olmayaraq eyni olmalıdır; yalnız tarix fərqli olacaq. Əvvəlcə `issue54` branch-nı birləşdirməyi seçir:

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README           |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

Heç bir problem yaranmır; göründüyü kimi sadə və sürətli birləşmə idi. İndi Jessica John'un `origin/master` branch-da oturan John'un əvvəllər əldə edilmiş işlərini birləşdirərək local birləşmə prosesini tamamlayır.

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by the 'recursive' strategy.
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Hər şey təmiz birləşir və Jessica'nın tarixi indi belə görünür:

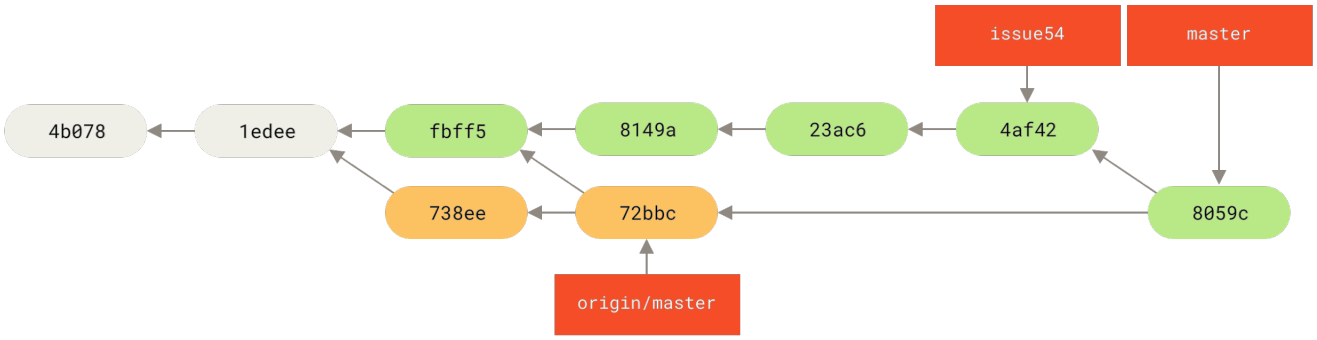


Figure 63. John dəyişikliklərini birləşdirdikdən sonra Jessica'nın tarixi

İndi `origin/master` Jessica'nın `master` branch-dan əldə edilə bilər, buna görə uğurla push etməyi bacarmalıdır (Johnun bu vaxt başqa dəyişiklik etmədiyini güman edərək):

```
$ git push origin master
...
To jessica@githost:simplegit.git
   72bbc59..8059c15  master -> master
```

Hər bir developer bir neçə dəfə commit etdi və bir-birinin işlərini uğurla birləşdirdi.

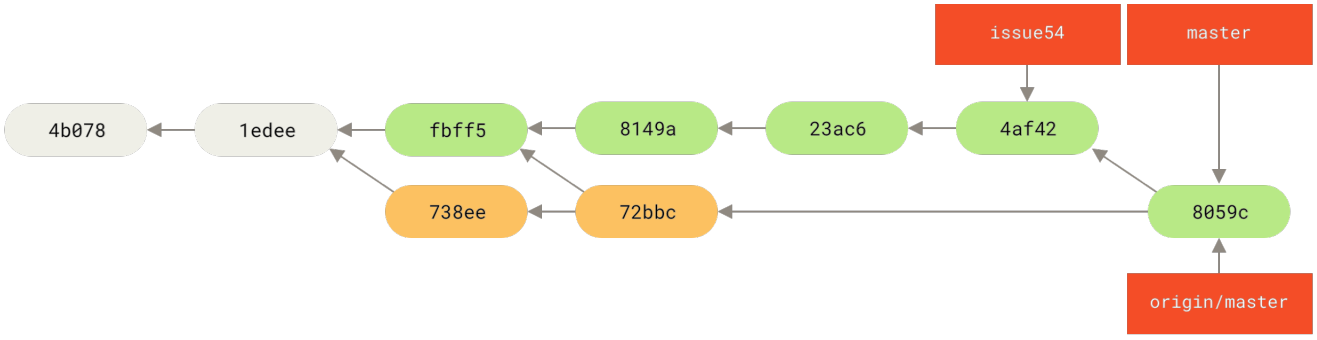


Figure 64. Bütün dəyişiklikləri serverə push etdikdən sonra Jessica'nın tarixi

Bu ən sadə iş axınlarından biridir. Bir müddət (ümumiyyətlə bir mövzu branch-da) işləyəcəksiniz və bu işi inteqrasiya olunmağa hazır olduqda `master` branch-nıza birləşdirəcəksiniz. Bu işi bölüşm ək istədikdə dəyişdirildiyi təqdirdə `origin/master`- dən `master`-i götürün və birləşdirin və nəhayət serverdəki `master` branch-na push edin. Ümumi ardıcıllıq belə olacaqdır:

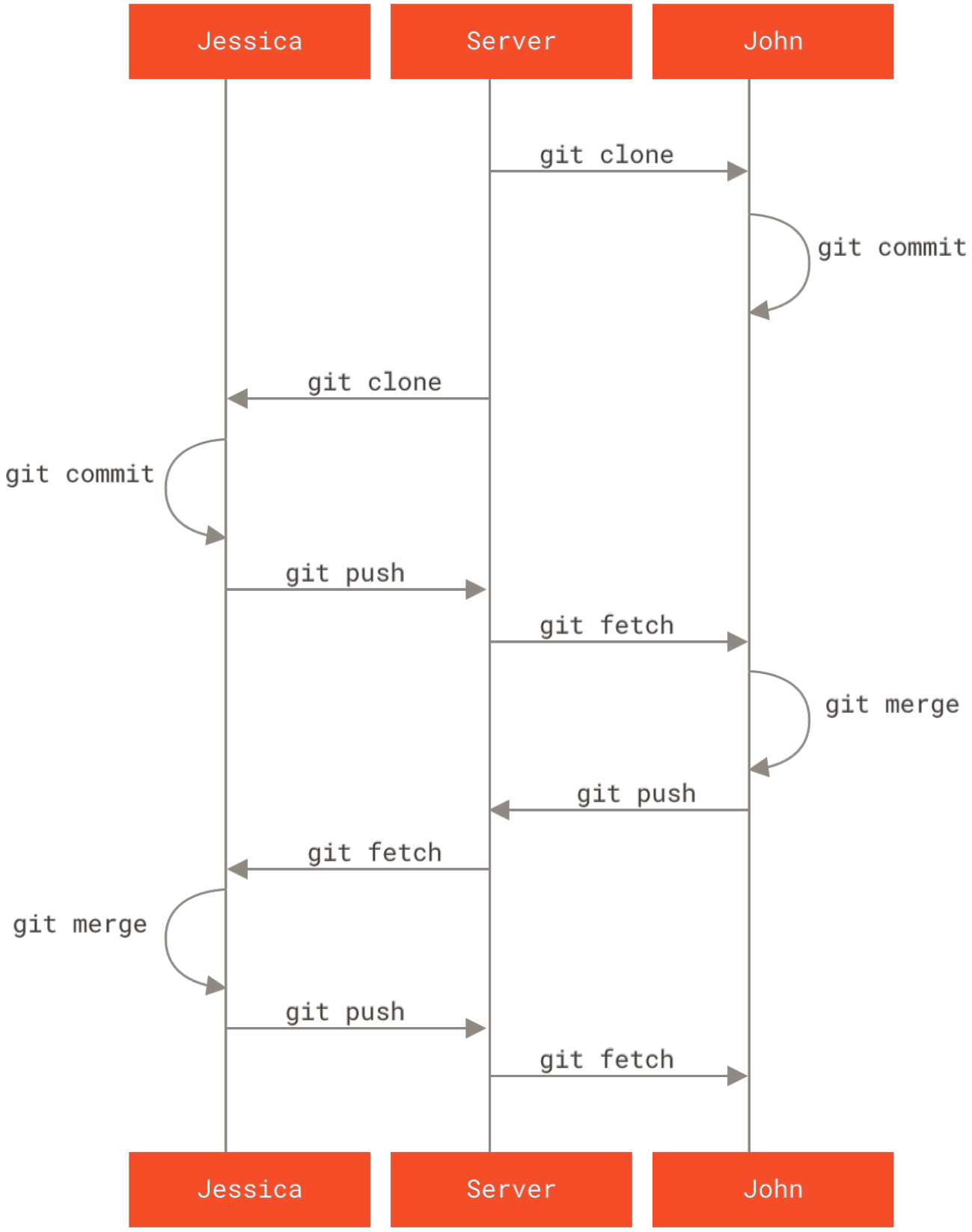


Figure 65. Sadə bir multiple-developer Git iş axını üçün hadisələrin ümumi ardıcıllığı

Private İdarə olunan Komanda

Bu növbəti ssenaridə daha böyük bir private qrupda iştirakçı rollarına baxacaqsınız. Kiçik qrupların xüsusiyyətlər üzərində əməkdaşlıq etdiyi bir mühitdə necə işləmək lazım olduğunu öyr ənəcəksiniz, bundan sonra bu komanda əsaslı töhfələr başqa tərəf tərəfindən birləşdirilir.

Deyək ki, John və Jessica bir xüsusiyyət üzərində işləyirlər (buna “featureA” deyək), Jessica və üçüncü developer Josie isə ikinci (“featureB” deyək) üzərində işləyir. Bu vəziyyətdə, şirkət ayrı-ayrı qrupların işi yalnız müəyyən mühəndislər tərəfindən inteqrasiya olunduğu və əsas reponun `master` branch-ı yalnız həmin mühəndislər tərəfindən yenilənə biləcəyi bir növ inteqrasiya-menecer iş axınından istifadə edir. Bu ssenaridə bütün işlər komanda əsaslı branch-larda aparılır və sonradan inteqratorlar tərəfindən bir yerə yığılır.

Bu mühitdə iki fərqli developer ilə paralel olaraq iki xüsusiyyəti üzərində işlədiyi üçün Jessica’nın iş axınını izləyək. Artıq öz depolarını klonlaşdırdığını düşünərək ilk olaraq `featureA` üzərində işləməyi qərara alır.

Xüsusiyyət üçün yeni bir branch yaradır və orada bəzi işlər görür:

```
# Jessica's Machine
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ vim lib/simplegit.rb
$ git commit -am 'Add limit to log function'
[featureA 3300904] Add limit to log function
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Bu anda, işini John ilə bölüşməlidir, ona görə də `featureA` branch-ı serverə commit edir. Jessica’nın `master` branch-a push etməyə icazəsi yoxdur - yalnız inteqratorlar bunu edir - ona görə də John ilə əməkdaşlıq etmək üçün başqa bir branch-a push etmək məcburiyyətindədir:

```
$ git push -u origin featureA
...
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

Jessica John’a e-poçt göndərərək `featureA` adlı bir branch-a bir iş push etdiyini və indi baxa biləcəyini yazar. John’dan rəy gözlədiyi müddətdə, Jessica Josie ilə `featureB` üzərində işləməyə qərar verdi. Başlamaq üçün o, serverin `master` branch-dan istifadə edərək yeni bir xüsusiyyət branch-na başlayır:

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch 'featureB'
```

İndi Jessica `featureB` branch-da bir neçə commit yaradır:

```
$ vim lib/simplegit.rb
$ git commit -am 'Make ls-tree function recursive'
[featureB e5b0fdc] Make ls-tree function recursive
 1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'Add ls-files'
[featureB 8512791] Add ls-files
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Jessica's deposu indi belə görünür:

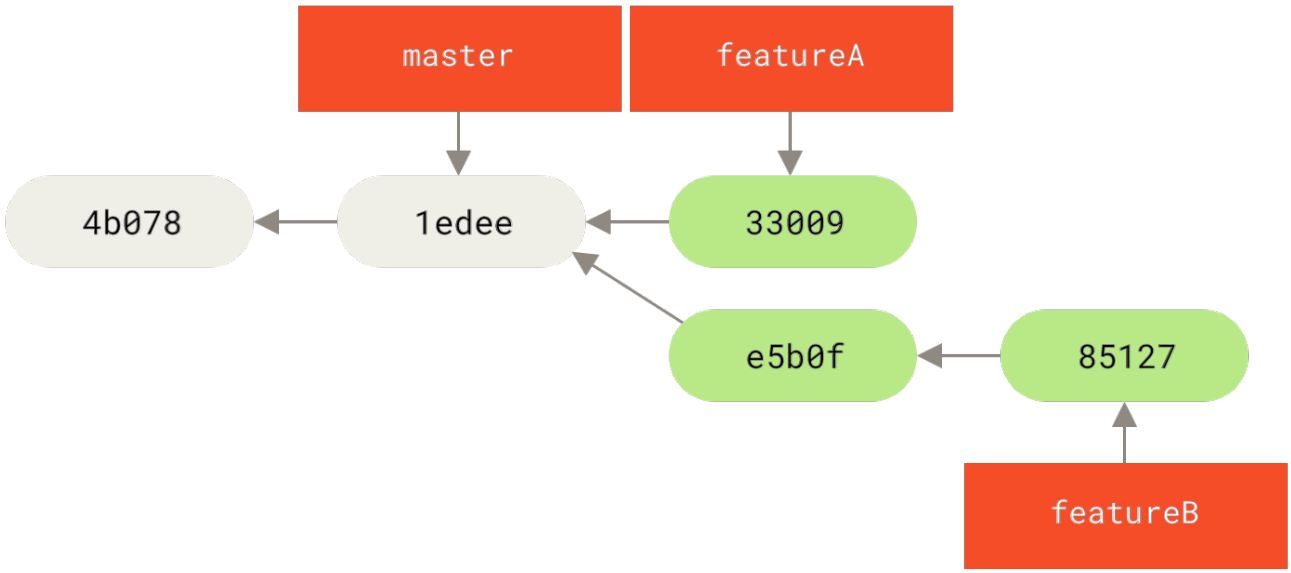


Figure 66. Jessica'nın ilkin commit tarixi

İşini push etməyə hazırdır, amma Josie-dən bir e-poçt alır ki, bunun üzərində bir neçə ilkin "featureB" işi olan bir branch artıq `featureBee` branch-ı olaraq serverə push edildi. Jessica, işini serverə push etməzdən əvvəl bu dəyişiklikləri özü ilə birləşdirməlidir. Jessica əvvəlcə Josie'nin d əyişikliklərini `git fetch` ilə qəbul edir:

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

Jessica'nın hələ yoxlanılmış `featureB` branch-ında olduğunu düşündükdə indi Josie'nin işini `git merge` ilə bu branch-a birləşdirə bilər:

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by the 'recursive' strategy.
 lib/simplegit.rb |    4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)
```

Bu anda, Jessica bu birləşdirilmiş “featureB” işinin hamısını yenidən serverə qaytarmaq istəyir, ancaq sadəcə öz `featureB` branch-nı push etmək istəmir. Əksinə, Josie artıq yuxarı bir `featureBee` branch-na başlamış olduğundan, Jessica özü ilə birlikdə etdiyi *bu* branch-a push etmək istəyir:

```
$ git push -u origin featureB:featureBee
...
To jessica@githost:simplegit.git
   fba9af8..cd685d1  featureB -> featureBee
```

Bu *refspec* adlanır. Git refspecs hnlarla edə biləcəyiniz fərqli şeylər barədə daha ətraflı müzakirə etmək üçün Refspec baxın. `-u` bayrağına da diqqət yetirin; bu, branch-ları daha sonra push və pull etmək üçün konfiqurasiya edən `--set-upstream` üçün qısaldılmış formasıdır.

Birdən, Jessica John’dan e-poçt alır, onun əməkdaşlıq etdikləri `featureA` branch-a bəzi dəyişikliklər etdiyini söyləyir və Jessica’dan onlara baxmağı xahiş edir. Yenə Jessica sadə bir *git fetch* işlədərək John’un son işi də daxil olmaqla serverdən *bütün* yeni məzmunu əldə edir:

```
$ git fetch origin
...
From jessica@githost:simplegit
   3300904..aad881d  featureA   -> origin/featureA
```

Jessica yeni gətirilən `featureA` branch-nın kontentini eyni branch-ın local kopyası ilə müqayisə edə rək John’un yeni işinə baxa bilər.

```
$ git log featureA..origin/featureA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 19:57:33 2009 -0700

    Increase log output to 30 from 25
```

Əgər Jessica gördüyünü bəyənsə, John’un yeni işini local `featureA` branch-a birləşdirə bilər:

```
$ git checkout featureA
Switched to branch 'featureA'
$ git merge origin/featureA
Updating 3300904..aad881d
Fast forward
 lib/simplegit.rb |   10 +++++++++-
1 files changed, 9 insertions(+), 1 deletions(-)
```

Nəhayət, Jessica birləşənlərin hamısına bir neçə kiçik dəyişiklik etmək istəyə bilər, buna görə də bu dəyişiklikləri etmək, local `featureA` branch-na tapşırmaq və nəticəni yenidən serverə push etmək pulsuzdur.

```
$ git commit -am 'Add small tweak to merged content'
[featureA 774b3ed] Add small tweak to merged content
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
...
To jessica@githost:simplegit.git
   3300904..774b3ed  featureA -> featureA
```

Jessica'nın commit tarixi indi belə görünəcək:

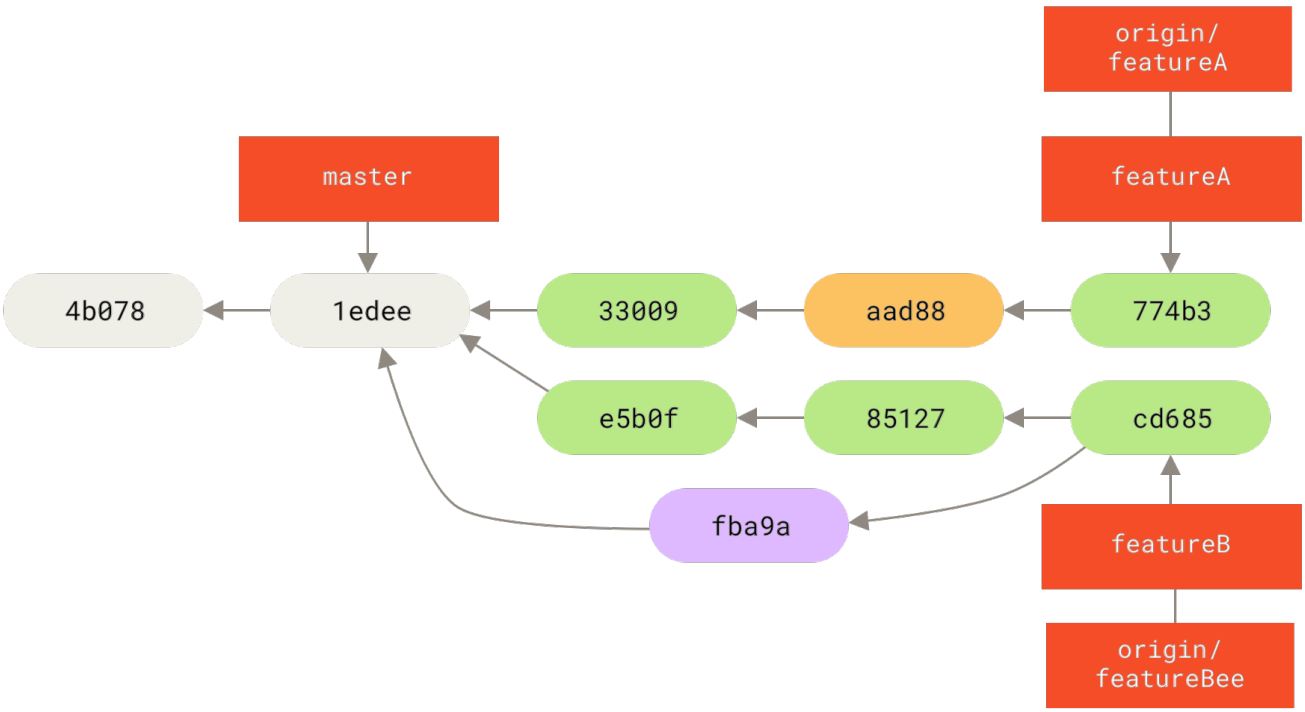


Figure 67. Jessica'nın bir xüsusiyyət branch-na commit etdikdən sonra tarixi

Bir anda Jessica, Josie və John inteqratorlara serverdəki `featureA` və `featureBee` branch-larının ana xəttə inteqrasiyaya hazır olduqlarını bildirirlər. İnteqrasiya edənlər bu branch-ları ana xəttə birl əşdirdikdən sonra getch yeni birləşdirmə commit-ni aşağı çəkəcək və tarix belə görünəcəkdir:

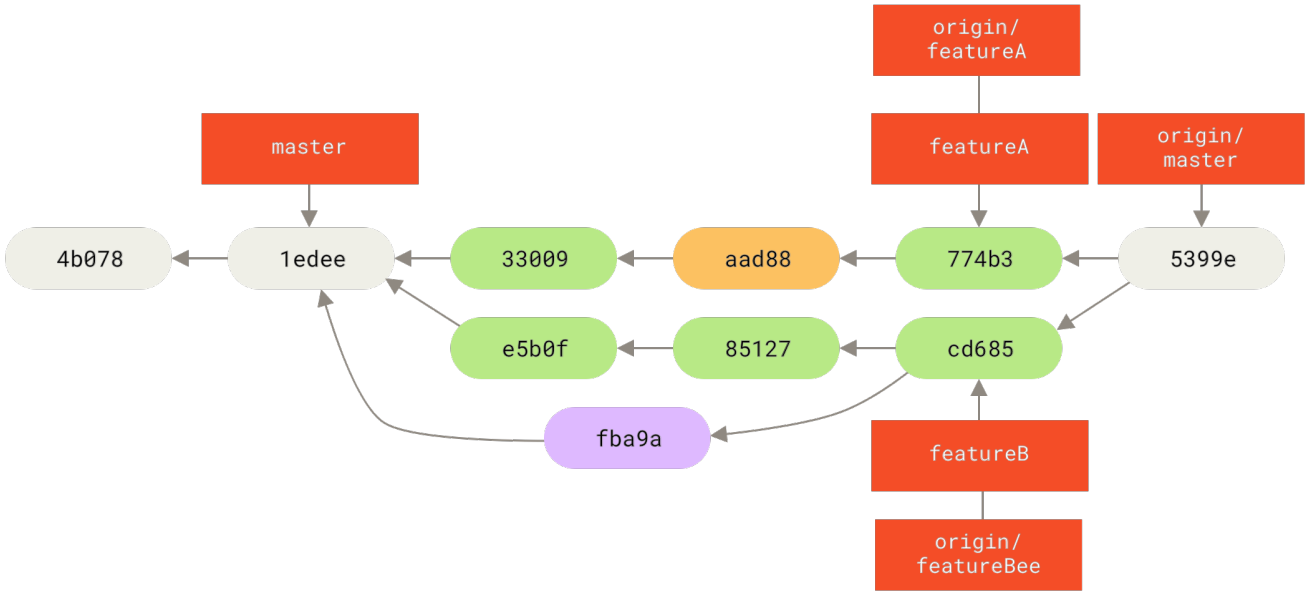


Figure 68. Hər iki mövzu branch-nı birləşdirdikdən sonra Jessica'nın tarixi

Bir çox qrup Git-ə keçdi, çünki bu müddətdə çox sayda komandanın paralel işləməsi, müddətin sonunda fərqli iş xətlərini birləşməsi mümkündür. Bir komandanın kiçik alt qruplarının bütün branch-ı cəlb etməməsi və maneə olmadan uzaq branch-larla işləmək bacarığı Git'in böyük bir faydalarından biridir. Burada gördüyünüz iş axınının ardıcıllığı belə olacaqdır:

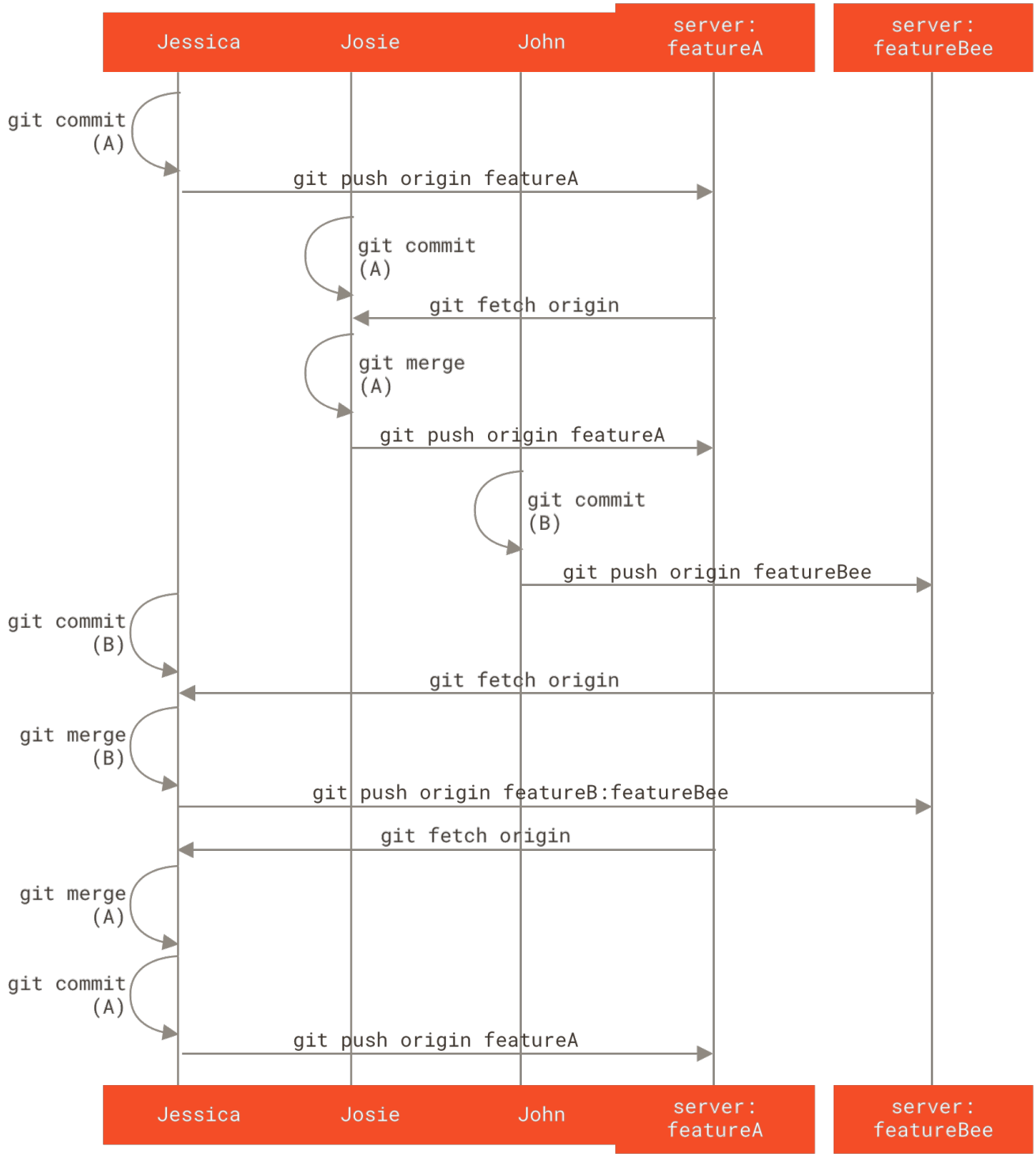


Figure 69. İdarə olunan komandanın iş axınının əsas ardıcıllığı

Forked Public Layihəsi

Public layihələrə töhfə vermək bir az fərqlidir. Layihə üzrə branch-ları birbaşa yeniləmək icazəniz olmadığına görə işinizi başqa yollarla təmirçilərə verməlisiniz. Bu ilk nümunə asan forking dəst əyini dəstəkləyən Git hostlarında forking vasitəsilə töhfəni təsvir edir. Bir çox hosting saytları (GitHub, BitBucket, repo.or.cz və başqaları daxil olmaqla) bunu dəstəkləyir və bir çox layihə aparıcısı bu töhfə tərzini gözləyirlər. Növbəti bölmə, e-poçt vasitəsilə töhfə patch-larını qəbul etm əyi üstün edən layihələr haqqındadır.

Birincisi, ehtimal ki, əsas depo klonlaşdırmaq, töhfə verməyi planlaşdırdığınız patch və ya patch

seriyası üçün bir mövzu branch-ı yaratmaq və orada işlərinizi etmək istəyəcəksiniz. Ardıcıllıq əsas ən belə görünür:

```
$ git clone <url>
$ cd project
$ git checkout -b featureA
  ... work ...
$ git commit
  ... work ...
$ git commit
```

İşinizi bir commit-ə endirmək üçün və ya təmirçinin nəzərdən keçirməsini asanlaşdırmaq məqsədi ilə işi yenidən düzəltmək üçün `rebase -i` istifadə edə bil ərsiniz - interaktiv reabilitasiya haqqında daha çox məlumat üçün Tarixi Yenidən Yazmaq-a baxın.

Branch-nızın işini bitirdikdə və onu yenidən təmin edənlərə töhfə verməyə hazır olduğunuzda, orijinal layihə səhifəsinə keçin və layihənin öz yazıla bilən fork-unu yaradaraq “Fork” düyməsini basın. Daha sonra bu depo URL-ni local depo üçün yeni bir remote kimi əlavə etməlisiniz; bu misalda onu `myfork` adlandıraq:

```
$ git remote add myfork <url>
```

Daha sonra yeni işinizi bu depoya köçürməlisiniz. Üzərində işlədiyiniz mövzu branch-nı forked deponuza push etmək, bu işi `master` branch-nızla birləşdirib onu push etmək yerinə daha asandır.

Səbəb, işinizin qəbul edilmədiyi və ya cherry-picked olduğu təqdirdə, `master` branch-nızı geri çevirməyiniz lazım deyil (Git `cherry-pick` əməliyyatına daha ətraflı Rebasing və Cherry-Picking İş Axınları baxa bilərsiniz). Əgər işçiləriniz `merge`, `rebase`, və ya `cherry-pick` işlərini görsələr, nəhayət öz depolarından pulling edərək geri qaytaracaqsınız. Hər halda işinizi aşağıdakılarla push edə bil ərsiniz:

```
$ git push -u myfork featureA
```

İşləriniz deponuzun fork-na push edildikdən sonra orijinal layihənin aparıcılarını birləşdirmək ist ədikləri iş barədə xəbərdar etməlisiniz. Buna genel olaraq *pull request* adlanır və ümumiyyətlə veb sayt vasitəsilə belə bir sorğu yaradırsınız - GitHub-un GitHub -dan keçəcəyimiz “Pull Request” mexanizmi var. - ya da `git request-pull` əmrini işlədərək sonrakı nəticəni layihə aparıcısına manual olaraq e-poçt ilə göndərə bilərsiniz.

`git request-pull` əmri mövzu branch-nızın pull edildiyini və Git depozit URL-nin pull edilməsini istədiyiniz əsas bölməni götürür və pull edilməsini xahiş etdiyiniz bütün dəyişikliklərin xülasəsini hazırlayır. Məsələn, Jessica John’a bir sorğu göndərmək istəsə və o, yalnız push etdiyi mövzu branch-da iki əmr yerinə yetirirsə, bunu edə bilər:

```
$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
Jessica Smith (1):
        Create new function

are available in the git repository at:

  git://githost/simplegit.git featureA

Jessica Smith (2):
      Add limit to log function
      Increase log output to 30 from 25

 lib/simplegit.rb |   10 +++++++++-
 1 files changed, 9 insertions(+), 1 deletions(-)
```

Bu çıxışı texniki işçiyə göndərmək olar - bu, işin haradan qaynaqlandığını, comiit-ləri ümumil əşdirir və yeni işin haradan pull ediləcəyini müəyyənləşdirir.

Təminatçı olmadığınız bir layihədə, ümumiyyətlə, `master` kimi bir branch-ın olması, `origin/master` kimi bir branch-ın olması və rədd edildiyi təqdirdə asanlıqla atıla biləcəyiniz mövzu branch-larında işlərinizi etmək daha asandır. İş temalarını mövzu branch-larına ayırmaq həm də əsas deponun ucu bu vaxt tərpənibsə və artıq təmiz tətbiq olunmadığı təqdirdə işinizi yenidən yazmağı asanlaşdırır. Məsələn, ikinci bir iş mövzusunu layihəyə təqdim etmək istəyirsinizsə, yalnız pushed up etdiyiniz mövzu şöbəsində işləməyin - əsas deponun `master` branch-ından başlayın:

```
$ git checkout -b featureB origin/master
  ... work ...
$ git commit
$ git push myfork featureB
$ git request-pull origin/master myfork
  ... email generated request pull to maintainer ...
$ git fetch origin
```

İndi mövzularınızın hər biri silos içərisindədir - patch növbəsinə bənzər - mövzuları bir-birinə qarışan və ya bir-birinə qarışmadan yenidən yaza, yenidən istifadə edə və dəyişdirə bilərsiniz:

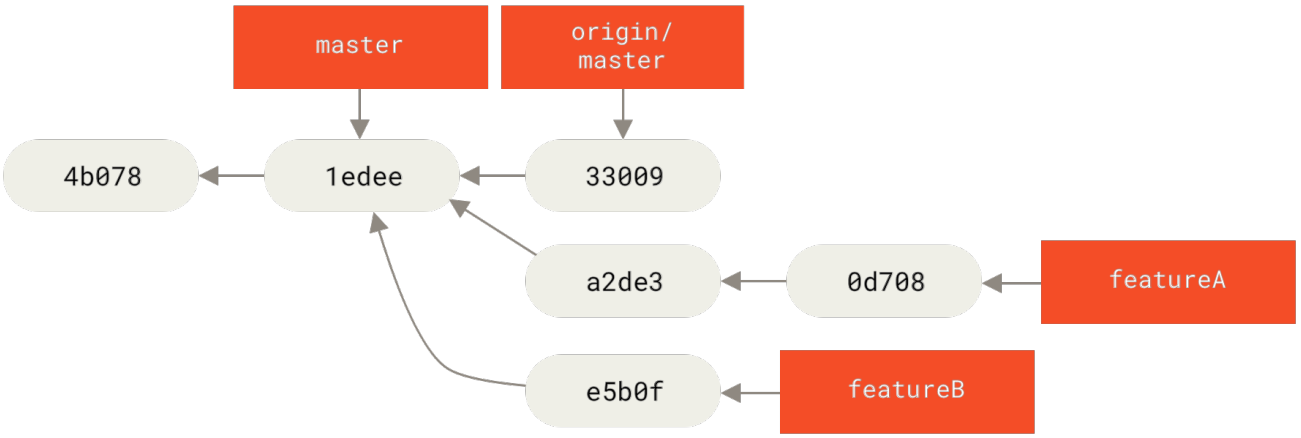


Figure 70. İlkin `featureB` işinin commit tarixi

Layihə aparıcısı bir dəstə başqa patch pulled etdi və ilk branch-nızı sınadı, amma artıq təmiz birl əşmə olmadı. Bu vəziyyətdə, bu branch-ı `origin/master`-in üstünə qaytarmağa, təmirçi üçün olan münaqişələri həll etməyə və dəyişikliklərinizi yenidən göndərməyə cəhd edə bilərsiniz:

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

İndi bu tarixiniz `featureA` işindən sonra commit tarixi kimi görünməsi üçün yenidən yazır.

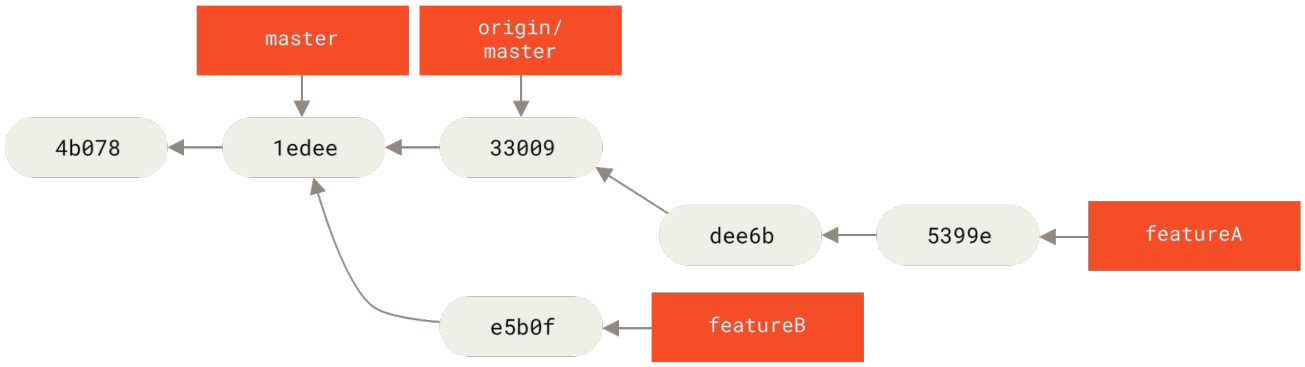


Figure 71. `featureA` işindən sonra commit tarixi

Branch-ı rebase etdiyinizə görə serverdəki `featureA` branch-nı bir nəsil olmayan bir commit ilə əvəz edə bilmək üçün `-f` seçimini əmrdə göstərməlisiniz. Alternativ bu yeni işi serverdəki fərqli bir branch-a (bəlkə də `featureAv2` adlandırmaq olar) yönəltmək olar. Gəlin daha bir ssenariyə baxaq: t əmirçi ikinci branch-ınızdakı işə baxdı və konsepsiyanı bəyəndi, ancaq bir icra detalını dəyişdirm əyinizi istədi. Layihənin indiki `master` branch-dan kənara qoyulacaq işi köçürmək üçün bu fürsətd ən istifadə edəcəksiniz. Mövcud `origin/master` branch-na əsaslanan yeni bir branch açırsınız, `featureB` dəyişikliklərini sıxışdırırsınız, hər hansı bir münaqişəni həll edirsiniz, tətbiqetməni d əyişdirirsiniz və sonra yeni bir branch olaraq push edirsiniz:

```
$ git checkout -b featureBv2 origin/master
$ git merge --squash featureB
  ... change implementation ...
$ git commit
$ git push myfork featureBv2
```

`--squash` seçimi birləşdirilmiş branch-dakı bütün işləri alır və bir birləşmə əməliyyatı etmədən h əqiqi birləşmə baş vermiş kimi bir depo vəziyyətini yaradan bir dəyişiklik halına gətirir. Bu, gələc ək commit-inizin yalnız bir valideynə sahib olacağını və yeni bir commiti qeyd etməzdən əvvəl bütün dəyişiklikləri başqa bir branch-a təqdim etməyinizi və daha sonra daha çox dəyişiklik etm əyinizi təmin edəcək deməkdir. Qeyri-adi birləşmə prosesi halında birləşməni təxirə salmaq üçün `--no-commit` seçimi faydalı ola bilər. Bu anda, tələb olunan dəyişiklikləri etdiyiniz barədə təmirçiyə xəbər verə bilərsiniz və bu dəyişiklikləri `featureBv2` branch-nızda tapa bilər.

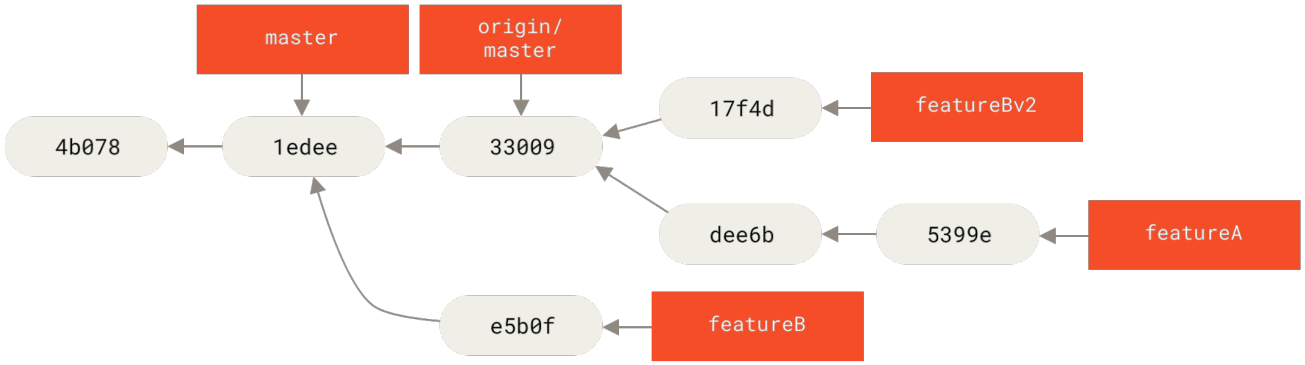


Figure 72. `featureBv2` işindən sonra commit tarixi

E-poçt Üzərindən Public Layihə

Bir çox layihədə patch-ları qəbul etmək üçün prosedurlar qurulmuşdur - hər bir layihə üçün xüsusi qaydaları yoxlamaq lazımdır, çünki onlar fərqli olurlar. Bir developer poçt siyahısı vasitəsilə patch-ları qəbul edən bir neçə daha köhnə, daha böyük layihələr olduğundan indi bir nümunəyə baxacağıq.

İş axını əvvəlki istifadə vəziyyətinə bənzəyir - işlədiyiniz hər patch seriyası üçün mövzu branch-ları yaradırsınız. Fərq onları layihəyə necə təqdim etməyinizdir. Layihəni forking etmək və öz yazıla bilən versiyanıza push etmək əvəzinə hər bir sıra seriyasının e-poçt versiyasını hazırlayır və onları developer poçt siyahısına göndərirsiniz:

```
$ git checkout -b topicA
  ... work ...
$ git commit
  ... work ...
$ git commit
```

İndi poçt siyahısına göndərmək istədiyiniz iki əmr var. Siyahıya e-poçt göndərə biləcəyiniz mbox formatlı faylları yaratmaq üçün `git format-patch` istifadə edirsiniz - hər bir tapşırığı mövzu mesajının birinci sətri olan bir e-poçt mesajına və mesajın qalan hissəsini əlavə edir commit etdiyi

patch body kimi təqdim olunur. Bunun xoş tərəfi budur ki, `format-patch` ilə yaradılan bir e-poçtdan bir patch tətbiq etmək bütün commit məlumatlarını düzgün saxlayır.

```
$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-increase-log-output-to-30-from-25.patch
```

`format-patch` əmri yaratdığı patch fayllarının adlarını yazdırır. `-M` seçimi Git-ə adları axtarmağı tapşırır. Fayllar belə görünür:

```
$ cat 0001-add-limit-to-log-function.patch
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] Add limit to log function

Limit log functionality to the first 20

---
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 76f47bc..f9815f1 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -14,7 +14,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log #{treeish}")
+    command("git log -n 20 #{treeish}")
   end

   def ls_tree(treeish = 'master')
--
2.1.0
```

Göndərmə mesajında göstərmək istəmədiyiniz e-poçt siyahısı üçün daha çox məlumat əlavə etmək üçün bu patch fayllarını düzəldə bilərsiniz. `---` sətri ilə patch-ın əvvəlində (`diff --git` xətti) arasında mətn əlavə etsəniz, developerlər onu oxuya bilər, amma patch prosesi tərəfindən yox sayılacaqdır.

Bunu bir poçt siyahısına göndərmək üçün faylı e-poçt proqramınıza yapışdıra və ya komanda xətti proqramı vasitəsilə göndərə bilərsiniz. Mətnin yapışdırılması tez-tez formatlaşdırma problemlə rinə səbəb olur.Xüsusən də yeni sətirləri və digər boş yerləri lazımi qaydada saxlamayan “smarter” müştərilərlə belə problemlər yaranır. Xoşbəxtlikdən, Git sizin üçün daha asan ola bilən IMAP vasit əsilə düzgün formatlı patch-lar göndərməyinizə kömək edəcək bir vasitə təqdim edir. Gmail vasit

əsilə necə bir yollama göndərəcəyimizi göstərəcəyik, bu da ən yaxşı tanıdığımız e-poçt agenti olur; Git mənbə kodunda yuxarıda göstərilən `Documentation/SubmittingPatches` faylının sonunda bir sıra poçt proqramları üçün ətraflı təlimatları oxuya bilərsiniz.

Əvvəlcə imap bölməsini `~/.gitconfig` faylınızda qurmalısınız. Hər bir dəyəri ayrıca bir sıra `git config` əmrləri ilə təyin edə bilərsiniz və ya onları manual əlavə edə bilərsiniz, lakin sonda konfiqurasiya faylınızda bu kimi bir şey görünməlidir:

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = YX]8g76G_2^sFbd
  port = 993
  sslverify = false
```

Əgər IMAP serveriniz SSL istifadə etmirsə, son iki sətir, ehtimal ki, lazım deyil və host dəyəri `imap://` yerinə `imaps://` olacaqdır. Qurulduqda patch seriyasını göstərilən IMAP serverinin Drafts folder-nə yerləşdirmək üçün `git imap-send` istifadə edə bilərsiniz:

```
$ cat *.patch |git imap-send
Resolving imap.gmail.com... ok
Connecting to [74.125.142.109]:993... ok
Logging in...
sending 2 messages
100% (2/2) done
```

Bu anda, Drafts folder-nə gedə, Kimə hissəsini patch-ı göndərdiyiniz poçt siyahısına dəyişdirə, bə lkə CC-ə bu bölməyə cavabdeh olan şəxsə göndərə biləsiniz.

Patch-ları bir SMTP serveri vasitəsilə də göndərə bilərsiniz. Əvvəllər olduğu kimi hər bir dəyəri ayrıca bir sıra `git config` əmrləri ilə müəyyənləşdirə bilərsiniz və ya onları manual olaraq `~/.gitconfig` sənədinizdəki göndərmə poçt bölməsinə əlavə edə bilərsiniz:

```
[sendemail]
  smtpencryption = tls
  smtpserver = smtp.gmail.com
  smtpuser = user@gmail.com
  smtpserverport = 587
```

Bu iş bitdikdən sonra patch-larınızı göndərmək üçün `git send-email` istifadə edə bilərsiniz:

```
$ git send-email *.patch
0001-add-limit-to-log-function.patch
0002-increase-log-output-to-30-from-25.patch
Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
Who should the emails be sent to? jessica@example.com
Message-ID to be used as In-Reply-To for the first email? y
```

Sonra Git göndərdiyiniz hər bir patch üçün bu kimi bir şey axtaran bir dəst log məlumatı verir:

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from
  \line 'From: Jessica Smith <jessica@example.com>'
OK. Log says:
Sendmail: /usr/sbin/sendmail -i jessica@example.com
From: Jessica Smith <jessica@example.com>
To: jessica@example.com
Subject: [PATCH 1/2] Add limit to log function
Date: Sat, 30 May 2009 13:29:15 -0700
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>
X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty
In-Reply-To: <y>
References: <y>

Result: OK
```

Nəticə

Bu bölmə, qarşınıza çıxa biləcəyiniz bir çox fərqli Git layihəsi ilə məşğul olmaq üçün bir sıra ümumi iş axınlarını əhatə etdi və bu prosesi idarə etməyə kömək edəcək bir neçə yeni vasitə t əqdim etdi. Sonra, coin-nin digər tərəfini necə işləyəcəyinizi görəcəksiniz: Git layihəsini qorumaq. Xeyirxah bir diktator və ya inteqrasiya meneceri olmağı öyrənəcəksiniz.

Layihənin Saxlanılması

Bir layihəyə necə effektiv dəstək verəcəyinizi bilməklə yanaşı, çox güman ki, onu necə qorumağı da bilməlisiniz. Bu, sizə göndərilən `format-patch` vasitəsi ilə yaradılan patch-ların qəbul edilməsindən və tətbiq edilməsindən və ya proyektinizə uzaqdan əlavə etdiyiniz depolar üçün uzaq filiallarda d əyişikliklərin birləşdirilməsindən ibarət ola bilər. Kanonik bir depo saxlamağınızdan və ya patchların doğrulanmasından və ya təsdiqlənməsindən kömək istəməyinizdən asılı olmayaraq, işinizi digər dəstəkçilər üçün ən aydın və uzun müddət ərzində davamlı olacaq şəkildə qəbul etm əli olduğunuzu bilməlisiniz.

Mövzu Branch'larında İşləmək

Yeni işə inteqrasiya etməyi düşünəndə ümumiyyətlə onu bir *mövzu branch-ında* yəni, bu yeni işi sınamaq üçün xüsusi hazırlanmış müvəqqəti branch-da sınamaq daha yaxşı fikirdir. Bu yolla, bir patch-ı xüsusi olaraq tweak etmək və işə qayıtmaq üçün vaxtınız olmadıqda qoyub getmək asandır.

Çalışacağınız işin mövzusuna, məsələn `ruby_client` və ya buna bənzər təsvir olunan bir şeyə əsaslanaraq sadə bir branch adını yaratsanız, bir müddət tərk etməli və daha sonra geri qayıtmalı olsanız belə asanlıqla yadda saxlaya bilərsiniz. Git layihəsinin aparıcısı bu branch-ları da `sc/ruby_client` kimi genişləndirməyə çalışır və bu işə dəstək verən şəxs üçün `sc` qısa formada olur. Yadınızdadırsa, bu şəkildə `master` branch-nıza əsaslanan branch yarada bilərsiniz:

```
$ git branch sc/ruby_client master
```

Və ya dərhal ona keçid etmək istəyirsinizsə, `checkout -b` seçimindən istifadə edə bilərsiniz:

```
$ git checkout -b sc/ruby_client master
```

İndi aldığınız dəstəklənmiş işi bu mövzu branch-na əlavə etməyə və daha uzunmüddətli branch-lar birləşdirməyə hazırsınız.

Elektron Poçtdan Patch'ların Tətbiq Olunması

Layihənizə inteqrasiya edilməsi lazım olan bir e-poçt üzərindən bir patch alsanız, qiymətləndirmə üçün mövzu branch-da patch tətbiq etməlisiniz. E-poçt patch-nı tətbiq etməyin iki yolu var:`git apply` və ya `git am` ilə.

Tətbiqetmə ilə Patch Tətbiq Olunması

Patchi `git diff` və ya Unix `diff` əmri ilə yaradan birisindən almış olsanız (tövsiyə edilmir; növbəti hissəyə baxın), `git apply` əmri ilə tətbiq edə bilərsiniz. Patch-ı `/tmp/patch-ruby-client.patch`-də saxladığınızı düşünürsünüzsə, belə birşey tətbiq edə bilərsiniz:

```
$ git apply /tmp/patch-ruby-client.patch
```

Bu, işlədiyiniz qovluqdakı faylları dəyişdirir. Patch tətbiq etmək demək olar ki, eynidir - tətbiq üçün `patch -p1` komanda daha paranoid olsa da, patch-dan daha az qeyri-səlis matçları qəbul edir. Ayrıca `git diff` formatında təsvir edildiyi təqdirdə fayl əlavə edir, silir və adını dəyişir, hansı ki `patch` bunu etmir. Nəhayət, `git apply` hər şeyin tətbiq olunduğu və ya heç birinin olmadığı "apply all or abort all" modelidir, halbuki `patch` qismən patchfiles tətbiq edə bilər. `git apply patch`-dan daha çox mühafizəkardır. Bu sizin üçün commit yaratmayacaq - onu işlədikdən sonra manual t əqdim olunan dəyişiklikləri səhnələşdirməli və etməlisiniz. Siz onu tətbiq etməyə çalışmadan əvvəl bir patch-ın təmiz tətbiq olunduğunu görmək üçün `git apply` ilə yoxlaya bilərsiniz - bu zaman `git apply --check`-i patch ilə yoxlayın:

```
$ git apply --check 0001-see-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

Əgər çıxış yoxdursa, patch təmiz tətbiq olunmalıdır. Bu əmr həmçinin çek uğursuz olduqda sıfır olmayan bir status ilə çıxır, bu zaman istədiyiniz təqdirdə skriptlərdə istifadə edə bilərsiniz.

Patch'ı `am` ilə Tətbiq Etmək

Əgər dəstəkçi Git istifadəçisidirsə və patch-ların düzəldilməsi üçün `format-patch` əmrindən istifadə etmək kifayətdirsə, sizin işiniz daha asandır, çünki patch-da müəllif məlumatları və sizin üçün commit mesajı var. Əgər edə bilsəniz, dəstəkçilərinizi sizin üçün patch-lar yaratmaq üçün fərqli olan `format-patch` istifadə etməyə təşviq edin. Siz yalnız köhnə patch-lar və bu kimi şeylər üçün `git apply` işlətməlisiniz.

`format-patch` tərəfindən yaradılan bir patch tətbiq etmək üçün `git am` istifadə edirsiniz (əmr "poçt qutusundan bir sıra patchlar tətbiq etmək üçün istifadə edildiyi üçün" `am` adlanır). Texniki olaraq, `git am` bir və ya daha çox e-poçt mesajını bir mətn sənədində saxlamaq üçün sadə, düz mətn formatı olan bir mbox faylını oxumaq üçün qurulmuşdur. Belə bir şey kimi görünəcəkdir:

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] Add limit to log function

Limit log functionality to the first 20
```

Bu əvvəlki hissədə gördüyünüz git format-patch əmrinin çıxışının başlanğıcıdır; eyni zamanda etibarlı bir mbox e-poçt formatını təmsil edir. Kimsə git göndərmə e-poçtundan istifadə edərək sizə patchdan elektron poçt göndərsə və bunu mbox formatına yükləsəniz, git am-ı o mbox faylına yönləndirə bilərsiniz və o, gördüyü bütün patchları tətbiq etməyə başlayacaq. Bir neçə e-poçtu mbox formatında saxlaya bilən bir poçt müştərisi işlətsəniz, bütün patch silsilələrini bir faylda saxlaya bilərsiniz və sonra onları bir-bir tətbiq etmək üçün `git am` istifadə edə bilərsiniz.

Ancaq kimsə `git format-patch` vasitəsi ilə yaradılan bir patch sənədini bilet sisteminə və ya bənzər bir şeyə yükləyibsə, yerli olaraq saxlaya bilər və sonra diskinizdə saxlanan həmin sənədi tətbiq etmək üçün ötürə bilərsiniz:

```
$ git am 0001-limit-log-function.patch
Applying: Add limit to log function
```

Təmiz tətbiq olunduğunu və avtomatik olaraq sizin üçün yeni bir commit yaratdığını görə bil ərsiniz. Müəllif haqqında məlumat e-poçtun `From` və `Date` başlıqlarından götürülür və commit mesajı e-poçtun `Subject` və gövdəsindən (patchdan əvvəl) götürülür. Məsələn, bu patch yuxarıdakı mbox nümunəsindən tətbiq olsaydı, əmələ gələn əməl buna bənzəyəcəkdi:

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author:     Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit:     Scott Chacon <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

   Add limit to log function

   Limit log functionality to the first 20
```

`Commit` məlumatında patch tətbiq edən şəxs və tətbiq olunan vaxt göstərilir. `Author` məlumatları ə vvəlcə patch yaradan və əvvəlcədən yaradan şəxsdir.

Ancaq patch-ın təmiz tətbiq edilməməsi mümkündür. Bəlkə də əsas branch-nız patch tikilmiş branch-dan çox uzaqlaşıb və ya patch hələ tətbiq etmədiyiniz başqa bir patchdan asılıdır. Bu və ziyyətdə `git am` prosesi uğursuz olacaq və nə etmək istədiyinizi soruşacaq:

```
$ git am 0001-see-if-this-helps-the-gem.patch
Applying: See if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

Bu əmr, ziddiyyətli birləşmə və ya yenidən işə salmaq kimi problemləri olan hər hansı bir sənəddə konflikt işarələri qoyur. Bu məsələni eyni şəkildə həll edə bilərsiniz - münaqişəni həll etmək üçün faylı düzəldin, yeni faylı hazırlayın və sonra `git am --resolved` əmrini işə salıb, növbəti patcha davam edin:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: See if this helps the gem
```

Git'in konflikti həll etmək üçün bir az daha ağıllı bir şəkildə cəhd etməsini istəyirsinizsə, ona `-3` seçimini yönləndirə bilərsiniz, bu da Git cəhdini üç tərəfli birləşməyə məcbur edir. Bu seçim standart olaraq edilmir, çünki patch-ın baza olaraq götürüldüyü əmr deponuzda yoxdursa o işləm əyəcək. Əgər bu commit-niz varsa - patch public bir commit üzərində qurulmuşdursa - o zaman `-3` seçimi ziddiyyətli patch-ın tətbiqi ilə bağlı daha ağıllı seçimdir:

```
$ git am -3 0001-see-if-this-helps-the-gem.patch
Applying: See if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

Bu vəziyyətdə, `-3` variant olmadan patch konflikt hesab edilə bilər. `-3` variant istifadə edildiyi üçün patch təmiz tətbiq olunur.

Bir mbox-dan bir sıra patchlar tətbiq etsəniz, `am` əmrini interaktiv rejimdə işlətmək olar, bu tapdığı hər patchda dayanır və tətbiq etməyinizi xahiş edir:

```
$ git am -3 -i mbox
Commit Body is:
--------------------------
See if this helps the gem
--------------------------
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

Bir çox patch-ın saxlanması yaxşıdır, çünki əvvəlcə patch-ın nə olduğunu xatırlamadığınız təqdirdə görə bilərsiniz və ya əvvəlcədən patch tətbiq etməyə bilərsiniz. Mövzunuz üçün bütün patch-lar t ətbiq edildikdə və branch-ınıza verildikdə onları daha uzun bir branch-a inteqrasiya edib etməyəc əyinizi və ya necə etməli olduğunuzu seçə bilərsiniz.

Uzaq Branch'ları Yoxlamaq

Sizin töhfəniz öz depolarını quran, bir sıra dəyişikliklər edən Git istifadəçisindən gəlirsə və URL-i depozitə göndərirsinizsə və dəyişikliklərin olduğu uzaq branch-ın adını çəksəniz, bunları əlavə edə bilərsiniz, uzaq və local olmaqla birləşdirəcəkdir.

Məsələn, Jessica sizə öz depolarının `ruby-client` branch-ında yeni bir xüsusiyyəti olduğunu söyləy ən bir e-poçt göndərsə, uzaqdan əlavə edərək local branch-ı yoxlamaqla sınaya bilərsiniz:

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

Daha sonra başqa bir əla bir xüsusiyyəti özündə birləşdirən başqa bir branch ilə yenidən sizə e-poçt göndərərsə, birbaşa quraşdırma olduğunuz üçün birbaşa `fetch` və `checkout` edə bilərsiniz.

Bir şəxslə ardıcıl işləmək ən faydalı seçimdir. Kimsə bir müddət ərzində bir qatqı təmin edəcək tək bir patch-a sahibdirsə, onu elektron poçtla qəbul etmək hər kəsdən öz serverini işə salmağı tələb etməkdən və bir neçə patch almaq üçün daim əlavə etmək və silmək məcburiyyətində qalmaqdan daha az vaxt tələb edə bilər. Hər biri yalnız bir patch və ya iki töhfə verən biri üçün yüzlərlə

uzaqdan istifadə etmək istəməyiniz mümkün deyil. Bununla birlikdə, skriptlər və ev sahibi xidmə tləri bunu asanlaşdıra bilər - bu, sizin necə inkişaf etdiyinizə və töhfəçilərinizin necə inkişaf etdiyinə bağlıdır.

Bu yanaşmanın digər üstünlüyü odur ki, commit-lərin tarixini də əldə etməyinizdir. Birləşmə ilə bağlı qanuni problemləriniz ola bilər, ancaq tarixinizdə işlərinin harada dayandığını bilirsiniz; düzgün bir üç tərəfli birləşmə bir `-3` təmin etmək əvəzinə standartdır və patchin daxil olacağınız bir public commit-dən yaradıldığına ümid edirik. Daimi bir adamla işləmirsinizsə, lakin hələ də bu şəkildə onlardan pull etmək istəsəniz, `git pull` əmrinə uzaqdakı depo URL-ni təqdim edə bilərsiniz. Bu birdəfəlik çəkim aparır və URL-i uzaqdan istinad kimi saxlamır:

```
$ git pull https://github.com/onetimeguy/project
From https://github.com/onetimeguy/project
 * branch            HEAD       -> FETCH_HEAD
Merge made by the 'recursive' strategy.
```

Nəyin Təqdim Olunduğunu Müəyyənləşdirmək

İndi dəstəkdə əməyi olanlardan ibarət bir mövzu branch-nız var. Bu anda nə etmək istədiyinizi t əyin edə bilərsiniz. Bu bölmə bir neçə əmrini yenidən nəzərdən keçirir ki, bunları əsas branch-ınıza birləşdirdiyiniz təqdirdə tətbiq edəcəyinizi nəzərdən keçirmək üçün necə istifadə etdiyinizi görə bilərsiniz.

Bu branch-da olan, lakin `master` branch-ınızda olmayan bütün commit-lərin icmalını almaq çox vaxt faydalıdır. Branch-ın adından əvvəl `--not` seçimi əlavə etməklə `master` branch-dakı commit-ləri istisna edə bilərsiniz. Bu, əvvəllər istifadə etdiyimiz `master..contrib` formatı ilə eyni şeyi edir. Məs ələn, töhfəçiniz sizə iki patch göndərirsə və orada həmin patch-ları tətbiq edən `contrib` adlı bir branch yaradırsınızsa, bunu edə bilərsiniz:

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700

    See if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700

    Update gemspec to hopefully work better
```

Hər bir tapşırığın nəyi dəyişdiyini görmək üçün `-p` seçimini `git log`-a keçə biləcəyinizi və hər bir commit-ə təqdim olunan fərqi əlavə edəcəyinizi unutmayın.

Bu mövzu branch-nı başqa bir branch-la birləşdirsəniz, nə olacağının tam fərqini görmək üçün düzgün nəticələr əldə etmək üçün qəribə bir hiylədən istifadə etməli ola bilərsiniz. Bunu idarə

etməyi düşünə bilərsiniz:

```
$ git diff master
```

Bu əmr sizə bir fərq verir, ancaq sizi yanılda bilər. Mövzu branch-nı ondan yaratdığınızdan bəri `master` branch-nız irəliləyibsə, qəribə görünən nəticələr əldə edəcəksiniz. Bu, Git birbaşa olduğunuz mövzu bölməsinin son əmrlərinin anketlərini və magistr bölməsindəki son əmrlərin snapshotlarını birbaşa müqayisə etməsi nəticəsində baş verir. Məsələn, `master` branch-da bir sətir əlavə etdinizsə, snapshotların birbaşa müqayisəsi mövzu branch-na bu sətri silmək kimi görünəcəkdir.

Əgər `master` mövzu branch-nın birbaşa əcdadıdırsa, bu problem deyil; lakin iki tarix ayrılıbsa, f ərqli görünəcək ki, mövzu branch-da bütün yeni materialları əlavə etməyiniz və `master` branch-na xas olan hər şeyi silmək olar.

Həqiqətən görmək istədiyiniz mövzu barnch-na əlavə edilmiş dəyişikliklər - bu branch-ı `master` ilə birləşdirsəniz təqdim edəcəyiniz işlərdir. Bunu Git-in mövzu branch-dakı son commiti ana branch-ı ilə olan ilk ortaq əcdadı ilə müqayisə etməklə etməlisiniz.

Texniki cəhətdən, ortaq əcdadını aydın şəkildə müəyyənləşdirə və sonra fərqinizi işlətməklə bunu edə bilərsiniz:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

və ya:

```
$ git diff $(git merge-base contrib master)
```

Ancaq bunların heç biri xüsusilə əlverişli deyildir, buna görə Git eyni şeyi etmək üçün başqa bir stend təqdim edir: üç nöqtəli sintaksis. `git diff` əmri kontekstində başqa bir branch-dan sonra üç dövr qoya bilər və olduğunuz branch-ın son törəməsi ilə başqa bir branch ilə ümumi əcdadı arasında fərq qoymaq üçün:

```
$ git diff master...contrib
```

Bu əmr yalnız cari mövzu branch-ın `master` ilə ortaq əcdadından bəri tanıtdığı işləri göstərir. Yadda saxlamaq üçün çox faydalı bir sintaksisdir.

İşə İnteqrasiya

Mövzu branch-nızdakə bütün işlər daha təməl branch-a birləşdirilməyə hazır olduqda bunu belə etmək olar; Bundan əlavə, layihənizi qorumaq üçün hansı ümumi iş axını istifadə etmək ist əyirsiniz? Bir neçə seçiminiz var, buna görə onlardan bir neçəsini əhatə edəcəyik.

İş Axınlarının Birləşdirilməsi

Bir əsas iş axını, sadəcə bütün işləri birbaşa `master` branch-ınıza birləşdirməkdir. Bu ssenaridə, ə sasən sabit kodu ehtiva edən `master` branch-nız var. Tamamladığınızı düşündüyünüz bir mövzu branch-da işləmisinizsə və ya başqasının töhfəsini verdiyinizi və təsdiq etdiyinizi görsəniz, onu `master` branch-nıza birləşdirirsiniz, birləşdirilmiş mövzu branch-nı silib yenidən t əkrarlayacaqsınız.

Məsələn, əgər `ruby_client` və `php_client` adlı Bir neçə mövzu branch-ı olan tarix kimi görünən iki branch-da işləyən bir depomuz varsa və `ruby_client`-i və ardınca `php_client`-i birləşdirsəniz tarixniz Mövzu branch-ı birləşmədən sonra kimi görünəcəkdir.

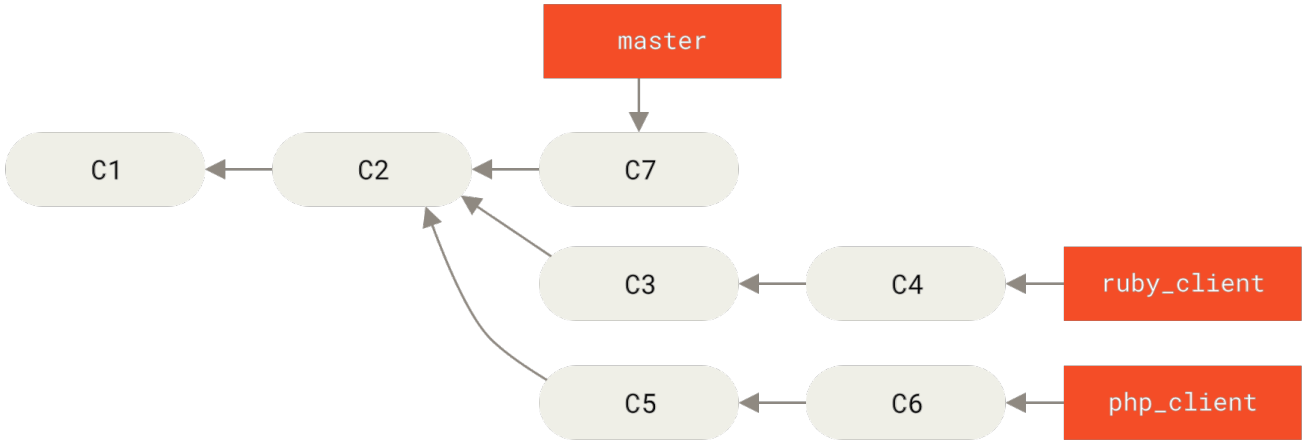


Figure 73. Bir neçə mövzu branch-ı olan tarix

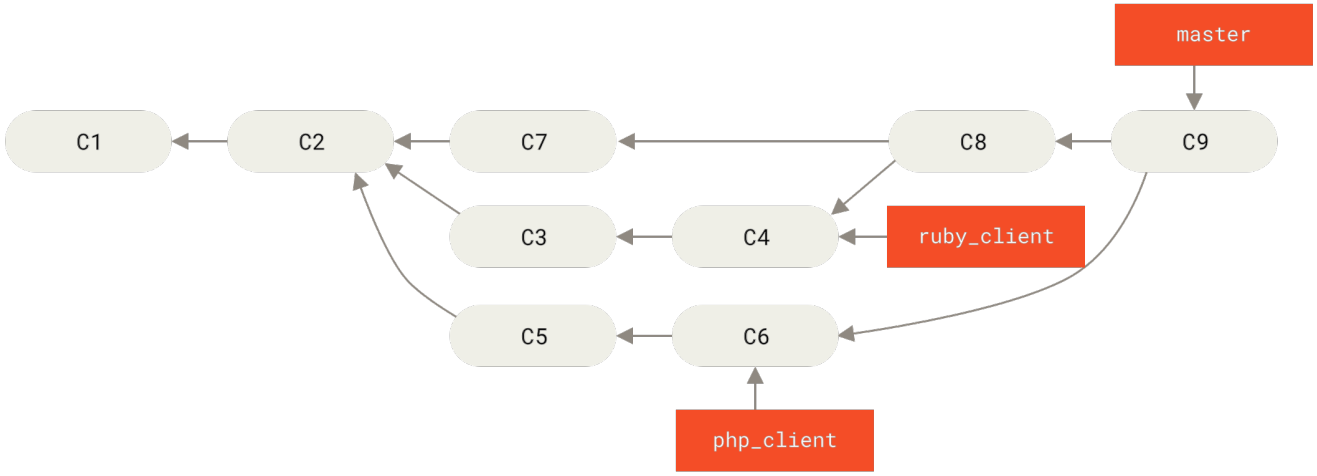


Figure 74. Mövzu branch-ı birləşmədən sonra

Bu, bəlkə də ən sadə iş axınlarıdır, amma tanıdıb təqdim etdiyiniz şeylərə diqqətli olmaq ist ədiyiniz daha böyük və ya daha sabit layihələrlə məşğul olsanız, problemli ola bilər.

Daha vacib bir layihəniz varsa, iki fazalı birləşmə dövründən istifadə etmək istəyə bilərsiniz. Bu ssenaridə `master` and `develop` olan iki uzun filial var, `master` yalnız çox sabit bir buraxılma kə sildikdə və bütün yeni kod `develop` branch-na inteqrasiya edildikdə yeniləndiyini müəyyənl əşdirirsiniz.

Mütəmadi olaraq bu branch-ların hər ikisini public depolarına aparırsınız. Hər dəfə (Mövzu

branch-ı birləşmədən əvvəl) birləşmək üçün yeni bir mövzu branch-ı varsa, onu `develop`-a (Mövzu branch-ı birləşmədən sonra) birləşdirirsiniz; sonra bir etiketi etiketlədikdə, stabil `develop` branch-ın olduğu yerə `master` sürətlə irəliləyir (Mövzu branch-ı buraxıldıqdan sonra).

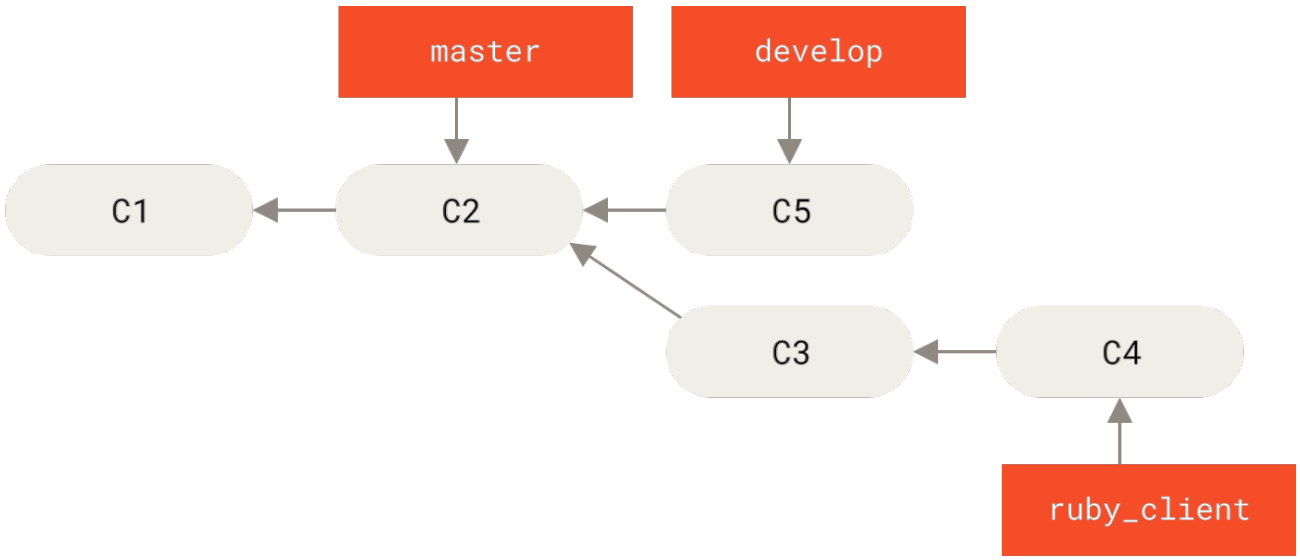


Figure 75. Mövzu branch-ı birləşmədən əvvəl

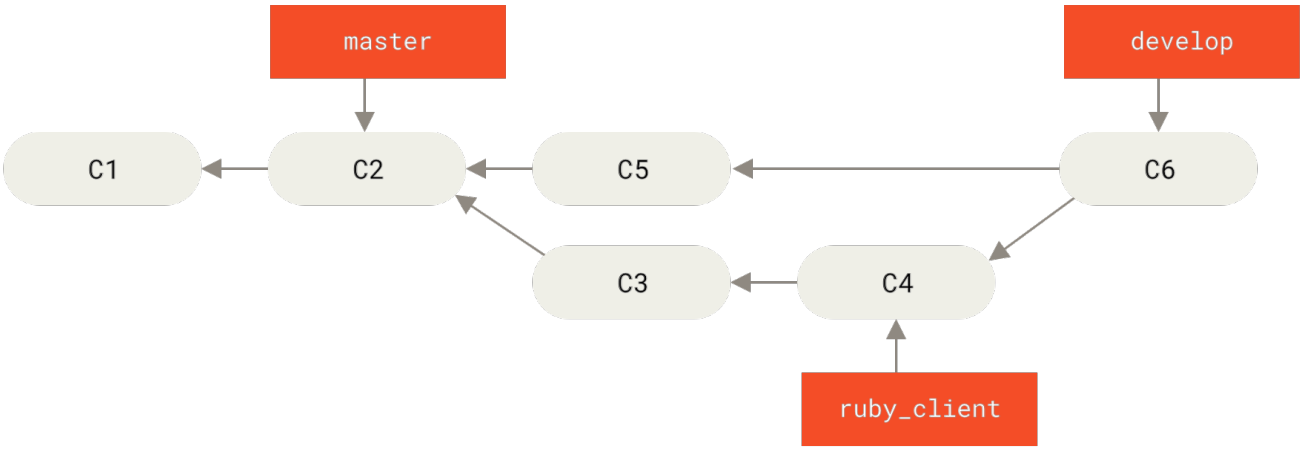


Figure 76. Mövzu branch-ı birləşmədən sonra

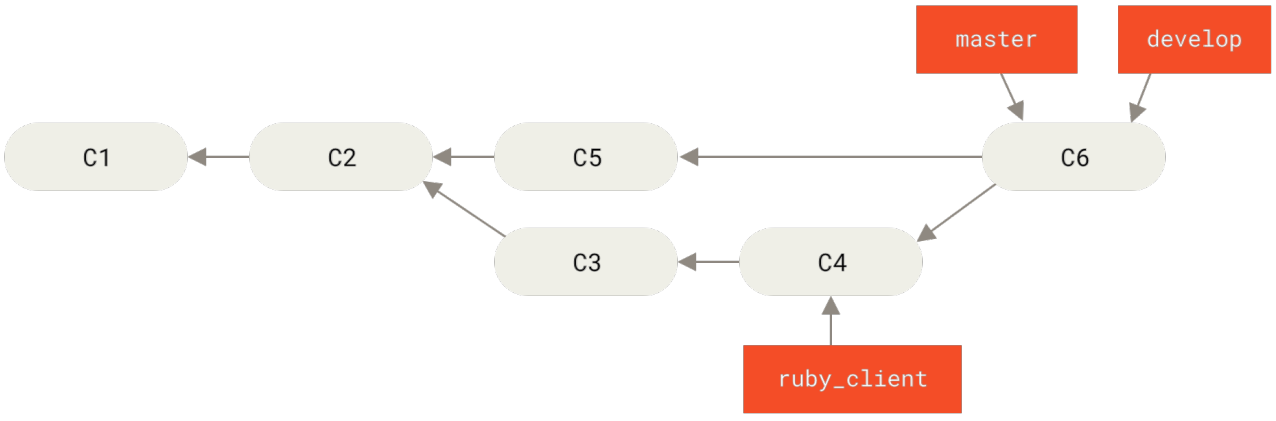


Figure 77. Mövzu branch-ı buraxıldıqdan sonra

Bu yolla, insanlar proyektlərinizin depolarını klonlaşdırdıqda son sabit versiyasını hazırlamaq

üçün `master`-i yoxlaya və asanlıqla bu günə qədər davam etdirə bilər və ya daha inkişaf etmiş məzmun olan `develop`-u yoxlaya bilərlər. Ayrıca, bütün işlərin birləşdirildiyi bir `integrate` branch-na sahib olmaqla bu anlayışı genişləndirə bilərsiniz. Sonra, bu branch-dakı kod bazası sabit olduqda və testlərdən keçdikdə, onu `develop` branch-na birləşdirirsiniz; və bu, bir müddət sabit olduqda, `master` branch-nızı sürətlə irəliyə aparırsınız.

Böyük Birləşən İş Axınları

Git layihəsinin dörd uzun branch-ı var: `master`, `next`, və `seen` (əvəllər *pu* --təklif olunan yeniləmələr) yeni iş üçün, və `maint` yeni iş yerləri üçün. Yardımçılar tərəfindən yeni bir iş təqdim edildikdə, təsvir etdiyimizə bənzər bir şəkildə tərtibatçı depolarında mövzu şöbələrinə toplanır (Paralel töhfə verilmiş mövzu şöbələrinin kompleks seriyasını idarə etmək bax). Bu nöqtədə, təhlükəsiz və istehlaka hazır olub olmadığını və ya daha çox işə ehtiyacı olub olmadığını müəyyən etmək üçün mövzular qiymətləndirilir. Etibarlı olduqları təqdirdə, `next` birləşdirilir və hər kəs bir-birinə inteqrasiya olunan mövzuları sınaya bilməsi üçün bu branch yuxarı göndərilir.

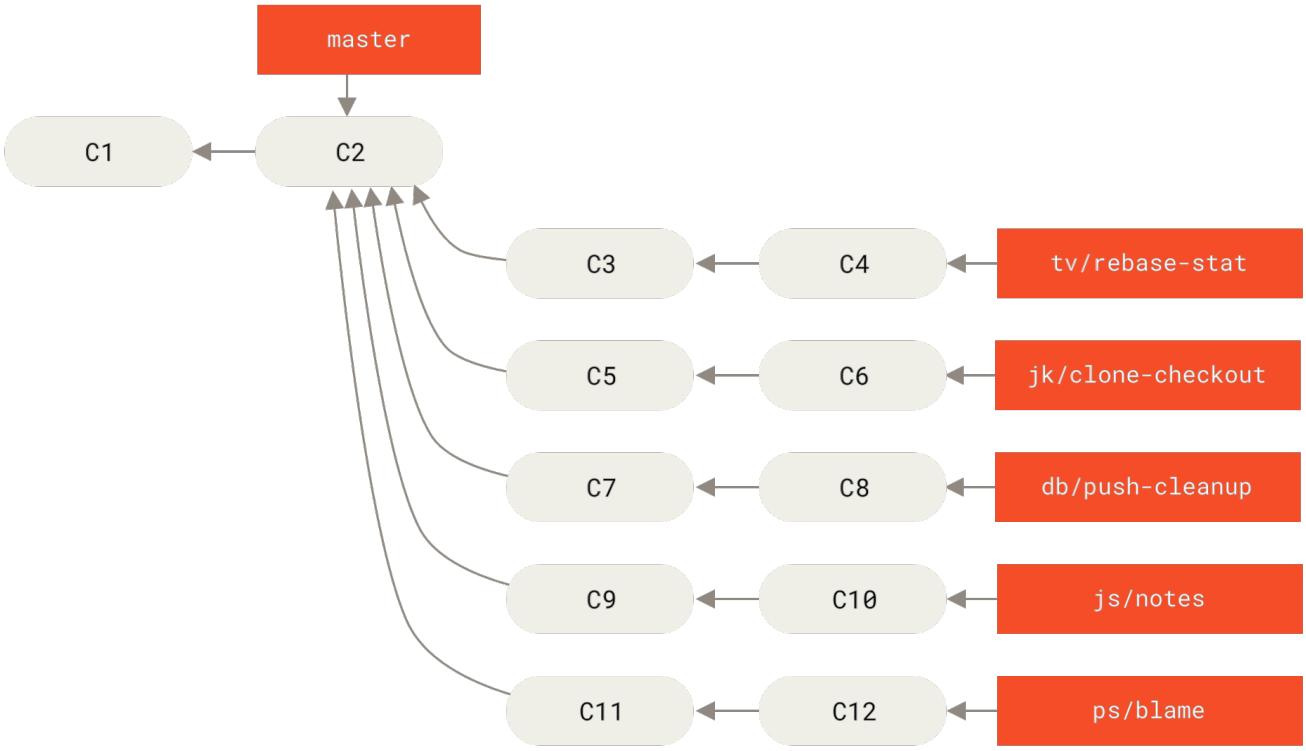


Figure 78. Paralel töhfə verilmiş mövzu şöbələrinin kompleks seriyasını idarə etmək

Mövzular hələ də işə ehtiyac duyursa, əvəzinə `seen`-ə birləşdirilir. Tamamilə sabit olduqları müəyyən edildikdə, mövzular yenidən `master`-ə birləşdirilir. `next` və `seen` branch-ları `master` tərəfindən yenidən qurulur. Bu o deməkdir ki, `master` demək olar ki, həmişə irəliləyir, `next` bəzən rebase olunur və `senn` daha tez-tez dəyişdirilir:

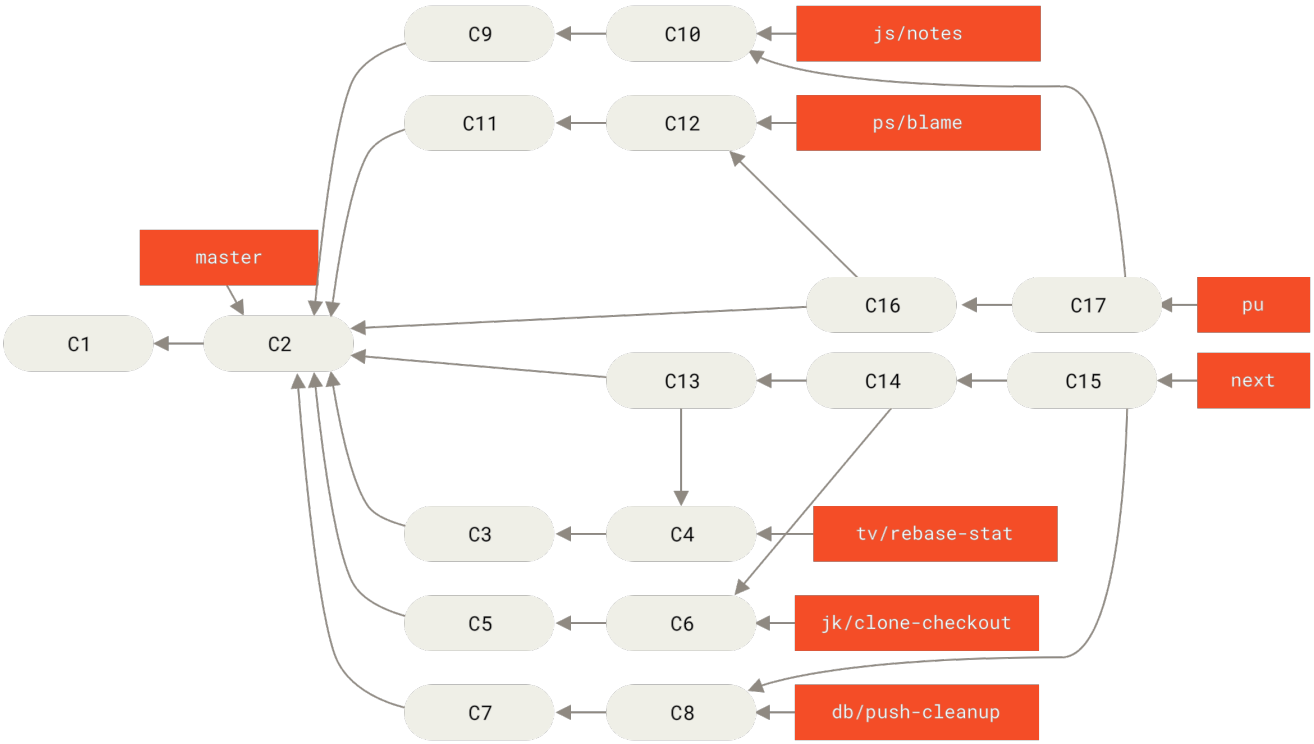


Figure 79. Mövzu şöbələrini uzunmüddətli inteqrasiya filiallarına birləşdirmək

Bir mövzu branch-ı nəhayət `master` ilə birləşdirildikdə, depo içərisindən silinir. Git layihəsində, texniki xidmət tələb olunduğu təqdirdə geri çəkilmiş patchları təmin etmək üçün son buraxılışdan kənarlaşdırılan bir `maint` branch-ı var. Beləliklə, Git depolarını klonlaşdırdığınız zaman, necə olmaq istədiyinizdən və necə töhfə vermək istədiyinizdən asılı olaraq, müxtəlif inkişaf mərhələl ərində layihəni qiymətləndirmək üçün yoxlaya biləcəyiniz dörd branch-nız var; və təmirçi onlara yeni töhfələr verməyə kömək etmək üçün strukturlaşdırılmış bir workflow-a malikdir. Git layih əsinin workflow-u ixtisaslaşmışdır. Bunu aydın başa düşmək üçün Git Maintainer's guide baxa bil ərsiniz.

Rebasing və Cherry-Picking İş Axınları

Digər təmirçilər, əsasən xətti bir tarix saxlamaq üçün `master` branch-nın üstünə töhfə verən işləri rebase və ya cherry-pick-ə üstünlük verirlər. Bir mövzu branch-ında işlədiyinizdə və onu birl əşdirmək istədiyinizi müəyyənləşdirdiyinizdə, o branch-a keçin və dəyişiklikləri hazırkı `master` (və ya `develop` və s.) branch-ınızda yenidən qurmaq üçün rebase əmrini işə salırsınız. Yaxşı olarsa, `master` branch-ı sürətlə irəli göndərə bilərsiniz və xətti bir layihə tarixi ilə başa çatacaqsınız.

Təqdim olunan işi bir branch-dan digərinə keçirməyin başqa yolu cherry-picki seçməkdir. Git'də bir cherry-pick, yalnız bir commit üçün bir rebase kimidir. Bu commit-ə daxil edilmiş patchi götürür və onu hazırda işlədiyiniz branch-da yenidən tətbiq etməyə çalışır. Bir mövzu banch-ında bir sıra commit-ləriniz varsa və onlardan yalnız birini birləşdirmək istəsəniz və ya bir mövzu banch-ında yalnız bir əmriniz varsa və yenidən yazmağı yerinə, cherry-picking-i üstün tutursunuzsa,bu daha faydalıdır. Məsələn, belə görünən bir layihəniz var deyək:

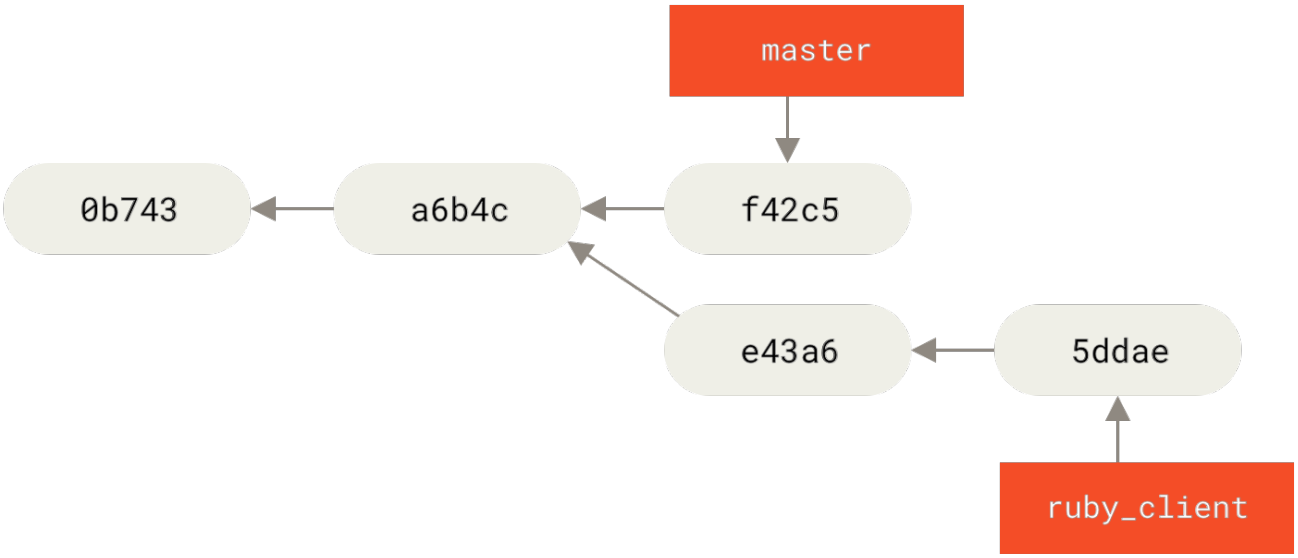


Figure 80. Bir cherry-pickdən əvvəl nümunə tarixi

`e43a6` commit-ni `master` branch-nıza pull etmək istəirsinizsə, bunu işlədə bilərsiniz:

```
$ git cherry-pick e43a6
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
 3 files changed, 17 insertions(+), 3 deletions(-)
```

Bu `e43a6`-da tətbiq olunan eyni dəyişikliyi irəli çəkir, ancaq tətbiq olunan tarix fərqli olduğundan yeni bir SHA-1 dəyəri əldə edirsiniz. Onda tarixiniz belə görünür:

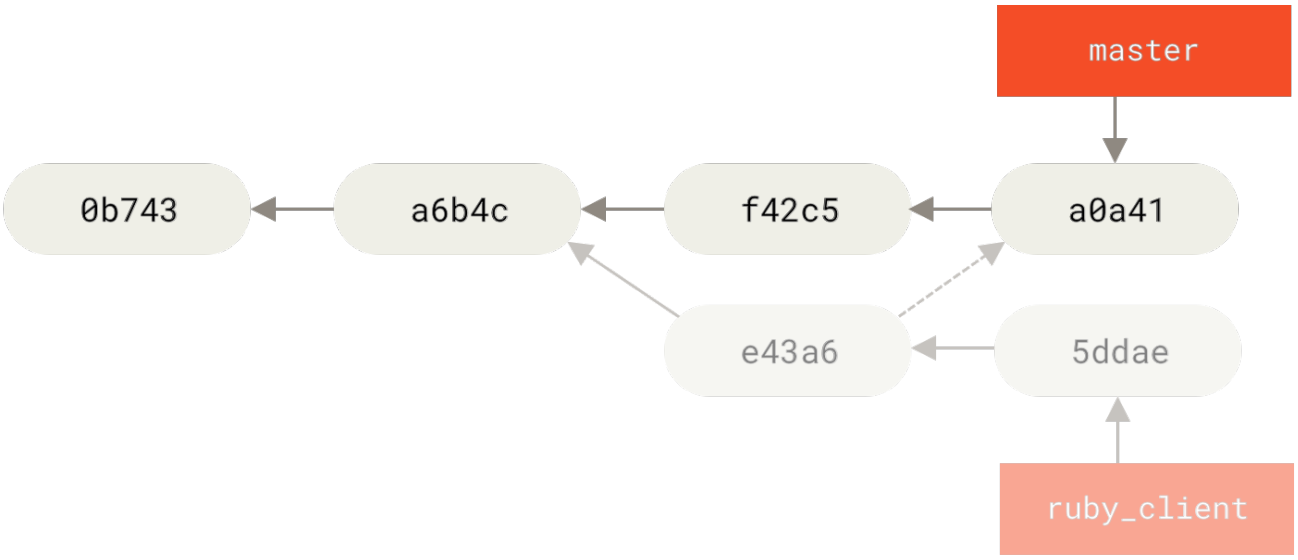


Figure 81. Cherry-pickdən sonra bir mövzu branch-ından bir commit tarixi

İndi mövzu şöbənizi silə və daxil etmək istəmədiyiniz əmrləri ata bilərsiniz.

Rerere

Birləşmə və rebasing mövzusunda çox işlər görürsünüzsə və ya uzun müddətdir davam edən bir mövzu branch-ına davam etdirirsinizsə, Git'in kömək edə biləcəyi "rerere" adlı bir xüsusiyyəti var.

Rerere açılışı "reuse recorded resolution"-dur-- bu, manual konflikt həllini qısaldır. Yenidən işə salındıqda, Git uğurlu birləşmədən əvvəl və sonrakı görüntülər dəstini saxlayacaq və əgər əvvəlcə dən düzəltdiyinizə bənzər bir ziddiyyət olduğunu görsəniz, sizi narahat etmədən yalnız son dəfə düzəlişdən istifadə edəcək.

Bu xüsusiyyət iki hissədən ibarətdir: bir konfiqurasiya qəbulu və bir əmr. Konfiqurasiya qəbulu `rerere.enabled` və qlobal konfiqurasiyanızı qoymaq üçün əlverişlidir:

```
$ git config --global rerere.enabled true
```

İndi, münaqişələri həll edən bir birləşmə etdikdə, gələcəkdə ehtiyacınız olduğu halda qətnamə cache-də qeyd ediləcəkdir. Lazım olsa `git rerere` əmrindən istifadə edərək rerere cache ilə qarşılıqlı əlaqə qura bilərsiniz. Tək başına çağırıldıqda, Git qətnamələr bazasını yoxlayır və cari birləşmə ziddiyyətləri ilə bir eynilik tapmağa və onları həll etməyə çalışır (`rerere.enabled` doğru olduqda bu avtomatik `true` olaraq edilir). Yazılacağını görmək, cache-dən xüsusi bir qətnaməni silmək və bütün cache-i təmizləmək üçün alt qruplar var. Rerere-da daha ətraflı əhatə edəcəyik.

Buraxılışlarınızı Etiketləmək

Buraxılışı kəsmək qərarına gəldiyinizdə, yəqin ki, etiket təyin etmək istəyərsiniz ki, irəliləyişin ist ənilən nöqtəsində yenidən yarada bilərsiniz. Git'in Əsasları-də müzakirə edildiyi kimi yeni bir etiket yarada bilərsiniz. Etiketi qoruyucu olaraq imzalamaq qərarına gəlsəniz, etiketləmə bu kimi bir görünə bilər:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gmail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

Etiketlərinizi imzalamısınızsa, etiketlərinizi imzalamaq üçün istifadə olunan ümumi PGP key-ini paylamaqda probleminiz ola bilər. Git layihəsinin aparıcısı ümumi key-ini depo içərisinə bir qabda kimi əlavə edərək birbaşa həmin məzmuna işarə edən etiket əlavə etməklə bu məsələni həll etdi. Bunu etmək üçün `gpg --list-keys` işlədərək hansı key-i istədiyinizi anlaya bilərsiniz:

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
---------------------------------
pub   1024D/F721C45A 2009-02-09 [expires: 2010-02-09]
uid                  Scott Chacon <schacon@gmail.com>
sub   2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

Sonra açarı birbaşa Git verilənlər bazasına ixrac edərək boru kəməri ilə ixrac edə bilərsiniz və `git`

`hash-object` vasitəsilə, bu məzmunu Git içərisinə yeni bir blob yazan və blob-un SHA-1-ni geri qaytara bilərsiniz.

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Git-də açarınızın məzmunu olduğundan, `hash-object` əmrinin sizə verdiyi yeni SHA-1 dəyərini göstərərək birbaşa ona bir etiket yarada bilərsiniz:

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

`git push --tags` işlədirsinizsə, `maintainer-pgp-pub` etiketi hamıya paylanacaq. Əgər kimsə etiketi yoxlamaq istəyirsə, birbaşa PBP key-nizi bazanı birbaşa verilənlər bazasından çıxararaq GPG-yə idxal edə bilər:

```
$ git show maintainer-pgp-pub | gpg --import
```

Bu key-i bütün imzalanmış etiketləri yoxlamaq üçün istifadə edə bilərlər. Ayrıca, etiket mesajına t əlimatlar daxil edərsənsə, `git show <tag>` istifadə edərək, son istifadəçiyə etiket yoxlaması ilə bağlı daha dəqiq göstərişlər verməyə imkan verəcəkdir.

Bir Build Nömrəsi Yaratmaq

Git'in *v123* kimi monoton olaraq artan nömrələri və ya hər bir commit ilə birlikdə getmək üçün ekvivalenti olmadığı üçün, bir commit ilə getmək üçün insan tərəfindən oxunan bir ada sahib olmaq istəsəniz, bu commit-in üzərində `git describe` işlədə bilərsiniz. Buna cavab olaraq, Git bu əməldən daha erkən son etiketin adından ibarət bir sətir yaradır, sonra bu etiketdən bəri verilənl ərin sayı, sonra təsvir olunan commit-in qismən SHA-1 dəyəri ilə izlənilir (Git mənasını verən "g" hərf ilə əvvəl verilir) :

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

Bu yolla, bir görüntü ixrac edə və ya insanlar üçün başa düşülən bir şey qura və adlandıra bil ərsiniz. Əslində, Git-i Git deposundan klonlanmış mənbə kodundan qursanız, `git --version` sizə b ənzər bir şey verəcəkdir. Birbaşa etiketlədiyiniz bir commit-i təsvir edirsinizsə, sadəcə etiket adını verir.

Varsayılan olaraq, `git describe` əmrində əlavə etiketlər tələb olunur (`-a` və ya `-s` flag-ı ilə yaradılan etiketlər); yüngül (qeyd olunmayan) etiketlərdən də faydalanmaq istəyirsinizsə, `--tags` seçimini əmrə əlavə edin. Ayrıca, bu simli bir `git checkout` və ya `git show` əmrlərini hədəf kimi istifadə edə bilərsiniz, baxmayaraq ki, sonunda qısaldılmış SHA-1 dəyərinə dayanır, buna görə də daimi olaraq etibarlı olmaya bilər. Məsələn, yaxınlarda Linux nüvəsi SHA-1 obyektinin bənzərsizliyini təmin etmək üçün 8-dən 10 simvola qədər atlayır, buna görə köhnə `git describe` çıxış adları etibarsız sayılır.

Buraxılış Hazırlamaq

İndi bir qurğunu buraxmaq istəyirsiniz. Etmək istədiyiniz işlərdən biri Git istifadə etməyən bu yoxsul ruhlar üçün kodunuzun snapshotunda arxiv yaratmaqdır. Bu əmr `git archive`-dir:

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

Kimsə bu tarballı açarsa, layihənizin alt hissəsində layihənizin snapshot-larını əldə edər. Eyni ş əkildə bir zip arxivini də yarada bilərsiniz, `--format=zip` seçimini `git archive`əmrinə ötürərək bu mümkündür:

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

İndi gözəl bir tarballa və veb saytınıza və ya e-poçtla insanlara yükləyə biləcəyiniz bir layihə buraxılışının bir zip arxiviniz var.

Qısa Yol

Layihənizdə nələrin baş verdiyini bilmək istəyən şəxslərin poçt siyahısına e-poçt göndərməyin vaxtı gəldi. Son buraxılışınızdan və ya e-poçtunuzdan bəri proyektinizə əlavə edilmiş bir növ d əyişikliyi tez bir şəkildə əldə etməyin gözəl bir yolu `git shortlog` əmri istifadə etməkdir. Bu verdiyiniz diapazonda göstərilənlərin hamısını ümumiləşdirir; məsələn, sonuncu buraxılışınız v1.0.1 adlandırılıbsa, aşağıdakılar sizə son buraxılışınızdan bəri görülən işlərin xülasəsini verir:

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (6):
      Add support for annotated tags to Grit::Tag
      Add packed-refs annotated tag support.
      Add Grit::Commit#to_patch
      Update version and History.txt
      Remove stray `puts`
      Make ls_tree ignore nils

Tom Preston-Werner (4):
      fix dates in history
      dynamic version method
      Version bump to 1.0.2
      Regenerated gemspec for version 1.0.2
```

Siyahıya e-poçt göndərə biləcəyiniz müəllif tərəfindən qruplaşdırılmış v1.0.1-dən bəri verilən commit-lərin təmiz bir xülasəsini alırsınız.

Qısa Məzmun

Öz layihənizi qorumaq və ya digər istifadəçilərin töhfələrini inteqrasiya etməklə yanaşı, Git'dəki bir layihəyə töhfə vermək üçün kifayət qədər rahat hiss etməlisiniz. Effektiv bir Git developer'i olduğunuz üçün sizi təbrik edirəm! Növbəti fəsildə ən böyük və ən populyar Git hosting xidməti olan GitHub'dan necə istifadə edəcəyinizi öyrənəcəksiniz.

GitHub

GitHub, Git depoları üçün yeganə ən böyük ev sahibidir və milyonlarla developer və layihə üçün iş birliyinin əsas nöqtəsidir. Bütün Git depolarının böyük bir hissəsi GitHub'da yerləşdirilir və bir çox açıq mənbəli layihələr onu Git hosting, problem izləmə, kod nəzərdən keçirmək və digər şeylər üçün istifadə edir. Buna görə Git açıq mənbə layihəsinin birbaşa bir hissəsi olmasa da, Git'i peşəkar olaraq istifadə edərkən bir nöqtədə GitHub ilə qarşılıqlı əlaqə qurmağınızı istəməyiniz və ya etm əyiniz üçün yaxşı bir şans var.

Bu fəsil GitHub'dan səmərəli istifadə ilə bağlıdır. Bir hesab üçün qeydiyyatdan keçməyi və idarə etməyi, Git depolarını, layihələrə töhfə vermək və özünüzə töhfələr qəbul etmək üçün ümumi iş axınlarını yaratmaq və istifadə etməyi, GitHub-un proqramatik interfeysini və ümumiyyətlə h əyatınızı asanlaşdırmaq üçün bir çox kiçik tövsiyələri əhatə edəcəyik.

Öz layihələrinizə ev sahibliyi etmək və ya GitHub'da yerləşdirilən digər layihələrlə əməkdaşlıq etmək üçün GitHub'ı istifadə etməklə maraqlanmırsınızsa, təhlükəsiz şəkildə Git Alətləri-ə keçə bil ərsiniz.

İnterfeyslər Dəyişir

Qeyd etmək vacibdir ki, bir çox aktiv veb sayt kimi, bu ekran görüntülərindəki istifadəçi interfeysi elementləri də zamanla dəyişəcəkdir. Ümid edirik, burada h əyata keçirməyə çalışdığımız şey haqqında ümumi fikir hələ də qalacaq, ancaq bu ekranların ən yeni versiyalarını istəsəniz, bu kitabın onlayn versiyaları daha yeni ekran şəkillərinə sahib ola bilər.

Hesab Qurma və Konfiqurasiya

Etməli olduğunuz ilk şey pulsuz istifadəçi hesabı qurmaqdır. Sadəcə https://github.com saytına daxil olun, öncədən alınmayan bir istifadəçi adını seçin, bir e-poçt ünvanı və şifrə ilə təmin edin və böyük yaşıl "Sign up for GitHub" düyməsinə vurun.

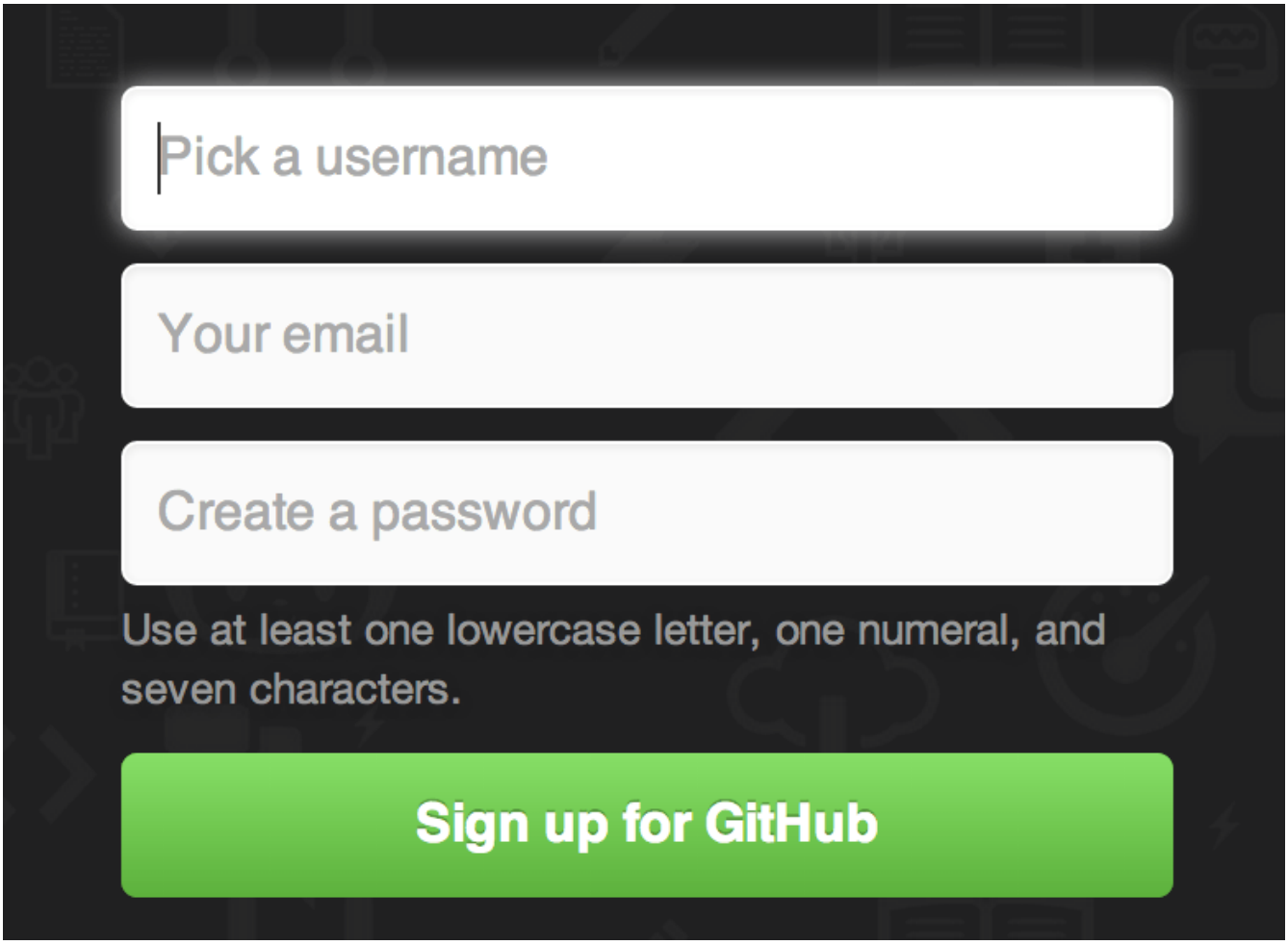


Figure 82. GitHub qeydiyyat forması

Gördüyünüz növbəti şey təkmilləşdirilmiş planlar üçün qiymət səhifəsidir, lakin indi bunu görm əməzlikdən gəlmək təhlükəsizdir. GitHub sizə göstərdiyiniz ünvanı yoxlamaq üçün sizə bir e-poçt göndərəcəkdir. Gedin və bunu edin; olduqca vacibdir (daha sonra görəcəyik).

GitHub, bəzi inkişaf etmiş xüsusiyyətlərdən başqa demək olar ki, bütün funksiyalarını pulsuz hesablara təmin edir.

GitHub-un pullu planlarına qabaqcıl alətlər və xüsusiyyətlər, həmçinin pulsuz xidmətlərə qoyulan məhdudiyyətlər daxildir, lakin biz bu kitabdakıları əhatə etm əyəcəyik. Mövcud planlar və onların müqayisəsi haqqında daha çox məlumat əldə etmək üçün https://github.com/pricing saytına daxil olun.

Ekranın yuxarı sol hissəsindəki Octocat logosuna basaraq sizi dashboard səhifəsinə aparacaq. İndi GitHub-dan istifadə etməyə hazırsınız.

SSH Girşi

Bu andan etibarən, `https://` protokolundan istifadə edərək və yalnız quraşdırdığınız istifadəçi adınızı və şifrənizi təsdiq edərək Git depoları ilə tamamilə əlaqə qura bilərsiniz.

As of right now, you're fully able to connect with Git repositories using the `https://` protocol, authenticating with the username and password you just set up. Bununla birlikdə, sadəcə public layihələri klonlaşdırmaq üçün qeydiyyatdan keçməyinizə ehtiyac yoxdur - layihələrimizi fork

etdiyimizdən və bir az sonra fork-larımıza basdığımızda yaratdığımız hesab açılır.

SSH uzaqdan istifadə etmək istəyirsinizsə, açıq açarı konfiqurasiya etməlisiniz. Hələ yoxdursa,Sizin öz SSH Public Key'nizi yaratmaq-ə baxın. Pəncərənin yuxarı sağındakı linki istifadə edərək hesab parametrlərinizi açın:

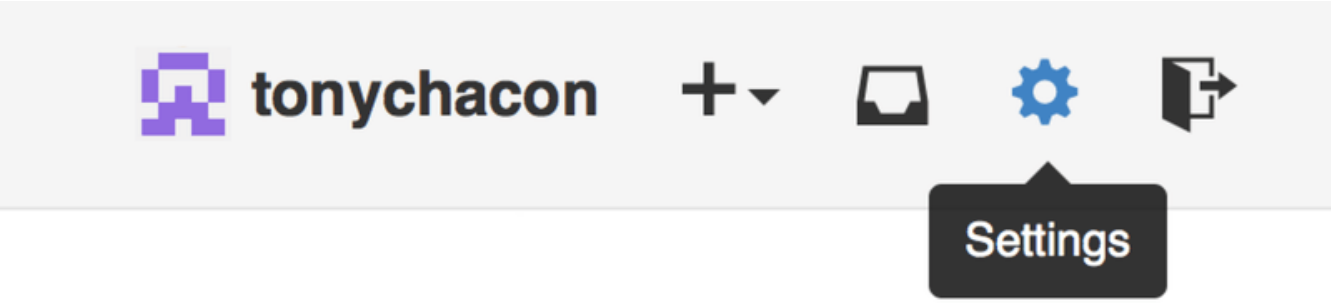


Figure 83. "Account settings" linki

Sonra sol tərəfdən "SSH keys" bölməsini seçin.

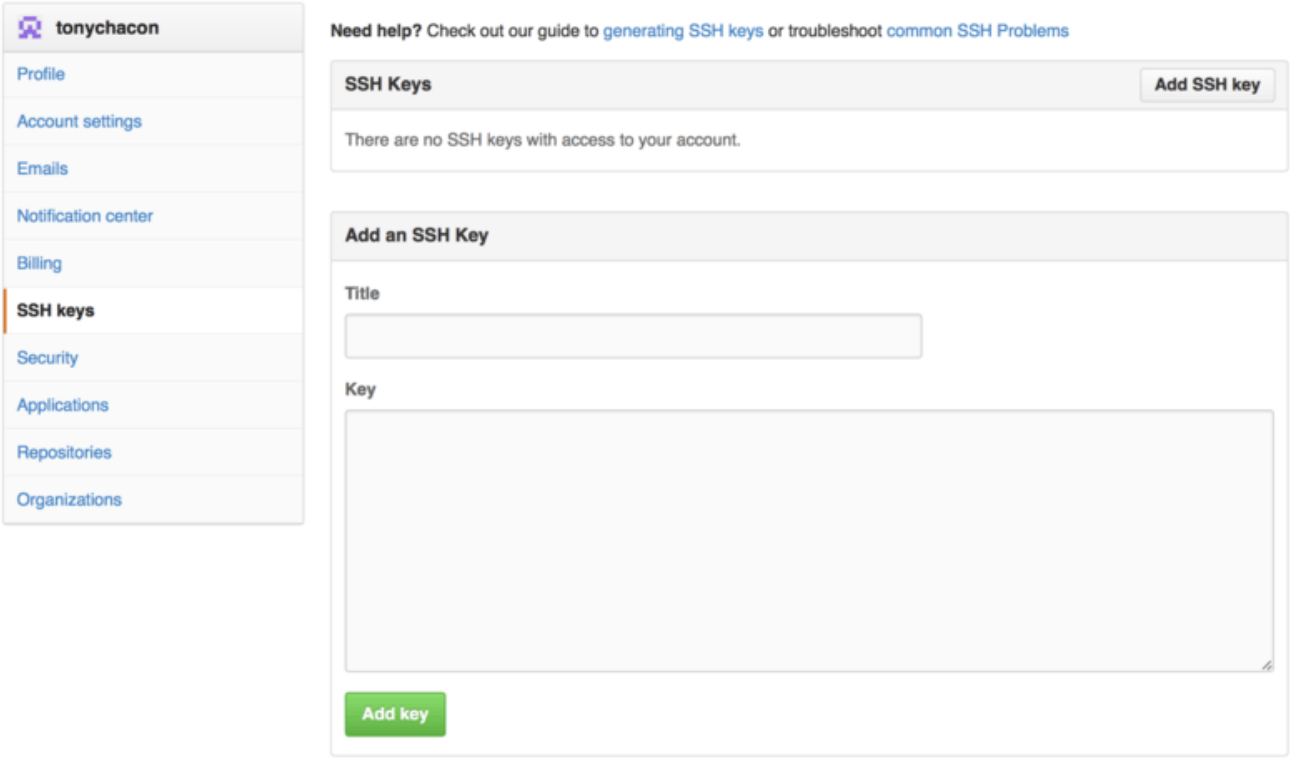


Figure 84. "SSH keys" linki

Oradan "`Add an SSH key`" düyməsini vurun, açarınıza bir ad verin, `~/.ssh/id_rsa.pub`un (və ya adını verdiyiniz hər bir) məzmununu mətnə yapışdırın ərazini seçin və "Add key" düyməsini basın.

SSH key-nizin adını xatırlaya biləcəyiniz bir şey olacağından əmin olun. Açarlarınızın hər birinə (məsələn, "My Laptop" və ya "Work Account") adını verə bilərsiniz ki, belə olduqda bir açarı sonra ləğv etmək lazımdırsa, hansını axtaracağınızı asanlıqla tapa bilərsiniz.

Avatar-ınız

Sonrakı, istəsəniz, sizin üçün yaradılan avatarı seçdiyiniz bir şəkil ilə əvəz edə bilərsiniz. Əvvəlcə “Profile” bölməsinə (SSH keys bölməsinin üstündədir) gedin və “Upload new picture” düyməsinə vurun.

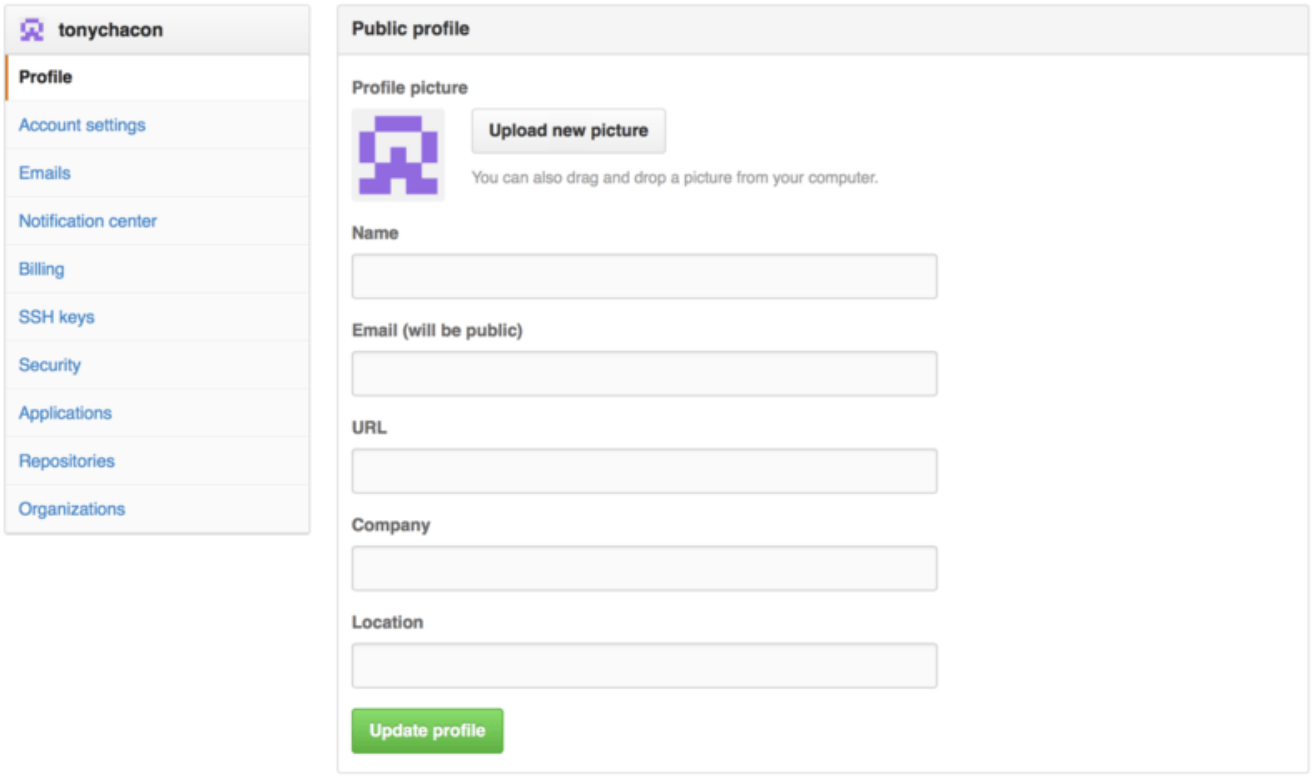


Figure 85. “Profile” linki

Sabit diskimizdə olan Git logonun bir nüsxəsini seçəcəyik və sonra onu kəsmək şansımız olacaq.

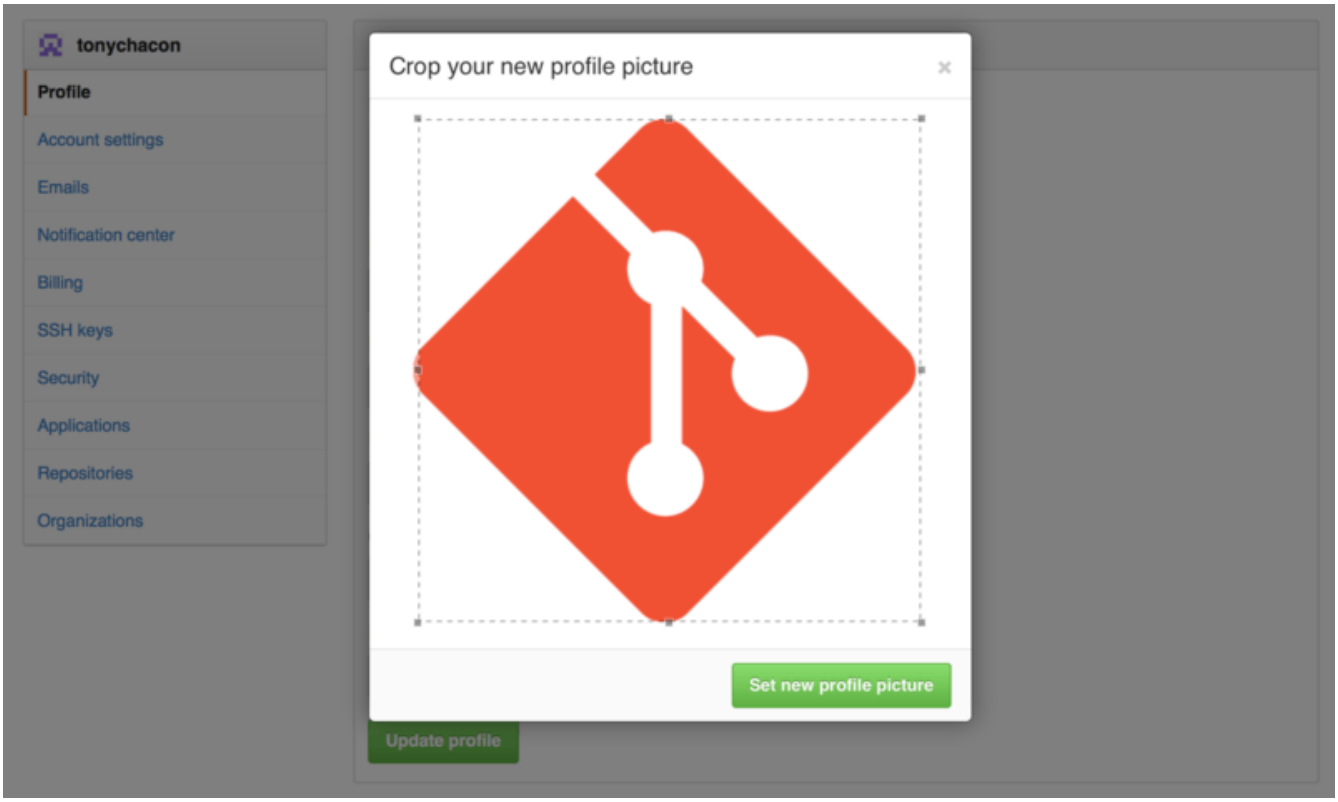


Figure 86. Avatarınızı kəsin

İndi saytda əlaqə qurduğunuz zaman insanlar avatarınızı istifadəçi adınızın yanında görəcəklər.

Məşhur Gravatar xidmətinə bir avatar yükləmisinizsə (tez-tez Wordpress hesabları üçün istifadə olunur), o avatar default olaraq istifadə olunacaq və bu addımı etmək lazım olmayacaq.

E-poçt ünvanlarınız

GitHub'un Git'inizi istifadəçinizə təqdim etməsi yolu e-poçt adresidir. Verilən tapşırıqlarda birdən çox e-poçt adresindən istifadə edirsinizsə və GitHub'un onları düzgün bir şəkildə əlaqələndirm əsini istəyirsinizsə, istifadə etdiyiniz bütün e-poçt ünvanlarını admin bölməsinin E-poçtlar bölm əsinə əlavə etməlisiniz.

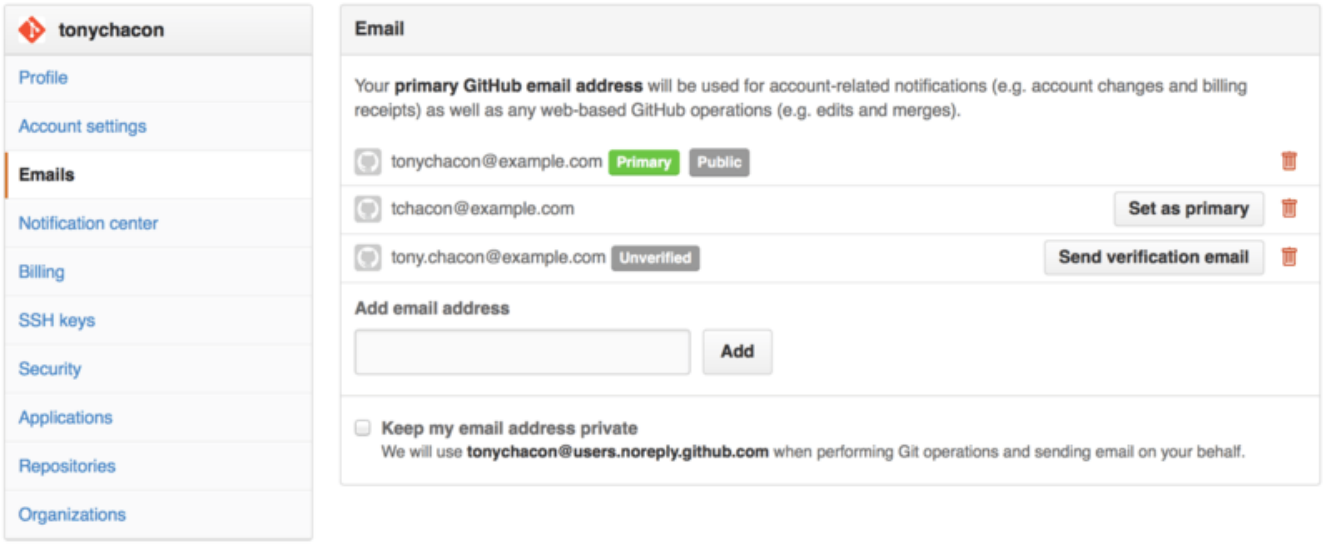


Figure 87. E-poçt ünvanlarını əlavə edin

E-poçt ünvanlarını əlavə edin-də mümkün olan bəzi fərqli vəziyyətləri görə bilərik. Üst ünvan t əsdiqləndi və əsas ünvan olaraq təyin olundu, yəni hər hansı bir bildiriş və qəbz alacağınız yerdir. İkinci ünvan təsdiqlənir və onları dəyişdirmək istəsəniz əsas olaraq təyin edilə bilərsiniz. Son ünvan təsdiqlənməmişdir, yəni onu əsas ünvanı edə bilməyəcəksiniz.r. GitHub bunlardan birini saytdakı hər hansı bir depoda mesaj göndərməkdə görürsə, indi istifadəçinizlə əlaqələndiriləc əkdir.

İki Faktorlu İdentifikasiya

Sonda əlavə təhlükəsizlik üçün mütləq iki faktorlu identifikasiya və ya "2Fİ" qurmalısınız. İki faktorlu identifikasiya, şifrənizi oğurlandığı təqdirdə hesabınızı pozma riskini azaltmaq üçün son zamanlarda daha çox populyarlaşan bir identifikasiya mexanizmidir. Onu yandırmaq GitHub sizd ən iki fərqli identifikasiya metodu tələb edəcəkdir ki, onlardan biri pozulubsa, təcavüzkar hesabınıza daxil ola bilməyəcək.

Hesab ayarlarınızın Təhlükəsizlik bölməsində İki faktorlu identifikasiya quraşdırmasını tapa bil ərsiniz.

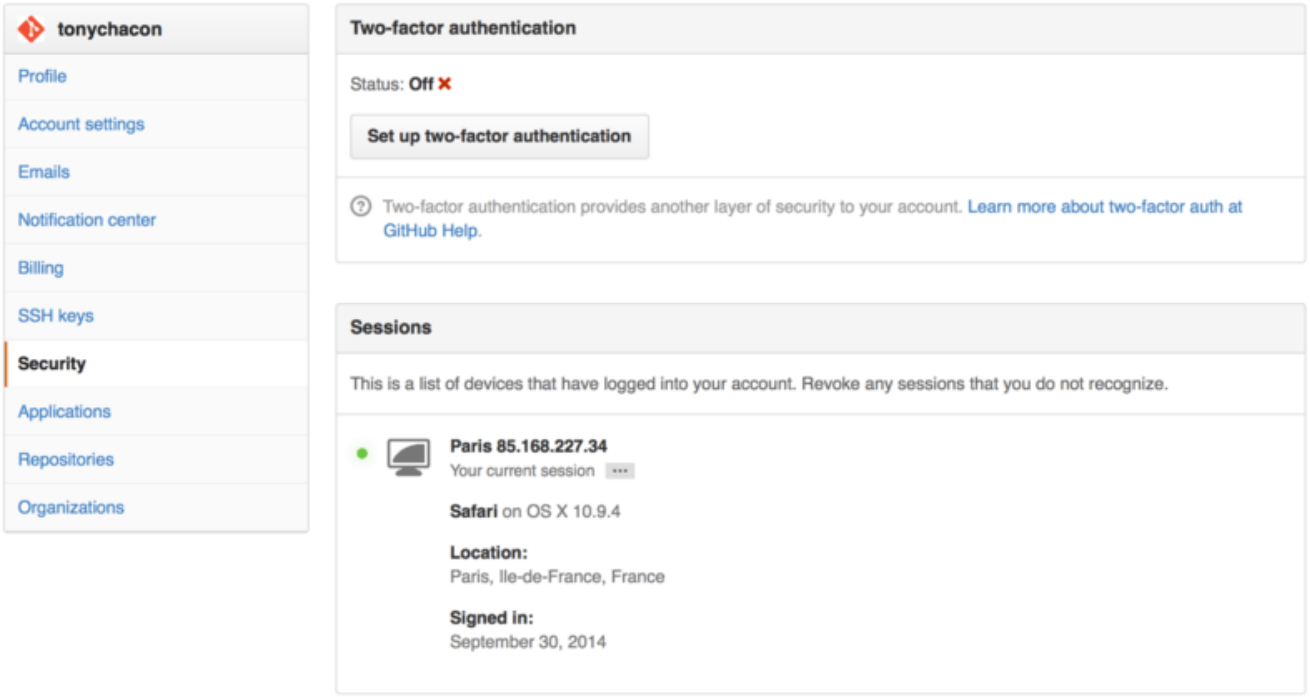


Figure 88. Təhlükəsizlik tabında 2Fİ

"İki faktorlu identifikasiya qurmaq" düyməsini kliklləsəniz, ikinci kodunuzu ("vaxt əsaslı birdəfəlik parol") yaratmaq üçün telefon tətbiqindən istifadə edə biləcəyiniz bir konfiqurasiya səhifəsinə aparacaqsınız və ya GitHub daxil olmağınız üçün hər dəfə SMS vasitəsilə bir kod göndərə bil ərsiniz.

Metodu seçdikdən və 2Fİ qurmaq üçün təlimatları izlədikdən sonra hesabınız bir az daha etibarlı olacaq və GitHub-a daxil olduqda parolunuzdan əlavə bir kod təqdim etməli olacaqsınız.

Bir Layihəyə Töhfə Vermək

Hesabımız qurulduğuna görə mövcud bir layihəyə töhfə verməyinizdə faydalı ola biləcək bəzi

detalları nəzərdən keçirək.

Forking Layihələr

Push giriş imkanı olmayan bir layihəyə töhfə vermək istəyirsinizsə, layihəni “fork” edə bilərsiniz. Bir layihəni “fork” etdikdə, GitHub tamamilə sizə məxsus olan bir layihənin bir nüsxəsini düzəldə cəkdir; o sizin namespace-nizdə yaşayır və ona push edə bilərsiniz.

Tarixən, “fork” termini kontekstdə bir qədər mənfi olmuşdur, yəni kimsə açıq bir mənbə layihəsini fərqli bir istiqamətə apardığı, bəzən rəqabətçi bir layihə yaratdığı və iştirakçıları böldüyü anlamına gəlir. GitHub-da, “fork” sadəcə öz namespace-nizdəki eyni layihədir, daha açıq bir şəkildə töhfə verməyin bir yolu olaraq bir layihədə public olaraq dəyişikliklər etməyə imkan verir.

Bu yolla, layihələr push etmək üçün istifadəçilərin tərəfdaş kimi əlavə edilməsindən narahat olmur. İnsanlar bir layihə fork edə bilər, onu push edə bilər və dəyişiklikləri orijinal depoda geri qaytara biləcəyimiz Pull Request yaradaraq geri qaytara bilər. Bu kod nəzərdən keçirilmiş müzakirə mövzusunu açır və sonra sahibi və töhfəçisi dəyişiklik barədə sahibindən razı qalana q ədər əlaqə saxlaya bilər, bu zaman sahibi onu birləşdirə bilər.

Bir layihə fork etmək üçün layihə səhifəsinə daxil olun və səhifənin yuxarı sağındakı “Fork” düym əsini basın.

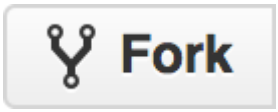


Figure 89. “Fork” düyməsi

Bir neçə saniyədən sonra kodun özünün yazıla bilən nüsxəsi ilə yeni layihə səhifənizə aparılacaqsınız.

GitHub Axını

GitHub, Pull Requests mərkəzində, müəyyən bir əməkdaşlıq işinin ətrafında hazırlanmışdır. Bu axın tək paylaşılan bir depo içərisində sıx birləşən bir komanda ilə işləməyinizə və ya qlobal miqyasda bölüşdürülən bir şirkətə və ya onlarla fork vasitəsilə bir layihəyə töhfə verən kənar şəb əkələrə işləməyinizə kömək edir. Bu Git’də Branch ilə əhatə olunmuş Mövzu Branch-ları iş axınına yönəldilmişdir.

Ümumillikdə bu necə işləyir.

1. Layihəni fork et.
2. `master`-dən mövzu branch-ı yarat.
3. Layihəni yaxşılaşdırmaq üçün bəzi commit-lər verin.
4. Bu branch-ı GitHub layihənizə push edin.
5. Github-da Pull Request-i açın.
6. Müzakirə edin və commit etməyə davam edin.

7. Layihə sahibi Pull Request-i birləşdirir və ya bağlayır.
8. Yenilənmiş master-i fork-a sinxronlaşdırın.

Bu əsasən İnteqrasiya-Menecer İş Axınları-da İnteqrasiya Meneceri iş axınıdır, lakin dəyişiklikləri ünsiyyət və nəzərdən keçirmək üçün e-poçtdan istifadə etmək əvəzinə komandalar GitHub-un veb əsaslı vasitələrindən istifadə edirlər.

Bu axını istifadə edərək GitHub'da açıq bir mənbə layihəsinə dəyişiklik təklif etmək nümunəsini n əzərdən keçirək.

Pull Request Yaratmaq

Tony, Arduino ilə proqramlaşdırıla bilən mikrokontrolleri işə salmaq üçün kod axtarır və Github-da https://github.com/schacon/blink-də əla program faylı tapır.

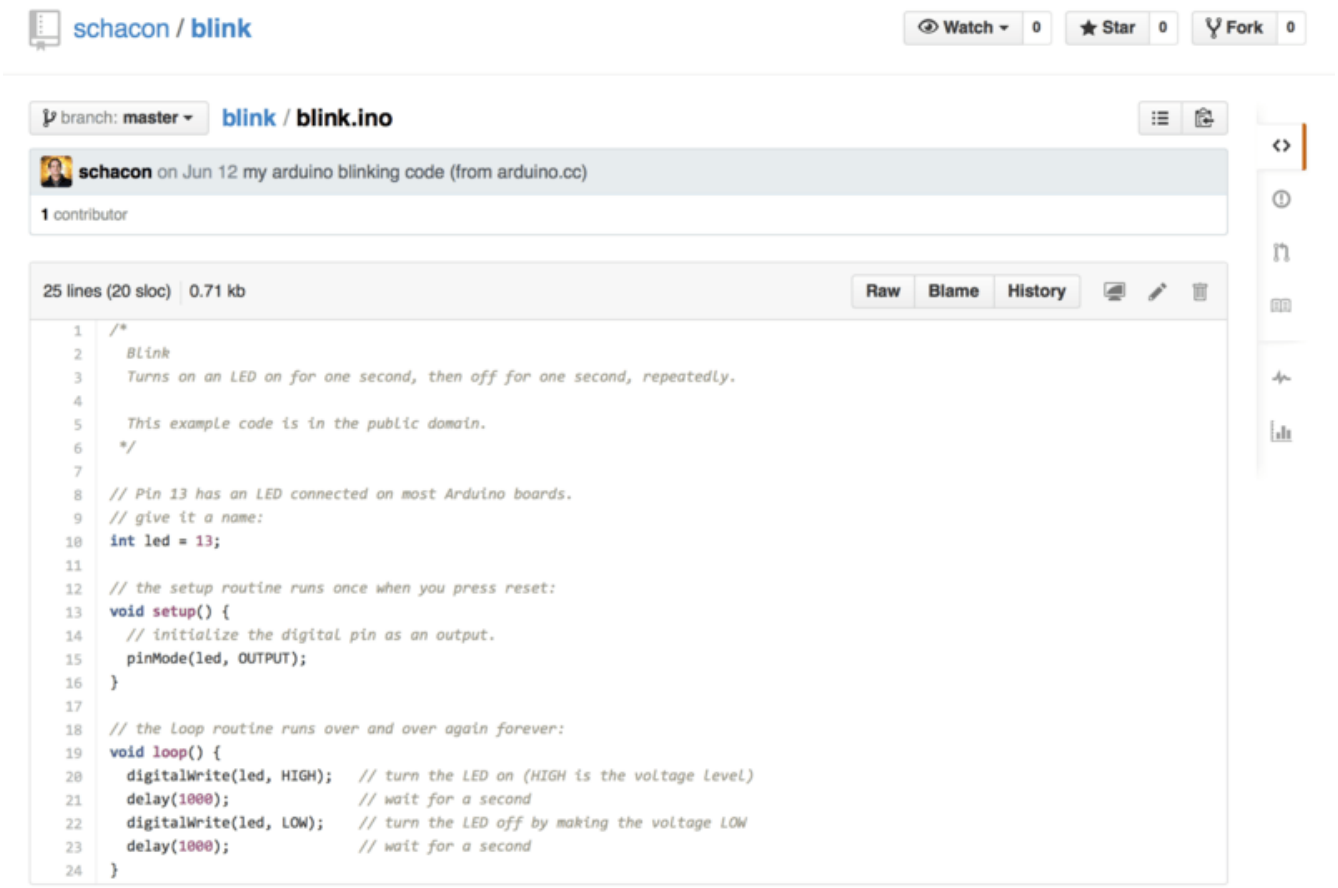


Figure 90. Töhfə vermək istədiymiz layihə

Yeganə problem yanıb-sönmə sürətinin çox sürətli olmasıdır. Düşünürük ki, hər bir vəziyyət d əyişikliyi arasındakı 1-i əvəzinə 3 saniyə gözləmək daha yaxşıdır. Beləliklə, proqramı təkmilləşdir ək və təklif olunan dəyişiklik kimi yenidən layihəyə təqdim edək.

Birincisi, layihənin öz nüsxəsini əldə etmək üçün əvvəlcədən qeyd edildiyi kimi *Fork* düyməsini sıxırıq. Burada istifadəçi adımız "tonychacon" olduğundan bu layihənin kopyası və onu redaktə edə biləcəyimiz yer `https://github.com/tonychacon/blink`-dır . Local şəkildə klonlaşdıracağıq, bir mövzu branch-ı yaradacağıq, kod dəyişikliyini edib nəhayət bu dəyişikliyi yenidən GitHub-a köçürəcəyik.

```
$ git clone https://github.com/tonychacon/blink ①
Cloning into 'blink'...

$ cd blink
$ git checkout -b slow-blink ②
Switched to a new branch 'slow-blink'

$ sed -i '' 's/1000/3000/' blink.ino (macOS) ③
# If you're on a Linux system, do this instead:
# $ sed -i 's/1000/3000/' blink.ino ③

$ git diff --word-diff ④
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  [-delay(1000);-]{+delay(3000);+}               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  [-delay(1000);-]{+delay(3000);+}               // wait for a second
}

$ git commit -a -m 'Change delay to 3 seconds' ⑤
[slow-blink 5ca509d] Change delay to 3 seconds
 1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin slow-blink ⑥
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tonychacon/blink
 * [new branch]      slow-blink -> slow-blink
```

① Layihənin fork-nu local olaraq klonlayın.

② Açıqlayıcı mövzu branch-ı yaradın.

③ Kod dəyişikliyimizi edin.

④ Dəyişiklikin yaxşı olduğunu yoxlayın.

⑤ Dəyişikliklərimizi mövzu branch-na commit edin.

⑥ Yeni mövzu branch-mızı GitHub fork-na geri push edin.

İndi GitHub üzərindəki fork-muza qayıtsaq, görərik ki, GitHub-ın yeni bir mövzu branch-nı

yuxarıya push etdiyimizi fərq etdiyini və dəyişikliklərimizi yoxlamaq və orijinal layihəyə Pull Request açmaq üçün bizə böyük bir yaşıl düyməni təqdim etdiyini görə bilərik.

Alternativ olaraq, `https://github.com/<user>/<project>/branches`-da “Branches” səhifəsinə keçərək branch-ınızı tapmaq və oradan yeni Pull Request açmaq olar.

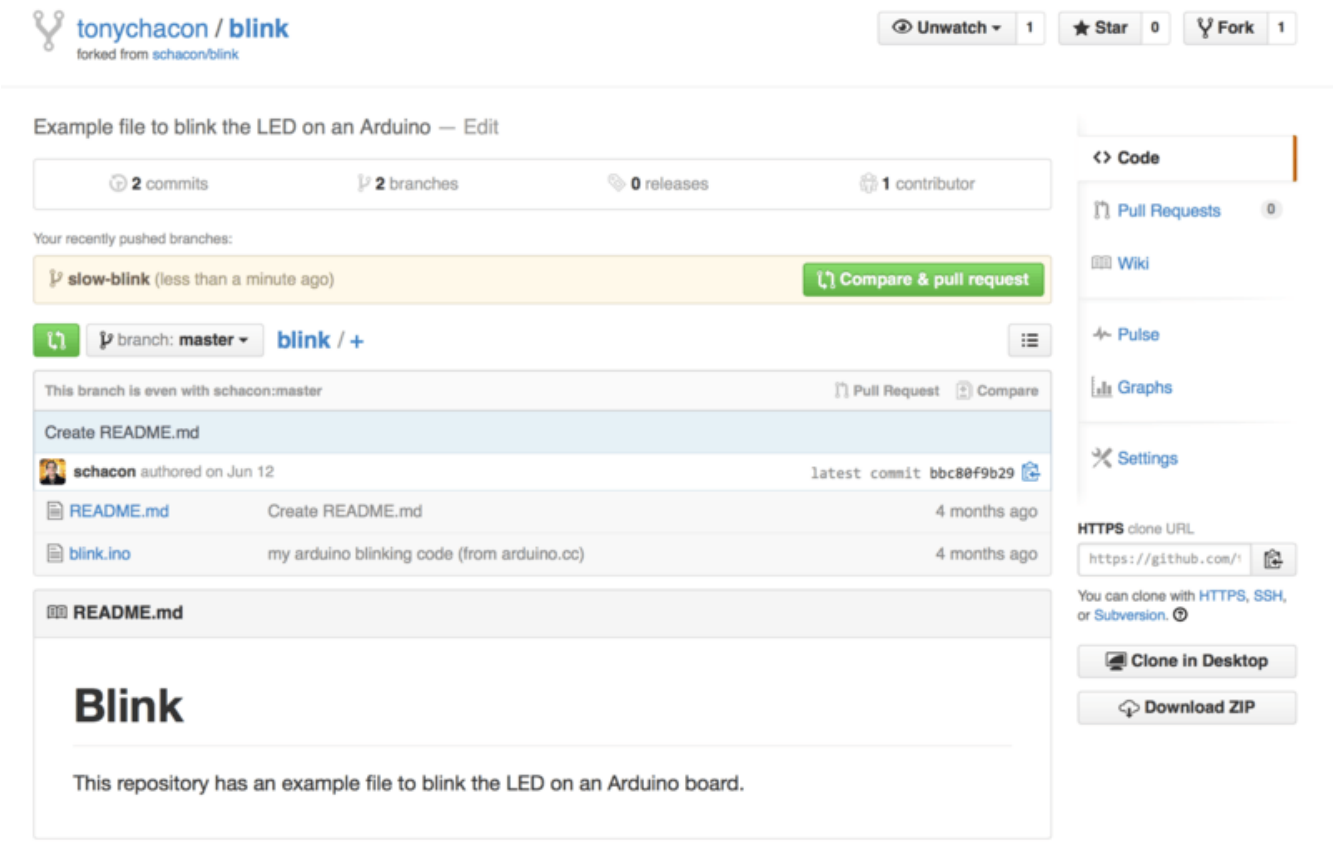


Figure 91. Pull Request düyməsi

Bu yaşıl düyməni bassaq, Pull Request’ə başlıq və təsvir verməyimizi xahiş edən bir ekran görərik. Buna bir az effort göstərmək demək olar ki, həmişə dəyərlidir, çünki yaxşı bir təsvir orijinal layih ənin sahibinə nə etmək istədiyinizi, təklif olunan dəyişikliklərin düzgün olub olmadığını və d əyişikliklərin qəbul edilməsinin orijinal layihəni yaxşılaşdıracağını müəyyən etməyə kömək edir.

Mövzu branch-mızda da `master` branch-nın (bu vəziyyətdə yalnız bir) “əvvəlində” olan commit-l ərin bir hissəsini və layihə sahibi tərəfindən birləşdirilməsi durumunda bu branch əldə edəcəyi bütün dəyişikliklərin vahid fərqliliyini görürük.

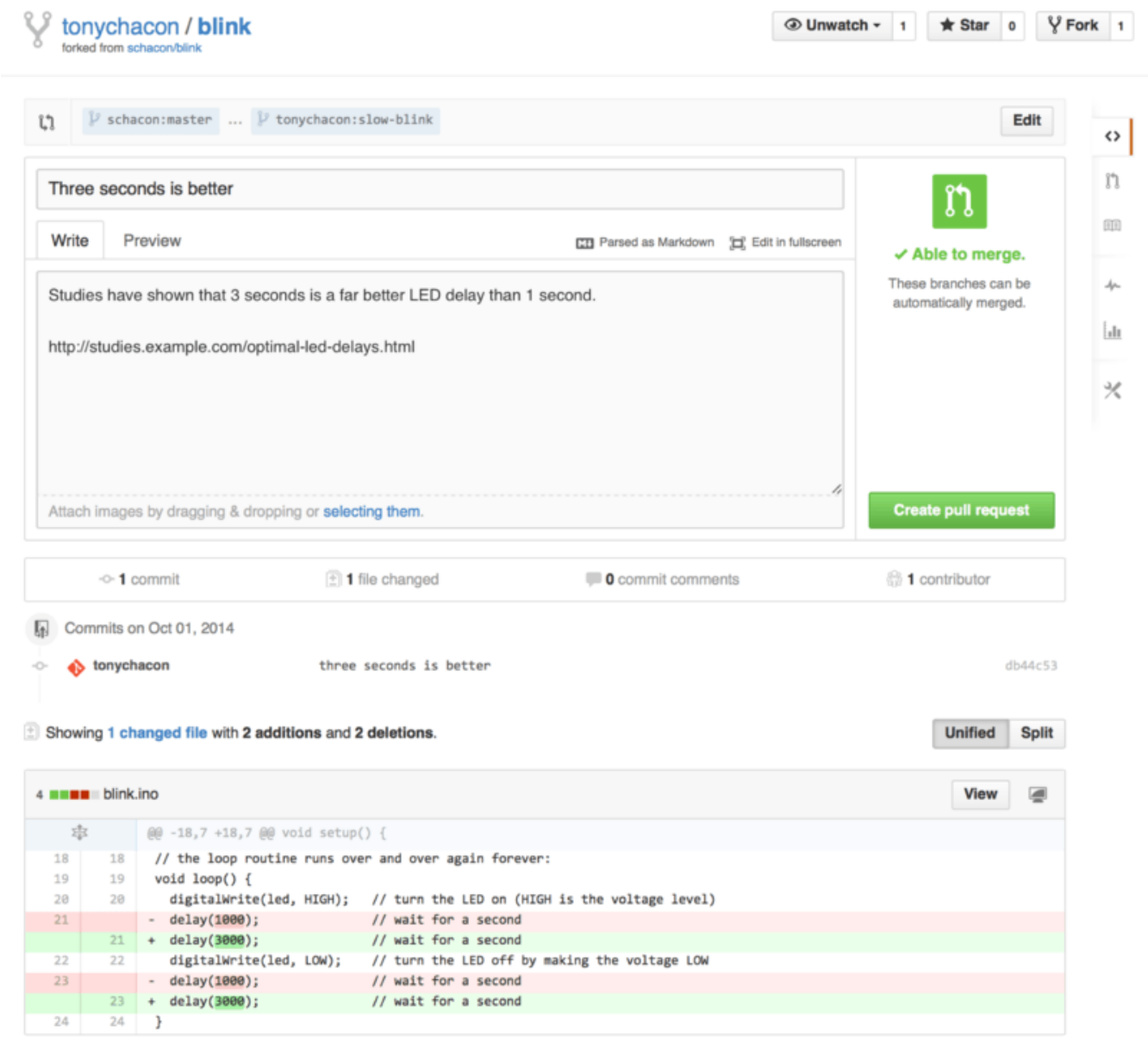


Figure 92. Pull Request yaratma səhifəsi

Bu ekrandakı *Create pull request* düyməsini basdığınızda, forked etdiyiniz layihənin sahibi kimsə dəyişiklik təklif etdiyi barədə bir bildiriş alacaq və bu məlumatların hamısını özündə saxlayan s əhifəyə bağlayacaq.

Töhfə edəcək şəxs dəyişikliyə hazır olduqda bu kimi public layihələr üçün tez-tez istifadə olunur, baxmayaraq ki, bu, inkişaf dövrünün *başlanğıcında* olan daxili layihələrdə də istifadə olunur. Pull Request açıldıqdan **sonra** mövzu bölməsinə pushing davam edə bildiyiniz üçün,ümumulikdə tez açılır və prosesin sonunda açılmaq əvəzinə, bir kontekstdə bir komanda şəklində işləməyi təkrarlamaq üçün istifadə olunur.

Pull Request ilə Təkrarlama

Bu anda, layihə sahibi təklif olunan dəyişikliyə baxa bilər və onu birləşdirə bilər, rədd edə və ya ş ərh verə bilər. Deyək ki, o fikri bəyənir, amma işığın sönülü qalması üçün biraz daha uzun müddə tə üstünlük verir.

GitHub-da Paylanmış Git-də təqdim olunan iş axınlarında e-poçt üzərində və bu söhbətin baş verə biləcəyi yerlərdə bu onlayn rejimdə baş verir. Layihə sahibi vahid fərqi nəzərdən keçirə bilər və s ətirlrdən hər hansı birini tıklayaraq rəy yaza bilər.

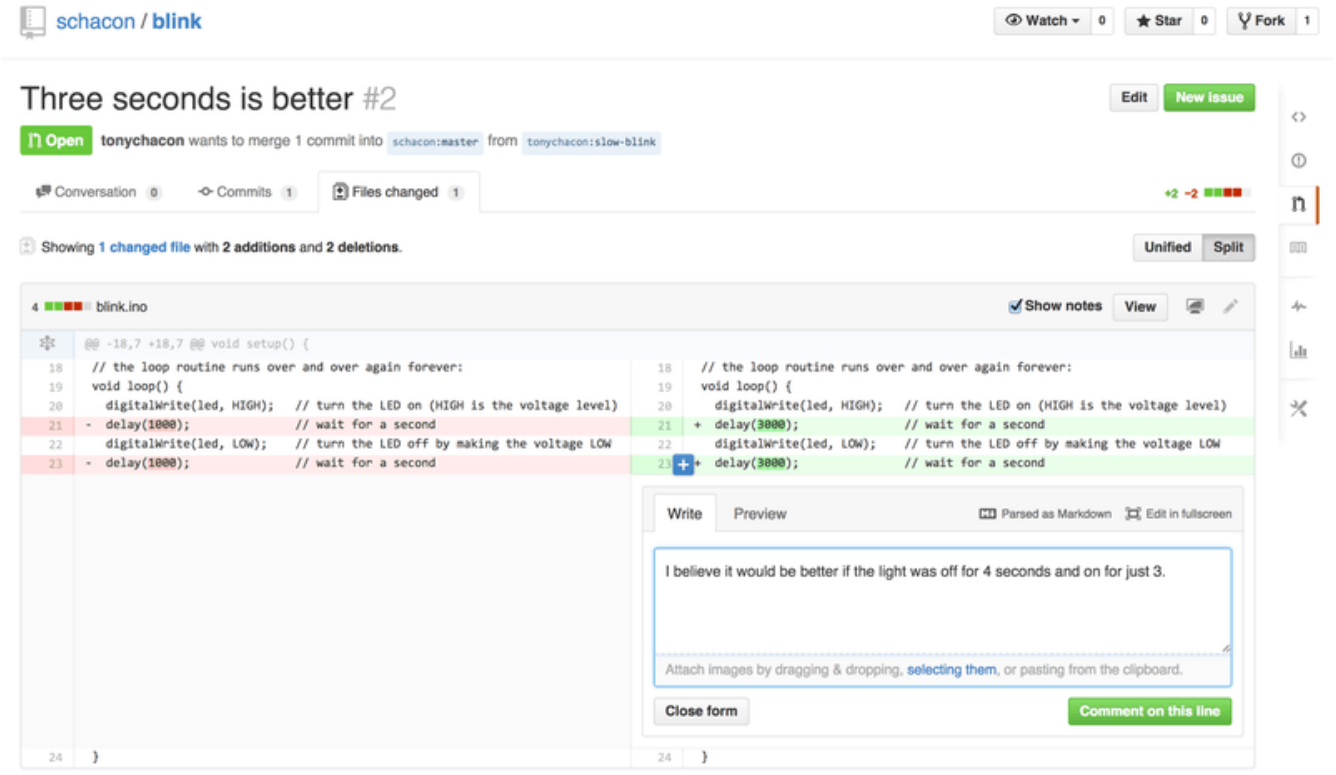


Figure 93. Pull Request-də müəyyən bir kod xəttinə şərh verin

Təminatçı bu açıqlamanı verdikdən sonra Pull Request-i açan şəxs (və həqiqətən, deponu seyr edən hər kəs) bir bildiriş alacaq. Daha sonra bunu özəlləşdirməyə keçəcəyik, amma e-poçt bildirişləri olsaydı, Tony bu kimi bir e-poçt alacaq:

Figure 94. E-poçt bildirişləri olaraq göndərilən şərhlər

Hər kəs Pull Request-də ümumi şərhlər buraxa bilər. Pull Request müzakirə səhifəsi-də layihə sahibinin həm kod xəttini şərh etdiyini, həm də müzakirə bölməsində ümumi bir şərh buraxmağını görə bilərik. Kod şərhlərinin də söhbətə gətirildiyini görə bilərsiniz.

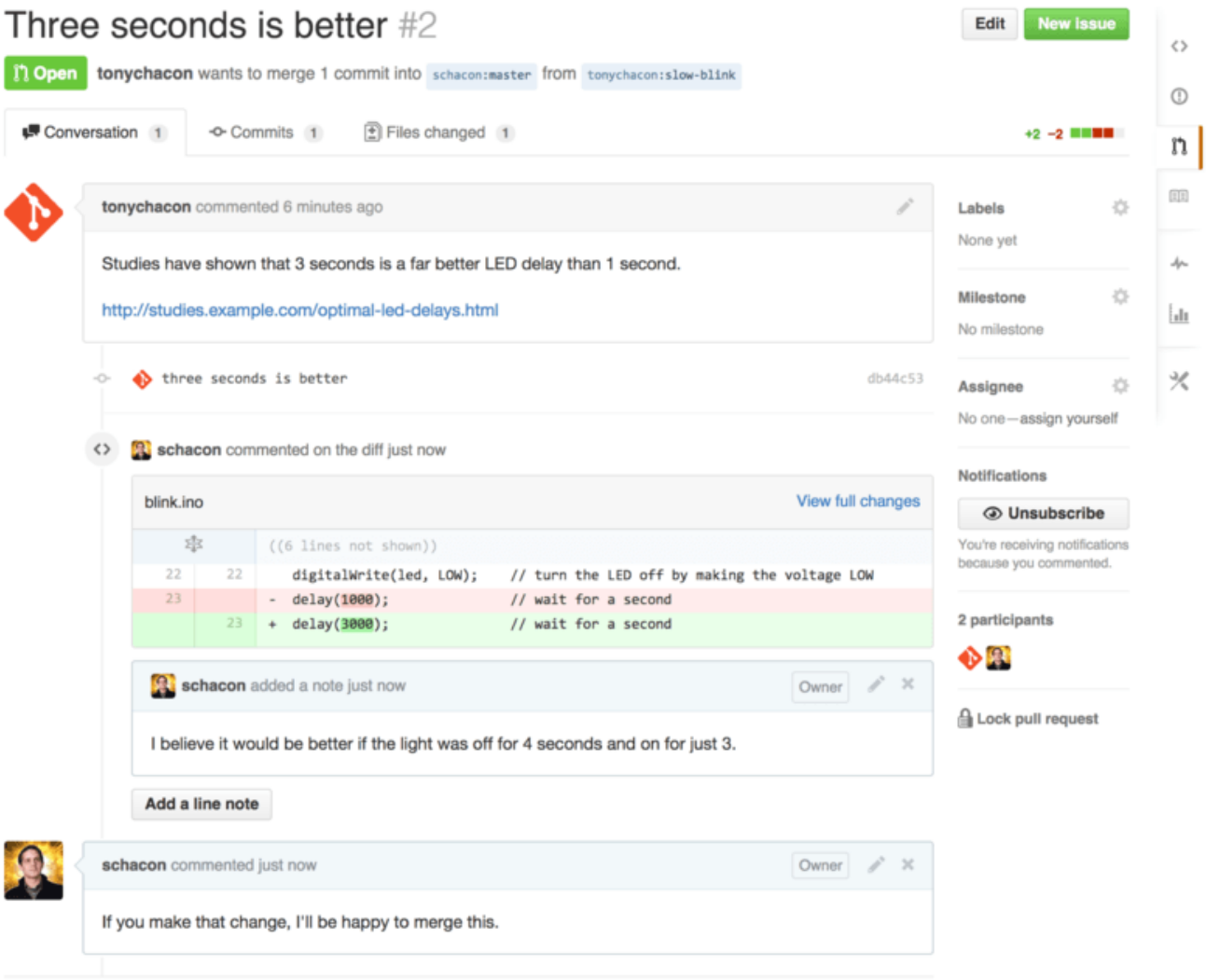


Figure 95. Pull Request müzakirə səhifəsi

İndi töhfəçi dəyişikliklərin qəbul edilməsi üçün nə etməli olduqlarını görə bilər. Xoşbəxtlikdən bu çox sadədir. E-poçt üzərində seriyalarınızı yenidən yuvarlaqlaşdırmaq və poçt siyahısına yenidən göndərmək məcburiyyətində ola bilərsiniz, GitHub ilə mövzu branch-na yenidən qoşulun və avtomatik olaraq Pull Request-i yeniləyin. Pull Request final da köhnə kod şərhinin yenilənmiş Pull Request-də daraldığını görə bilərsiniz, çünki o vaxtdan bəri dəyişdirilmiş bir xətt üzərində hazırlanmışdır.

Mövcud Pull Request commit etməyə bildiriş vermir, buna görə Tony düzəlişlərini push etdikdən sonra layihə sahibinə tələb olunan dəyişikliyi etdiyini bildirmək üçün rəy bildirməyə qərar verir.

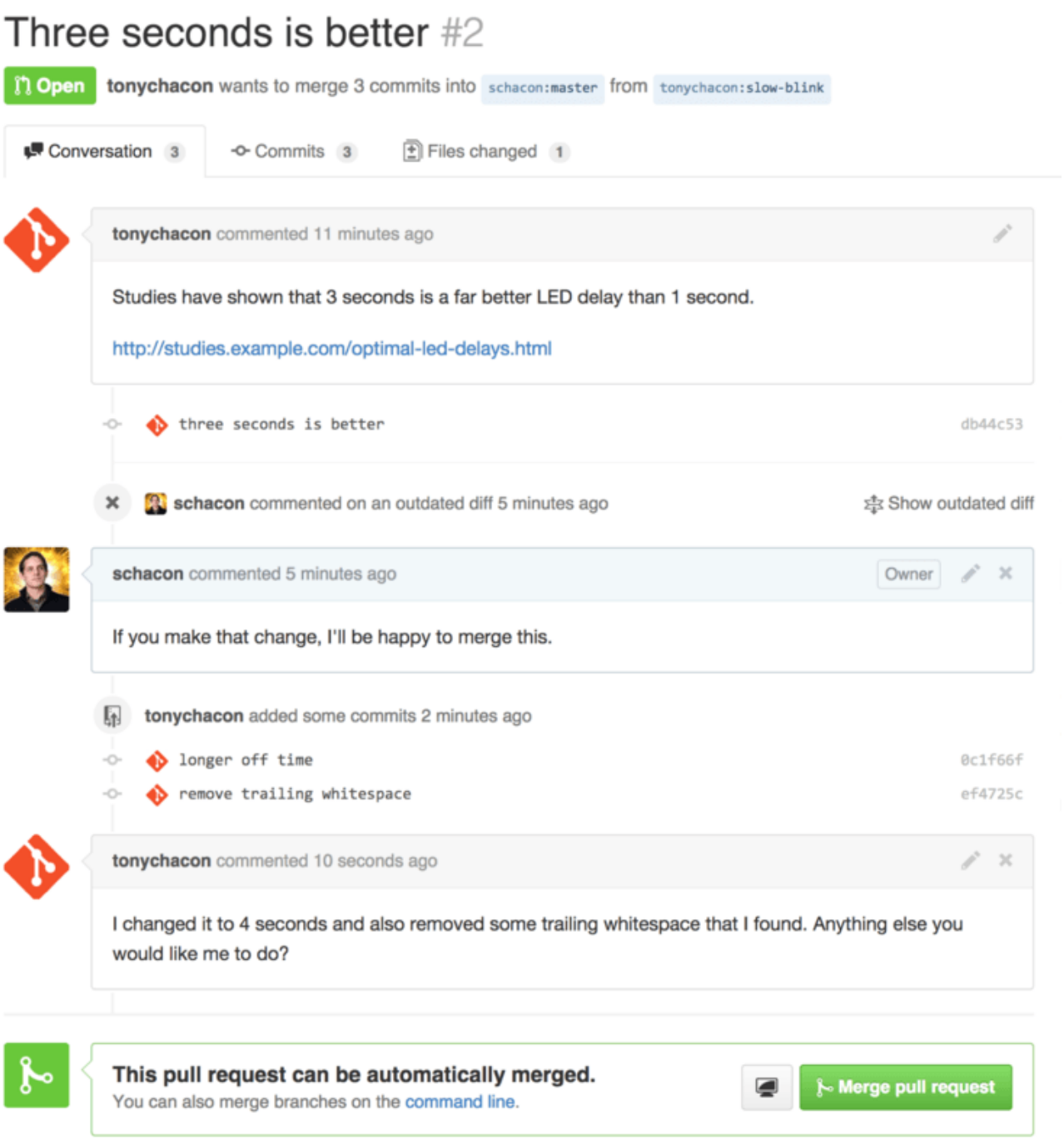


Figure 96. Pull Request final

Diqqəti cəlb edən bir maraqlı məqam budur ki, bu Pull Request-dəki “Files Changed” sekmesini klikləsəniz, “unified” edilmiş fərqini əldə edəcəksiniz - yəni sizə təqdim ediləcək ümumi məcmu f ərq Bu mövzu branch-ı birləşdirildiyi təqdirdə əsas branchdır. `git diff` baxımından, bu avtomatik olaraq bu Pull Request-in dayandığı branch üçün avtomatik olaraq `git diff master...<branch>` göstərir. Bu növ fərq haqqında daha çox məlumat üçün Nəyin Təqdim Olunduğunu Müəyyənl əşdirmək səhifəsinə baxın.

Gördüyünüz digər bir şey, GitHub Pull Request-nin təmiz bir şəkildə birləşdiyini və serverdə birl əşmə üçün bir düyməni təmin etdiyini yoxlamaqdır. Bu düymə yalnız depoya yazma imkanınız varsa və mənasız birləşmə mümkündürsə göstərilir. Bu düyməni basarsanız, GitHub birləşmə sür ətli ola bilsə də, yenə də birləşmə commit-i yaradacaq deməkdir.

İstəsəniz, branch-ı sadəcə pull down edib local olaraq birləşdirə bilərsiniz. Bu branch-ı `master`

branch-ına birləşdirsəniz və GitHub'a push etsəniz, Pull Request-i avtomatik olaraq bağlanacaqdır.

Bu GitHub layihələrinin çoxunun istifadə etdiyi əsas iş axınlarıdır. Mövzu branch-ları yaradılır, Pull Request-lər açılır, müzakirə baş verir, bəlkə branch üzərində daha çox iş görülür və nəticədə sorğu ya bağlanır, ya da birləşdirilir.

Yalnız Forks Deyil

Qeyd etməyiniz vacibdir ki, eyni depo içərisində iki branch arasında Pull Request açа bilərsiniz. Biri ilə bir xüsusiyyət üzərində işləyirsinizsə və hər ikinizin də layihəyə yazılı giriş imkanınız varsa, bir mövzu branch-nı depoya göndərə bil ərsiniz və kodu başlatmaq üçün eyni layihənin `master` branch-na Pull Request açа bilərsiniz. Forking-ə ehtiyac yoxdur.

Ətraflı Pull Request-lər

İndi GitHub-da bir layihəyə töhfə vermək əsaslarını izah etdik. İndi isə onlardan istifadə etməkdə daha effektiv olmağınız üçün Pull Requests ilə bağlı bir neçə maraqlı məsləhət və tövsiyələri nəz ərdən keçirək.

Pull Requests Patch-lar kimi

Bir çox layihənin həqiqətən Pull Requests üçün təmiz tətbiq edilməli mükəmməl patch-lar sırası kimi düşünmədiklərini başa düşmək vacibdir, çünki əksər poçt siyahısına əsaslanan layihələr patch seriyalarını düşünür. GitHub layihələrinin əksəriyyəti birləşmə ilə tətbiq olunan vahid bir f ərqlə sona çatan təklif olunan dəyişiklik ətrafında iterativ söhbətlər kimi Pull Request branch-ları haqqında düşünür.

Bu vacib bir fərqdir, çünki dəyişiklik ümumiyyətlə kodun mükəmməl olduğu düşünülməmişdən əvvəl təklif olunur, bu da poçt siyahısına əsaslanan patch seriyası töhfələri ilə daha nadirdir. Bu, t əmirçilərlə əvvəlcədən söhbət etməyə imkan verir ki, lazımi həll yolu tapmaq daha çox cəmiyyətin efortunu artırsın. Pull Request ilə kod təklif edildikdə və texniki işçilər və ya icma bir dəyişiklik t əklif edərsə, patch seriyası ümumiyyətlə yenidən yuvarlanmır, əksinə fərqi branch-a yeni bir commit kimi push edilir, söhbəti kontekstində irəli aparır.

Məsələn, geri qayıdıb yenidən Pull Request final -ə baxsanız, əmanətçinin etdiyi commit-i geri qaytarmadığını və başqa Pull Request göndərmədiyini görəcəksiniz. Bunun əvəzinə yeni commit-l ər əlavə etdilər və mövcud branch-a push etdilər. Bu yolla geri dönsəniz və gələcəkdə bu Pull Request-ə baxsanız, qərarların niyə verildiyi kontekstində asanlıqla tapa bilərsiniz. Saytdakı "Merge" düyməsini basaraq Pull Request-ə istinad edən birləşmə əməliyyatı yaradır ki, geri qayıtmaq və lazım olduqda orijinal söhbəti araşdırmaq asandır.

Upstream ilə Davam Etmək

Pull Request köhnəlirsə və ya başqa cür təmiz şəkildə birləşmirsə, təmirçi asanlıqla birləşə bilməsi üçün onu düzəltmək istəyəcəksiniz. GitHub bunu sizin üçün sınayacaq və Pull Request-in altındakı birləşmənin mənasız olub-olmadığı barədə sizə bildirəcək.

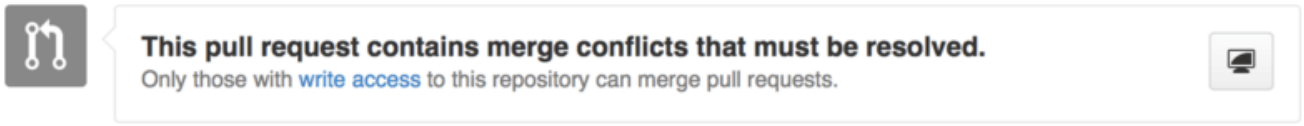


Figure 97. Pull Request təmiz birləşmir

Əgər Pull Request təmiz birləşmir kimi bir şey görsəniz, branch-ınızı yaşıl rəngə çevirməsini və t əmirçi əlavə iş görməməsi üçün düzəltmək istəyəcəksiniz.

Bunu etmək üçün iki əsas seçiminiz var. Siz ya da branch-nızı hədəf branch-ın hər hansı birinin üstünə qaytara bilərsiniz (normal olaraq forked etdiyiniz depo-nun `master` branch-ı) və ya hədəf branch-ı branch-nıza birləşdirə bilərsiniz.

GitHub-da developerlərin əksəriyyəti əvvəlki hissədə danışdığımız səbəblərə görə son hissəni etm əyi seçəcək. Əhəmiyyətli olan tarix və son birləşmə, buna görə reabilitasiya biraz daha təmiz bir tarixdən daha çox şey almır və bunun əvəzinə **çox daha** çətin və səhvlərə meyllidir.

Pull Request-nizi birləşdirilə bilmək üçün hədəf branch-na birləşdirmək istəyirsinizsə, orijinal deponu yeni bir uzaqdan əlavə edin, ondan götürün, həmin deponun əsas branch-nı mövzu branch-na birləşdirin, hər hansı bir problemi həll edin və nəhayət geri push edin Pull Request açdığınız eyni branch-a geri göndərin.

Məsələn, deyək ki, əvvəllər istifadə etdiyimiz “tonychacon” nümunəsində, orijinal müəllif Pull Request-ində konflikt yaradan bir dəyişiklik etdi. Gəlin bu addımlardan keçək.

```
$ git remote add upstream https://github.com/schacon/blink ①

$ git fetch upstream ②
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
Unpacking objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
From https://github.com/schacon/blink
 * [new branch]      master     -> upstream/master

$ git merge upstream/master ③
Auto-merging blink.ino
CONFLICT (content): Merge conflict in blink.ino
Automatic merge failed; fix conflicts and then commit the result.

$ vim blink.ino ④
$ git add blink.ino
$ git commit
[slow-blink 3c8d735] Merge remote-tracking branch 'upstream/master' \
    into slower-blink

$ git push origin slow-blink ⑤
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 682 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To https://github.com/tonychacon/blink
   ef4725c..3c8d735  slower-blink -> slow-blink
```

① Orijinal depoya `upstream` adı verilən bir remote kimi əlavə edin.

② Ən yeni işi remote fetch edin.

③ Həmin deponun əsas branch-nı mövzu branch-na birləşdirin.

④ Baş verən konflikti həll edin.

⑤ Eyni mövzu branch-na geri push edin.

Bunu etdikdən sonra Pull Request avtomatik olaraq yenilənir və təmiz birləşdiyini yoxlamaq üçün yenidən yoxlanılır.

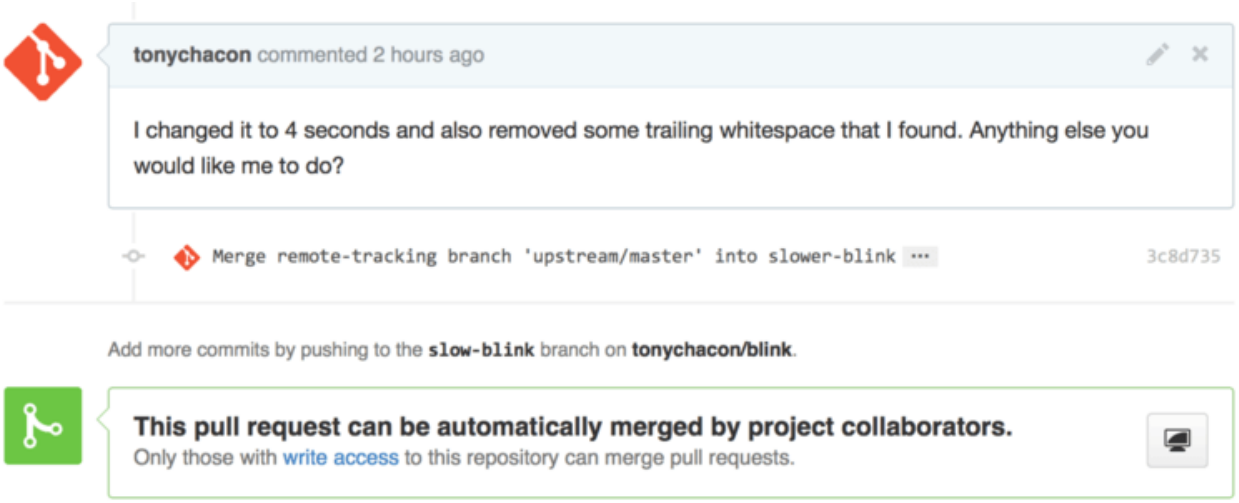


Figure 98. Pull Request indi təmiz birləşir

Git ilə əlaqəli ən gözəl şeylərdən biri də davamlı bunu edə biləcəyinizdir. Çox uzun müddətdir davam edən bir layihəniz varsa, hədəf branch-ından asanlıqla təkrar-təkrar birləşə bilərsiniz və yalnız birləşdiyiniz müddətdən bəri yaranan konfliktləri həll etmək məcburiyyətindəsiniz və bu prosesi çox idarə edə bilərsiniz.

Branch-ı təmizləmək üçün istədiyiniz kimi tamamilə dəyişdirə bilərsiniz, ancaq Pull Request-in artıq açıldığı branch-ı push etməməyiniz tövsiyə olunur. Başqa insanlar onu aşağı pull edib daha çox iş görsələr, Rebasing-in Təhlükələri-də göstərilən məsələlərin hamısına qoşulursunuz. Bunun əvəzinə, yenidən satılan branch-ı GitHub-dakı yeni bir branch-a push edin və köhnəsinə istinad ed ən yeni bir Pull Request açın, sonra orijinalı bağlayın.

İstinadlar

Növbəti sualınız “How do I reference the old Pull Request?” ola bilər. GitHub-da yaza biləcəyiniz başqa yerlərə istinad etmək üçün bir çox yolun var.

Başqa bir Pull Request-i və ya bir məsələni necə cross-reference edəcəyimizdən başlayaq. Bütün Pull Request və məsələlərinə nömrələr verilir və onlar layihə çərçivəsində unikaldır. Məsələn, Pull Request #3 və Issue #3.yoxdur. Başqa birindən hər hansı bir Pull Request və ya Issue-a istinad etm ək istəyirsinizsə, sadəcə hər hansı bir şərhdə və ya təsvirdə `#<num>` qoya bilərsiniz. Issue və ya Pull request başqa bir yerdə yaşayırsa, daha konkret ola bilərsiniz; Girdiyiniz deponun bir fork-unda Bir Issue və ya Pull Request-nə müraciət etsəniz, `username#<num>` yazın və ya başqa bir depoda bir şeyə istinad etmək üçün `username/repo#<num>` yazın.

Bir nümunəyə baxaq. Əvvəlki misalda branch-ı yenidən düzəltdik, bunun üçün yeni bir pull request yaratdıq və indi köhnə pull request-ni yenisindən istinad etmək istədiyimizi varsayaq. Ayrıca deponun fork-undakı bir məsələyə və tamamilə fərqli bir layihədəki bir məsələyə istinad etmək istəyirik. Təsviri yalnız Cross references in a Pull Request kimi doldura bilərik.

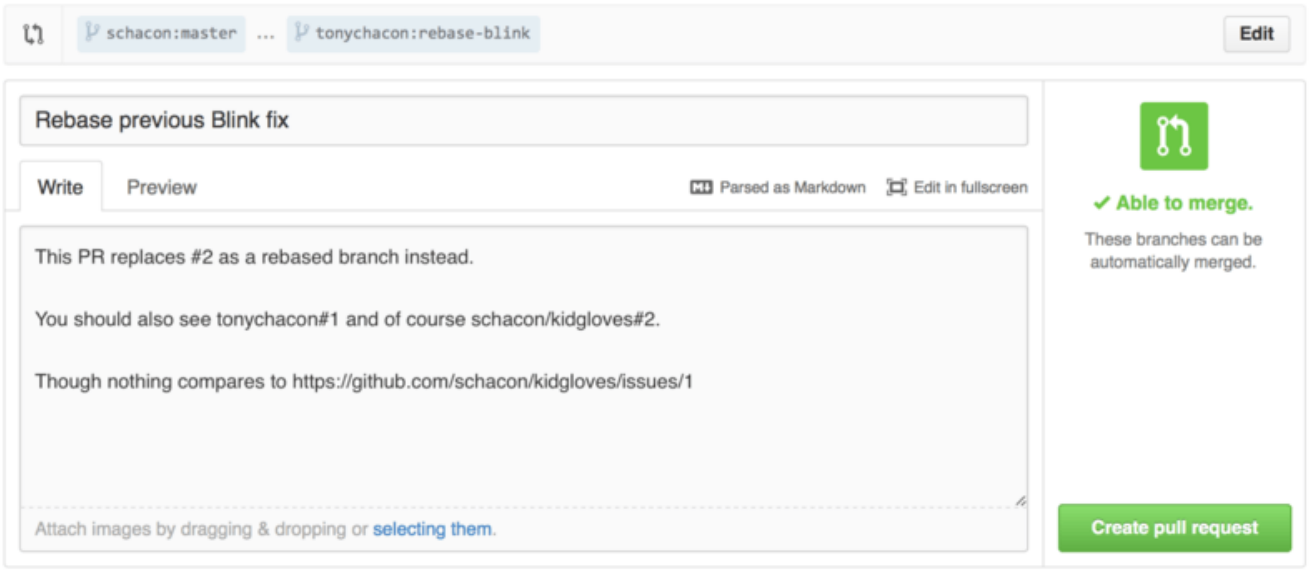


Figure 99. Cross references in a Pull Request

When we submit this pull request, we'll see all of that rendered like Pull Request-də göstərilən arayışlar.

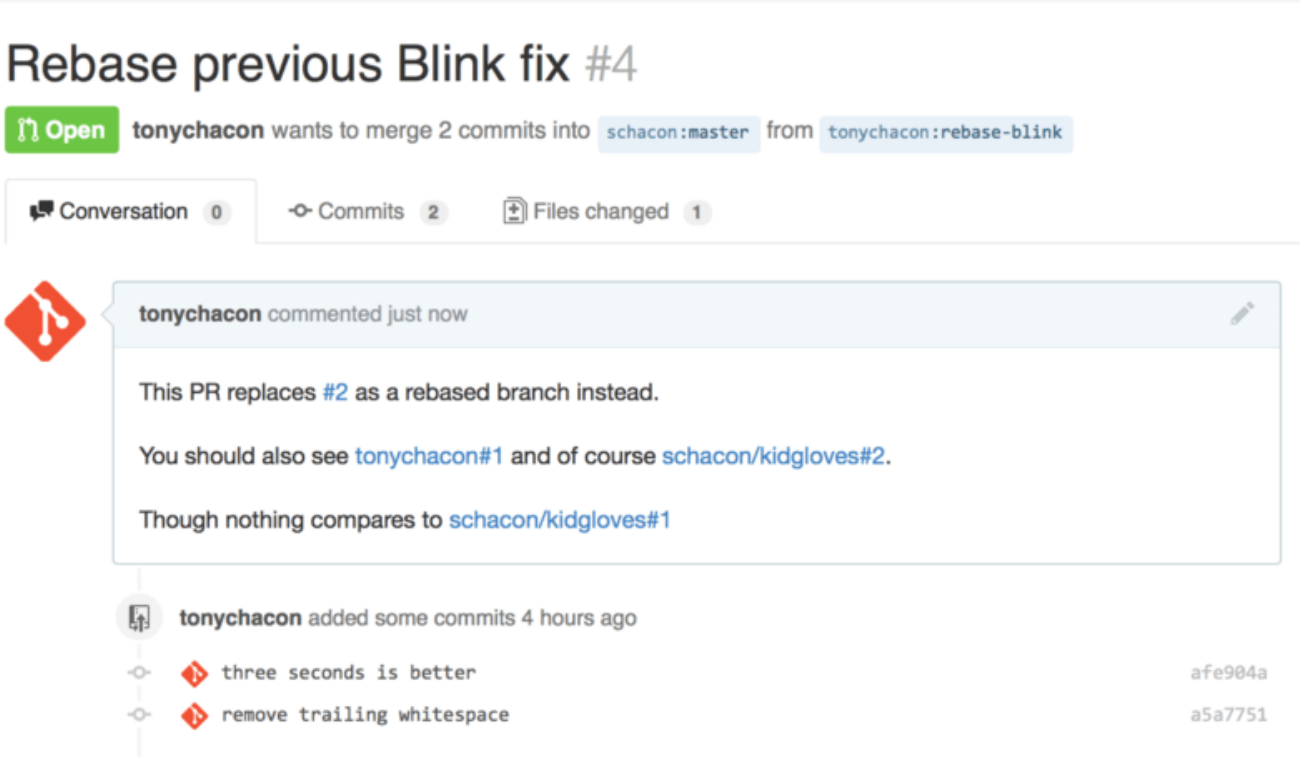


Figure 100. Pull Request-də göstərilən arayışlar

Diqqət yetirin ki, oraya qoyduğumuz GitHub URL-i yalnız lazım olan məlumatlara qısaldılmışdır.

İndi Tony geri qayıtsa və orijinal Pull Request-i bağlasa, GitHub'ın avtomatik olaraq Pull Request timeline-da bir izləmə hadisəsi yaratdığını görə bilərik. Bu o deməkdir ki, bu Pull Request-i ziyarət edən və qapalı olduğunu görən hər kəs onu əvəz edənə asanlıqla geri qayıda bilər. Bağlantı Qapalı Pull Request-i qrafikində yeni Pull Request ilə yenidən əlaqələndirin kimi bir şeyə bənzəyəcəkdir.

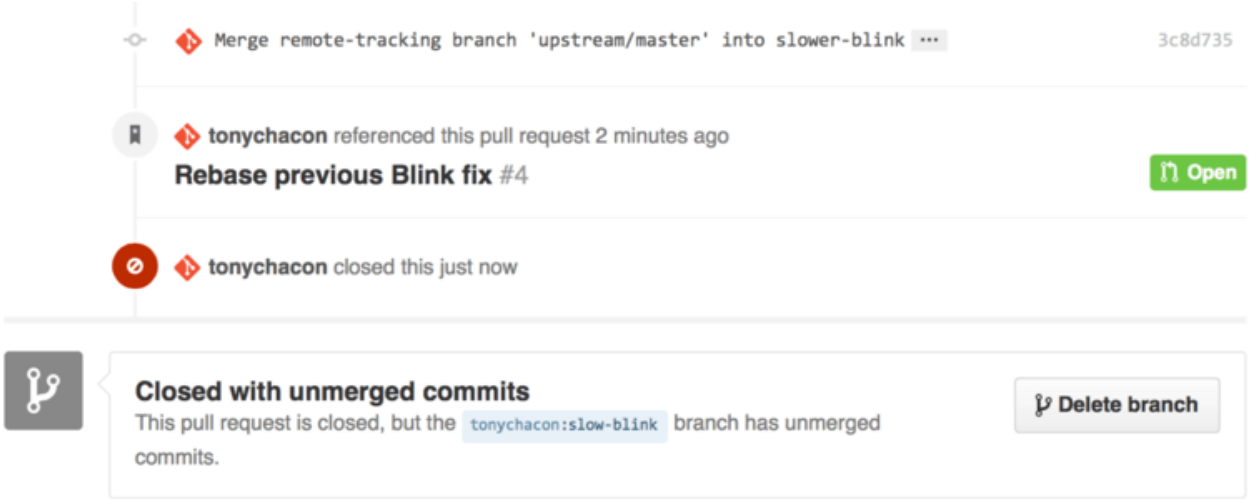


Figure 101. Qapalı Pull Request-i qrafikində yeni Pull Request ilə yenidən əlaqələndirin

Nömrələrin verilməsindən əlavə, SHA-1 tərəfindən müəyyən bir commit-ə də istinad edə bilərsiniz. Tam 40 simvol SHA-1 göstərməlisiniz, ancaq GitHub bunu bir şərhdə görsə birbaşa commit-ə bağlayacaqdır. Yenə də, məsələlər ilə əlaqəli şəkildə fork-lar və ya digər depolarda olan istinadlara istinad edə bilərsiniz.

GitHub Flavored Markdown

Digər məsələlərlə əlaqələndirmək GitHub-da demək olar ki, hər hansı bir mətn qutusu ilə edə bilə cəyiniz maraqlı işlərin başlanğıcıdır. Issue və Pull Request təsvirləri, şərhlər, kod şərhləri və daha çox da “GitHub Flavored Markdown” adlandırılandan istifadə edə bilərsiniz. Markdown düz mə tnlə yazmaq kimidir, lakin zəngin şəkildə ifadə olunur.

Markdown istifadə edərək şərhlərin və ya mətnin necə yazılacağına və sonra göstərildiyinə dair bir nümunə üçün Yazılı və göstərildiyi kimi GitHub Flavored Markdown nümunəsi baxın.

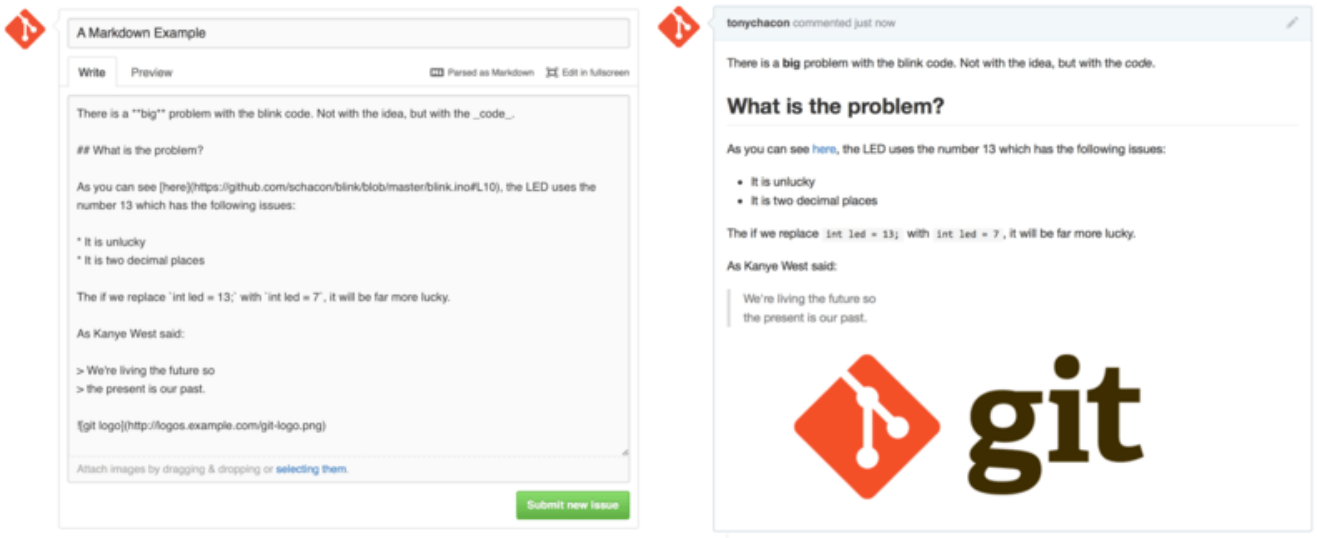


Figure 102. Yazılı və göstərildiyi kimi GitHub Flavored Markdown nümunəsi

Markdownun GitHub özəlliyi əsas Markdown sintaksisindən kənarda edə biləcəyiniz daha çox şey əlavə edir. FaydalıPull Request və ya Issue ilə əlaqədar şərhlər və ya açıqlamalar yaratdıqda bunların hamısı həqiqətən faydalı ola bilər.

Tapşırıq Siyahıları

GitHub-a məxsus Markdown xüsusiyyəti Xüsusilə Pull Requests-də istifadə üçün həqiqətən birinci faydalı Tapşırıqlar Siyahısıdır. Tapşırıq siyahısı işinizi yerinə yetirmək istədiyiniz checkbox siyahısıdır. Onları bir Issue və ya Pull Request salmaq maddənin tamamlanmadığını düşünmədən əvvəl nə etmək istədiyinizi göstərir.

Bu kimi bir tapşırıq siyahısı yarada bilərsiniz:

```
- [X] Write the code
- [ ] Write all the tests
- [ ] Document the code
```

Pull Request və ya Issue təsvirinə daxil etsək, onun Bir Markdown şərhində göstərilən tapşırıq siyahıları kimi göstərildiyini görərik.

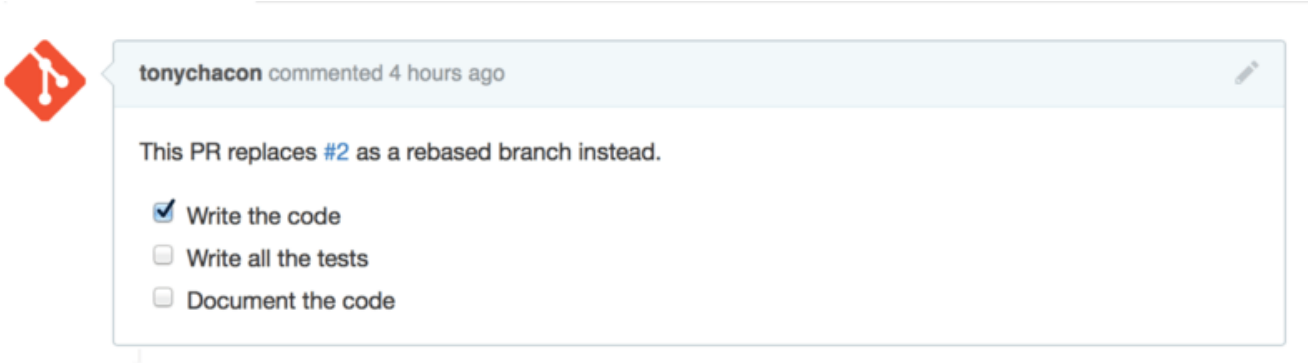


Figure 103. Bir Markdown şərhində göstərilən tapşırıq siyahıları

Bu tez-tez Pull Requests-də birləşməyə hazır olmamışdan əvvəl branch-da nə etmək istədiyinizi göstərmək üçün istifadə olunur. Əla tərəf odur ki, rəyi yeniləmək üçün checkbox-ları klik edə bil ərsiniz - tapşırıqları yoxlamaq üçün birbaşa Markdown-u düzəltməyə ehtiyac yoxdur.

Bundan əlavə, GitHub sizinIssues and Pull Requests-nizdə tapşırıq siyahılarını axtaracaq və onları siyahıya alan səhifələrdə metadata kimi göstərəcəkdir. Məsələn, tapşırıqları olan Pull Request varsa və bütün Pull Request-lərin Baxış səhifəsinə baxsanız, bunun nə qədər yerinə yetirildiyini görə bilərsiniz. Bu, insanların Pull Requests-in alt hissələrə ayrılmasına kömək edir və digər insanlara branch-ın inkişafını izləməyə kömək edir. Bunun bir nümunəsini Pull Request siyahısında tapşırıq siyahısı xülasəsi-də görə bilərsiniz.

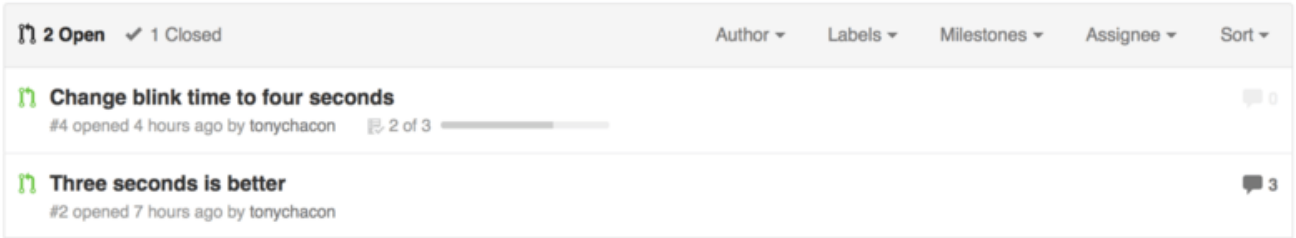


Figure 104. Pull Request siyahısında tapşırıq siyahısı xülasəsi

Erkən Pull Request-i açdığınızda və xüsusiyyəti həyata keçirməklə inkişafı izləmək üçün istifadə edərkən bunlar olduqca faydalıdır.

Kod Parçaları

Ayrıca şərhlərə kod parçaları əlavə edə bilərsiniz. Bu, branch-ınızda commit kimi yerinə yetirməd ən əvvəl etməyə çalışdığınız bir şeyi təqdim etmək üçün Pull Request-in həyata keçirə biləcəyi nümunə kodu əlavə etmək üçün istifadə olunur.

Bir parça parça əlavə etmək üçün onu arxa hissələrdə “hasar” etməlisiniz.

```
```java
for(int i=0 ; i < 5 ; i++)
{
   System.out.println("i is : " + i);
}
```
```

Orada *java* ilə etdiyimiz kimi bir dil adını əlavə etsəniz, GitHub parçanı vurğulamaq üçün sintaksis etməyə çalışacaqdır. Yuxarıda göstərilən nümunə vəziyyətində Hasarlanmış kod nümunəsi göst ərildi kimi göstərməyə son verərdi.

Figure 105. Hasarlanmış kod nümunəsi göstərildi

Sitat Gətirmək

Uzun bir şərhin kiçik bir hissəsinə cavab verirsinizsə, digər şərhdən seçilmiş şəkildə > işarəsi ilə s ətirləri öncədən çıxara bilərsiniz. Əslində, bu o qədər yaygın və faydalıdır ki, bunun üçün bir klaviatura qısa yol var. Bir şərhdə mətni birbaşa qeyd etmək istəsəniz və r düyməsini vursanız, o mətni sizin üçün şərh qutusundan sitat gətirəcəkdir.

Sitatlar bu kimi bir şeyə bənzəyir:

```
> Whether 'tis Nobler in the mind to suffer
> The Slings and Arrows of outrageous Fortune,

How big are these slings and in particular, these arrows?
```

Göstəriləndən sonra şərh Sitat gətirməyə nümunə kimi görünəcəkdir.

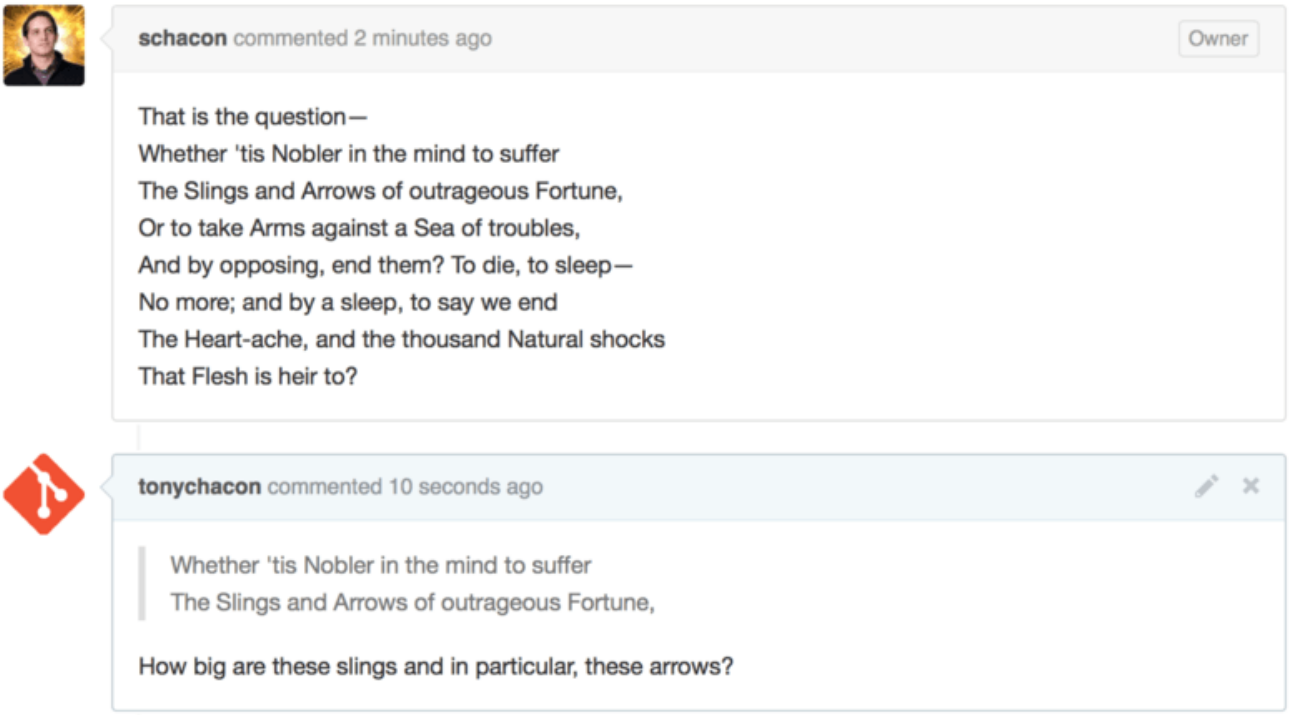


Figure 106. Sitat gətirməyə nümunə

Emoji

Nəhayət, şərhlərdə emoji istifadə edə bilərsiniz. Bu, həqiqətən, bir çox GitHub Issues and Pull Requests gördüyünüz şərhlərdə olduqca geniş istifadə olunur.

Finally, you can also use emoji in your comments. This is actually used quite extensively in comments you see on many GitHub Issues and Pull Requests. GitHub'da hətta bir emoji köməkçisi var. Bir şərh yazırsınızsa və `:` simvolu ilə başlasanız, bir avtomatik tamamlayıcı axtardığınızı tapmağa kömək edəcəkdir.

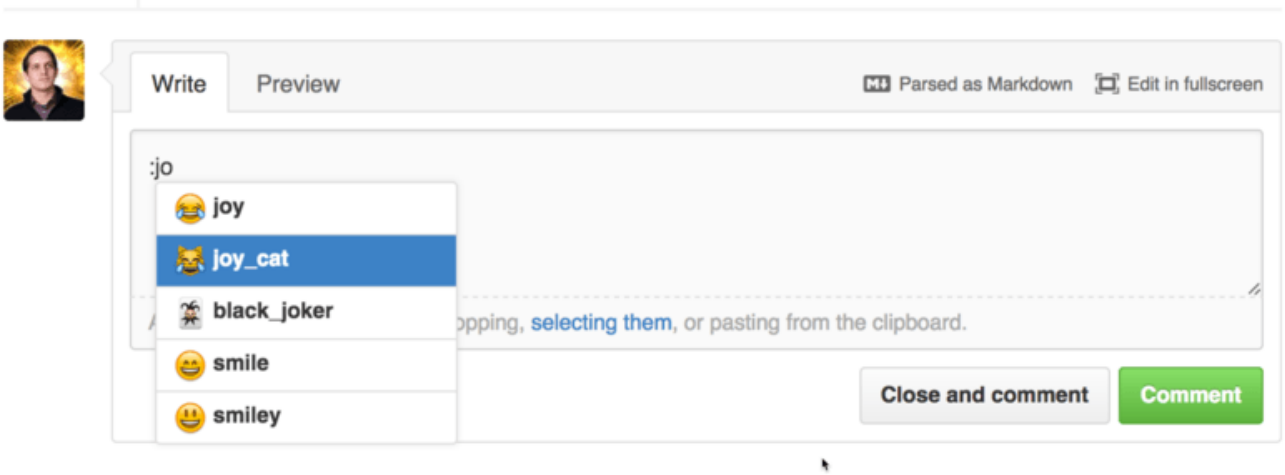


Figure 107. Əməliyyatda emoji autocompleter

Emojilər şərhin hər hansı bir yerində `:<name>:` şəklini alırlar. Məsələn, bu kimi bir şey yaza bil ərsiniz:

```
I :eyes: that :bug: and I :cold_sweat:.

:trophy: for :microscope: it.

:+1: and :sparkles: on this :ship:, it's :fire::poop:!

:clap::tada::panda_face:
```

Təqdim edildikdə, Heavy emoji commenting kimi bir şey görünür.

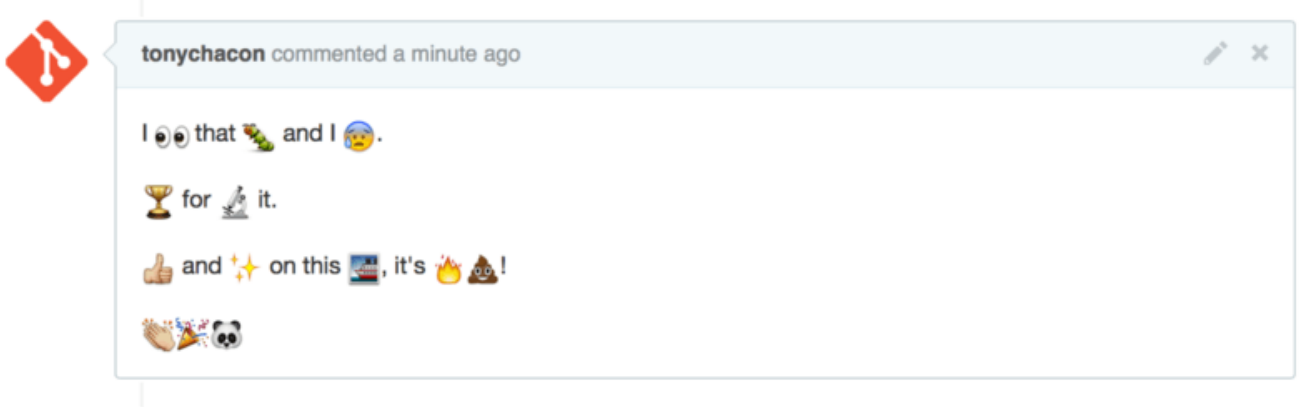


Figure 108. Heavy emoji commenting

Bu inanılmaz dərəcədə faydalı olduğundan deyil, amma duyğuları çatdırmaq çətin olan bir mühitə əyləncə və duyğu elementi əlavə edir.

Bu günlərdə emoji simvollarından istifadə edən bir çox veb xidmətləri var. Demək istədiyinizi ifadə edən emoji tapmaq üçün istinad üçün səhifəni burdan tapa bil ərsiniz:

https://www.webfx.com/tools/emoji-cheat-sheet/

Şəkillər

Bu texniki olaraq GitHub Flavored Markdown deyil, amma olduqca faydalıdır. URL-lərini tapmaq və yerləşdirmək çətin ola biləcək şərhlərə Markdown şəkil bağlantılarını əlavə etməklə GitHub ş əkilləri onları daxil etmək üçün mətn sahələrinə drag və drop etməyə imkan verir.

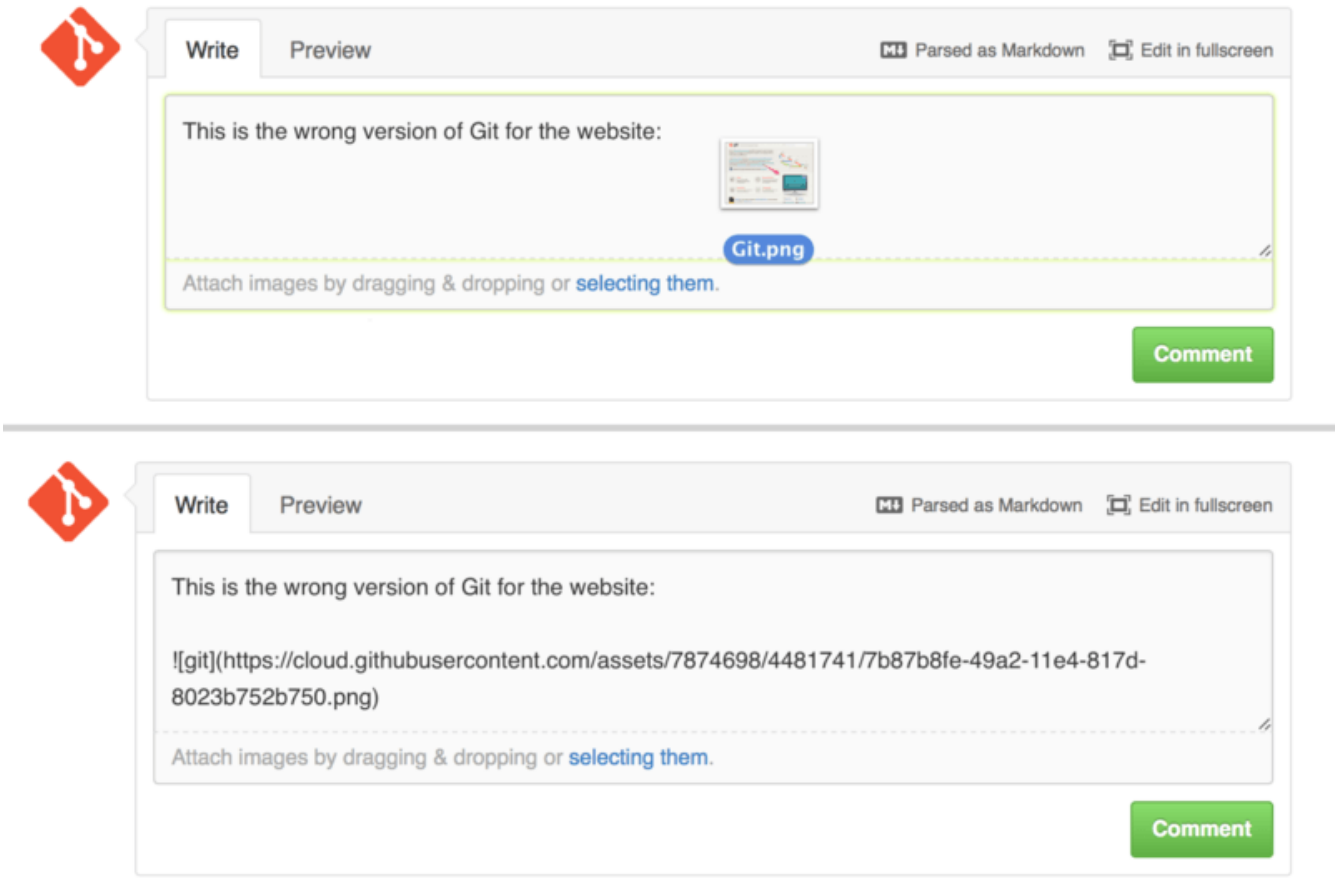


Figure 109. Şəkilləri yükləmək və avtomatik yerləşdirmək üçün drag və drop edin

Şəkilləri yükləmək və avtomatik yerləşdirmək üçün drag və drop edin -ə baxsanız, mətn sahəsinin üstündəki kiçik bir “Parsed as Markdown” işarəsini görə bilərsiniz. Bunun üzərinə basaraq GitHub-da Markdown ilə edə biləcəyiniz hər şeyin dolğun bir vərəqini verəcəksiniz.

GitHub Public Depolarınızı Yeniləyin

Bir GitHub deposunu fork etdikdən sonra, depo (sizin "fork") orijinaldan asılı olmayaraq mövcuddur. Xüsusilə, orijinal deponuzda yeni commitlər olduqda, GitHub sizə aşağıdakı kimi bir mesajla məlumat verir:

```
This branch is 5 commits behind progit:master.
```

Lakin GitHub deponuz GitHub tərəfindən avtomatik olaraq yenilənməyəcəkdir; bu özünüzün etməli olduğunuz bir şeydir. Xoşbəxtlikdən, bunu etmək çox asandır.

Bunu etmək üçün bir konfiqurasiya tələb olunmur. Məsələn, `https://github.com/progit/progit2.git`-dən ayrılmısınızsa, bu `master` branch-nızı bu cür aktuallaşdıra bilərsiniz:

```
$ git checkout master ①
$ git pull https://github.com/progit/progit2.git ②
$ git push origin master ③
```

① Əgər başqa branch-dasınızsa, `master`-ə qayıdın.

② `https://github.com/progit/progit2.git`-dən dəyişiklikləri fetch edin və onları `master`-ə birləşdirin.

③ `master` branch-nızı `origin`-ə push edin.

Bu işləyir, ancaq URL almaq üçün hər dəfə yazmaq məcburiyyətində qalır. Bu işi bir az konfiqurasiya ilə avtomatlaşdıra bilərsiniz:

```
$ git remote add progit https://github.com/progit/progit2.git ①
$ git branch --set-upstream-to=progit/master master ②
$ git config --local remote.pushDefault origin ③
```

① Mənbə deposunu əlavə edin və ona bir ad verin. Budur, onu `progit` adlandırmağı seçdim..

② `progit` remote-undan fetch etmək üçün `master` branch-nızı qurun.

③ Defolt deponu `origin` olaraq təyin edin.

Bunu etdikdən sonra iş axını daha asan olur:

```
$ git checkout master ①
$ git pull ②
$ git push ③
```

① Əgər başqa branch-dasınızsa, `master`-ə qayıdın.

② `progit`-dən dəyişiklikləri fetch edin və dəyişiklikləri `master`-ə birləşdirin.

③ `master` branch-nızı `origin`-ə push edin.

Bu yanaşma faydalı ola bilər, amma heç bir yararsız yanı yox deyil. Git səmimi şəkildə bu işi sizin üçün edəcək, ancaq `master`-ə commit etsəniz,`progit`-dən pull edin, sonra `origin`- push etsəniz sizə x əbərdarlıq etməz - bütün bu əməliyyatlar bu quraşdırma ilə etibarlıdır.

Bir Layihənin Saxlanılması

İndi bir layihəyə töhfə verməkdə daha rahat olduğumuz üçün digər tərəfə baxaq: öz layihənizi yaratmaq, saxlamaq və idarə etmək.

Yeni bir Depo Yaratmaq

Layihə kodumuzu bölüşmək üçün yeni bir depo yaradaq. Vasitə panelinin sağ tərəfindəki “New repository” düyməsinə və ya “Yeni Depo” açılışı-da göründüyü kimi istifadəçi adınızın yanındakı yuxarı alətlər panelindəki `+` düyməsindən basaraq başlaya bilərsiniz.

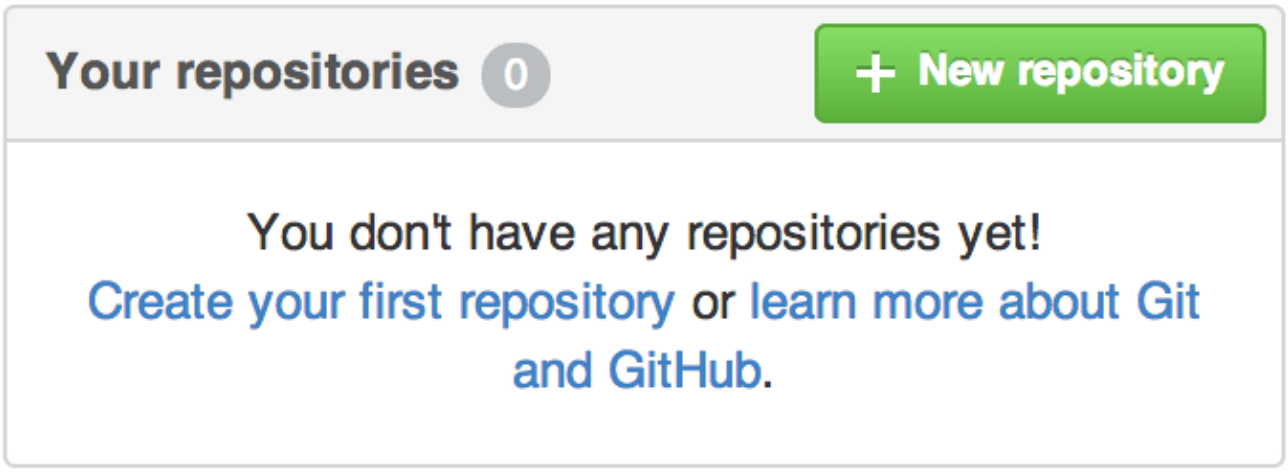


Figure 110. “Depolarınız” sahəsi

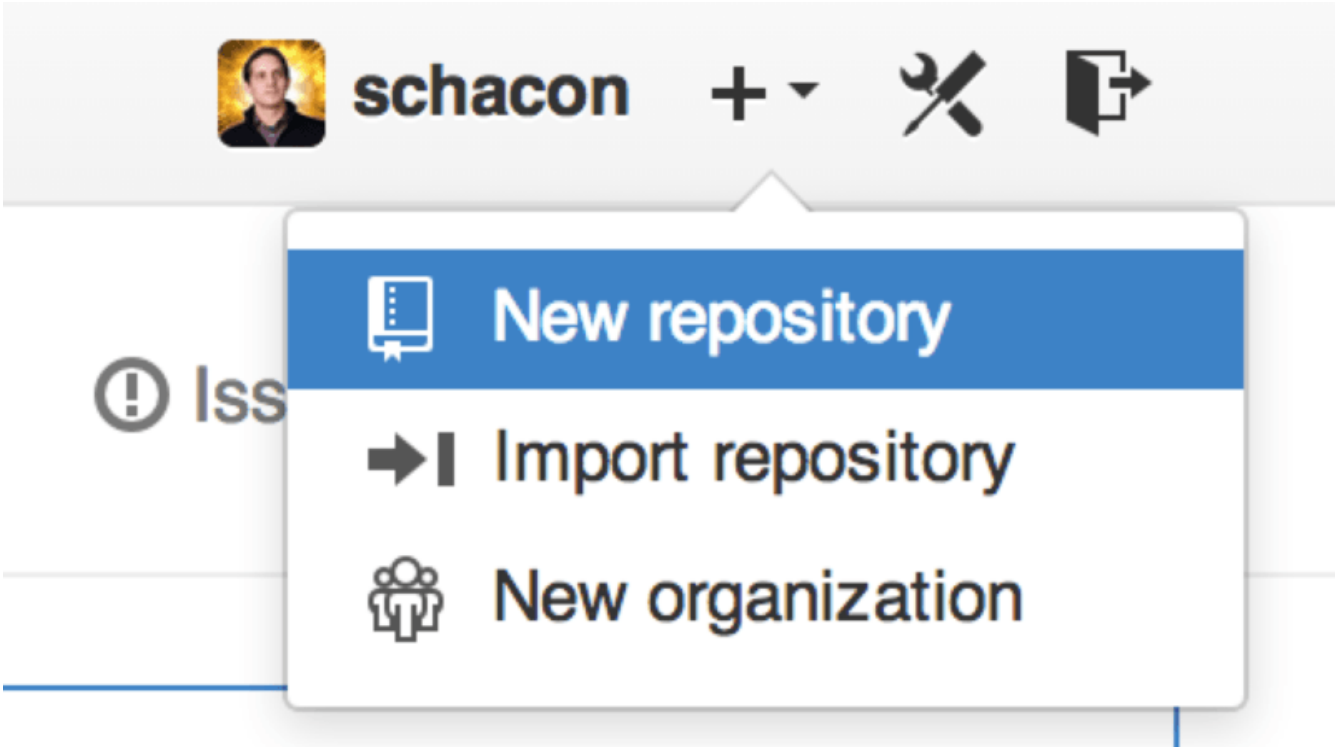


Figure 111. “Yeni Depo” açılışı

Bu sizi “yeni depo” formasına aparır:

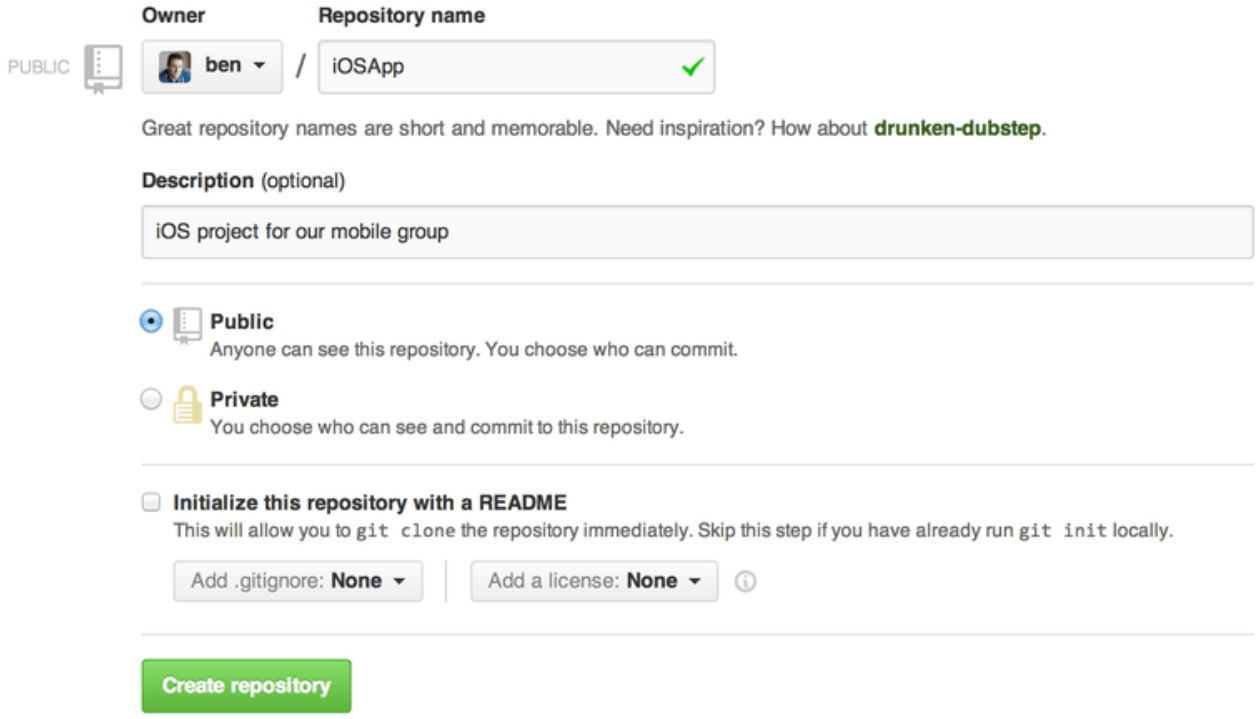


Figure 112. "yeni depo" forması

Burada həqiqətən etməli olduğunuz bir şey bir layihə adını təqdim etməkdir; qalan sahələr tamamilə istəyə bağlıdır. Hələlik, sadəcə "Create Repository" düyməsini vurun və bum - GitHub-da `<user>/<project_name>` adlı yeni bir depo var.

Hələ orada kodunuz olmadığından, GitHub sizə yeni Git deposunuyaratmaq və ya mövcud Git layihəsini necə bağlamaq barədə təlimatları göstərəcəkdir. Biz burada işləməyəcəyik; yeniləməyə ehtiyacınız varsa, Git'in Əsasları-ə baxın.

İndi layihəniz GitHub-a ev sahibliyi etdiyindən, layihənizi bölüşmək istədiyiniz şəxsə URL-i verə bilərsiniz. GitHub'dakı hər bir layihəyə HTTPS üzərindən `https://github.com/<user>/<project_name>` və SSH üzərindən `git@github.com:<user>/<project_name>` kimi müraciət etmək mümkündür. Git bu URL-lərin hər ikisindən də fetch və push edə bilər, lakin onlara qoşulan istifadəçinin etimadnaməsinə əsaslanaraq girişlə idarə olunur.

HTTPS əsaslı bir URL-i public bir layihə üçün paylaşmağa çox vaxt üstünlük verilir, çünki klonlama üçün istifadəçinin GitHub hesabına sahib olmaq m əcburiyyətində deyil. İstifadəçilərə SSH URL-i versəniz, layihənizə daxil olmaq üçün bir hesab və yüklənmiş SSH key-i olmalıdır. HTTPS də layihəni orada görmək üçün brauzerə yapışdıracaqları URL-lə eynidir.

Əməkdaşlar Əlavə Etmək

Giriş icazəsi vermək istədiyiniz digər insanlarla işləyirsinizsə, onları "əməkdaş" olaraq əlavə etm əlisiniz. Ben, Jeff və Louise hamısı GitHub-dakı hesablara daxil olsalar və depolarınıza push etmək imkanı vermək istəsəniz, onları layihənizə əlavə edə bilərsiniz. Bunu etmək onlara "push" girişi verəcəkdir ki, bu da həm layihə, həm də Git depolarına oxumaq və yazmaq imkanı əldə edir.

Sağ tərəfdən çubuğun altındakı `*Parametrlər* 'bağlantısını klik edin.

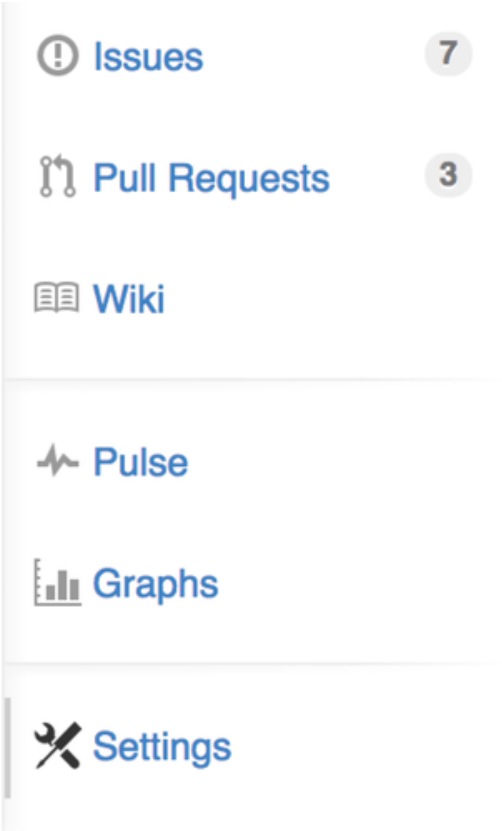


Figure 113. Depo parametrləri linki

Sonra sol tərəfdəki menyudan “Collaborators” seçin. Sonra, qutuya sadəcə bir istifadəçi adı yazın və “Add collaborator.” düyməsini basın. İstədiyiniz hər kəsə giriş vermək istədiyiniz qədər bunu t əkrar edə bilərsiniz. Girişi ləğv etmək lazımdırsa, sıralarının sağ tərəfindəki “X” düyməsini basın.

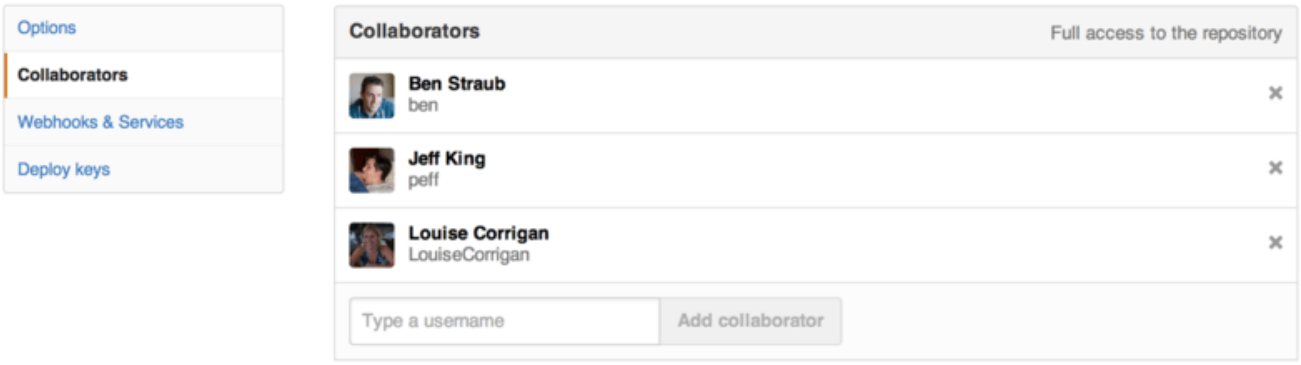


Figure 114. Depo əməkdaşları

Pull Requests İdarə etmək

İndi içərisində bir kodu olan bir layihəniz və bəlkə də push etmə imkanı olan bir neçə əməkdaşınız var, özünüzü Pull Request-i aldığınız zaman nə edəcəyinizə baxaq.

Pull Request-ləri ya depolarınızın bir fork-undakı bir branch-dan və ya eyni depodakı başqa bir branch-dan gələ bilər. Yeganə fərq, fork içərisində olanlar çox vaxt sizin branch-a basa bilm ədiyiniz və sizin tərəfinizə push edə bilməyəcəkləri şəxslər olmalıdır, halbuki daxili Pull Request-l əri ilə ümumiyyətlə hər iki tərəf branch-a daxil ola bilər.

Bu misallar üçün, “tonychacon” olduğunuzu və “fade” adlı yeni Arduino kod layihəsini yaratdığınız

barədə düşünək.

E-poçt Bildirişləri

Kimsə gəlir və kodunuza dəyişiklik edir və Pull Request göndərir. Yeni Pull Request barədə sizi xə bərdar edən bir e-poçt almalısınız və bu Yeni Pull Request barədə e-poçt bildirişi kimi bir şey görünməlidir.

Figure 115. Yeni Pull Request barədə e-poçt bildirişi

Bu e-poçtla əlaqəli bir neçə şey var. Kiçik bir diffstat verəcəkdir - Pull Request-da nə qədər d əyişdiyiniz sənədlərin siyahısı. GitHub üzərindəki Pull Request-ə bir link verir. Ayrıca, əmr sətrind ən istifadə edə biləcəyiniz bir neçə URL verir.

`git pull <url> patch-1` deyilən xətti görsəniz, bu, uzaq bir branch-a qoşulmanın sadə bir yoldur. Biz bu işi tez bir zamanda Uzaq Branch'ları Yoxlamaq-də keçdik. İstəyirsinizsə, bir mövzu branch-ı yarada və həmin branch-a keçə bilərsiniz və sonra Pull Request dəyişikliklərində birləşmək üçün bu əmri işlədə bilərsiniz.

Digər maraqlı URL-lər `.diff` və `.patch` URL-ləridir, ehtimal ki, Pull Request-nin vahid diff və patch versiyalarını təmin edir. Texniki olaraq Pull Request işini bu kimi bir şeylə birləşdirə bilərsiniz:

```
$ curl https://github.com/tonychacon/fade/pull/1.patch | git am
```

Pull Request üzrə Əməkdaşlıq

GitHub Axını-da bəhs etdiyimiz kimi Pull Request-ni açan şəxslə söhbət edə bilərsiniz. Hər yerdə GitHub Flavored Markdown istifadə edərək kodun müəyyən sətirlərini şərh edə bilərsiniz, bütün commit-ləri şərh edə bilərsiniz və ya Pull Request-in özünə şərh verə bilərsiniz.

Pull Request-nə başqaları hər dəfə rəy verdikdə fəaliyyətin baş verdiyini bildiyiniz üçün e-poçt bildirişləri almağa davam edəcəksiniz. Onların hər birində fəaliyyətin baş verdiyi yerdən Pull Request-ə bir keçid olacaq və siz də Pull Request mövzusuna şərh vermək üçün e-poçta birbaşa cavab verə bilərsiniz.

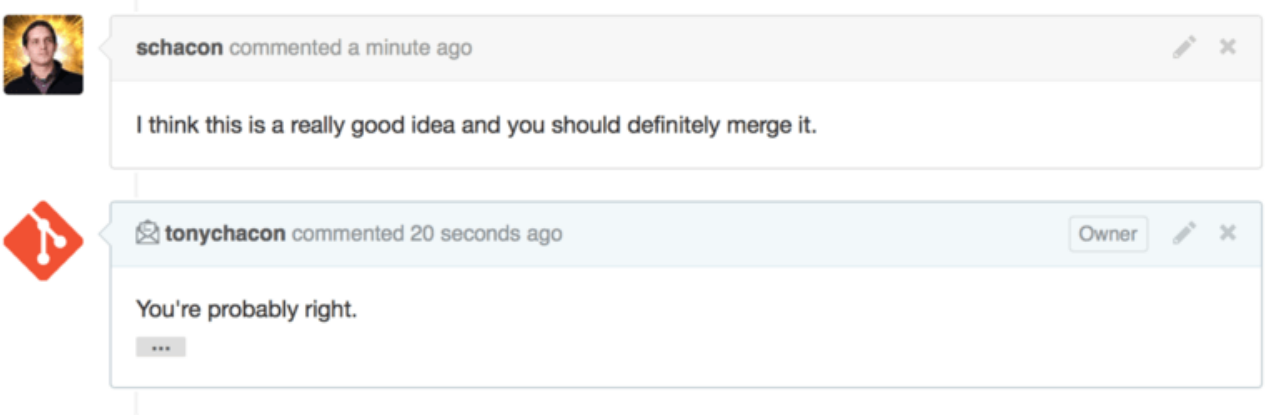


Figure 116. E-poçtlara cavablar mövzuya daxil edilmişdir

Kod istədiyiniz yerdə olanda və onu birləşdirmək istədikdə daha əvvəl gördüyünüz `git pull <url> <branch>` əmri ilə və ya kodu əlavə edərək local olaraq və ya fork ilə uzaqdan fetch edə, birləşdirə bilərsiniz.

Birləşmə mənasızdırsa, GitHub saytındakı “Merge” düyməsini vura bilərsiniz. Bu sürətli bir irəli daxil olma ehtimalı olsa belə, birləşmə əməliyyatı yaradaraq, “sürətli olmayan irəli” birləşməsini həyata keçirəcəkdir. Bu o deməkdir ki, nə olursa olsun, birləşmə düyməsini vurduğunda, birləşmə əməliyyatı yaradılır. Birləşdirmək düyməsini və Pull Request-in manual birləşməsi üçün təlimatları da gördüyünüz kimi, işarə bağlantısını vurursanız, GitHub bu məlumatların hamısını verir.

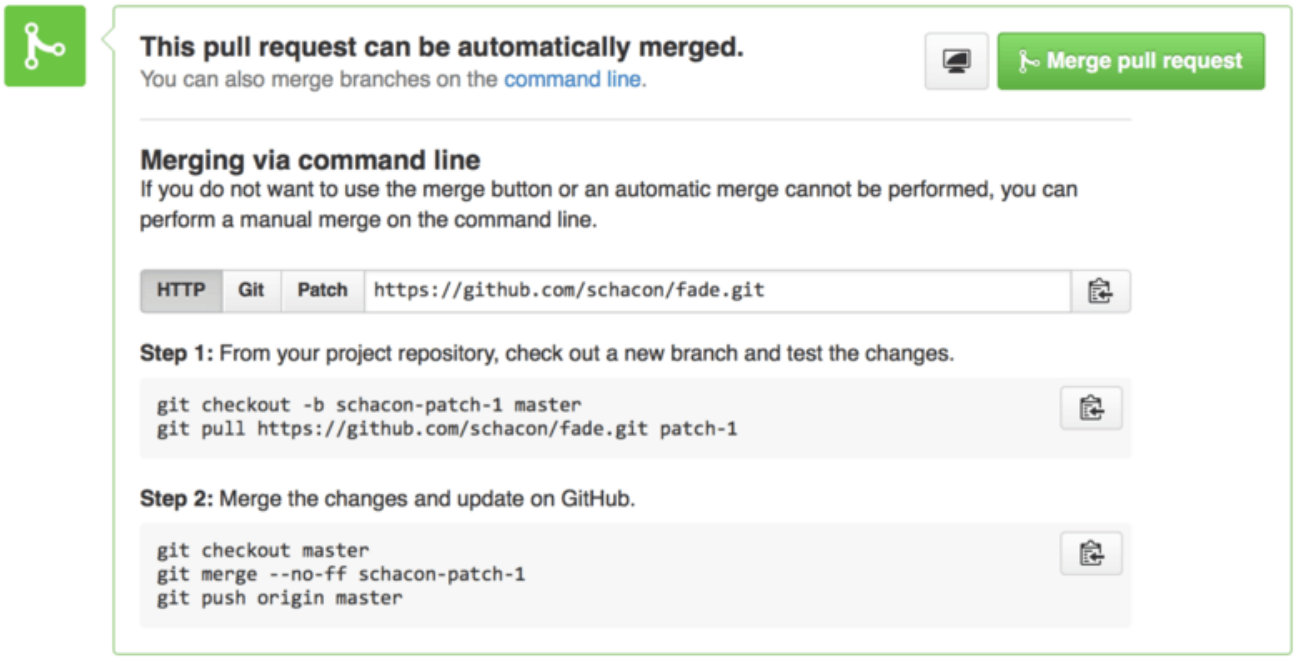


Figure 117. Birləşdirmək düyməsini və Pull Request-in manual birləşməsi üçün təlimatları

Birləşdirmək istəmədiyinizə qərar verdiyiniz təqdirdə yalnız Pull Request-i bağlaya bilərsiniz və bu onu açan şəxsə bildiriləcəkdir.

Pull Request Referləri

Pull Request-lərin bir çoxu ilə məşğul olursunuzsa və bir qrup uzaqdan əlavə etmək və ya hər səf ərində bir dəfə pull etmək istəmirsinizsə, GitHub-ın etməynizə sizə imkan verəcək səliqəli bir hiylə var. Bu bir az inkişaf etmiş bir hiylədir və bunun təfərrüatlarını bir az daha Refspec-də araşdıracağıq, bu olduqca faydalı ola bilər.

GitHub, əslində serverdəki pseudo-branch-lar kimi bir depo üçün Pull Request branch-larını reklam edir. Varsayılan olaraq, onları klonladığınız zaman almırsınız, ancaq gizli bir şəkildə var və onlara olduqca asanlıqla daxil ola bilərsiniz.

Bunu nümayiş etdirmək üçün `ls-remote` adlı aşağı səviyyəli bir əmrdən (daha çoxPlumbing və Porcelain-də oxuyacağımız bir “plumbing” əmri olaraq xatırlanır.) istifadə edəcəyik. Bu əmr ümumiyyətlə gündəlik Git əməliyyatlarında istifadə edilmir, lakin serverdə hansı istinadların mövcud olduğunu bizə göstərmək faydalıdır.

Bu əmri əvvəllər istifadə etdiyimiz “yanıb-sönmə” deposuna qarşı işləsək, depodakı bütün branch-ların və etiketlərin və digər istinadların siyahısını alacağıq.

```
$ git ls-remote https://github.com/schacon/blink
10d539600d86723087810ec636870a504f4fee4d    HEAD
10d539600d86723087810ec636870a504f4fee4d    refs/heads/master
6a83107c62950be9453aac297bb0193fd743cd6e    refs/pull/1/head
afe83c2d1a70674c9505cc1d8b7d380d5e076ed3    refs/pull/1/merge
3c8d735ee16296c242be7a9742ebfbc2665adec1    refs/pull/2/head
15c9f4f80973a2758462ab2066b6ad9fe8dcf03d    refs/pull/2/merge
a5a7751a33b7e86c5e9bb07b26001bb17d775d1a    refs/pull/4/head
31a45fc257e8433c8d8804e3e848cf61c9d3166c    refs/pull/4/merge
```

Əlbəttə ki, depo içərisindəsinizsə və `git ls-remote origin` və ya yoxlamaq istədiyiniz hər hansı bir uzaqdan çalışırsınızsa, bu sizə bənzər bir şey göstərəcəkdir. Depo GitHub-dadırsa və açılan hər hansı bir Pull Request-ləriniz varsa, `refs/pull/` ilə prefiks edilmiş bu istinadları alacaqsınız. Bunlar əsasən branch-lardır, amma `refs/heads/` altında olmadığından onları klonlaşdırdıqda və ya serverdən götürəndə normal qəbul etmirsiniz - fetching prosesi onları normal qəbul etmir.

Pull Request-ə iki müraciət var - `/head` nöqtəsində sonu Pull Request branch-dakı sonuncu commit-lə eyni.

Beləliklə, kimsə depoda Pull Request açsa və onların branch-ı `bug-fix` adlanırsa və `a5a775` əm əliyyatını göstərirsə, onda **bizim** depoda `bug-fix` branch-ı olmayacaq (çünki bu onların fork-larıdı), amma biz `a5a775` işarəsini verən `pull/<pr#>/head` olacaqdır. Bu o deməkdir ki, uzaqdan bir qrup əlavə etmədən hər Pull Request branch-ını bir dəfəyə asanlıqla pull down edə bilərik.

İndi referansı birbaşa fetching edə bilərsiniz.

```
$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
 * branch            refs/pull/958/head -> FETCH_HEAD
```

Bu, Git-ə ‘` `origin` remote-a qoşulun və `refs/pull/958/head` adlı ref-i yükləyin.” Git xoşbəxt şəkildə itaət edir və bu ref düzəltmək üçün lazım olan hər şeyi yükləyir və `.git/FETCH_HEAD` altından ist ədiyiniz commit-ə göstərici qoyur. Bunu test etmək istədiyiniz bir branch-a `git merge FETCH_HEAD` ilə təqib edə bilərsiniz, lakin bu birləşmə mesajı bir az qəribə görünür. Ayrıca, **çox** pull request-ləri n əzərdən keçirirsinizsə, bu yorucu olur.

Pull request-lərin *hamısını* fetch etməyin və uzaqdan əlaqə qurduğunuz zaman onları yeniləməyin bir yolu da var. Sevdiyiniz redaktorda`.git/config` açın və uzaqdan `origin`-i axtarın. Bu belə görünməlidir:

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2
    fetch = +refs/heads/*:refs/remotes/origin/*
```

`fetch =` ilə başlayan xətt “refspec.”-dir. Bu uzaqdakə adları local `.git` qovluğunuzdakı adlarla ç əkilməsinin bir yoludur. Bu, xüsusilə Git-ə "uzaqdan idarə olunanlar `refs/heads` altındakı şeylər local depolarımda `refs/remotes/origin` altında getməlidir." deyir. Bu hissəni başqa bir refspec əlavə

etmək üçün dəyişdirə bilərsiniz:

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2.git
    fetch = +refs/heads/*:refs/remotes/origin/*
    fetch = +refs/pull/*/head:refs/remotes/origin/pr/*
```

Bu son sətir Git'ə '` refs/pull/123/head kimi görünən bütün istinadlar,refs/remotes/origin/pr/123 kimi local olaraq saxlanılmalıdır." İndi həmin faylı yadda saxlasanız, `git fetch` edin:

```
$ git fetch
# …
 * [new ref]         refs/pull/1/head -> origin/pr/1
 * [new ref]         refs/pull/2/head -> origin/pr/2
 * [new ref]         refs/pull/4/head -> origin/pr/4
# …
```

İndi uzaqdan pull request-lərin hamısı local olaraq, branch-ları izləyən branch-lar kimi təmsil olunur; yalnız oxuyursunuz və fetch etdiyiniz zaman yeniləyirlər. Bu, local bir pull request-də kodu sınamağı asanlaşdırır:

```
$ git checkout pr/2
Checking out files: 100% (3769/3769), done.
Branch pr/2 set up to track remote branch pr/2 from origin.
Switched to a new branch 'pr/2'
```

Aranızdakı qartal gözlü, refspecin uzaq hissəsinin sonundakı `head` hissəsini qeyd edəcək. GitHub tə rəfində `refs/pull/#/merge` ref də var, saytdakı "merge" düyməsini basdığınız zaman nəticəni təmin edəcəkdir. Bu, düyməni vurmadan əvvəl birləşməni sınamağa imkan verə bilər.

Pull Request-lər üzərindəki Pull Request-lər

Yalnız əsas və ya `master` branch-nı hədəf alan Pull Request-ləri deyil, həm də şəbəkədəki istənilən branch-ı hədəf alan Pull Request aça bilərsiniz. Əslində, başqa bir Pull Request-i də hədəfə ala bil ərsiniz.

Əgər düzgün istiqamətdə irəlilədiyini görsəniz və ona bağlı bir dəyişiklik üçün bir fikriniz varsa və ya yaxşı bir fikir olduğuna əmin deyilsinizsə və ya sadəcə hədəf branch-ına push etmək imkanı yoxdursa, birbaşa Pull Request aça bilərsiniz.

Pull Request açdığınız zaman, səhifənin yuxarısında hansı branch-ı pull etmək istədiyinizi və haradan pull etmək istəyinizi göstərən bir qutu var. Həmin qutunun sağındakı "Edit" düyməsini vurursanız, nəinki branch-ları, həm də fork-ları da dəyişdirə bilərsiniz.

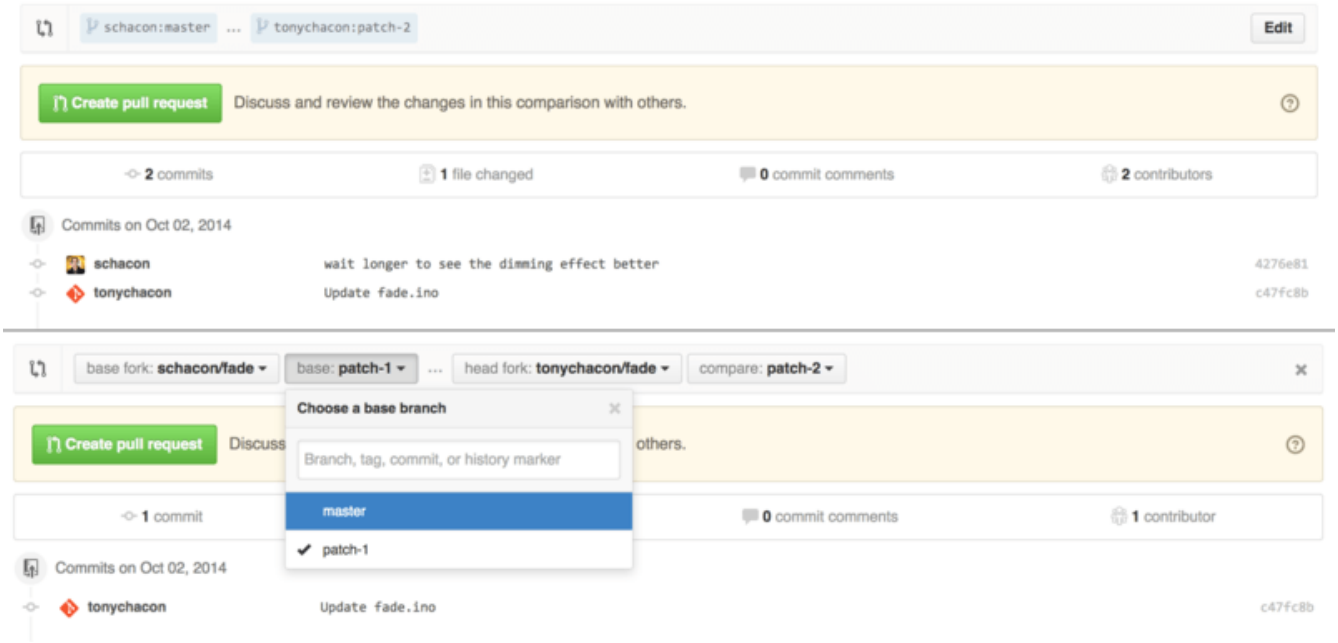


Figure 118. Pull Request hədəf fork və branch-ını manual olaraq dəyişdirin

Burada yeni branch-ınızı başqa bir Pull Request-ə və ya layihənin başqa bir fork-una birləşdirmək üçün asanlıqla göstərə bilərsiniz.

Mention-lar və Bildirişlər

GitHub ayrıca suallarınız olduqda və ya müəyyən bir şəxs və ya komandanın rəyinə ehtiyac duyduğunuzda yararlana biləcək olduqca gözəl bir bildiriş sisteminə malikdir.

Hər hansı bir şərhdə bir "@" simvolu yazmağa başlaya bilərsiniz və layihə ilə əməkdaşlıq edən və ya töhfə verən insanların adları və istifadəçi adları ilə avtomatik tamamlanmağa başlayacaq.

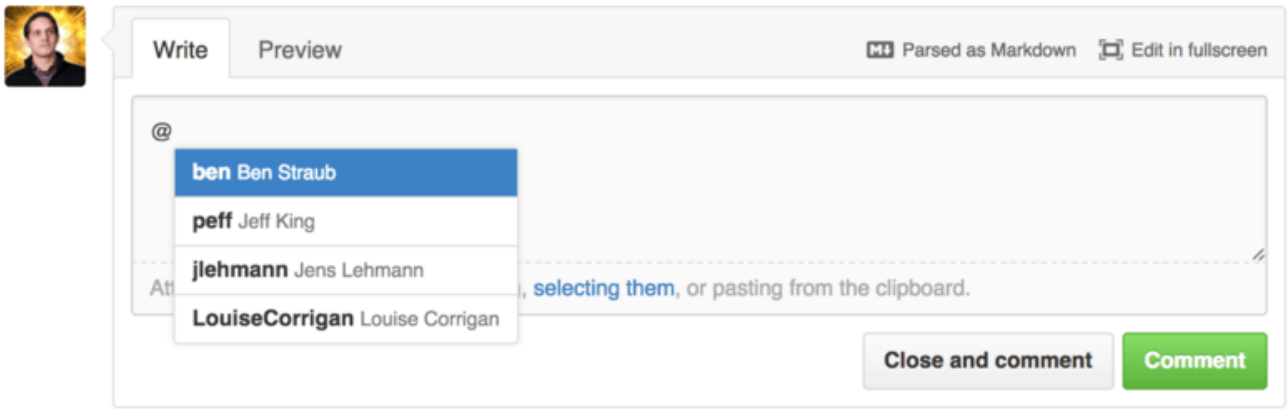


Figure 119. Birini xatırlatmaq üçün @ yazmağa başlayın

Ayrıca, bu açılan siyahıda olmayan bir istifadəçini də qeyd edə bilərsiniz, lakin əksər hallarda avtoköçürmə bunu daha sürətli edə bilər.

Bir istifadəçi qeydi ilə bir şərh göndərdikdən sonra bu istifadəçiyə bildiriş veriləcəkdir. Bu o dem əkdir ki, bu, insanları sorğu-sual etməkdənsə söhbətə cəlb etməyin həqiqətən effektiv yolu ola bil ər. Çox vaxt GitHub'un Pull Requests-də insanlar bir qrupu və ya şirkətindəki bir insanı bir Issue və ya Pull Request-i nəzərdən keçirmək üçün pull edəcəklər.

Kimsə Pull Request və ya Issue barədə məlumat verərsə, ona “subscribed” olacaq və hər hansı bir fəaliyyət baş verdikdə bildiriş almağa davam edəcəkdir. Bir şeyi açmısınızsa, depoya baxırsınızsa və ya bir şeyə şərh verərsinizsə, ona abunə olacaqsınız. Artıq bildiriş almaq istəmirsinizsə, səhifə də yeniləmələri almağı dayandırmaq üçün tıklaya biləcəyiniz “Unsubscribe” düyməsi var.

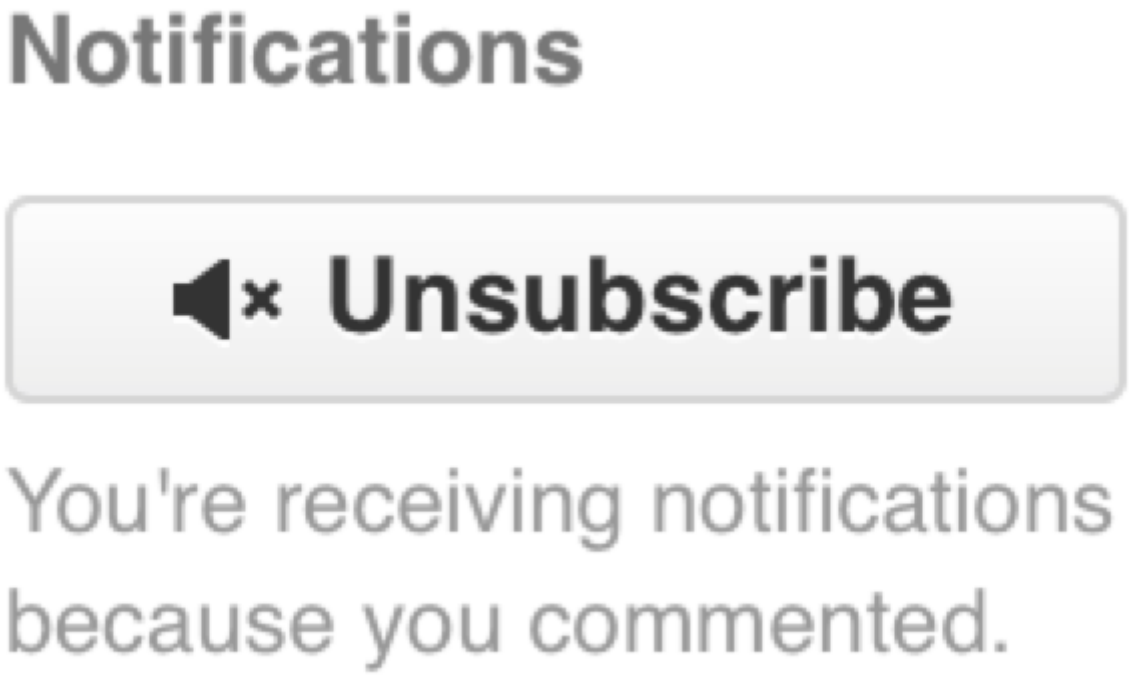


Figure 120. Issue və ya Pull Request-də Unsubscribe olmaq

Bildrişlər Səhifəsi

GitHub ilə əlaqədar burada “bildirişlər” dedikdə GitHub-da hadisələr baş verdikdə sizinlə əlaqə qurmağa çalışdıqlarını və onları konfiqurasiya edə biləcəyiniz bir neçə fərqli yol olduğunu söyl əyirik. Parametrlər səhifəsindəki “Notification center” bölməsinə keçsəniz, əlinizdə olan bəzi variantları görə bilərsiniz.

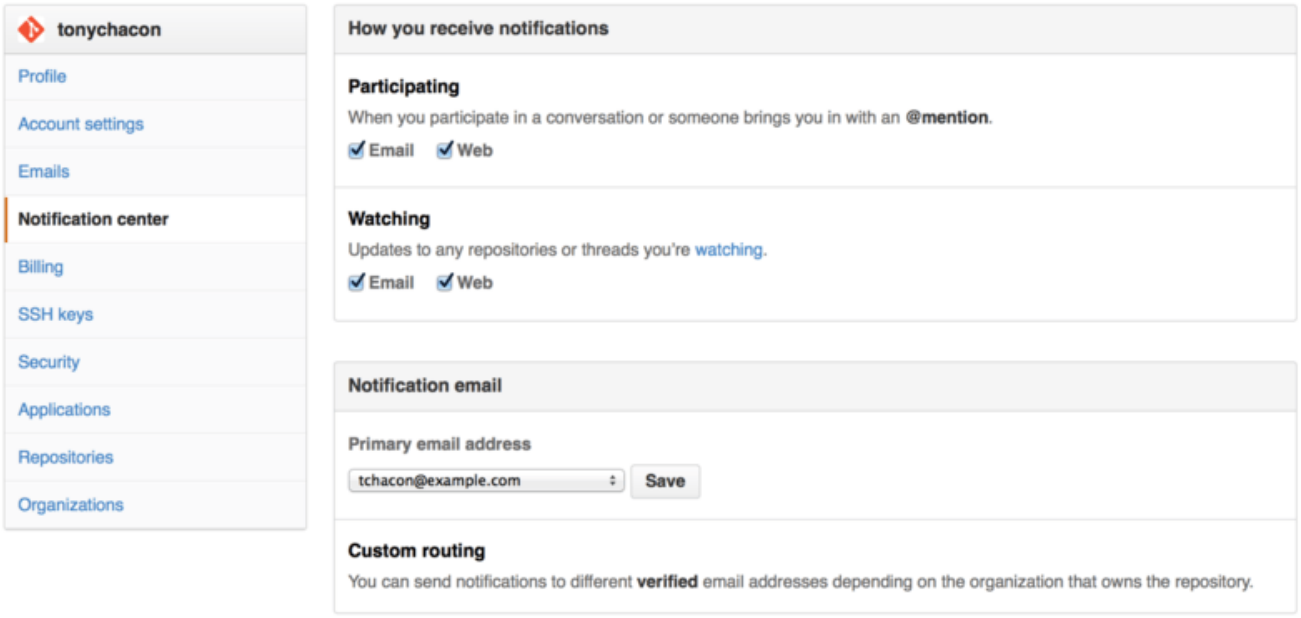


Figure 121. Bildiriş mərkəzi seçimləri

İki seçim, “E-poçt” və “Web” üzərindən bildirişlər almaqdır və işlərdə fəal iştirak etdiyiniz zaman və seyr etdiyiniz depolardakı fəaliyyət üçün ya birini ya da ikisini də seçə bilərsiniz.

Veb bildirişləri

Veb bildirişləri yalnız GitHub-da mövcuddur və onları yalnız GitHub-da yoxlaya bilərsiniz. Tərcihl ərinizdə bu seçim seçilibsə və sizin üçün bildiriş tətiklənirsə, Bildiriş mərkəzi da göründüyü kimi ekranınızın yuxarı hissəsindəki bildirişlər nişanınızın üzərində kiçik bir mavi nöqtə görəcəksiniz.

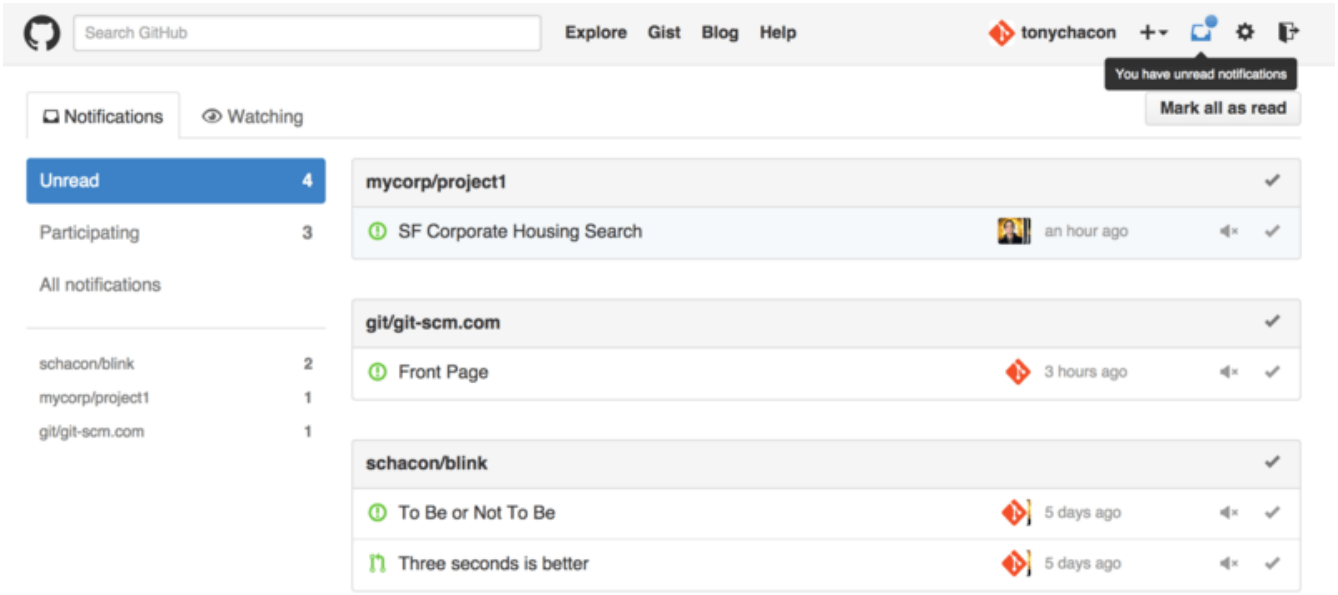


Figure 122. Bildiriş mərkəzi

Bunun üzərinə klikləsəniz, layihə üzrə qruplaşdırılmış, sizə bildirilmiş bütün maddələrin siyahısını görəcəksiniz. Sol tərəfdəki çubuğundakı adını tıklayaraq müəyyən bir layihənin bildirişlərinə süzgəcdən keçirə bilərsiniz.

Hər hansı bir bildirişin yanındakı işaret nişanını vuraraq bildirişi qəbul edə bilərsiniz və ya qrupun üst hissəsindəki işarət düyməsini vuraraq bildirişlərin *hamısını*-ı təsdiq edə bilərsiniz. Bu element haqqında əlavə bildiriş almamaq üçün vura biləcəyiniz hər bir işaretin yanında səssiz bir düymə də var.

Bu vasitələrin hamısı çox sayda bildirişlə işləmək üçün çox faydalıdır. GitHub güc istifadəçilərinin çoxu e-poçt bildirişlərini tamamilə söndürəcək və bütün bildirişlərini bu ekran vasitəsilə idarə ed əcəkdir.

E-poçt Bildirişləri

E-poçt bildirişləri GitHub vasitəsilə bildirişləri idarə edə biləcəyiniz başqa bir yoldur. Əgər bu varsa, hər bildiriş üçün e-poçt alacaqsınız. Bunun nümunələrini E-poçt bildirişləri olaraq göndəril ən şərhlər və Yeni Pull Request barədə e-poçt bildirişi-də gördük. E-poçtlar da düzgün bir şəkildə yığılar, bu da bir threading e-poçt müştərisi istifadə edirsinizsə yaxşı olur.

Xüsusi filtrlər və qaydaları qurmaq üçün həqiqətən faydalı ola biləcək GitHub'un sizə göndərdiyi e-poçtların başlıqlarına kifayət qədər miqdarda metadataya malikdir.

Məsələn, Tony-ə göndərilən faktiki e-poçt başlıqlarına Yeni Pull Request barədə e-poçt bildirişi-da göstərilən e-poçtda baxsaq, göndərilən məlumatlar arasında aşağıdakıları görəcəyik:

```
To: tonychacon/fade <fade@noreply.github.com>
Message-ID: <tonychacon/fade/pull/1@github.com>
Subject: [fade] Wait longer to see the dimming effect better (#1)
X-GitHub-Recipient: tonychacon
List-ID: tonychacon/fade <fade.tonychacon.github.com>
List-Archive: https://github.com/tonychacon/fade
List-Post: <mailto:reply+i-4XXX@reply.github.com>
List-Unsubscribe: <mailto:unsub+i-XXX@reply.github.com>,...
X-GitHub-Recipient-Address: tchacon@example.com
```

Burada bir neçə maraqlı şey var. Bu layihəyə e-poçtları vurğulamaq və ya yenidən yönəltmək ist əyirsinizsə və ya hətta Pull Request etmək istəsəniz, `Message-ID`-də olan məlumatlar sizə `<user>/<project>/<type>/<id>`-dəki bütün məlumatları verir. Bu, məsələn, bir problem olsaydı, mə sələn, `<type>` sahəsi "pull" əvəzinə "issues" olardı.

`List-Post` və `List-Unsubscribe` sahələri o deməkdir ki, bunları başa düşən bir poçt müştəriniz varsa,asanlıqla siyahıya bir mesaj göndərə və ya mövzudan "Unsubscribe" deməkdir. Bu, bildirişin veb versiyasındakı "mute" düyməsini basmaqla və ya Issue və ya Pull Request səhifəsindəki "Unsubscribe" düyməsini basmaqla eyni olacaqdır.

Həm e-poçt, həm də veb bildirişləriniz aktivdirsə və bildirişin e-poçt versiyasını oxusanız, poçt müştərinizdə icazə verilən şəkillər varsa veb versiyası da oxunmuş kimi qeyd olunur.

Xüsusi Fayllar

GitHub depolarınızda olduqda fərq edəcəyi bir neçə xüsusi fayl var.

README

Birincisi, GitHub'ın nəsr kimi tanıdığı hər hansı bir formatda ola bilən `README 'sənədidir. Məsələn, `README`, `README.md`, `README.asciidoc` və s. ola bilər. GitHub mənbəyinizdə README faylını görürsə, onu layihənin açılış səhifəsində göstərəcəkdir.

Bir çox komanda depo və ya layihə üçün yeni ola biləcək birisi üçün bütün müvafiq layihə m əlumatlarını saxlamaq üçün bu faylı istifadə edir. Bura ümumiyyətlə aşağıdakılar daxildir:

- Layihə nədir
- Konfiqurasiya və quraşdırılma qaydaları
- Onu istifadə və ya işlətmək üçün bir nümunə
- Layihə çərçivəsində təklif olunan lisenziya
- Buna necə töhfə vermək olar

GitHub bu faylı göstərəcəyi üçün əlavə anlaşma rahatlığı üçün şəkillər və ya bağlantılar yerləşdirə bilərsiniz.

CONTRIBUTING

GitHub'un tanıdığı digər xüsusi fayl `CONTRIBUTING` faylıdır. Hər hansı bir fayl uzantısı ilə `CONTRIBUTING` adlı bir faylınız varsa, hər kəs Pull Request açmağa başlayanda GitHub CONTRIBUTING file olduqda Pull Request açmaq

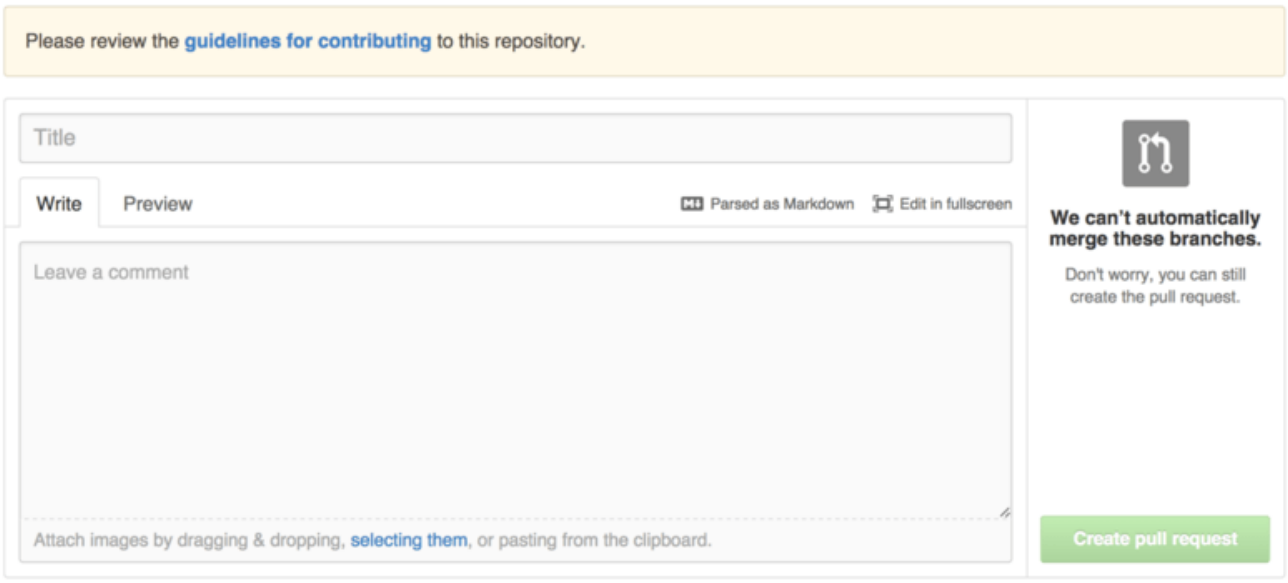


Figure 123. CONTRIBUTING file olduqda Pull Request açmaq

Burada fikir budur ki, istədiyiniz və ya istəmədiyiniz konkret şeyləri proyektinizə göndərilən Pull Request-də göstərə bilərsiniz. Bu yolla insanlar Pull Request-i açmadan əvvəl təlimatları həqiqətən oxuya bilərlər.

Layihə İdarəçiliyi

Ümumiyyətlə bir layihə ilə edə biləcəyiniz bir çox inzibati iş yoxdur, ancaq maraq doğuran bir

neçə maddə var.

Default Branch-ı Dəyişdirmək

Standart branch kimi “master” dən başqa bir branch istifadə edirsinizsə, insanların Pull Request-l əri açmasını və ya vrasayılan olaraq görməsini istəyirsinizsə, bunu “Options” bölməsində deponun parametrlər səhifəsində dəyişə bilərsiniz.

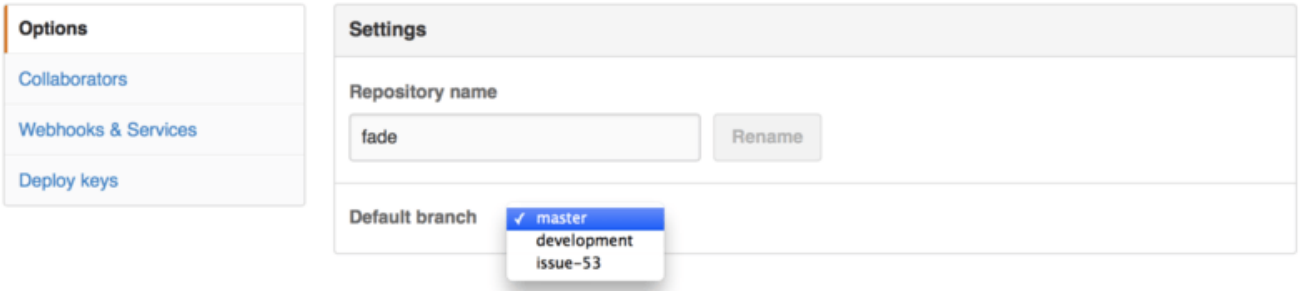


Figure 124. Bir layihə üçün default branch dəyişdirin

Açılan menudan standart branch-ı dəyişdirin və o zamandan etibarən bütün əsas əməliyyatlar üçün, o cümlədən kiminsə deponu klonlaşdırdıqda bu barnch-ın yoxlanılması da daxil olmaqla standart olacaqdır.

Bir Layihəni Köçürmək

Bir layihəni GitHub-dakı başqa bir istifadəçiyə və ya bir təşkilata köçürmək istəyirsinizsə, bunu etməyə imkan verən depo parametrləri səhifənizin eyni “Options” bölməsi altındakı “Transfer ownership” seçimi var.

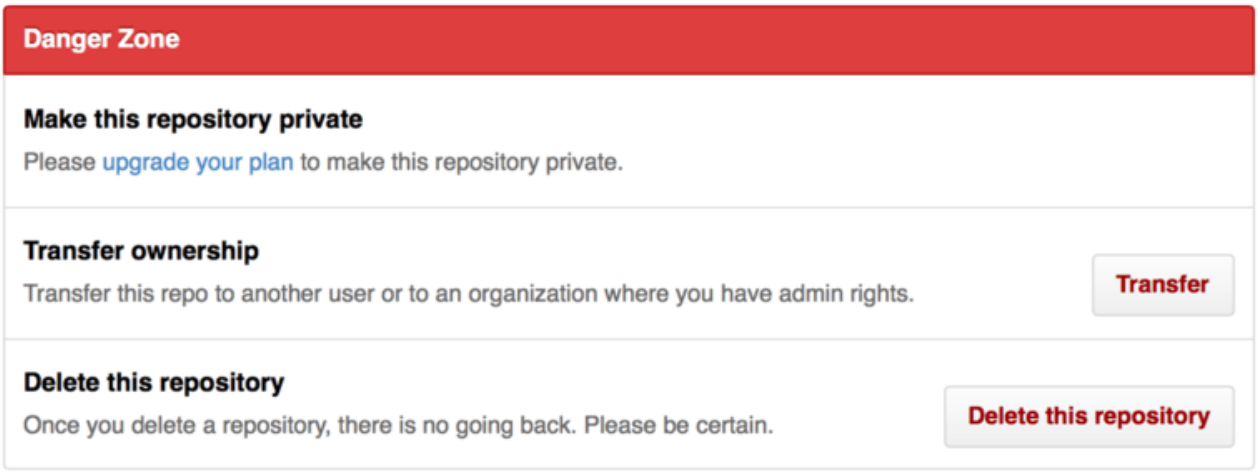


Figure 125. Bir layihəni başqa bir GitHub istifadəçisinə və ya Təşkilata köçürün

Bir layihədən imtina edirsinizsə və kimsə onu öz üzərinə götürmək istəyirsə və ya layihəniz böyüyürsə, onu bir təşkilata keçirməyiniz daha faydalıdır.

Bu, təkcə bütün izləyiciləri və ulduzları ilə birlikdə deponu başqa yerə köçürmür, eyni zamanda URL-dən yeni yerə yönləndirmə qurur. Bu, yalnız veb sorğularını deyil, Git-dən klonları və alış-verişi yönləndirəcəkdir.

Bir Təşkilatı Idarə Etmək

Tək istifadəçi hesablarından əlavə olaraq, GitHub-da Təşkilatlar adlandırılan hesablar da var. Şəxsi hesablar kimi Təşkilat hesabları da bütün layihələrinin mövcud olduğu bir namespace-ə malikdir, lakin bir çox şeylər fərqlidir. Bu hesablar layihələrin ortaq mülkiyyəti olan bir qrup insanı təmsil edir və bu insanların alt qruplarını idarə etmək üçün bir çox vasitə var. Normalda bu hesablar Açıq Mənbə qrupları (məsələn, “perl” və ya “rails”) və ya şirkətlər (məsələn,“google” və ya “twitter”) üçün istifadə olunur.

Təşkilat Əsasları

Bir təşkilat yaratmaq olduqca asandır; hər hansı bir GitHub səhifəsinin yuxarı sağ hissəsindəki “+” işarəsini vurun və menyudan “New organization” seçin.

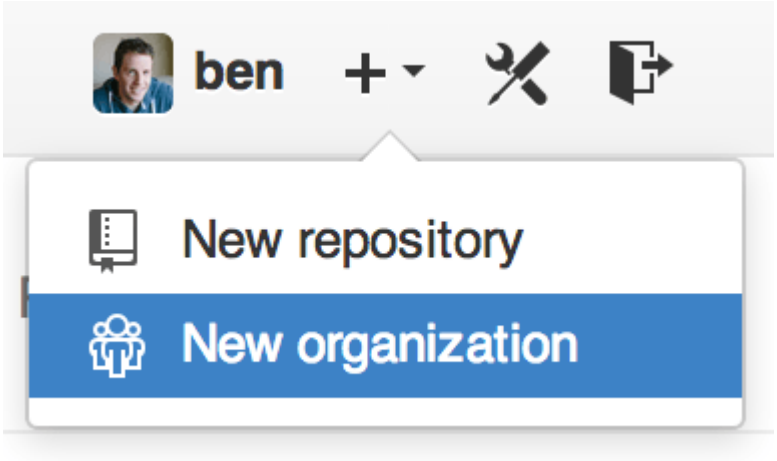


Figure 126. “New organization” menyusu

Əvvəlcə təşkilatınızı adlandırmalı və qrup üçün əsas əlaqə nöqtəsi üçün bir e-poçt ünvanı təqdim etməlisiniz. Sonra istəsəniz digər istifadəçiləri hesabın sahibi olmağa dəvət edə bilərsiniz.

Bu addımları izləyin və tezliklə yeni bir təşkilatın sahibi olacaqsınız. Şəxsi hesablardakı kimi, orada saxlamağı planlaşdırdığınız hər şey açıq mənbə olacağı təqdirdə təşkilatlar pulsuzdur.

Bir təşkilatın sahibi olaraq, bir depo fork etdikdə, təşkilatınızın namespace-nə fork seçmək seçiminiz olacaqdır. Yeni depolar yaratdıqda onları ya şəxsi hesabınız altında və ya sahibi olduğunuz təşkilatlardan yarada bilərsiniz.

Bu təşkilatlar altında yaradılan hər hansı bir yeni deponu avtomatik olaraq “seyr edirsiniz”.

Yalnız Avatar-ınız-da olduğu kimi, bir az fərdiləşdirmək üçün təşkilatınız üçün bir avatar yükləyə bilərsiniz. Həm də şəxsi hesablar kimi, bütün depolarınızı siyahıya alan və digər insanlar tərəfind ən görünə bilən təşkilat üçün açılış səhifəsi var.

İndi bir az fərqli olan bəzi şeyləri təşkilati hesabla əhatə edək.

Komandalar

Təşkilatlar ayrı-ayrı insanlarla, sadəcə təşkilat daxilindəki fərdi istifadəçi hesabları və depoların bir qruplaşması və bu insanların bu depolarda hansı növ giriş imkanları olduğu qruplar yolu ilə

əlaqələndirilir.

Məsələn, şirkətinizin üç depoya sahib olduğunu deyək: `frontend`, `backend` və `deployscripts`. HTML/CSS/JavaScript developerlərinizin `frontend` və bəlkə də `backend`-ə daxil olmasını istərdiniz, və Əməliyyatlarınızın `backend` və `deployscripts`-ə daxil olmasını istəyirsiniz.

Təşkilat səhifəsi sizə bu təşkilatın tərkibindəki bütün depoların, istifadəçilərin və komandaların sadə bir tablosunu göstərir.

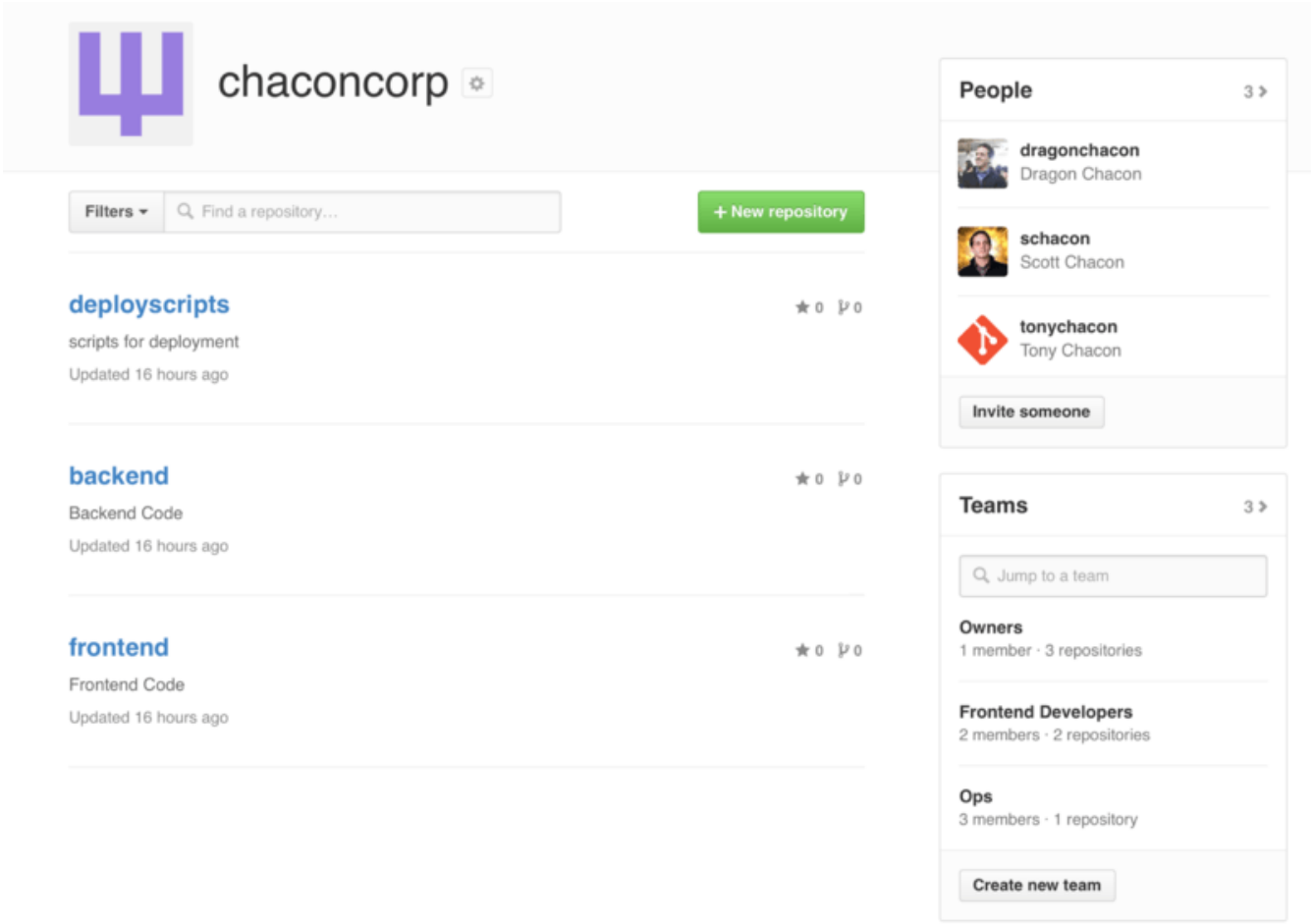


Figure 127. Təşkilat səhifəsi

Komandaları idarə etmək üçün, Təşkilat səhifəsi səhifədəki sağ tərəfdəki Komandalar düyməsini vura bilərsiniz. Bu, komandaya üzvlər əlavə etmək, komandaya depo yerləri əlavə etmək və ya komanda üçün parametrləri və giriş nəzarət səviyyələrini idarə etmək üçün istifadə edə biləcəyiniz bir səhifəyə gətirəcəkdir. Hər komanda yalnız oxumaq, oxumaq/yazmaq və ya depolara inzibati giriş əldə edə bilər. Bu səviyyəni Komanda səhifəsi-dəki' Parametrlər “Settings” vuraraq dəyişə bil ərsiniz.

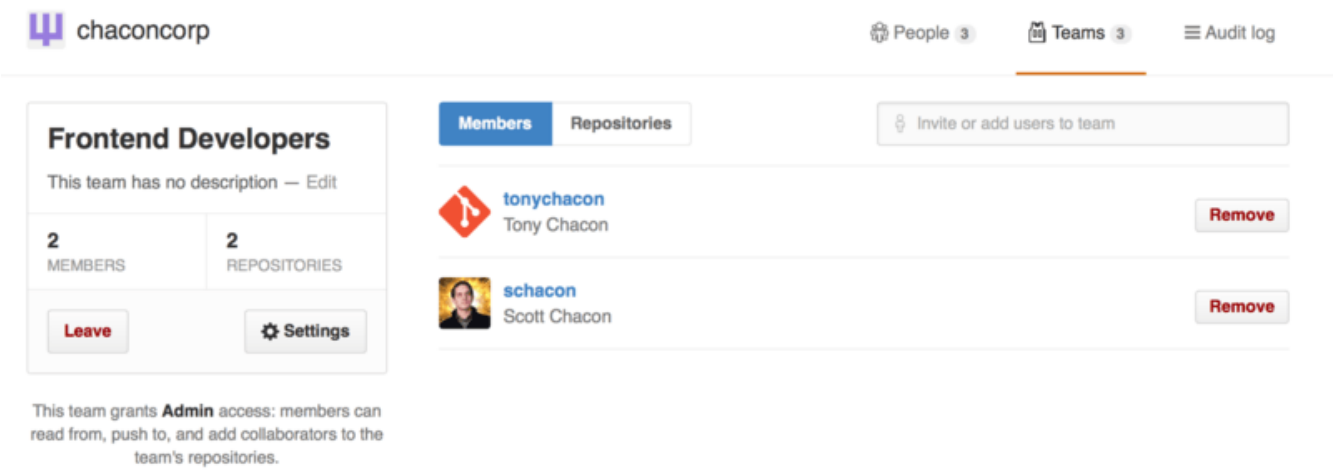


Figure 128. Komanda səhifəsi

Kimisə bir komandaya dəvət etdikdə hımin şəxslər dəvət olunduğunu bildirən bir e-poçt alacaqlar.

Bundan əlavə, `@mentions` komandası (məsələn, `@acmecorp/frontend`) fərdi istifadəçilərlə eyni işləyir, yalnız komandanın **bütün** üzvləri sonra mövzuya qoşulur. Bir komandada kiminsə diqqətini çə kmək istəsəniz, faydalıdır, ancaq kimin soruşacağını dəqiq bilmirsiniz.

Bir istifadəçi istənilən sayda komandaya aid ola bilər, buna görə özünüzü yalnız giriş-nəzarət qrupları ilə məhdudlaşdırmayın. Xüsusi maraq qrupları, `ux`, `css` və ya `refactoring` müəyyən suallar üçün, digərləri isə `legal` və `colorblind` üçün tamamilə fərqli bir növ üçün faydalıdır.

Audit Log

Təşkilatlar sahiblərinə təşkilat daxilində baş verənlər haqqında bütün məlumatları əldə etmək imkanı verir. *Audit Log* bölməsinə girib bir təşkilat səviyyəsində hansı hadisələrin baş verdiyini, kimlərin dünyada nə etdiklərini görə bilərsiniz.

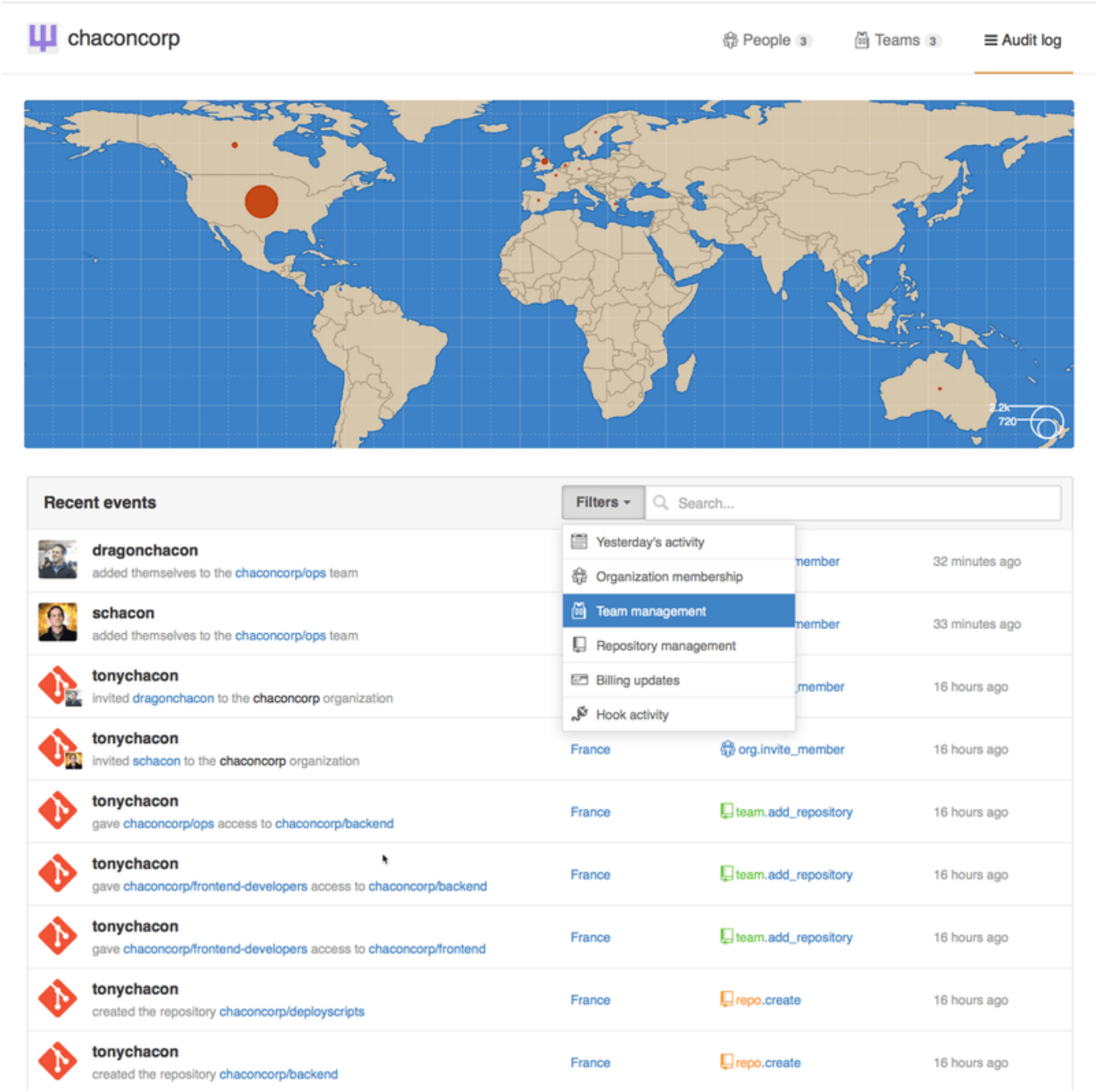


Figure 129. Audit log

Ayrıca hadisələrin müəyyən növlərini, müəyyən yerləri və ya müəyyən insanları süzgəcdən keçirə bilərsiniz.

GitHub Skriptləmə

Beləliklə, indi GitHub'un əsas xüsusiyyətləri və iş axınlarının hamısını əhatə etdik, ancaq hər hansı bir böyük qrup və ya layihənin istədikləri və ya inteqrasiya etmək istədikləri xarici xidmətlər ola bilər.

Bizim üçün xoşbəxtlikdən, GitHub bir çox cəhətdən həqiqətən hack-lənə bilər. Bu hissədə GitHub-un istədiyi şəkildə işləməsi üçün GitHub hooks sistemindən və onun API-dən necə istifadə edəc əyimizi əhatə edəcəyik.

Servislər and Hook-lar

GitHub depo idarəsinin Servislər and Hook-larr bölməsi GitHub'un xarici sistemlərlə qarşılıqlı ə laqəsinin ən asan yoludur.

Servislər

Əvvəlcə Servislərə nəzər salacağıq. Həm Hook-lar, həm də Servislər inteqrasiyasını əvvəlcədən H əmkarlar əlavə etmək və layihənizin standart bölməsini dəyişdirməyə baxdığımız deponun Ayarlar bölməsində tapa bilərsiniz. "Webhooks and Services" bölməsində Servislər and Hook-lar konfiqurasiya bölməsi kimi bir şey görəcəksiniz.

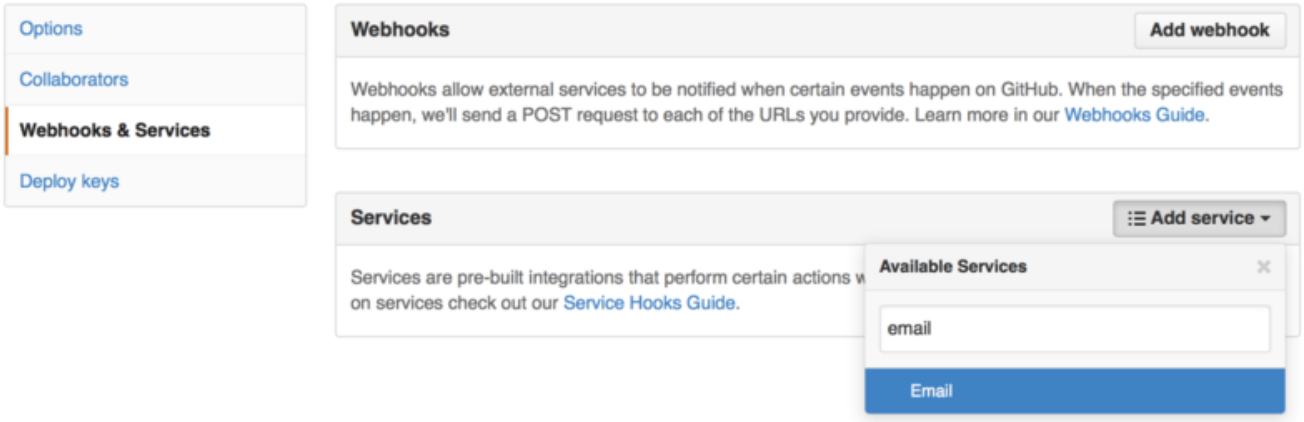


Figure 130. Servislər and Hook-lar konfiqurasiya bölməsi

Seçdiyiniz onlarla servis var, əksəriyyəti digər kommersiya və açıq mənbə sistemlərinə inteqrasiya edir. Onların əksəriyyəti Davamlı İnteqrasiya xidmətləri, səhv və problem izləyiciləri, söhbət otağı sistemləri və sənədləşdirmə sistemləri üçündür. E-poçt fork-unu quraşdırmaqla çox sadə bir şə kildə keçəcəyik. "Add Service" açılan menyudan "email" seçsəniz, E-poçt xidmətinin konfiqurasiyası kimi bir konfiqurasiya ekranı alacaqsınız.

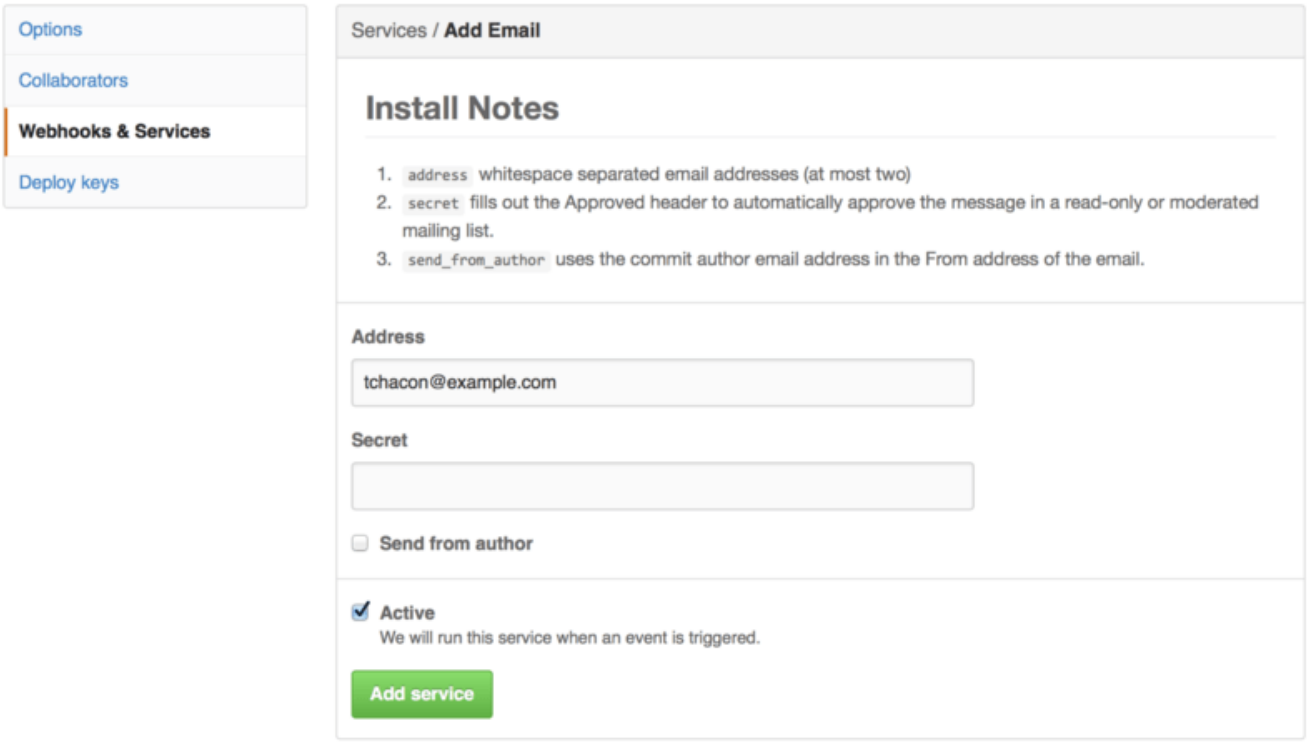


Figure 131. E-poçt xidmətinin konfiqurasiyası

Bu vəziyyətdə, “Add service” düyməsini vursaq, göstərilən e-poçt ünvanı, hər kimsə depoya push edərkən hər dəfə bir e-poçt alacaq. Servislər çox sayda müxtəlif hadisəni dinləyə bilər, ancaq çoxu push etmə hadisələrini dinləyir və sonra bu məlumatlarla bir şey edə bilər.

GitHub ilə inteqrasiya etmək istədiyiniz bir sistem varsa, mövcud bir servis inteqrasiyasının olub olmadığını görmək üçün buranı yoxlamalısınız. Məsələn, kod bazasında testlər aparmaq üçün Jenkins-dan istifadə edirsinizsə, Jenkins-inin servis inteqrasiyasını kimsə depolarına push etdikdə hər dəfə bir test işə salmağa imkan verə bilərsiniz.

Hook-lar

Daha spesifik bir şeyə ehtiyacınız varsa və ya bu siyahıya daxil olmayan bir servis və ya saytla inteqrasiya etmək istəsəniz, bunun əvəzinə daha ümumi hook-lar sistemindən istifadə edə bil ərsiniz. GitHub depo hook-ları olduqca sadədir. Bir URL göstərmisinizsə və GitHub istədiyiniz hər hansı bir hadisədə HTTP yüklənməsini həmin URL-ə göndərəcəkdir.

Ümumiyyətlə bu iş üsulu, GitHub fork yükünü dinləmək və qəbul edildikdə məlumatla bir şey etm ək üçün kiçik bir veb xidməti qura bilərsiniz.

Bir fork aktivləşdirmək üçün Servislər and Hook-lar konfiqurasiya bölməsi-dakı “Add webhook” düyməsini klikləyin. Bu sizi Web hook konfiqurasiyası kimi görünən bir səhifəyə gətirəcəkdir.

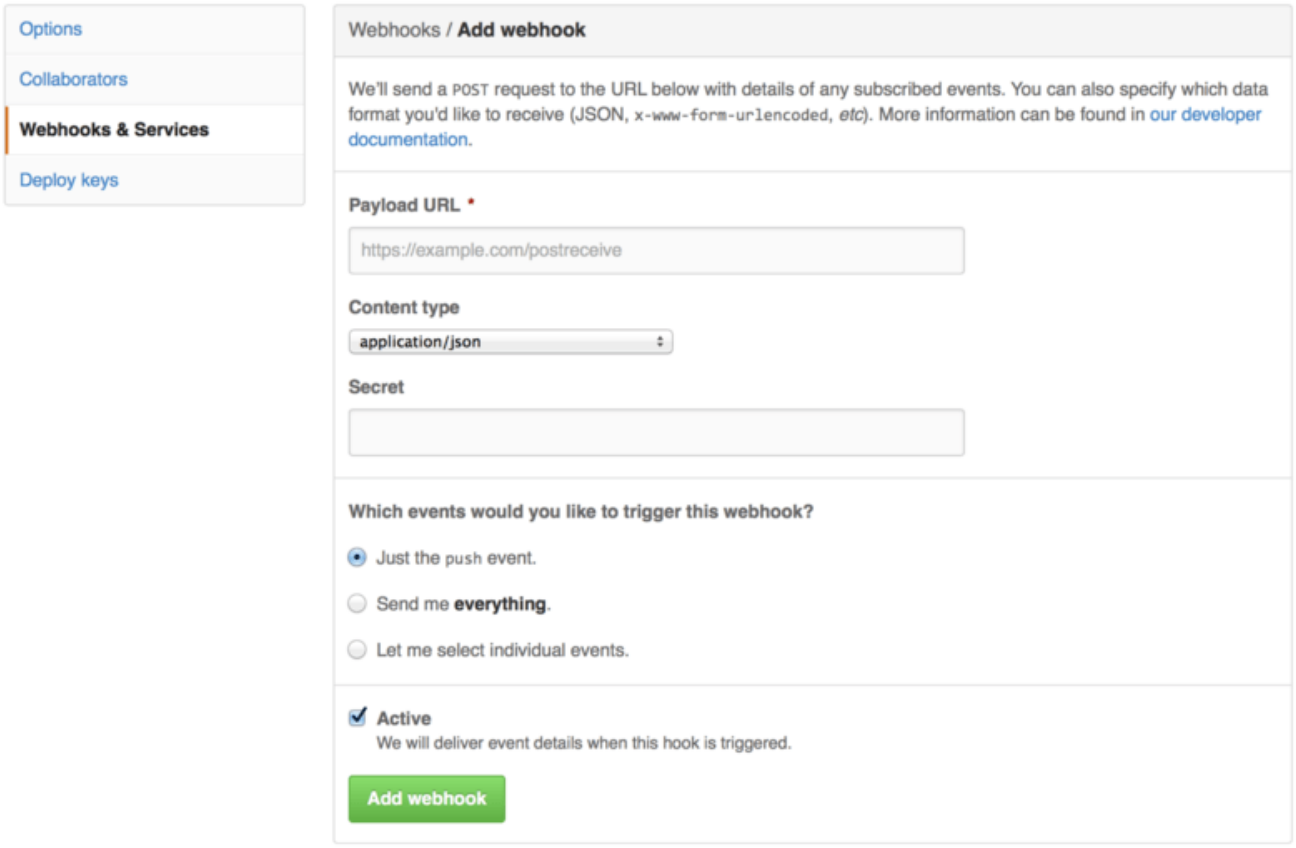


Figure 132. Web hook konfiqurasiyası

Bir web hook üçün konfiqurasiya olduqca sadədir. Əksər hallarda sadəcə bir URL və gizli bir açar daxil edir və “Add webhook” düyməsini vurursunuz. GitHub-ın yükləməsini istədiyiniz hadisələrin bir neçə variantı var - varsayılan kimsə deponuzun hər hansı bir branch-ına yeni kodu `push` hadisəsi üçün bir yük almaqdır.

Bir web hook-u idarə etmək üçün qura biləcəyiniz bir veb servisin kiçik bir nümunəsini baxaq. Ruby web framework Sinatra’dan istifadə edəcəyik, çünki olduqca qısadır və etdiyimiz işləri asanlıqla görə bilərsiniz.

Müəyyən bir şəxs müəyyən bir sənəd dəyişdirərək layihəmizin müəyyən bir branch-na push etsə, e-poçt almaq istədiyimizi deyək. Bunu kifayət qədər asanlıqla bu kimi kodla edə bilərik:

```
require 'sinatra'
require 'json'
require 'mail'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON

  # gather the data we're looking for
  pusher = push["pusher"]["name"]
  branch = push["ref"]

  # get a list of all the files touched
  files = push["commits"].map do |commit|
    commit['added'] + commit['modified'] + commit['removed']
  end
  files = files.flatten.uniq

  # check for our criteria
  if pusher == 'schacon' &&
     branch == 'ref/heads/special-branch' &&
     files.include?('special-file.txt')

    Mail.deliver do
      from     'tchacon@example.com'
      to       'tchacon@example.com'
      subject  'Scott Changed the File'
      body     "ALARM"
    end
  end
end
```

Burada GitHub-un bizə verdiyi JSON yükünü götürürük və onu kimin push etdiyini, hansı branch-a push etdiyini və push etdikləri bütün sənədlərdə hansı sənədlərə toxunduğunu axtarırıq. Sonra meyarlarımıza uyğun olduğunu yoxlayırıq və uyğun olduqda bir e-poçt göndəririk.

Bu kimi bir şeyi inkişaf etdirmək və sınamaq üçün, fork hazırladığınız eyni ekranda gözəl bir inkişaf etdirici konsolunuz var. GitHub'un webhook üçün etdiyi cəhdləri görə bilərsiniz.

Hər bir fork üçün təhvil verildiyi anda istəyə və cavaba cavab verən body və başlıqları incəliyə bil ərsiniz. Bu, hook-ları test və debug etməyi olduqca asanlaşdırır.

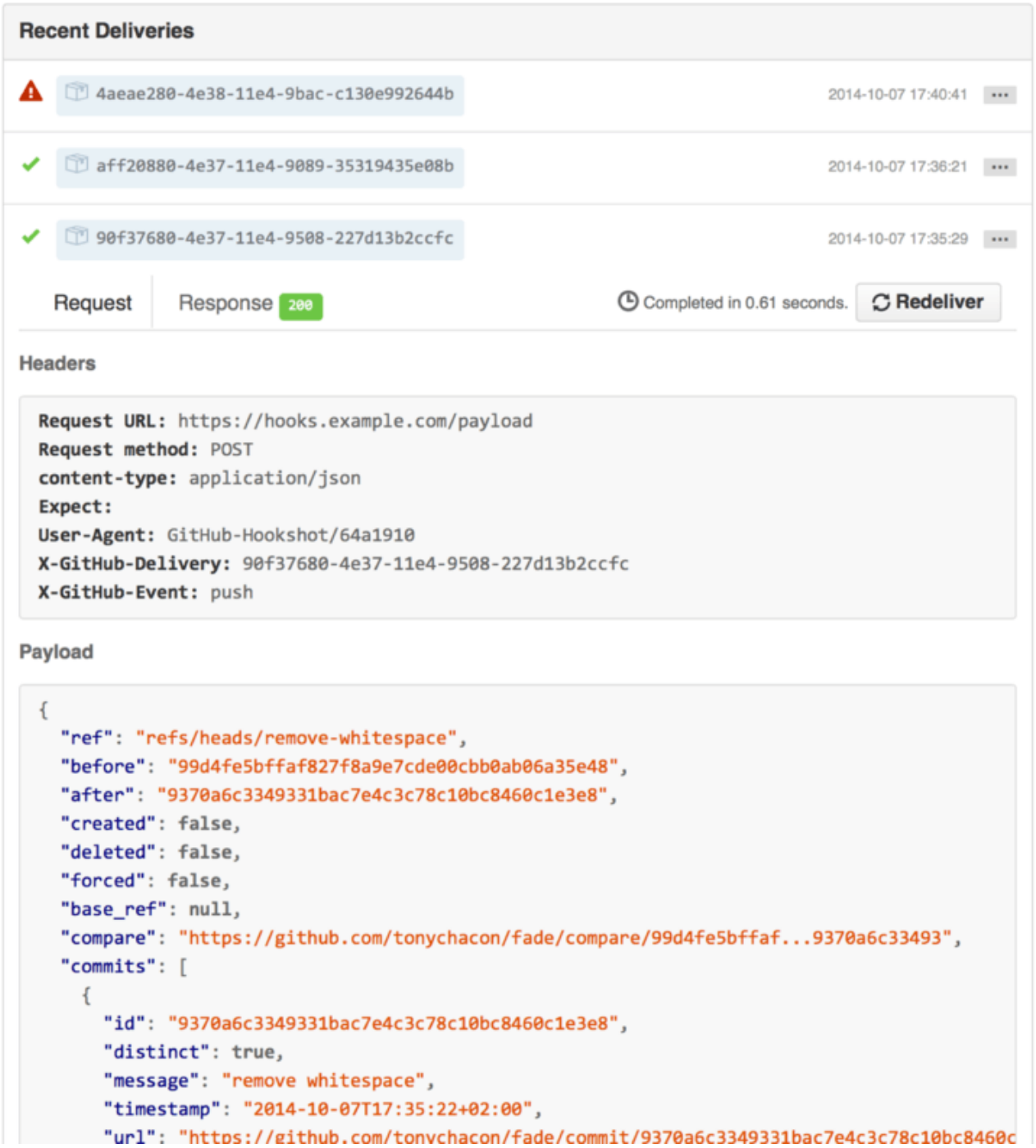


Figure 133. Web hook debugging informasiyası

Bunun digər böyük xüsusiyyəti, xidmətinizi asanlıqla sınamaq üçün hər hansı bir payload-ı geri qaytara biləcəyinizdir.

Webhook-ların necə yazılacağı və dinləyə biləcəyiniz bütün fərqli hadisə növləri haqqında daha çox məlumat üçün https://developer.github.com/webhooks/ saytından GitHub Developer sənədlə rinə baxın.

GitHub API

Servislər və hook-lar, depolarınızda baş verən hadisələr barədə push bildirişlərini almaq üçün bir yol verir, ancaq bu hadisələr haqqında daha çox məlumat lazımdırsa, nə etməlisiniz? Əm əkdaşlarınızı və ya etiketləmə məsələlərini əlavə etmək kimi bir şeyi avtomatlaşdırmaq lazımdırsa,

nə etməli?

GitHub API lazımlı olduğu yerdir. GitHub-da veb saytında avtomatlaşdırılmış bir şəkildə edə biləc əyiniz hər şeyi etmək üçün çox sayda API nöqtəsi var. Bu bölmədə doğrulamağı və API-yə necə qoşulacağımızı, bir məsələyə necə şərh verəcəyinizi və API vasitəsilə Pull Request-in vəziyyətini necə dəyişdirəcəyinizi öyrənəcəyik.

Əsas İstifadə

Siz edə biləcəyiniz ən əsas şey doğrulama tələb etməyən endpoint-də sadə bir GET request-dir. Bu istifadəçi və ya açıq mənbəli bir layihədə yalnız oxu məlumatı ola bilər. Məsələn, “schacon” adlı bir istifadəçi haqqında daha çox bilmək istəyiriksə, belə bir şey işlədə bilərik:

```
$ curl https://api.github.com/users/schacon
{
  "login": "schacon",
  "id": 70,
  "avatar_url": "https://avatars.githubusercontent.com/u/70",
# …
  "name": "Scott Chacon",
  "company": "GitHub",
  "following": 19,
  "created_at": "2008-01-27T17:19:28Z",
  "updated_at": "2014-06-10T02:37:23Z"
}
```

Təşkilatlar, layihələr, məsələlər, commit-lər haqqında məlumat almaq üçün bu kimi endpoint-lər var - GitHub-da hər şey haqqında açıq şəkildə görə biləcəksiniz . Siz hətta ixtiyari Markdown göst ərmək üçün API-dən istifadə edə və ya `.gitignore` şablonunu tapa bilərsiniz.

```
$ curl https://api.github.com/gitignore/templates/Java
{
  "name": "Java",
  "source": "*.class

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
*.ear

# virtual machine crash logs, see
https://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
"
}
```

Bir Məsələyə Münasibət Bildirmək

Bununla birlikdə, bir Issue or Pull Request şərhini şərh etmək kimi bir veb saytında bir hərəkət etmək istəsəniz və ya şəxsi məzmuna baxmaq və ya qarşılıqlı əlaqə qurmaq istəyirsinizsə, identifikasiya lazımdır.

İdentifikasiyanın bir neçə yolu var. Əsas identifikasiyanı yalnız istifadəçi adınızı və şifrənizi istifadə edə bilərsiniz, ancaq ümumiyyətlə şəxsi giriş tokenindən istifadə etmək daha yaxşı bir fikirdir. Bunu parametrlər səhifənizdəki “Applications” bölməsindən əldə edə bilərsiniz.

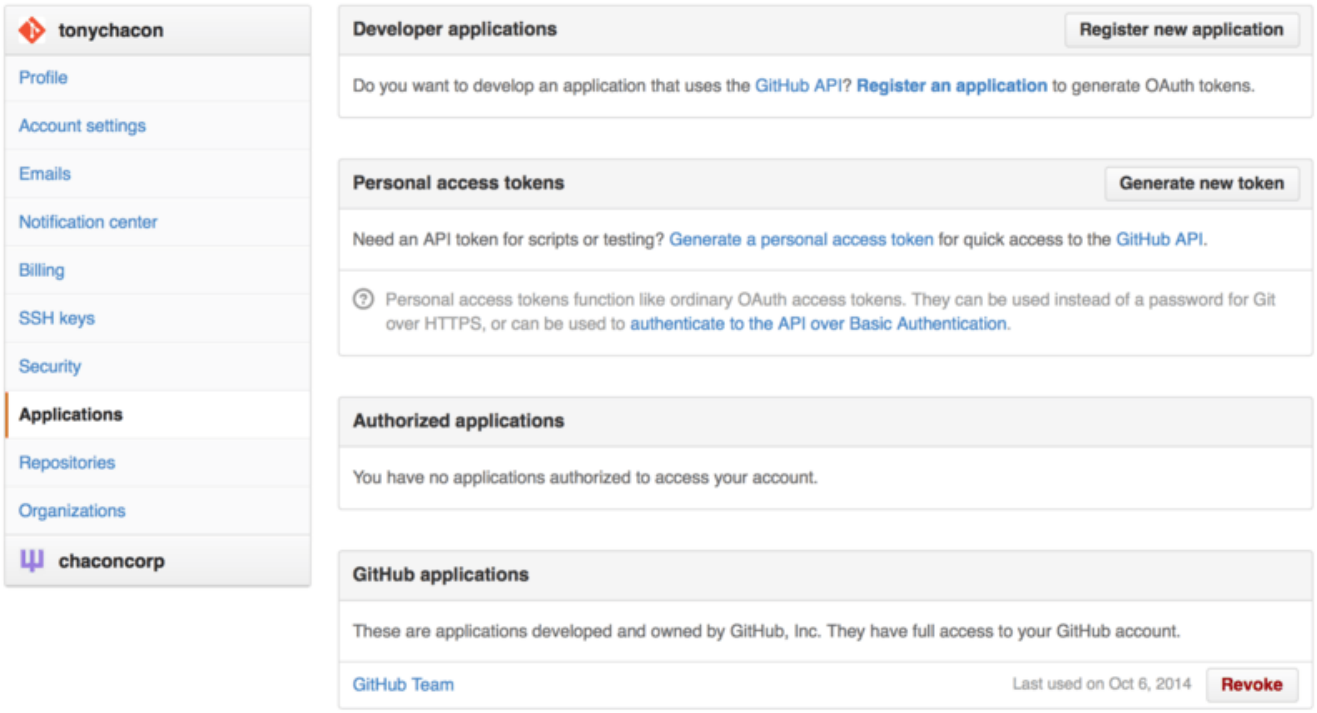


Figure 134. Parametrlər səhifənizdəki “Applications” bölməsindən giriş tokeninizi yaradın

Bu işarə və təsvir üçün hansı sahələri istədiyinizi soruşacaqdır. Skriptiniz və ya tətbiqiniz artıq istifadə edilmədiyi zaman token-i çıxartmağı rahat hiss etdiyiniz üçün yaxşı bir təsviri istifadə etdiyinizə əmin olun.

GitHub sizə bir token-i yalnız bir dəfə göstərəcəkdir, buna görə də onu kopyalayın. İndi istifadə ed ərək istifadəçi adı və şifrə istifadə etmək əvəzinə skriptinizdə identifikasiya etmək üçün istifadə edə bilərsiniz. Bu, çox xoşdur, çünki nə etmək istədiyinizi məhdudlaşdıra bilərsiniz və token-in l əğvi mümkündür.

Bu da dərəcəniz həddini artırmağın əlavə üstünlüyünə malikdir. Doğrulama olmadan, saatda 60 sorğu ilə məhdudlaşacaqsınız. Doğruladığınız təqdirdə saatda 5000 müraciət edə bilərsiniz.

Beləliklə, məsələlərimizdən birinə şərh etmək üçün istifadə edək. Deyək ki, 6-cı sayda müəyyən bir məsələyə münasibət bildirmək istəyirik. Bunu etmək üçün, yalnız Avtorizasiya başlığı olaraq hazırladığımız işarə ilə `repos/<user>/<repo>/issues/<num>/comments`-a bir HTTP POST tələb etm əliyik.

```
$ curl -H "Content-Type: application/json" \
       -H "Authorization: token TOKEN" \
       --data '{"body":"A new comment, :+1:"}' \
       https://api.github.com/repos/schacon/blink/issues/6/comments
{
  "id": 58322100,
  "html_url": "https://github.com/schacon/blink/issues/6#issuecomment-58322100",
  ...
  "user": {
    "login": "tonychacon",
    "id": 7874698,
    "avatar_url": "https://avatars.githubusercontent.com/u/7874698?v=2",
    "type": "User",
  },
  "created_at": "2014-10-08T07:48:19Z",
  "updated_at": "2014-10-08T07:48:19Z",
  "body": "A new comment, :+1:"
}
```

İndi bu məsələyə keçsəniz, yalnız GitHub API-dən bir şərh göndərildi -da uğurla yayımladığımızı görə bilərsiniz.

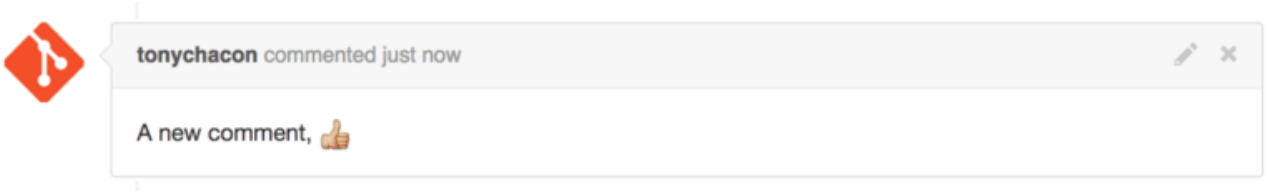


Figure 135. GitHub API-dən bir şərh göndərildi

Veb saytından edə biləcəyiniz hər şeyi etmək üçün API-dən istifadə edə bilərsiniz - mərhələləri yaratmaq və təyin etmək, İnsanları Issues və Pull Requests-a təyin etmək, etiketlər yaratmaq və d əyişdirmək, commit məlumatlarına daxil olmaq, yeni commit-lər və branch-lar yaratmaq, açmaq, bağlamaq və ya etmək Pull Requests-in birləşməsi, qruplar yaratmaq və redaktə etmək, Pull Request-dəki kod sətirlərinə şərh vermək, saytı axtarmaq və s.

Pull Request Statusunun Dəyişdirilməsi

Pull Request-ləri ilə işləyirsinizsə, həqiqətən faydalı olduğundan baxacağımız bir son nümunə var. Hər bir commit-in onunla əlaqəli bir və ya daha çox statusu ola bilər və bu statusu əlavə etmək və soruşmaq üçün bir API var.

Davamlı İnteqrasiya və sınaq servislərinin əksəriyyəti bu API-dən istifadə edilən kodu sınamaqla push etməərə reaksiya vermək üçün istifadə edir və sonra bu commit-in bütün sınaqlardan keçdiyini geri bildirir. Əgər təqdimatçı bütün töhfələr qaydalarına əməl edirsə, commit düzgün imzalanıbsa, göndərmə mesajının düzgün formatlandığını yoxlamaq üçün istifadə edə bilərsiniz.

Deyək ki, əmanət mesajınızdakı `Signed-off-by` sətirini yoxlayan kiçik bir veb xidmətə xitabən depo qurduq.

```
require 'httparty'
require 'sinatra'
require 'json'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON
  repo_name = push['repository']['full_name']

  # look through each commit message
  push["commits"].each do |commit|

    # look for a Signed-off-by string
    if /Signed-off-by/.match commit['message']
      state = 'success'
      description = 'Successfully signed off!'
    else
      state = 'failure'
      description = 'No signoff found.'
    end

    # post status to GitHub
    sha = commit["id"]
    status_url = "https://api.github.com/repos/#{repo_name}/statuses/#{sha}"

    status = {
      "state"       => state,
      "description" => description,
      "target_url"  => "http://example.com/how-to-signoff",
      "context"     => "validate/signoff"
    }
    HTTParty.post(status_url,
      :body => status.to_json,
      :headers => {
        'Content-Type'  => 'application/json',
        'User-Agent'    => 'tonychacon/signoff',
        'Authorization' => "token #{ENV['TOKEN']}" }
    )
  end
end
```

Ümid edirik, bunu izləmək olduqca sadədir. Bu web hook işlədicisinə, sadəcə push etdikləri hər bir commit-ə baxırıq, commit mesajında *Signed-off-by* sətirini axtarırıq və nəhayət HTTP vasitəsilə `/repos/<user>/<repo>/statuses/<commit_sha>` vəziyyəti olan API endpointi tapırıq.

Bu halda bir vəziyyət (*success*, *failure*, *error*), baş verənlərin təsviri, istifadəçinin əlavə məlumat üçün gedə biləcəyi hədəf URL və bir commit üçün çox statuslu “context” göndərə bilərsiniz.. Məsəl ən, bir test servisi bir status təqdim edə bilər və bu kimi bir yoxlama servisi də bir status təqdim edə bilər - “context” sahəsi necə fərqləndiyini göstərir.

Kimsə GitHub-da yeni Pull Request açırsa və bu fork qurulubsa, API vasitəsilə Commit status kimi bir şey görə bilərsiniz.

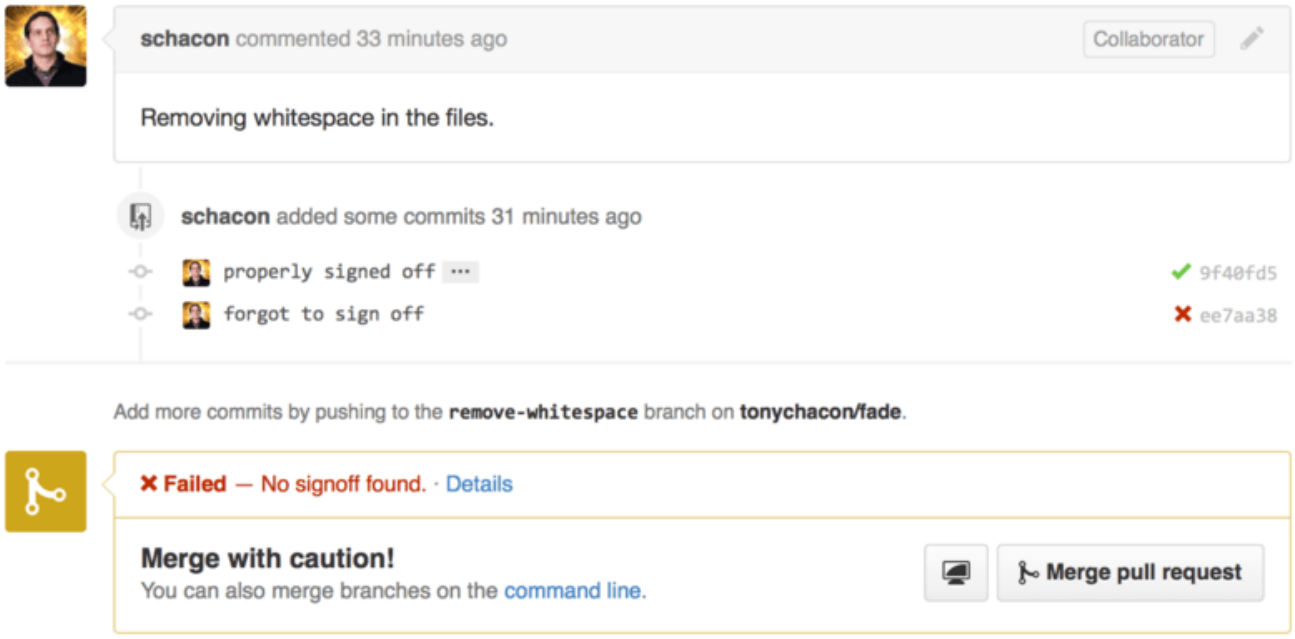


Figure 136. API vasitəsilə Commit status

İndi mesajdakı “Signed-off-by” sətri olan və yazarın imzalamağı unutduğu yerdən qırmızı xaç olan işarənin yanında bir az yaşıl bir işarə görə bilərsiniz. Ayrıca Pull Request branch-dakı son commitin vəziyyətini aldığını və uğursuz olduqda xəbərdarlıq etdiyini görə bilərsiniz. Bu API-ni test nəticələri üçün istifadə edirsinizsə, sonuncu commit-in uğursuz olduğu testləri təsadüfən birl əşdirməməyiniz üçün çox faydalıdır.

Octokit

Bu nümunələrdə `curl` və sadə HTTP sorğuları ilə demək olar ki, hər şeyi etdiyimizə baxmayaraq bu API-ni daha idiomatik şəkildə təqdim edən bir neçə açıq mənbəli kitabxana mövcuddur. Bu yazı zamanı dəstəklənən dillərə Go, Objective-C, Ruby və .NET daxildir. Bunlar haqqında daha çox m əlumat üçün https://github.com/octokitsəhifəsinə baxın, çünki sizin üçün HTTP-nin çox hissəsi idarə olunur.

Ümid edirik, bu vasitələr xüsusi iş axınlarınız üçün daha yaxşı işləmək üçün GitHub-u düzəltməyə və dəyişdirməyə kömək edə bilər. Bütün API sənədləri və ümumi tapşırıqlar üçün təlimatlar üçün https://developer.github.com səhifəsinə baxın.

Qısa Məzmun

İndi bir GitHub istifadəçisisiniz. Bir hesab yaratmağı, bir təşkilatı necə idarə etməyi, depo’lar yaratmağı və push etməyi, başqalarının layihələrinə töhfə verməyi və başqalarının töhfələrini q əbul etməyi bilirsiniz. Növbəti fəsildə, sizi həqiqətən Git ustası halına gətirəcək mürəkkəb vəziyyə tlərlə məşğul olmaq üçün daha güclü alətlər və tövsiyələr öyrənəcəksiniz.

Git Alətləri

İndiyə qədər mənbə kodu nəzarəti üçün Git deposunu idarə etməli və ya saxlamağınız lazım olan gündəlik əmrlərin və iş axınlarının çoxunu öyrəndiniz. Faylları tracking və committing əsas tapşırıqlarını yerinə yetirdiniz və quruluş sahəsinin gücünü və yüngül mövzunun branching'ni və birləşməsini istifadə etdiniz.

İndi Git'in gündəlik olaraq istifadə edə bilməyəcəyiniz, ancaq bir anda ehtiyacınız ola biləcəyi çox güclü şeyləri araşdıracaqsınız.

Reviziya Seçimi

Git, bir sıra commit-lərə, commit-lər dəstinə və ya commit-lərə istinad etməyə imkan verir. Bunlar mütləq açıq deyil, bilmək faydalıdır.

Tək Reviziyalar

Tamamilə 40 xarakterli SHA-1 hash ilə hər hansı bir commit-ə istinad edə bilərsiniz, lakin commit-ləri ifadə etmənin daha çox insan dostu yolları var. Bu bölüm, hər hansı bir commit-ə istinad edə biləcəyiniz müxtəlif yolları əks etdirir.

Qısa SHA-1

Git, SHA-1 hash-nın ilk bir neçə simvolunu verdiyiniz təqdirdə nəyi nəzərdə tutduğunuzu başa düşmək üçün kifayət qədər ağıllıdır, qismən qarışıq ən azı dörd simvol uzun və birmənalıdır; Başqa sözlə, obyekt verilənlər bazasındakı heç bir obyektdə eyni prefikslə başlayan bir hash ola bilməz.

Məsələn, müəyyən bir funksionallıq əlavə etdiyinizi bildiyiniz xüsusi bir commit-i araşdırmaq üçün əvvəlcə commit-i tapmaq üçün `git log` əmrini işə sala bilərsiniz:

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    Fix refs handling, add gc auto, update tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    Add some blame and merge stuff
```

Bu vəziyyətdə, hash `1c002dd...` ilə başlayan commit-lə maraqlandığınızı varsayaq. Aşağıdakı `git show` varyasyonlarından hər hansı biri ilə əlaqəli olanı yoxlaya bilərsiniz (daha qısa versiyaların birmənalı olduğunu düşünərək):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git, SHA-1 dəyərləriniz üçün qısa, bənzərsiz bir qısaltmanı müəyyən edə bilər. `--abbrev-commit git log` əmrinə keçsəniz, çıxış daha qısa dəyərlərdən istifadə edəcək, lakin onları unikal saxlayır; yeddi simvol istifadə etmək üçün standartdır, lakin SHA-1-in birmənalı olması üçün onları daha uzun edir:

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d Change the version number
085bb3b Remove unnecessary test code
a11bef0 Initial commit
```

Ümumiyyətlə, səkkiz-on simvol bir proyektdə bənzərsiz olmaq üçün kifayətdir. Məsələn, 2019-cu ilin fevral ayından etibarən, Linux kernelinin (olduqca əhəmiyyətli bir layihədir) 875.000-dən çox commit-i və obyekt bazasında təxminən yeddi milyon obyekti var, ilk 12 simvolda SHA-1'ləri eyni olan iki obyekt yoxdur.

SHA-1 HAQQINDA QISA QEYD

Bir çox insan təsadüfi bir şəkildə, eyni SHA-1 dəyərinə qarışan depolarında iki f ərqli obyektə sahib olacaqlarından bir anda narahat olurlar. Bəs onda nə etmək lazımdır? Əgər deponuzdakı əvvəlki *fərqli* obyekti ilə eyni SHA-1 dəyərinə bərab ər olan bir obyekt törətmisinizsə, Git əvvəlki obyekti Git verilənlər bazanızda gör əcək, artıq yazıldığını düşünün və sadəcə yenidən istifadə edin. Bir nöqtədə yenidən həmin obyekti yoxlamağa çalışsanız, həmişə ilk obyektin məlumatlarını əldə edəcəksiniz.

Bununla birlikdə, bu ssenarinin nə qədər gülünc bir şəkildə ehtimal olunmadığının fərqində olmalısınız. SHA-1 həcmi 20 bayt və ya 160 bitdir. Tək bir toqquşma ehtimalının 50% olmasını təmin etmək üçün lazım olan təsadüfi yığılmış obyektlərin sayı təxminən 2^{80}-dir (toqquşma ehtimalını müəyyənləşdirm ək üçün düstur `p = (n(n-1)/2) * (1/2^160))`. 2^{80} 1.2 x 10^{24} təşkil edir və ya 1 milyon milyard. Bu, yer üzündə qum dənələrinin sayından 1,200 dəfə çoxdur.

Burada SHA-1 toqquşması üçün nə lazım olduğunu düşünmək üçün bir nümunə var. Yer üzündəki 6,5 milyard insanın hamısı proqramlaşdırma aparsaydı və hər saniyədə hər biri bütün Linux nüvə tarixinə (6.5 milyon Git obyekt) bərabər olan bir kod istehsal etsəydi və onu böyük bir Git deposuna salsaydı, təxminən 2 il çək ərdi. Bu depoda bir SHA-1 obyektinin toqquşma ehtimalı 50% -ə çatacaq qədər obyekt var qədər. Beləliklə, SHA-1 toqquşması, proqramlaşma komandanızın hər bir üzvünün eyni gecədə əlaqəsi olmayan hadisələrdə canavarların hücumuna m əruz qalması və öldürülməsi ehtimalı daha azdır.

Branch Referansları

Müəyyən bir commit-ə istinad etməyin bir sadə yolu branch-ın ucundakı commit-in olmasıdır; bu halda, sadəcə bir commit-ə istinad gözləyən hər hansı bir Git əmrində branch adını istifadə edə bil ərsiniz. Məsələn, bir branch-dakı son commit obyektini araşdırmaq istəyirsinizsə, aşağıdakı mövzuda əmrlər ekvivalentdir ki, `topic1` branch-ının `ca82a6d...` işarə etdiyini göstərir:

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

Bir branch-ın hansı konkret SHA-1-ə işarə etdiyini görmək və ya bu nümunələrdən hər hansı birinin SHA-1-lər baxımından nəyə bənzədiyini görmək istəyirsinizsə, `rev-parse` adlı Git santexnika alətindən istifadə edə bilərsiniz. Santexnika alətləri haqqında daha çox məlumat üçün Git'in Daxili İşləri-ə baxa bilərsiniz; əsasən, `rev-parse` aşağı səviyyəli əməliyyatlar üçün mövcuddur və gündəlik əməliyyatlarda istifadə üçün nəzərdə tutulmayıb. Ancaq bəzən həqiqətən nələrin baş verdiyini görmək lazım olduqda faydalı ola bilər. Burada branch-ınızda `rev-parse` işlədə bilərsiniz.

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

RefLog Qısa Adları

Gitin arxada işləyərkən arxa planda gördüyü işlərdən biri də `reflog` saxlamaqdır - HEAD və branch istinadlarınızın son bir neçə ayda olduğu bir qeyd.

Reflogunuzu `git reflog` istifadə edərək görə bilərsiniz:

```
$ git reflog
734713b HEAD@{0}: commit: Fix refs handling, add gc auto, update tests
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by the 'recursive' strategy.
1c002dd HEAD@{2}: commit: Add some blame and merge stuff
1c36188 HEAD@{3}: rebase -i (squash): updating HEAD
95df984 HEAD@{4}: commit: # This is a combination of two commits.
1c36188 HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

Branch ucunuz hər hansı bir səbəbdən yeniləndikdə Git bu məlumatları sizin üçün bu müvəqqəti tarixdə saxlayır. Köhnə commit-lərə də istinad etmək üçün reflog məlumatlarınızı istifadə edə bil ərsiniz. Məsələn, deposunuzun HEAD-in əvvəlki beşinci dəyərini görmək istəyirsinizsə, reflog çıxışında gördüyünüz `@{5}` istinadından istifadə edə bilərsiniz:

```
$ git show HEAD@{5}
```

Bu sintaksisdən branch-ın müəyyən bir müddət əvvəl harada olduğunu görmək üçün də istifadə edə bilərsiniz. Məsələn, dünən `master` branch-ınızın harada olduğunu görmək üçün yaza bilərsiniz:

```
$ git show master@{yesterday}
```

Bu, dünən "`master` branch-ınızın ucunun harada olduğunu göstərəcəkdir. Bu texnika yalnız hələ də qeydlərinizdə olan məlumatlar üçün işləyir, buna görə də bir neçə aydan yuxarı commit-lər axtarmaq üçün istifadə edə bilməzsiniz.

Reflog məlumatlarını `git log` çıxışı kimi formatlanmış şəkildə görmək üçün `git log -g` işlədə bil ərsiniz:

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: Fix refs handling, add gc auto, update tests
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    Fix refs handling, add gc auto, update tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

Qeyd etmək vacibdir ki, reflog məlumatları ciddi şəkildə localdır - bu, yalnız *deponuzda* etdiyiniz işlərin bir qeydidir. Referanslar başqasının deposunun kopyasında eyni olmayacaq; ayrıca, əvvəlcə bir deponu klonladıqdan dərhal sonra, depoda hələ heç bir fəaliyyət baş verm ədiyi üçün boş bir refloqa sahib olacaqsınız. `git show HEAD@{2.months.ago}`-ı işə salmaq, sizə yalnız ən azı iki ay əvvəl layihəni klonlaşdırdığınız təqdirdə uyğunlaşma commit-ini göstərəcəkdir - daha yaxınlarda klonlaşdırsanız, yalnız ilk local commit-i görəcəksiniz.

Reflogu Git-in shell tarixinin versiyası kimi düşünün

UNIX və ya Linux arxa planınız varsa, reflog-u Git-in shell tarixinin versiyası olaraq düşünə bilərsiniz, burada olanların yalnız sizin və sizin “sessiyanız” üçün açıq şəkildə əlaqəli olduğunu vurğulayan və eyni maşında işləyə başqa heç kimlə ilə əlaqəsi yoxdur.

Ancestry Referansları

Bir commit-i müəyyənləşdirməyin digər əsas yolu əcdadı ilə bağlıdır. Bir referansın sonunda bir `^` (caret) qoysanız, Git, bu commit-in valideynini ifadə etmək üçün onu təhlil edir. Tutaq ki, layih ənizin tarixinə nəzər yetirdiniz:

```
$ git log --pretty=format:'%h %s' --graph
* 734713b Fix refs handling, add gc auto, update tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd Add some blame and merge stuff
|/
* 1c36188 Ignore *.gem
* 9b29157 Add open3_detach to gemspec file list
```

Daha sonra, “the parent of HEAD” mənasını verən `HEAD^` göstərərək əvvəlki commit-i görə bil

ərsiniz:

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

Craet-i Windows-dan xilas etmək

Windows-da `cmd.exe`, `^` xüsusi bir xarakter daşıyır və fərqli davranılmalıdır. Ya ikiqat edə bilərsiniz, ya da commit arayışını quote-lara daxil edə bilərsiniz:

```
$ git show HEAD^     # will NOT work on Windows
$ git show HEAD^^    # OK
$ git show "HEAD^"   # OK
```

İstədiyiniz valideynin *hansı* olduğunu müəyyən etmək üçün `^`-dən sonra bir rəqəm də göstərə bil ərsiniz; məsələn, `d921970^2` "d921970-in ikinci valideynidir." deməkdir. Bu sintaksis yalnız birdən çox valideynə sahib olan birləşmə commit-ləri üçün faydalıdır - birləşdirmə commit-inin *birinci* valideyni birləşdikdə olduğunuz branch-dan (tez-tez `master`), birləşmə commit-nin *ikinci* valideyn hissəsi isə birləşdirilmiş branch-dan (`topic` deyək):

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    Add some blame and merge stuff

$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date:   Wed Dec 10 22:22:03 2008 +0000

    Some rdoc changes
```

Digər əsas əcdad spesifikasiyası `~` (tilde)-dir. Bu da birinci valideynə aiddir, buna görə `HEAD~` və `HEAD^` bərabərdir. Fərq bir rəqəm göstərdiyiniz zaman aydın olur. `HEAD~2`, "ilk valideynin birinci valideyni" və ya "nənə və baba" deməkdir - ilk valideynlərə göstərdiyiniz vaxt keçir. Məsələn, əvv əllər sadalanan tarixdə `HEAD~3`:

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    Ignore *.gem
```

Yenidən ilk valideynin ilk valideyninin ilk valideyni olan `HEAD~~~` yazıla bilər:

```
$ git show HEAD~~~
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    Ignore *.gem
```

Bu sintaksisləri də birləşdirə bilərsiniz - əvvəlcədən istinadın ikinci əsas hissəsini (birləşdirmə əmri olduğunu düşünərək) `HEAD~3^2` və s. istifadə edərək əldə edə bilərsiniz.

Commit Aralıqları

İndi fərdi commit-lər təyin edə bildiyinizə görə, commit-lərin hüdudlarını necə təyin edəcəyimizə baxaq. Bu, branch-larınızı idarə etmək üçün xüsusilə faydalıdır - çox sayda branch-ınız varsa, “Bu branch-da hələ əsas branch-a birləşdirmədiyim hansı iş var?” kimi suallara cavab vermək üçün spesifikasiyalardan istifadə edə bilərsiniz.

Cüt nöqtə

Ən geniş yayılmış spesifikasiya cüt nöqtəli sintaksisdir. Bu, əsasən Git-dən bir commit-dən əldə edilə bilən, digərinə çatmayan bir sıra commit-ləri həll etməsini xahiş edir. Məsələn, Aralıq seçimi üçün nümunə tarixçəsi kimi görünən commit tarixçəniz olduğunu söyləyin.

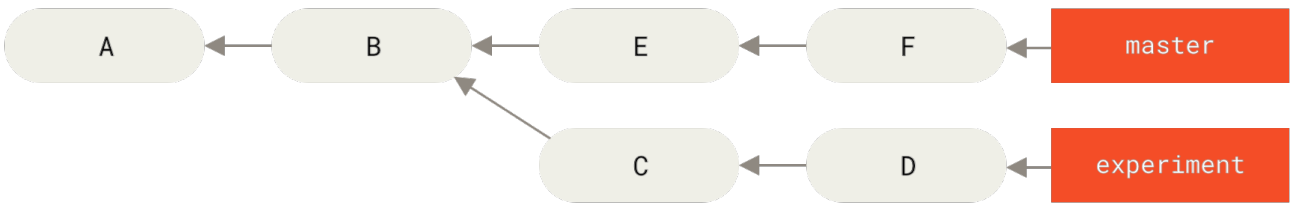


Figure 137. Aralıq seçimi üçün nümunə tarixçəsi

`experiment` branch-ınızda hələ `master` branch-ınıza birləşdirilməyənləri görmək istədiyinizi söyl əyin. Git-dən sizə yalnız `master..experiment` ilə işləyənlərin bir jurnalını göstərməsini xahiş edə bil ərsiniz - bu, ‘` `master`-dən əldə edilə bilməyən `experiment`-dən əldə edilə bilən bütün commit-lər" deməkdir. Bu nümunələrdə qısalıq və aydınlıq üçün, diaqramdakı commit obyektlərinin hərfləri göstərəcəkləri qaydada həqiqi log çıxışı yerinə istifadə olunur:

```
$ git log master..experiment
D
C
```

Digər tərəfdən bunun əksini görmək istəyirsinizsə - bütün commit-lər `experiment`-də olmayan `master`-də işləyirsə - branch adlarını tərsinə çevirə bilərsiniz. `experiment..master` sizə `experiment`-d ən əlçatmaz olan hər şeyi `master`-də göstərir:

```
$ git log experiment..master
F
E
```

Bu, `experiment` branch-ını yeniləmək və birləşdirmək istədiklərinizi önizləmək istəsəniz faydalıdır. Bu sintaksisin başqa bir tez-tez istifadəsi uzaq məsafəyə nəyi push edəcəyinizi görməkdir:

```
$ git log origin/master..HEAD
```

Bu əmr sizə cari branch-ınızdakı `origin` remote-dakı `master` branch-ında olmayan hər hansı bir commit-i göstərir. Bir `git push` işə salırsınızsa və mövcud branch-ınız `origin/master` izləyirsə, `git log origin/master..HEAD` tərəfindən sadalanan commit-lər serverə ötürülən commit-lərdir. Git-in `HEAD` olduğunu qəbul etməsi üçün sintaksisin bir tərəfini də tərk edə bilərsiniz. Məsələn, `git log origin/master..` yazaraq əvvəlki nümunədəki ilə eyni nəticələr əldə edə bilərsiniz - bir tərəfi yoxdursa, `HEAD` əvəzlə.

Birdən Çox Pal

İkili nöqtəli sintaksis shorthand kimi faydalıdır, lakin bəlkə də hazırda olduğunuz branch-da olmayan bir neçə branch-dan birinin nə olduğunu görmək kimi düzəlişlərinizi göstərmək üçün ikidən çox branch göstərmək istəyirsiniz. Git, əlçatan commit-lər görmək istəmədiyiniz hər hansı bir istinaddan əvvəl `^` simvolunu və ya `--not` istifadə edərək bunu etməyə imkan verir. Beləliklə, aşağıdakı üç əmr bərabərdir:

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

Bu çox yaxşıdır, çünki bu sintaksislə sorğunuzda ikiqat nöqtəli sintaksis ilə edə bilməyəcəyiniz ikidən çox istinad daxil edə bilərsiniz. Məsələn, `refA` ya da `refB`-dən əldə edilə bilən, ancaq `refC`-dən edilməyən, bütün commit-ləri görmək istəyirsinizsə, aşağıdakılardan birini istifadə edə bilərsiniz:

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

Bu, branch-larınızda nə olduğunu anlamanıza kömək edəcək çox güclü bir revizyon sorğu sistemi

yaradır.

Üçqat Nöqtə

Son böyük aralıq seçmə sintaksisi, hər ikisindən deyil, iki istinadın hər ikisi tərəfindən də əldə edilə bilən bütün commit-ləri təyin edən üç nöqtəli sintaksisdir. Aralıq seçimi üçün nümunə tarixçəsi -dakı commit tarixçəsinə baxın. `master` və ya `experiment`-də olanları görmək istəsəniz, lakin ümumi istinadları yox görmək istəməsəniz, işlədə bilərsiniz:

```
$ git log master...experiment
F
E
D
C
```

Yenə də, bu sizə normal bir `log` çıxışı verir, ancaq ənənəvi commit tarixi sifarişində görünən yalnız bu dörd commit üçün commit məlumatlarını göstərir.

Bu halda `log` əmri ilə istifadə olunan ümumi bir keçid, hər bir commit aralığın hansı tərəfində olduğunu göstərən `--left-right`-dır. Bu, nəticənin daha faydalı olmasına kömək edir:

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

Bu vasitələrlə Git-ə nəyi yoxlamaq istədiyinizi və ya commit-lərinizi daha asanlıqla bildirə bil ərsiniz.

Interaktiv Səhnələşdirmə

Bu bölmədə, yalnız müəyyən kombinasiyalar və faylların hissələrini daxil etməyi commit etməyə kömək edə biləcək bir neçə interaktiv Git əmrlərinə baxacaqsınız. Bu alətlər bir sıra faylları geniş şəkildə dəyişdirsəniz, o dəyişikliklərin bir böyük qarışıqlıq yerinə daha çox diqqət mərkəzinə düşməsini istəməyinizə qərar verməyiniz daha yararlıdır. Bu sayədə verdiyiniz tapşırıqların m əntiqi olaraq ayrı bir dəyişiklik olduğunu və sizinlə işləyən tərtibatçılar tərəfindən asanlıqla nəz ərdən keçiriləcəyinə əmin ola bilərsiniz.

`-i` və ya `--interactive` seçimi ilə `git add` tətbiq etsəniz, Git bu kimi bir şey göstərərək interaktiv shell moda girir:

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb

*** Commands ***
  1: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now>
```

Görə bilərsiniz ki, bu əmr sizə yayımlandığınız ərazidən daha çox fərqli bir mənzərəni göstərir - əsasən, `git status` ilə əldə etdiyiniz eyni, lakin bir az daha qısa və məlumatlıdır. Solda hazırladığınız dəyişiklikləri və sağdakı sabit olmayan dəyişiklikləri sadalayır.

Bundan sonra sənədlərin qurulması və ləğv edilməsi, faylların hissələrinin qurulması, yığılmamış sənədlərin əlavə edilməsi və hazırlanmışların fərqlərini göstərmək kimi bir sıra işləri görməyə imkan verən bir "Commands" bölməsi gəlir.

Staging və Unstaging Fayllar

`What now>` yerində `u` və ya `2` (yeniləmə üçün) yazsanız, hansı sənədləri səhnələşdirmək istədiyinizi soruşursunuz:

```
What now> u
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Update>>
```

`TODO` və `index.html` fayllarını səhnələşdirmək üçün nömrələri yaza bilərsiniz:

```
Update>> 1,2
           staged     unstaged path
* 1:    unchanged        +0/-1 TODO
* 2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Update>>
```

Hər bir sənədin yanında olan `*` sənədin səhnəyə qoyulması üçün seçildiyini bildirir. `Update>>` sorğusuna heç bir şey yazmadan sonra Enter düyməsini bassanız, Git seçilmiş bir şeyi götürür və sizin üçün mərhələləndirir:

```
Update>>
updated 2 paths

*** Commands ***
  1: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now> s
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

İndi `TODO` və `index.html` fayllarının səhnəyə qoyulduğunu və `simplegit.rb` faylının hələ də işləm ədiyini görə bilərsiniz. Bu anda `TODO` faylını çıxartmaq istəyirsinizsə, `r` və ya `3` (geri dönüş üçün) seçimini istifadə edirsiniz:

```
*** Commands ***
  1: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now> r
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Revert>> 1
           staged     unstaged path
* 1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Revert>> [enter]
reverted one path
```

Git statusunuza yenidən baxaraq `TODO` faylını açmadığınızı görə bilərsiniz:

```
*** Commands ***
  1: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now> s
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

Hazırladığınız şeyin fərqini görmək üçün `d` və ya `6` (fərq üçün) əmrindən istifadə edə bilərsiniz. Bu əhnələşdirilmiş sənədlərinizin siyahısını göstərir və mərhələli fərqi görmək istədiklərini seçə bil ərsiniz. Bu, əmr sətrində `git diff --cached` göstərməyə çox bənzəyir:

```
*** Commands ***
  1: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now> d
           staged     unstaged path
  1:        +1/-1      nothing index.html
Review diff>> 1
diff --git a/index.html b/index.html
index 4d07108..4335f49 100644
--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

 <p id="out">...</p>

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

 <script type="text/javascript">
```

Bu əsas əmrlərlə quruluş sahənizi bir az daha asan həll etmək üçün interaktiv əlavə rejimi istifadə edə bilərsiniz.

Staging Patch-lar

Git'in müəyyən *hissə* sənədlərini səhnələşdirməsi də mümkündür qalanlarının deyil. Məsələn, `simplegit.rb` faylınıza iki dəyişiklik etsəniz və onlardan birini digərinə deyil, digərinə düzəltmək istəyirsinizsə, Git-də bunu etmək çox asandır. Əvvəlki hissədə izah edilən eyni interaktiv istəkdən, `p` və ya `5` (patch üçün) yazın. Git sizə hansı sənədləri qismən səhnələşdirmək istədiyinizi soruşacaq; sonra, seçilmiş faylların hər bölməsi üçün fərqlənən faylı göstərəcək və bunları bir-bir düzəltmək istəyəcəyinizi soruşacaq:

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log -n 25 #{treeish}")
+    command("git log -n 30 #{treeish}")
   end

   def blame(path)
Stage this hunk [y,n,a,d,/,j,J,g,e,?]?
```

Bu nöqtədə bir çox seçiminiz var. Yazaraq `?` edə biləcəyiniz işlərin siyahısını görə bilərsiniz:

```
Stage this hunk [y,n,a,d,/,j,J,g,e,?]? ?
y - stage this hunk
n - do not stage this hunk
a - stage this and all the remaining hunks in the file
d - do not stage this hunk nor any of the remaining hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

Ümumiyyətlə,Hər bir parçanı səhnələşdirmək istəsəniz,`y` və ya `n` yazacaqsınız, ancaq hamısını mü əyyən fayllarda yerləşdirmək və ya bir qədər sonra bir qərarı atmaq da faydalı ola bilər. Faylın bir hissəsini düzəldib başqa bir hissəsini maneəsiz qoyarsanız, status çıxışınız belə olacaq:

```
What now> 1
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:        +1/-1      nothing index.html
  3:        +1/-1        +4/-0 lib/simplegit.rb
```

`Simplegit.rb` faylının vəziyyəti maraqlıdır. Bu, bir neçə sətin səhnələndiyini və bir neçə sətrin sə hnələnmədiyini göstərir. Bu faylı qismən səhnələşdirdiniz. Bu anda, interaktiv əlavə skriptdən çıxa və qismən mərhələli faylları düzəltmək üçün `git commit` işlədə bilərsiniz.

Qismən fayl quruluşunu etmək üçün interaktiv əlavə rejimində olmağınıza ehtiyac yoxdur - əmr s ətrində `git add -p` və ya `git add --patch` istifadə edərək eyni skriptə başlaya bilərsiniz.

Bundan əlavə, `git checkout --patch` əmri ilə faylların hissələrini yoxlamaq üçün və`git stash save --patch` ilə faylların hissələrini zədələmək üçün `git reset --patch` əmri ilə faylları qismən sıfırlamaq üçün istifadə edə bilərsiniz. Bu əmrlərin daha qabaqcıl istifadəsinə keçdikcə bunların h ər biri haqqında daha ətraflı məlumat əldə edəcəyik.

Stashing və Təmizləmə

Tez-tez, layihənizin bir hissəsi üzərində işlədiyiniz zaman işlər qarışıq vəziyyətdədir və başqa bir şey üzərində işləmək üçün branch-ları biraz dəyişdirmək istəyirsiniz. Məsələ burasındadır ki, bu məqama daha sonra qayıtmaq üçün yarımçıq iş görmək istəmirsiniz. Bu məsələnin cavabı `git stash` əmridir.

Stashing iş qovluğunuzun çirkli vəziyyətini alır - yəni dəyişdirilmiş izlənilmiş sənədlərinizi və m ərhələli dəyişikliklərinizi - və istənilən vaxt yenidən tətbiq edə biləcəyiniz tamamlanmamış d əyişikliklər yığınında saxlayır (fərqli bir branch-da da).

`git stash push`-ə *köçmək*

2017-ci il Oktyabr ayının sonundan etibarən Git poçt siyahısında geniş şəkildə müzakirələr aparıldı, burada `git stash save` əmri mövcud alternativ `git stash push` yerinə ləğv edildi. Bunun əsas səbəbi, `git stash push` seçilmiş *pathspecs*-ni saxlama seçimini təqdim etməsidir, `git stash save`-i dəstəkləmir.

`git stash save` tezliklə getməyəcək, buna görə birdən yox olmasından narahat olmayın. Ancaq yeni funksionallıq üçün `push` alternativinə keçməyə başlamaq ist əyə bilərsiniz.

İşinizi Stashing Etmək

Stashing-i nümayiş etdirmək üçün layihənizə girib bir neçə sənəd üzərində işləməyə başlayacaqsınız və ola bilsin dəyişikliklərdən birini səhnələşdirəsiniz. `git status` işlədirsinizsə, çirkli vəziyyətinizi görə bilərsiniz:

```
$ git status
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   lib/simplegit.rb
```

İndi branch-ları dəyişdirmək istəyirsiniz, amma hələ işlədiyin şeyi etmək istəmirsiniz, buna görə də dəyişiklikləri gizlədəcəksiniz. Yeni bir yığını yığınınıza göndərmək üçün `git stash` və ya `git stash push` əmrlərini işlədin:

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 Create index file"
HEAD is now at 049d078 Create index file
(To restore them type "git stash apply")
```

İndi iş qovluğunuzun təmiz olduğunu görə bilərsiniz:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

Bu nöqtədə branch-ları dəyişə və başqa yerdə işləyə bilərsiniz; dəyişiklikləriniz stack-da saxlanılır. Hansı zibil saxladığınızı görmək üçün `git stash list`-dən istifadə edə bilərsiniz:

```
$ git stash list
stash@{0}: WIP on master: 049d078 Create index file
stash@{1}: WIP on master: c264051 Revert "Add file_size"
stash@{2}: WIP on master: 21d80a5 Add number to log
```

Bu vəziyyətdə, əvvəl iki stash qeyd olundu, beləliklə üç fərqli stash işinə giriş əldə edə bilərsiniz. Orijinal stash əmrinin kömək çıxışında göstərilən əmrdən istifadə edərək təzə saxladığınızı yenid ən tətbiq edə bilərsiniz: `git stash apply`.

Daha köhnə işarələrdən birini tətbiq etmək istəyirsinizsə, onu belə adlandıraraq təyin edə bil ərsiniz:`git stash apply stash@{2}`. Bir saxlama yeri təyin etməsəniz, Git ən son stash-ı götürür və t ətbiq etməyə çalışır:

```
$ git stash apply
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   index.html
    modified:   lib/simplegit.rb

no changes added to commit (use "git add" and/or "git commit -a")
```

Git'in stash-ı saxladığınız zaman geri qaytardığınız faylları yenidən dəyişdirdiyini görə bilərsiniz. Bu vəziyyətdə, stash-ı tətbiq etməyə çalışarkən təmiz bir iş qovluğuna sahib oldunuz və onu saxladığınız eyni branch-a tətbiq etməyə çalışdınız. Təmiz bir iş qovluğuna sahib olmaq və eyni branch-da tətbiq etmək, bir stash-a müvəffəqiyyətlə tətbiq etmək üçün lazım deyil. Bir branch-da bir stash saxlaya bilərsiniz, daha sonra başqa bir branch-a keçə və dəyişiklikləri yenidən tətbiq etməyə çalışa bilərsiniz. Bir stash tətbiq edərkən iş qovluğunuzda dəyişdirilmiş və sənədləşdirilm əmiş fayllara da sahib ola bilərsiniz - Git bir şeyi artıq təmiz bir şəkildə tətbiq etmirsə, konfliktləri birləşdirəcəkdir.

Fayllarınızdakı dəyişikliklər yenidən tətbiq edildi, lakin əvvəllər səhnələşdirdiyiniz fayl yenidən b ərpa olunmadı. Bunu etmək üçün mərhələli dəyişiklikləri yenidən tətbiq etməyini demək üçün `--index` seçimi ilə `git stash apply` əmrini çalıştırmalısınız. Bunun əvəzinə işləsəniz, əvvəlki vəziyy ətinizə qayıdardınız:

```
$ git stash apply --index
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   lib/simplegit.rb
```

Tətbiq etmə seçimi yalnız stash edilmiş işi tətbiq etməyə çalışır - onu stack-nızda saxlamağa davam edirsiniz. Silmək üçün stash-ın adı ilə `git stash drop`-u işlədə bilərsiniz:

```
$ git stash list
stash@{0}: WIP on master: 049d078 Create index file
stash@{1}: WIP on master: c264051 Revert "Add file_size"
stash@{2}: WIP on master: 21d80a5 Add number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

Stash-ı tətbiq etmək üçün `git stash pop`-ı da işə sala və dərhal stack-a ata bilərsiniz.

Kreativ Stashing

Faydalı ola biləcək bir neçə stash variantı var. Kifayət qədər populyar olan ilk seçim, `git stash` əmrinə `--keep-index` seçimidir. Bu, Git-dən yalnız hazırlanmış bütün məzmunu yaratmaq üçün deyil, həm də indeksləşdirməsini tələb edir.

```
$ git status -s
M  index.html
 M lib/simplegit.rb

$ git stash --keep-index
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
M  index.html
```

Stash ilə etmək istəyə biləcəyiniz başqa bir ümumi şey izlənilməmiş faylların yanında izlənilməmiş sənədləri də zibilə salmaqdır. Varsayılan olaraq, `git stash` yalnız dəyişdirilmiş və səhnələşdirilmiş *tracked* fayllarını saxlayır. Əgər `--include-untracked` və ya `-u` qeyd etsəniz, Git, yaradılan depoda izlənməmiş faylları əlavə edəcəkdir. Bununla birlikdə, izlənməmiş faylları stash-a daxil etmək hələ açıq şəkildə *ignored* fayllarını daxil etməyəcək; əlavə olaraq məhəl qoyulmayan sənədləri daxil

etmək üçün `--all` (və ya sadəcə `-a`) istifadə edin.

```
$ git status -s
M  index.html
 M lib/simplegit.rb
?? new-file.txt

$ git stash -u
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
$
```

Nəhayət, `--patch` flag-nı təyin etsəniz, Git dəyişdirilmiş hər şeyi stash etmiyəcək, əksinə dəyişiklə rdən hansını saxlamağınızı və iş qovluğunuzda saxlamaq istədiyinizi interaktiv şəkildə təklif edəc əkdir.

```
$ git stash --patch
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 66d332e..8bb5674 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -16,6 +16,10 @@ class SimpleGit
         return `#{git_cmd} 2>&1`.chomp
       end
     end
+
+    def show(treeish = 'master')
+      command("git show #{treeish}")
+    end

 end
 test
Stash this hunk [y,n,q,a,d,/,e,?]? y

Saved working directory and index state WIP on master: 1b65b17 added the index file
```

Stash-dan bir Branch Yaratmaq

İşin bir hissəsini saxlasanız, bir müddət orada qoyun və işi saxladığınız branch-da davam edin, işi yenidən tətbiq etməkdə çətinlik çəkə bilərsiniz.

Tətbiq dəyişdirdiyiniz bir faylı dəyişdirməyə çalışırsa, birləşmə konflikti yaranacaq və onu həll etməyə çalışmalı olacaqsınız. Gizlənmiş dəyişiklikləri yenidən sınamaq üçün daha asan bir yol ist əyirsinizsə, seçdiyiniz branch adı ilə sizin üçün yeni bir branch yaradan `git stash branch <new branchname>` düyməsini işə sala bilərsiniz. İşləyin, işinizi orada yenidən tətbiq edin və sonra müvə ffəqiyyətlə tətbiq olunarsa saxlayın:

```
$ git stash branch testchanges
M   index.html
M   lib/simplegit.rb
Switched to a new branch 'testchanges'
On branch testchanges
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   lib/simplegit.rb

Dropped refs/stash@{0} (29d385a81d163dfd45a452a2ce816487a6b8b014)
```

Bu stashed işi asanlıqla bərpa etmək və yeni bir branch-da işləmək üçün gözəl bir qısayoldur.

İş Qovluğunuzun Təmizlənməsi

Nəhayət, işinizdəki bəzi işləri və ya faylları saxlamaq istəməyəcəksiniz, o zaman sadəcə bunlardan qurtulun; `git clean` əmri bunun üçündür.

İş qovluğunuzu təmizləməyin bəzi ümumi səbəbləri birləşmə və ya xarici alətlər nəticəsində əmələ gələn qırıntıları aradan qaldırmaq və ya təmiz bir quruluşu idarə etmək üçün tikinti əsərlərini silmək ola bilər.

Bu əmrlə olduqca diqqətli olmaq istərdiniz, çünki işləmə qovluğundan izlənilməyən faylları silmək üçün hazırlanmışdır. Fikrinizi dəyişirsinizsə, bu faylların məzmununun alınmasına çox vaxt rast gəlinmir. Daha etibarlı bir seçim, hər şeyi aradan qaldırmaq, ancaq bir yerə yığmaq üçün `git stash --all` işlətməkdir.

Cruft fayllarını silmək və ya iş qovluğunuzu təmizləmək istədiyinizi düşünsək, bunu `git clean` ilə edə bilərsiniz. İşləmə qovluğundakı bütün izlənilməmiş faylları silmək üçün, bütün faylları və nə ticədə boş qalan subdirectory-ləri silən `git clean -f -d` düyməsini işə sala bilərsiniz. `-f`, *force* və ya "həqiqətən bunu et" deməkdir və Git konfiqurasiya dəyişəninin `clean.requireForce` açıq şəkildə false olaraq ayarlanmadığı təqdirdə tələb olunur.

Nə edəcəyini görmək istəyirsinizsə, əmri `--dry-run` (və ya `-n`) seçimi ilə işlədə bilərsiniz, yəni "quru işləyin və nəyi çıxardığınızı mənə deyin.".

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

Varsayılan olaraq, `git clean` əmri yalnız göz ardı edilməyən yoxlanılmamış faylları siləcəkdir.

`.gitignore`-dakı bir nümunə ilə uyğun gələn hər hansı bir fayl silinməyəcək. Bu faylları da silmək istəyirsinizsə, məsələn, bir yığımdan yaradılan bütün `.o` fayllarını silmək istəyirsinizsə, beləliklə tamamilə təmiz bir yığın yarada bilərsiniz, təmiz əmrinə `-x` əlavə edə bilərsiniz.

```
$ git status -s
 M lib/simplegit.rb
?? build.TMP
?? tmp/

$ git clean -n -d
Would remove build.TMP
Would remove tmp/

$ git clean -n -d -x
Would remove build.TMP
Would remove test.o
Would remove tmp/
```

`git clean` əmrinin nə ediləcəyini bilmirsinizsə, `-n`-i `-f`-ə dəyişdirmədən və real olaraq yerinə yetirmədən əvvəl həmişə bir `-n` ilə işləyin. Prosesə diqqətli olmağınızın başqa bir yolu, `-i` or "interactive" flag-la işlətməkdir.

Bu, təmiz əmri interaktiv bir rejimdə işlədəcəkdir.

```
$ git clean -x -i
Would remove the following items:
  build.TMP  test.o
*** Commands ***
    1: clean                2: filter by pattern    3: select by numbers    4: ask
each             5: quit
    6: help
What now>
```

Bu şəkildə hər bir faylı ayrı-ayrılıqda nəzərdən keçirə və ya interaktiv şəkildə silmək üçün nümun ələri təyin edə bilərsiniz.

Git-dən iş qovluğunuzu təmizləməsini istəməyinizdə daha güclü olmanız lazım ola biləcəyi qəribə bir vəziyyət var. Digər Git depolarını (bəlkə də submodul kimi) kopyaladığınız və ya klonladığınız bir iş qovluğundasınızsa, hətta `git clean -fd` bu qovluqları silməkdən imtina edəcəkdir. Belə hallarda vurğu üçün ikinci bir `-f` seçimi əlavə etməlisiniz.

İşinizin İmzalanması

Git kriptoqrafik cəhətdən etibarlıdır, lakin axmaq deyil. İnternette başqalarından iş götürürsənsə və commit-lərin həqiqətən etibarlı bir mənbədən olduğunu yoxlamaq istəyirsənsə, Gitin GPG istifadə edərək işi imzalamaq və yoxlamaq üçün bir neçə yolu var.

GPG Girişi

Hər şeydən əvvəl, hər hansı bir şeyi imzalamaq istəyirsinizsə, GPG-nin konfiqurasiya edilməsi və ş əxsi açarınızın quraşdırmağınız lazımdır.

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
---------------------------------
pub   2048R/0A46826A 2014-06-04
uid                  Scott Chacon (Git signing key) <schacon@gmail.com>
sub   2048R/874529A9 2014-06-04
```

If you don't have a key installed, you can generate one with `gpg --gen-key`.

```
$ gpg --gen-key
```

İmzalanacaq xüsusi bir açarınız olduqdan sonra `user.signingkey` konfiqurasiya parametrini təyin edərək Git'i birşeyləri imzalamağa istifadə etmək üçün konfiqurasiya edə bilərsiniz.

```
$ git config --global user.signingkey 0A46826A
```

İndi Git etiketi imzalamaq üçün default olaraq açarınızı istifadə edəcək və istəsəniz commit götürə cəksiniz.

Etiketlər İmzalamaq

GPG xüsusi açar quraşdırmanız varsa, indi yeni etiketlər imzalamaq üçün istifadə edə bilərsiniz. T ək etməli olduğunuz `-a` əvəzinə `-s` istifadə etməkdi:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'

You need a passphrase to unlock the secret key for
user: "Ben Straub <ben@straub.cc>"
2048-bit RSA key, ID 800430EB, created 2014-05-04
```

Bu etiketdə `git show` işlətsəniz, ona əlavə edilmiş GPG imzanızı görə bilərsiniz:

```
$ git show v1.5
tag v1.5
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:29:41 2014 -0700

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcBAABAgAGBQJTZbQlAAoJEF0+sviABDDrZbQH/09PfE51KPVPlanr6q1v4/Ut
LQxfojUWiLQdg2ESJItkcuweYg+kc3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b
hM1/PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLoWZRF8B0MfqX/YTMbm
ecorc4iXzQu7tupRihslbNkfvfciMnSDeSvzCpWAHl7h8Wj6hhqePmLm9lAYqnKp
8S5B/1SSQuEAjRZgI4IexpZoeKGVDptPHxLLS38fozsyi0QyDyzEgJxcJQVMXxVi
RUysgqjcpT8+iQM1PblGfHR4XAhuOqN5Fx06PSaFZhqvWFezJ28/CLyX5q+oIVk=
=EFTF
-----END PGP SIGNATURE-----

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number
```

Etiketləri Doğrulamaq

İmzalanmış etiketi doğrulamaq üçün `git tag -v <tag-name>` istifadə edin. Bu əmr imzanı təsdiqlə mək üçün GPG istifadə edir. Bunun düzgün işləməsi üçün açar zəncirinizdə imzanın public açarına ehtiyacınız var:

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg:                 aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7  4A7D C0C6 D9A4 F311 9B9A
```

İmzalananın public açarı yoxdursa, bunun əvəzinə belə bir şey əldə edəcəksiniz:

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

Commit-ləri İmzalamaq

Git-in daha yeni versiyalarında (v1.7.9 və yuxarı), indi ayrıca fərdi commit-lər də imzalaya bil ərsiniz. Yalnız etiketlər əvəzinə birbaşa imzalamaqla maraqlanırsınızsa, `git commit` əmrinizə bir `-S` əlavə edin.

```
$ git commit -a -S -m 'Signed commit'

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

[master 5c3386c] Signed commit
 4 files changed, 4 insertions(+), 24 deletions(-)
 rewrite Rakefile (100%)
 create mode 100644 lib/git.rb
```

Bu imzaları görmək və yoxlamaq üçün `git log` üçün bir `--show-signature` seçimi var.

```
$ git log --show-signature -1
commit 5c3386cf54bba0a33a32da706aa52bc0155503c2
gpg: Signature made Wed Jun  4 19:49:17 2014 PDT using RSA key ID 0A46826A
gpg: Good signature from "Scott Chacon (Git signing key) <schacon@gmail.com>"
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Jun 4 19:49:17 2014 -0700

    Signed commit
```

Əlavə olaraq tapdığı hər hansı bir imzanı yoxlamaq və onları `%G?` formatı ilə çıxışı siyahısına salmaq üçün `git log`-ı konfiqurasiya edə bilərsiniz.

```
$ git log --pretty="format:%h %G? %aN  %s"

5c3386c G Scott Chacon  Signed commit
ca82a6d N Scott Chacon  Change the version number
085bb3b N Scott Chacon  Remove unnecessary test code
a11bef0 N Scott Chacon  Initial commit
```

Burada yalnız son commit-in imzalanmış və etibarlı olduğunu və əvvəlki commit-lərin olmadığını görərik.

Git 1.8.3 və sonrakı versiyalarında, etibarlı bir GPG imzasını daşımayan bir commit-i `--verify`

`-signatures` əmri ilə birləşdirərkən yoxlamaq və rədd etmək üçün `git merge` və `git pull`-a işlətmək olar.

Bir branch-ı birləşdirərkən bu seçimdən istifadə edirsinizsə və imzalanmamış və etibarlı olmayan commit-lərdən ibarətdirsə, birləşmə nəticə verməyəcəkdir.

```
$ git merge --verify-signatures non-verify
fatal: Commit ab06180 does not have a GPG signature.
```

Birləşmə yalnız etibarlı imzalanmış commit-lərdən ibarətdirsə, birləşdirmə əmri sizə yoxladığı bütün imzaları göstərəcək və sonra birləşmə ilə irəliləyəcəkdir.

```
$ git merge --verify-signatures signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key)
<schacon@gmail.com>
Updating 5c3386c..13ad65e
Fast-forward
 README | 2 ++
 1 file changed, 2 insertions(+)
```

Nəticədə birləşmə commit-ini imzalamaq üçün `git merge` əmri ilə `-S` seçimindən də istifadə edə bil ərsiniz. Aşağıdakı nümunə həm birləşdiriləcək branch-dakı hər bir commit-in imzalandığını tə sdiqləyir, həm də nəticədə birləşmə commit-ini imzalayır.

```
$ git merge --verify-signatures -S  signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key)
<schacon@gmail.com>

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

Merge made by the 'recursive' strategy.
 README | 2 ++
 1 file changed, 2 insertions(+)
```

Hamı İmzalaya Bilər

Etiketlər və commit-lər imzalamaq çox yaxşıdır, ancaq bunu normal iş axınınızda istifadə etməyə qərar verərsinizsə, komandanızdakı hər kəsin bunu necə edəcəyini başa düşməyiniz lazımdır. Bunu etməsəniz, işlərini imzalanmış versiyalarla necə yenidən yazacaqlarına kömək etmək üçün çox vaxt sərf edəcəksiniz. Bunu standart iş axınınızın bir hissəsi kimi qəbul etməzdən əvvəl GPG-ni və şeyləri imzalamağın üstünlüklərini başa düşdüyünüzdən əmin olun.

Axtarış

Təxminən hər hansı bir ölçülü kod bazası ilə tez-tez bir funksiyanın çağırıldığı və ya təyin olunduğu yeri tapmaq və ya bir metodun tarixini göstərmək lazımdır. Git kodu araşdırmaq üçün bir neçə faydalı vasitə təqdim edir və verilənlər bazasında tez və asanlıqla saxlanılır. Onlardan bir neçəsindən keçəcəyik.

Git Grep

Hər hansı bir düzəldilmiş ağacdan, işçi qovluqdan və ya hətta bir simli və ya adi bir ifadə üçün asanlıqla axtarış etməyə imkan verən `grep` adlı bir əmr ilə birlikdı gəlir. Sonrakı nümunələr üçün Git mənbə koduna baxacağıq.

Varsayılan olaraq, `git grep` işçi qovluğunuzdakı fayllara baxacaqdır. Birinci variasiya olaraq, Git'in uyğunlaşmaları tapdığı sətir nömrələrini çap etmək üçün `-n` və ya `--line-number` seçimlərindən birini istifadə edə bilərsiniz:

```
$ git grep -n gmtime_r
compat/gmtime.c:3:#undef gmtime_r
compat/gmtime.c:8:      return git_gmtime_r(timep, &result);
compat/gmtime.c:11:struct tm *git_gmtime_r(const time_t *timep, struct tm *result)
compat/gmtime.c:16:     ret = gmtime_r(timep, result);
compat/mingw.c:826:struct tm *gmtime_r(const time_t *timep, struct tm *result)
compat/mingw.h:206:struct tm *gmtime_r(const time_t *timep, struct tm *result);
date.c:482:             if (gmtime_r(&now, &now_tm))
date.c:545:             if (gmtime_r(&time, tm)) {
date.c:758:             /* gmtime_r() in match_digit() may have clobbered it */
git-compat-util.h:1138:struct tm *git_gmtime_r(const time_t *, struct tm *);
git-compat-util.h:1140:#define gmtime_r git_gmtime_r
```

Yuxarıda göstərilən əsas axtarışa əlavə olaraq, `git grep` bir çox digər maraqlı variantları dəstəkl əyir.

Məsələn, bütün uyğunlaşmaları çap etmək əvəzinə axtarış sətirində hansı sənədlərin olduğunu və hər bir faylda `-c` və ya `--count` seçiminindən istifadə edərək neçə uyğunlaşmanın olduğunu göstər ən `git grep`-dən nəticəni ümumiləşdirməsini istiyə bilərsiniz.

```
$ git grep --count gmtime_r
compat/gmtime.c:4
compat/mingw.c:1
compat/mingw.h:1
date.c:3
git-compat-util.h:2
```

Bir axtarış sətirinin *konteksti* ilə maraqlanırsınızsa, hər bir uyğun sətir üçün enclosing metodunu və ya funksiyasını `-p` və ya `--show-function` seçimlərindən birini göstərə bilərsiniz:

```
$ git grep -p gmtime_r *.c
date.c=static int match_multi_number(timestamp_t num, char c, const char *date,
date.c:         if (gmtime_r(&now, &now_tm))
date.c=static int match_digit(const char *date, struct tm *tm, int *offset, int
*tm_gmt)
date.c:         if (gmtime_r(&time, tm)) {
date.c=int parse_date_basic(const char *date, timestamp_t *timestamp, int *offset)
date.c:         /* gmtime_r() in match_digit() may have clobbered it */
```

Gördüyünüz kimi, `gmtime_r` rutin `date.c` faylındakı `match_multi_number` və `match_digit` funksiyalarını çağırır (üçüncü uyğunlaşma yalnız şərhdə görünən sətri təmsil edir).

Ayrıca mətnin eyni sətirində birdən çox uyöunlaşmanın meydana gəlməsini təmin edən `--and` flag-ı ilə kompleks birləşmələri axtara bilərsiniz.

Məsələn, adı “LINK” və ya “BUF_MAX” alt hissələrindən hər hansı bir sabit olanı müəyyənləşdirən hər hansı bir sətri, özəlliklə də `v1.8.0` etiketi ilə təmsil edilən Git kod bazasında köhnə versiyasında axtaraq. (nəticənin daha oxunaqlı olması üçün kömək edən `--break` və `--heading` seçimlərindən istifadə edəcəyik)

```
$ git grep --break --heading \
    -n -e '#define' --and \( -e LINK -e BUF_MAX \) v1.8.0
v1.8.0:builtin/index-pack.c
62:#define FLAG_LINK (1u<<20)

v1.8.0:cache.h
73:#define S_IFGITLINK  0160000
74:#define S_ISGITLINK(m)       (((m) & S_IFMT) == S_IFGITLINK)

v1.8.0:environment.c
54:#define OBJECT_CREATION_MODE OBJECT_CREATION_USES_HARDLINKS

v1.8.0:strbuf.c
326:#define STRBUF_MAXLINK (2*PATH_MAX)

v1.8.0:symlinks.c
53:#define FL_SYMLINK  (1 << 2)

v1.8.0:zlib.c
30:/* #define ZLIB_BUF_MAX ((uInt)-1) */
31:#define ZLIB_BUF_MAX ((uInt) 1024 * 1024 * 1024) /* 1GB */
```

`git grep` əmrinin `grep` və `ack` kimi normal axtarış əmrləri ilə müqayisədə bir sıra üstünlüklərə malikdir. Birincisi, həqiqətən sürətlidir, ikincisi, yalnız işləyən qovluğda deyil, Git-də hər hansı bir ağacdan axtarış edə bilərsiniz. Yuxarıdakı nümunədə gördüyümüz kimi hazırda yoxlanılan versiyada deyil, Git mənbə kodunun köhnə bir versiyasında terminlər axtara bilirik.

Git Log Axtarışı

Bəlkə də bir terminin *harda* mövcud olduğunu deyil, *nə zaman* mövcud olduğunu və tətbiq edildiyini axtarırsınız. `git log` əmrində mesajlarının məzmununa və ya təklif etdikləri fərqə əsaslanaraq müəyyən hərəkətləri tapmaq üçün bir sıra güclü vasitələr var.

Məsələn, `ZLIB_BUF_MAX` sabitinin nə vaxt tətbiq olunduğunu bilmək istəyiriksə, Git-in yalnız bizə göstərməsini söyləmək üçün `-S` seçimini istifadə edə bilərik.

```
$ git log -S ZLIB_BUF_MAX --oneline
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time
```

Verilənlərin fərqinə baxsaq, görərik ki, `ef49a7a`-da sabitlik qoyulmuşdur və `e01503b` də d əyişdirilmişdir.

Daha konkret olmağınız lazımdırsa, `-G` seçimi ilə axtarış aparmaq üçün adi bir ifadə verə bilərsiniz.

Line Log Axtarışı

Olduqca faydalı olan daha bir inkişaf etmiş axtarış xətti tarixidir. Sadəcə `-L` seçimi ilə `git log`-u işl ədin və bu kod bazasında bir funksiya və ya kod xətti tarixini göstərəcəkdir.

Məsələn, `zlib.c` faylında `git_deflate_bound` funksiyasına edilən hər bir dəyişikliyi görmək istəsək, `git log -L :git_deflate_bound:zlib.c` işlədə bilərik. Bu funksiyanın hüdudlarının nə olduğunu anlamağa çalışacaq və sonra tarixə nəzər salacaq və funksiyaya edilən dəyişikliklərin hamısını funksiyanın ilk yaradıldığı vaxtdan geri qalmış bir sıra patch-lar şəklində göstərəcəkdir.

```
$ git log -L :git_deflate_bound:zlib.c
commit ef49a7a0126d64359c974b4b3b71d7ad42ee3bca
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:52:15 2011 -0700

    zlib: zlib can only process 4GB at a time

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -85,5 +130,5 @@
-unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+unsigned long git_deflate_bound(git_zstream *strm, unsigned long size)
 {
-       return deflateBound(strm, size);
+       return deflateBound(&strm->z, size);
 }


commit 225a6f1068f71723a910e8565db4e252b3ca21fa
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:18:17 2011 -0700

    zlib: wrap deflateBound() too

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -81,0 +85,5 @@
+unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+{
+       return deflateBound(strm, size);
+}
+
```

Git bir proqramlaşdırma dilində bir funksiya və ya metodla necə uyğunlaşa biləcəyini bilmirsə, onu regular bir ifadə ilə təmin edə bilərsiniz (və ya *regex*). Məsələn, yuxarıdakı nümunə ilə eyni işi görmüş olardı: `git log -L '/unsigned long git_deflate_bound/',/^}/:zlib.c`. Ayrıca bir sıra sətir və ya bir sətir nömrəsi verə bilərsiniz və eyni cür nəticə əldə edəcəksiniz.

Tarixi Yenidən Yazmaq

Git ilə işləyərkən dəfələrlə local commit-lər tarixinizi yenidən nəzərdən keçirmək istəyə bilərsiniz. Git ilə əlaqəli ən yaxşı şeylərdən biri, mümkün olan ən son anda qərar verməyə imkan verməsidir. Səhnələşdirmə sahəsinə başlamazdan əvvəl hansı faylların hansı qeydlərə keçəcəyini, hələ `git stash` ilə bir şey üzərində işləmək istəmədiyinizi qərar verə bilərsiniz və əvvəllər görülən prosesl ərin görünməsi üçün yenidən fərqli bir şəkildə baş vermiş kimi yaza bilərsiniz. Bu, işinizi başqaları ilə bölüşməzdən əvvəl əmrlərin dəyişdirilməsini, mesajların dəyişdirilməsini və ya commit edilən

fayların dəyişdirilməsini, bir yerə yığılmağı, ayrılmağı və ya tamamilə silinməsini ehtiva edə bilər.

Bu bölmədə bu tapşırıqları necə yerinə yetirəcəyinizi görürsünüz ki, başqaları ilə paylaşmazdan əvvəl commit tarixçənizi istədiyiniz şəkildə göstərə biləsiniz.

İşinizdən məmnun olana qədər push etməyin

Git-in əsas qaydalarından biri budur ki, klon daxilində bu qədər iş local olduğundan, tarixinizi yenidən *locally* olaraq yenidən yazmaq üçün çoxlu azadlığa sahibsiniz. Ancaq işinizi push etdikdən sonra tamamilə fərqli bir hekayədir və d əyişdirmək üçün əsaslı səbəbiniz olmadığı təqdirdə sövq edilən işi yekun hesab etməlisiniz. Bir sözlə, işinizdən məmnun olana qədər və onu dünyanın qalan hiss əsi ilə bölüşməyə hazır olana qədər push etməkdən çəkinməlisiniz.

Son Commit Dəyişdirilməsi

Ən son cimmit-i dəyişdirmək, bəlkə də tarixin ən çox yayılmış yenidən yazılmasıdır. Sonuncu commit-nizə görə tez-tez iki əsas işi görmək istəyəcəksiniz: sadəcə commit mesajını dəyişdirin və ya faylları əlavə etmək, silmək və dəyişdirməklə commit-in həqiqi məzmununu dəyişdirin.

```
$ git commit --amend
```

Yuxarıdakı əmr əvvəlki commit mesajını redaktor sessiyasına yükləyir, burada mesajda dəyişiklik edə, həmin dəyişiklikləri saxlaya və çıxa bilərsiniz. Redaktoru saxlayıb bağladığınız zaman, redaktor yenilənmiş commit mesajı olan yeni bir commit yazır və onu yeni son commit-nizə çevirir.

Digər tərəfdən, son commit-in əsl *content*-ini dəyişdirmək istəyirsinizsə, proses əsasən eyni şəkildə işləyir - əvvəl unutduğunuzu düşündüyünüz dəyişiklikləri edin, bu dəyişiklikləri səhnələşdirin və sonrakı `git commit --amend` yeni, yaxşılaşdırılmış commit-nizlə son commit-i olan *replaces* edəc əkdir.

Bu texnika ilə diqqətli olmalısınız, çünki dəyişikliklər commit-in SHA-1-ini dəyişdirir. Çox kiçik bir rebase-ə bənzəyir - son commit-nizin əvvəldən push etmsisnizsə, düzəliş etməyin.

Dəyişdirilmiş commit-in dəyişdirilmiş commit mesajına ehtiyacı ola bilər (ya da olmaya bilər)

Bir commit-i dəyişdirdiyiniz zaman həm commit mesajını, həm də commit—in m əzmununu dəyişdirmək imkanınız var. Commit-in məzmununu əsaslı şəkildə d əyişdirirsinizsə, demək olar ki, dəyişiklik edilmiş məzmunu əks etdirmək üçün commit mesajını yeniləməlisiniz.

Digər tərəfdən, düzəlişləriniz əvvəlcədən edilən mesajın çox yaxşı olması üçün (mənasız bir səhvi düzəltmək və ya səhnəyə qoymağı unutduğunuz bir faylı əlavə etmək) uyğun deyilsə, sadəcə dəyişikliklər edə, onları səhnələşdirə və lazımsız redaktordan qaça bilərsiniz:

```
$ git commit --amend --no-edit
```

Birdən Çox Commit Mesajının Dəyişdirilməsi

Keçmişinizə qayıdan bir prosesi dəyişdirmək üçün daha mürəkkəb alətlərə keçməlisiniz. Git-in bir dəyişdirmə tarixçəsi vasitəsi yoxdur, ancaq bir sıra commit-ləri başqasına köçürmək əvəzinə əvv əlcə əsaslandıqları HEAD üzərinə biraz bərpa etmək üçün rebase alətindən istifadə edə bilərsiniz. İnteraktiv rebase aləti ilə mesajı modify etmək və dəyişdirmək, fayl əlavə etmək və ya etmək ist ədiyiniz hər bir əməldən sonra dayandıra bilərsiniz. İnteraktiv olaraq `git rebase` seçiminə `-i` seçimi əlavə edərək reaktivliyi işə sala bilərsiniz. Yenidən yazmaq istədiyiniz əmri söyləyərək commit-ləri yenidən yazmaq istədiyinizi göstərməlisiniz.

Məsələn, son üç commit mesajını və ya bu qrupdakı commit mesajlarından birini dəyişdirmək ist əyirsinizsə, düzəliş etmək istədiyiniz son commit-in valideynini `git rebase -i` üçün bir arqument olaraq verirsiniz, yəni `HEAD~2^` və ya `HEAD~3`.

Son üç commit-i düzəltməyə çalışdığınız üçün `~3`-ü xatırlamaq daha asan ola bilər, ancaq düzəliş etmək istədiyiniz son commit-in əsas hissəsi olan dörd commit-i əvvəl təyin etdiyinizi unutmayın:

```
$ git rebase -i HEAD~3
```

Yenidən unutmayın ki, bu bir rebasing əmrdir - dəyişdirilmiş bir mesajla `HEAD ~ 3..HEAD` aralığında hər bir commit *və onun bütün nəsilləri* yenidən yazılacaqdır. Artıq mərkəzi bir serverə sövq etdiyiniz hər hansı bir commit-i daxil etməyin - bunu etmək eyni dəyişikliyin alternativ versiyasını təqdim edərək digər developerləri çaşdıracaqdır.

Bu əmri işə salmaq mətn redaktorunuzda aşağıdakı kimi görünən commit-lərin siyahısını verir:

```
pick f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Qeyd etmək vacibdir ki, bu commit-lər normal olaraq `log` əmrindən istifadə etdiyinizdən əks qaydada verilmişdir. Bir `log` işlədirsinizsə, belə bir şey görürsünüz:

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d Add cat-file
310154e Update README formatting and add blame
f7f3f6d Change my name a bit
```

Ters sıraya diqqət yetirin. İnteraktiv rebase sizə çalışacağı bir skript verir. Komanda xəttində göst ərdiyiniz commit-dən başlayacaq (`HEAD ~ 3`) və bu əmrlərin hər birində tətbiq olunan dəyişiklikləri yuxarıdan aşağıya doğru təkrarlayın. Ən yenisinin yerinə, ən köhnəsini ən üstdə siyahıya alır, çünki yenidən replay edəcəyi ilk budur.

Skripti redaktə etmək istədiyiniz commit-də dayandığı üçün düzəltməlisiniz. Bunu etmək üçün, ssenarinin dayandırılmasını istədiyiniz commit-lərin hər biri üçün ‘pick’ sözünü ‘edit’ sözünə çevirin. Məsələn, yalnız üçüncü commit mesajını dəyişdirmək üçün sənədi bu şəkildə dəyişirsiniz:

```
edit f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

Redaktoru saxladığınızda və çıxdığınızda, Git sizi həmin siyahıda sonuncu commit-ə qaytarır və aşağıdakı mesajla komanda xəttinə salır:

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... changed my name a bit
You can amend the commit now, with

       git commit --amend

Once you're satisfied with your changes, run

       git rebase --continue
```

Bu təlimatlar sizə tam olaraq nə edəcəyinizi bildirir. Növ:

```
$ git commit --amend
```

Commit mesajını dəyişdirin və redaktordan çıxın. Sonra çalıştırın:

```
$ git rebase --continue
```

Bu əmr digər iki commit-i avtomatik olaraq tətbiq edəcək və sonra işiniz biticək. Seçimi daha çox s ətirdə redaktə etmək üçün dəyişdirsəniz, dəyişdirmək üçün etdiyiniz hər bir commit üçün bu addımları təkrarlaya bilərsiniz. Hər dəfə Git dayanacaq, commit-i düzəltməyinizə icazə verin və bitirdikdən sonra davam edin.

Commit-lərin Yenidən Tənzimlənməsi

Commit-lərin yenidən sıralanması və ya tamamilə silinməsi üçün interaktiv reaksiyalardan da istifadə edə bilərsiniz. “added cat-file” commit-ini aradan qaldırmaq və digər iki commit-in təqdim olunduğu sıranı dəyişdirmək istəyirsinizsə, rebase skriptini burdan dəyişə bilərsiniz:

```
pick f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

buna:

```
pick 310154e Update README formatting and add blame
pick f7f3f6d Change my name a bit
```

Redaktoru saxladığınızda və çıxdığınızda, Git branch-nızı bu commit-lərin valideyninə geri qaytarır, `310154e` və sonra `f7f3f6d` tətbiq edir və sonra dayanır. Bu commit-in sırasını təsirli şəkildə dəyişdirir və "added cat-file" tamamilə aradan qaldırırsınız.

Squashing Commits

İnteraktiv rebasing alati vasitəsi ilə bir sıra commit-lər götürmək və onları tək bir işə salmaq da mümkündür. Skript rebase mesajında faydalı təlimatlar verir:

```
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Əgər "pick" və ya "edit" əvəzinə, "squash"-ı təyin etsəniz, Git həm bu dəyişikliyi, həm də dəyişikliyi birbaşa ondan əvvəl tətbiq edir və commit mesajlarını birləşdirməyə məcbur edir. Beləliklə, bu üç commit-dən bir commit götürmək istəyirsinizsə, ssenarini belə göstərəcəksiniz:

```
pick f7f3f6d Change my name a bit
squash 310154e Update README formatting and add blame
squash a5f4a0d Add cat-file
```

Redaktoru saxladıqda və çıxdıqda, Git hər üç dəyişikliyi tətbiq edir və sonra üç commit mesajını birləşdirmək üçün sizi yenidən redaktora qaytarır:

```
# This is a combination of 3 commits.
# The first commit's message is:
Change my name a bit

# This is the 2nd commit message:

Update README formatting and add blame

# This is the 3rd commit message:

Add cat-file
```

Bunu saxladığınızda əvvəlki üç commit-in dəyişikliklərini təqdim edən tək bir commit-niz var.

Bir Commit-i Bölmək

Bir commit-in bölünməsi bir commit-i ləğv edir, sonra qismən mərhələli edir və sona çatdırmaq istədiyiniz qədər commit yerinə yetirir. Məsələn, üç commit-nizin orta commit-ni bölmək ist ədiyinizi düşünək.

“README formatını yeniləyin və blame əlavə edin” əvəzinə, onu iki commit-ə bölmək istəyirsinizs ə: birincisi üçün “README formatını yeniləyin”, ikincisi üçün “Blame əlavə edin”. Bölmək ist ədiyiniz commit-in təlimatını “edit” etməklə dəyişdirərək bunu `rebase -i` skriptində edə bilərsiniz:

```
pick f7f3f6d Change my name a bit
edit 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

Sonra, skript sizi əmr sətirinə endirdikdə, commit-i yenidən qurursunuz, yenidən qurulmuş d əyişiklikləri götürürsünüz və onlardan bir çox commit yaradırsınız.

Redaktoru saxladığınızda və çıxdığınızda, Git siyahınızdakı ilk commit-in valideyninə geri qayıdır, ilk commit-i tətbiq edir (`f7f3f6d`), ikincisini (`310154e`) tətbiq edir və sizi konsola salır. Orada, bu commit-i bir hash ilə `git reset HEAD^` ilə yenidən qura bilərsiniz; bu commit-i effektiv şəkildə ləğv edir və dəyişdirilmiş sənədləri işlənmədən buraxır. İndi bir neçə hərəkətiniz olana qədər sənədləri hazırlayıb təqdim edə və bitdikdən sonra `git rebase --continue` düyməsini işə sala bilərsiniz:

```
$ git reset HEAD^
$ git add README
$ git commit -m 'Update README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'Add blame'
$ git rebase --continue
```

Git, ssenaridəki son commit-i (`a5f4a0d`) tətbiq edir və tarixiniz belə görünür:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd Add cat-file
9b29157 Add blame
35cfb2b Update README formatting
f7f3f6d Change my name a bit
```

Bu, siyahınızdakı ən son üç commit-in SHA-1-lərini dəyişdirir, buna görə heç bir dəyişdirilmiş commit-in həmin siyahıda artıq paylaşılan bir depoya sövq etdiyinizin görünməməsinə əmin olun. Diqqət yetirin ki, siyahıda sonuncu commit (`f7f3f6d`) dəyişməzdir. Bu commit-in ssenaridə göst ərilməsinə baxmayaraq "pick" kimi qeyd olunduğundan və hər hansı bir endirim dəyişikliyindən əvvəl tətbiq olunduğu üçün Git commit-i dəyişdirilmədən tərk edir.

Drew DeVault, `git rebase`-dən istifadə qaydalarını öyrənmək üçün praktik bir t əlimat hazırladı. Burdan tapa bilərsiniz: https://git-rebase.io/

Nuclear Seçimi: filter-branch

Misal üçün e-poçt adresinizi qlobal olaraq dəyişdirmək və ya hər bir commit-dən bir sənəd çıxarmaq üçün daha çox sayda commit-i yenidən yazmaq lazımdırsa, istifadə edə biləcəyiniz başqa bir tarix yazma seçimi var. Əmr `filter-branch`-dir və tarixinizin böyük hissələrini yenidən yaza bil ər, buna görə proyektiniz hələ public-ə açıq olmadıqca və digər insanlar sizin etdiyiniz vəzifələrin icrasına əsaslanmadığı təqdirdə istifadə etməməlisiniz.

Bununla birlikdə, çox faydalı ola bilər. Yaygın istifadə üsullarından bir neçəsini öyrənəcəksiniz, buna görə bacardığı bəzi şeylər haqqında bir fikir əldə edə bilərsiniz.

`git filter-branch`-in bir çox tələsi var və artıq tarixi yenidən yazmağın tövsiyə olunan yol deyil. Bunun əvəzinə, normal olaraq `filter-branch`-ə müraciət etdiyiniz bir çox tətbiqetmə üçün daha yaxşı iş görən bir Python skripti olan `git-filter-repo` istifadə etməyi düşünün. Sənədlərinə və mənbə koduna https://github.com/newren/git-filter-repo ünvanından baxmaq olar.

Hər Commit-dən Bir Sənədin Silinməsi

Bu olduqca yaygındır. Biri təsadüfən düşüncəsiz bir `git add .` İlə böyük bir ikili sənəd işlədir və onu hər yerdə silmək istəyirsən. Bəlkə də səhvən bir şifrə olan bir sənəd işlədmisiniz və layihənizi açıq mənbəyə çevirmək istəyirsiniz. `filter-branch`, yəqin ki, bütün tarixinizi təmizləmək üçün istifadə etmək istədiyiniz bir vasitədir. Bütün tarixinizdən `passwords.txt` adlı bir faylı silmək üçün `--tree-filter` seçimini `filter-branch` üçün istifadə edə bilərsiniz:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

`--tree-filter` seçimi, layihənin hər yoxlanışdan sonra göstərilən əmri işə salır və nəticələri tövsiyə edir. Bu vəziyyətdə, mövcud olub-olmamasından asılı olmayaraq hər snapshot-dan `passwords.txt`

adlı bir faylı çıxararsınız. Səhvən yadda qalan bütün redaktor backup sənədlərini silmək ist əyirsinizsə, `git filter-branch --tree-filter 'rm -f *~' HEAD` kimi bir şey işlədə bilərsiniz.

Git-in ağacları və commit-lərini yenidən yazdığını izləyə biləcəksiniz və sonra branch göstəricisini sonda hərəkət etdirə bilərsiniz. Bunu ümumiyyətlə bir test branch-ında etmək və nəticəni həqiqət ən istədiyiniz şeyi müəyyənləşdirdikdən sonra `master` branch-nızı yenidən bərpa etmək daha yaxşıdır. Bütün branch-larınızda `filter-branch` işlətmək üçün əmrinə `--all` keçə bilərsiniz.

Bir Subdirectory-nı Yeni Root Halına Gətirmək

Fərz edək ki, başqa bir mənbədən idarəetmə sistemindən bir idxal etdiniz və heç bir mənası olmayan subdirectory-ləriniz var (`trunk`, `tags` və s.). `trunk` subdirectory-ni hər bir commit üçün yeni layihə kökü etmək istəyirsinizsə, `filter-branch` bu halda sizə kömək edə bilər:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

İndi yeni layihə root-unuz hər dəfə `trunk` subdirectory-dadır. Git ayrıca subdirectory-ni təsir etmə yən commit-ləri avtomatik olaraq siləcəkdir.

E-poçt Ünvanlarını Qlobal Olaraq Dəyişdirmək

Başqa bir ümumi hal, işə başlamazdan əvvəl adınızı və e-poçt adresinizi təyin etmək üçün `git config`-i işə salmağı unutmusunuz və ya bəlkə də iş yerində bir layihə açaraq bütün iş e-poçt adreslərinizi şəxsi adresinizə dəyişdirmək istəyirsiniz.

Hər halda, birdən çox prosesdə e-poçt adreslərini bir toplu halında `filter-branch` ilə əvəz edə bil ərsiniz. Yalnız özünüzə aid e-poçt adreslərini dəyişdirərkən diqqətli olmalısınız, beləliklə `--commit -filter` istifadə etməlisiniz:

```
$ git filter-branch --commit-filter '
        if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
        then
                GIT_AUTHOR_NAME="Scott Chacon";
                GIT_AUTHOR_EMAIL="schacon@example.com";
                git commit-tree "$@";
        else
                git commit-tree "$@";
        fi' HEAD
```

Bu, yeni ünvanınız üçün hər bir commit-i yenidən yazır. Qeydlər rəhbərlərinin SHA-1 dəyərlərini ehtiva etdiyindən, yalnız e-poçt ünvanlarına uyğun olanları deyil, əmr tarixinizdəki hər SHA-1 qeydini əvəz edir.

Reset Demystified

Daha ixtisaslaşmış alətlərə keçməzdən əvvəl Git `reset` və `checkout` əmrləri barədə danışaq. Bu ə mrlər, Git-lə ilk qarşılaşdığınız zaman ən çaşdırıcı hissələrindən biridir. Onlar o qədər şeylər edirl ər ki, onları həqiqətən başa düşmək və düzgün işlətmək ümidsiz görünür. Bunun üçün sadə bir metafora tövsiyə edirik.

The Three Trees

`reset` və `checkout` barədə düşünməyin daha asan yolu üç fərqli ağac məzmun meneceri olmaq üçün Git'in zehni çərçivəsindədir. Buradakı "tree" ilə, həqiqətən, məlumatların quruluşunu deyil, "files toplusunu" nəzərdə tuturuq. Bir neçə hal var ki, indeks tam olaraq bir ağac kimi davranmır, amma məqsədlərimiz üçün bu gün bu şəkildə düşünmək daha asandır.

Sistem olaraq Git normal işləmə rejimində üç ağac idarə edir və manipulyasiya edir:

Tree	Rol
HEAD	Son commit-i çəkmək, növbəti valideyn
Index	Təklif olunan növbəti commit snapshot-u
Working Directory	Sandbox

The HEAD

HEAD, öz növbəsində həmin branch-da edilən son commit-ə göstərici olan cari branch arayışına işarədir. Demək ki, HEAD yaradılan növbəti commit-in valideyni olacaq. HEAD bu branch-dakı son commit-nizin snapshot-u kimi düşünmək ümumiyyətlə ən sadədir. Əslində bunun necə göründüyünə baxmaq çox asandır. Budur HEAD anlıq görüntüsündə hər bir fayl üçün faktiki qovluq siyahısının və SHA-1 yoxlama cədvəlinin alınmasına dair bir nümunə:

```
$ git cat-file -p HEAD
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
author Scott Chacon  1301511835 -0700
committer Scott Chacon  1301511835 -0700

initial commit

$ git ls-tree -r HEAD
100644 blob a906cb2a4a904a152...   README
100644 blob 8f94139338f9404f2...   Rakefile
040000 tree 99f1a6d12cb4b6f19...   lib
```

Git `cat-file` və `ls-tree` əmrləri aşağı səviyyə işlər üçün istifadə olunan və həqiqətən gündəlik işl ərdə istifadə edilməyən "plumbing" əmrləridir, lakin bunlar burada nələrin baş verdiyini görməyə kömək edir.

Index

Index təklif olunan növbəti commit-dir. Biz də bu anlayışa Git'in "Staging Area" olaraq müraciət etdik, çünki Git `git commit` işlətdiyiniz zaman bu hissə Git-in baxdığı yerdir.

Git, bu indeks iş sənədlərinizdə son yoxlanılan və əvvəlcə yoxlanıldıqda göründüyü bütün fayl m əzmunlarının siyahısı ilə doldurulur. Daha sonra həmin faylların bəzilərini yeni versiyaları ilə əv əz edirsiniz və `git commit` onları yeni bir commit üçün ağaca çevirir.

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0   README
100644 8f94139338f9404f26296befa88755fc2598c289 0   Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0   lib/simplegit.rb
```

Yenə də, burada indeksinizin göründüyünü göstərən pərdə arxasında daha çox olan `git ls-files` istifadə olunur.

İndeks texniki cəhətdən bir ağac quruluşunda deyil - əslində sadə bir təzahür şəklində tətbiq olunur - lakin məqsədlərimiz üçün kifayət qədər yaxındır.

İş Qovluğu

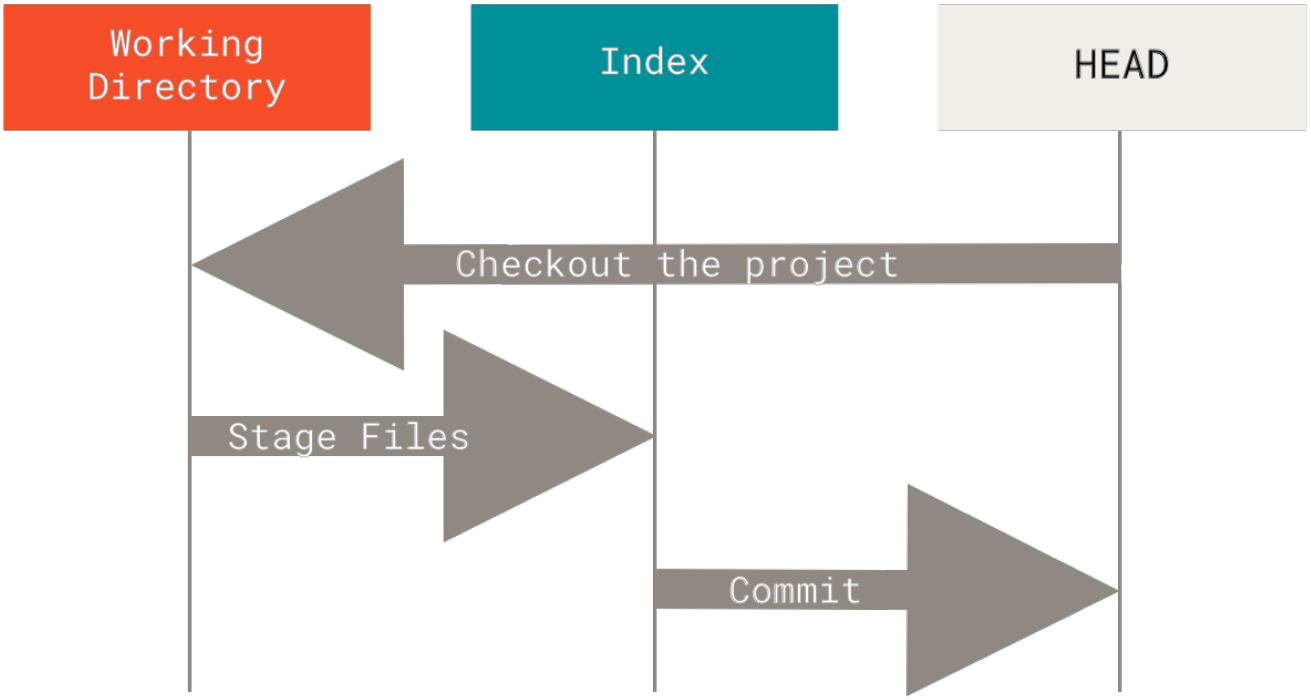
Nəhayət, bir iş qovluğunuz var (ümumiyyətlə "working tree" adlandırılır). Digər iki ağac, m əzmunlarını səmərəli, lakin əlverişsiz bir şəkildə, `.git` qovluğu içərisində saxlayır. İşçi qovluq onları faktiki fayllara ayırır, bu da onları redaktə etməyi asanlaşdırır. İşçi qovluğu bir **sandbox** olaraq düşünün, burada dəyişiklikləri sahəyə (indeks) və sonra tarixə verməzdən əvvəl sınaqdan keçirə bilərsiniz.

```
$ tree
.
├── README
├── Rakefile
└── lib
    └── simplegit.rb

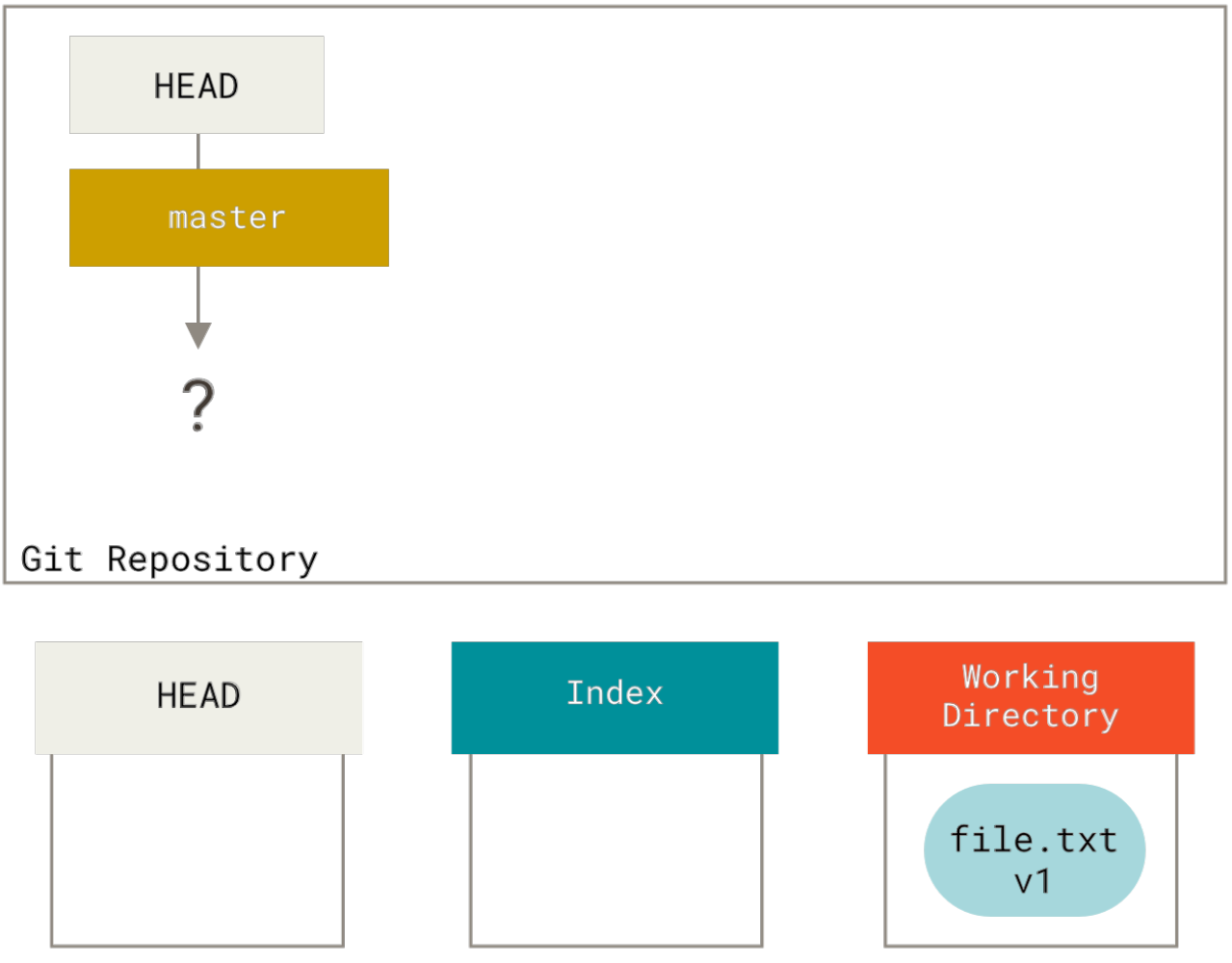
1 directory, 3 files
```

Workflow

Git-in tipik workflow-u bu üç ağacı manipulyasiya etməklə layihənizin görüntülərini ardıcıl olaraq daha yaxşı vəziyyətlərdə qeyd etməkdir.

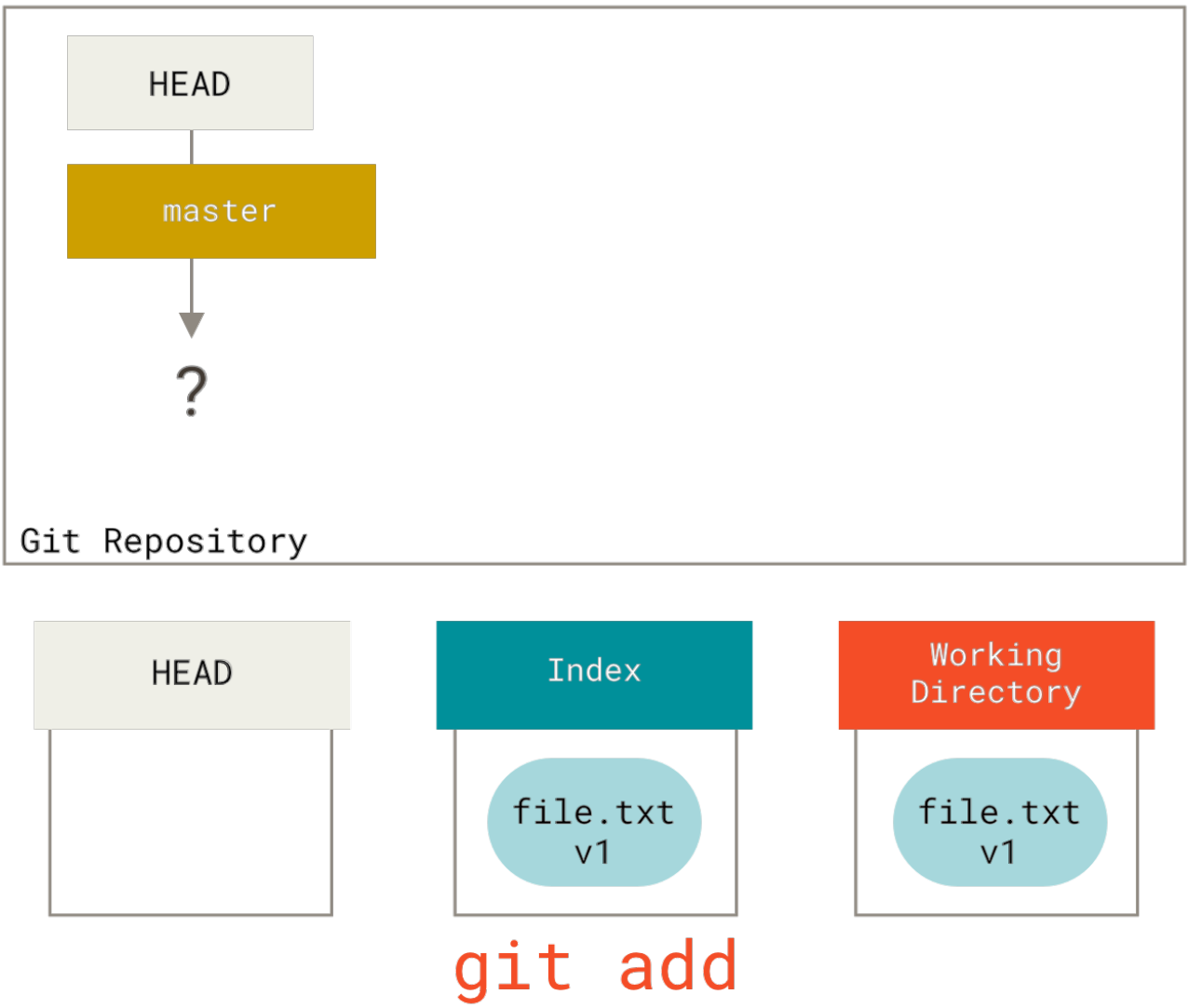


Bu prosesi görüntüləyək: sizin bir sənədlə yeni bir qovluğa girdiyinizi fərz edək. Bunu faylın **v1**-i adlandıracağıq və göy rəngdə göstərəcəyik. İndi biz doğmamış `master` brancha-a işarə edən bir HEAD arayışı ilə Git depo yaradacaq və `git init`-i işlədəcəyik.

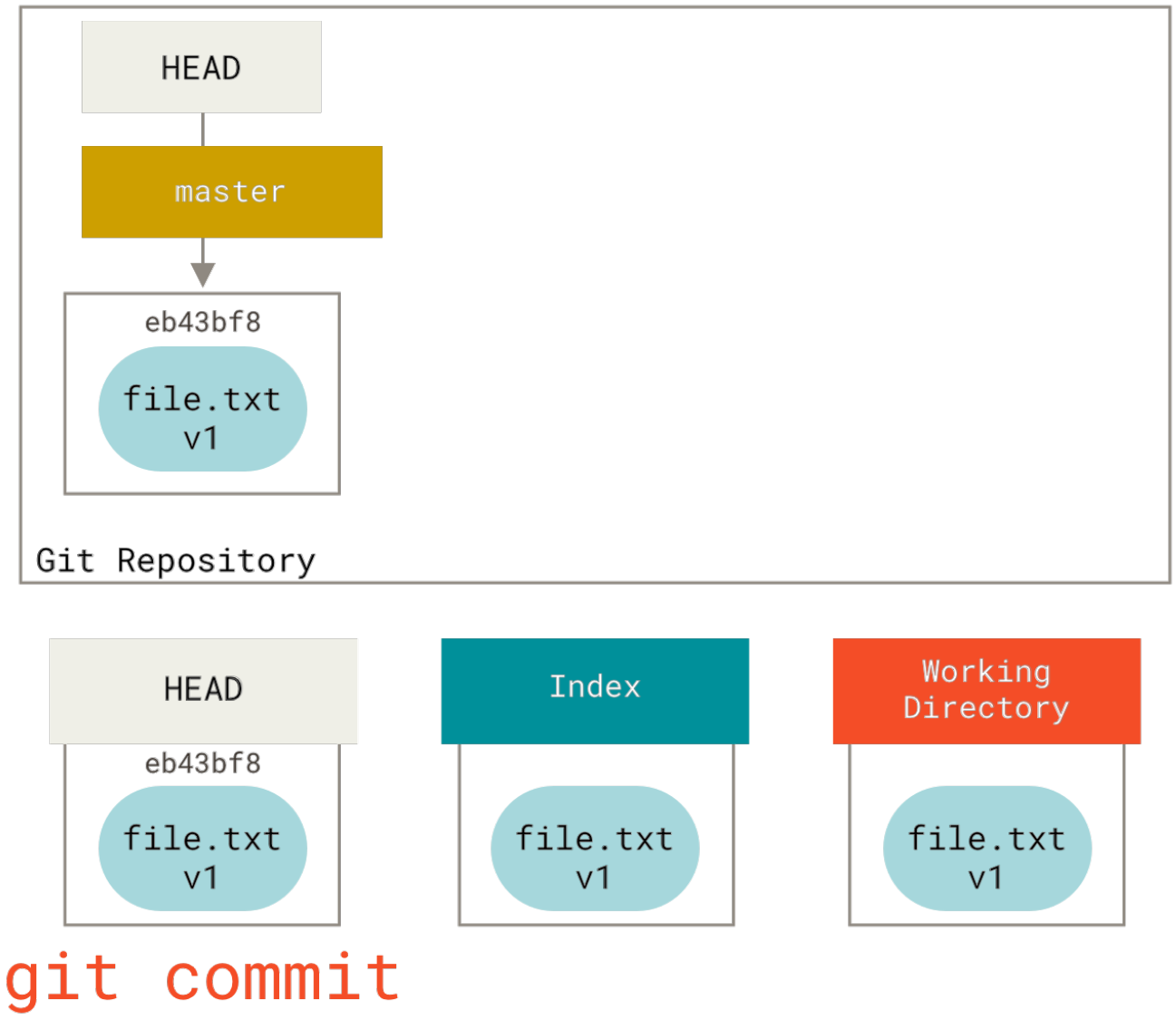


Bu anda yalnız iş qovluğu ağacı hər hansı bir məzmuna malikdir.

İndi bu faylı yerinə yetirmək istəyirik, buna görə işçi qovluğunda məzmun götürmək və indeksə kopyalamaq üçün `git add` istifadə edirik.

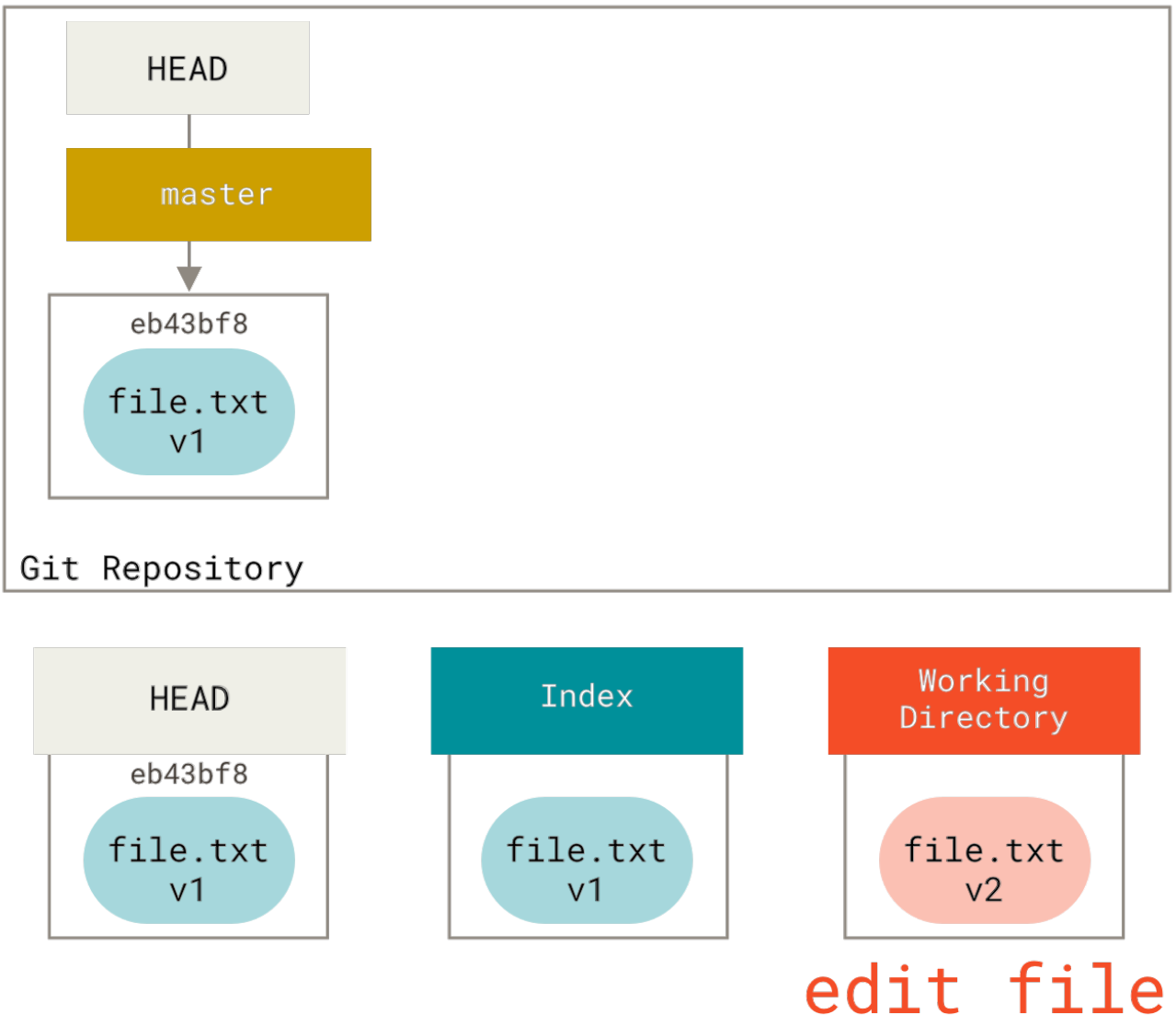


Sonra indeksin məzmununu götürən və onu daimi şəkli kimi saxlayan, həmin görüntüyə işarə ed ən bir commit obyekti yaradan və həmin commit-ə işarə etmək üçün `master` yeniləyən `git commit`-i işə salırıq.

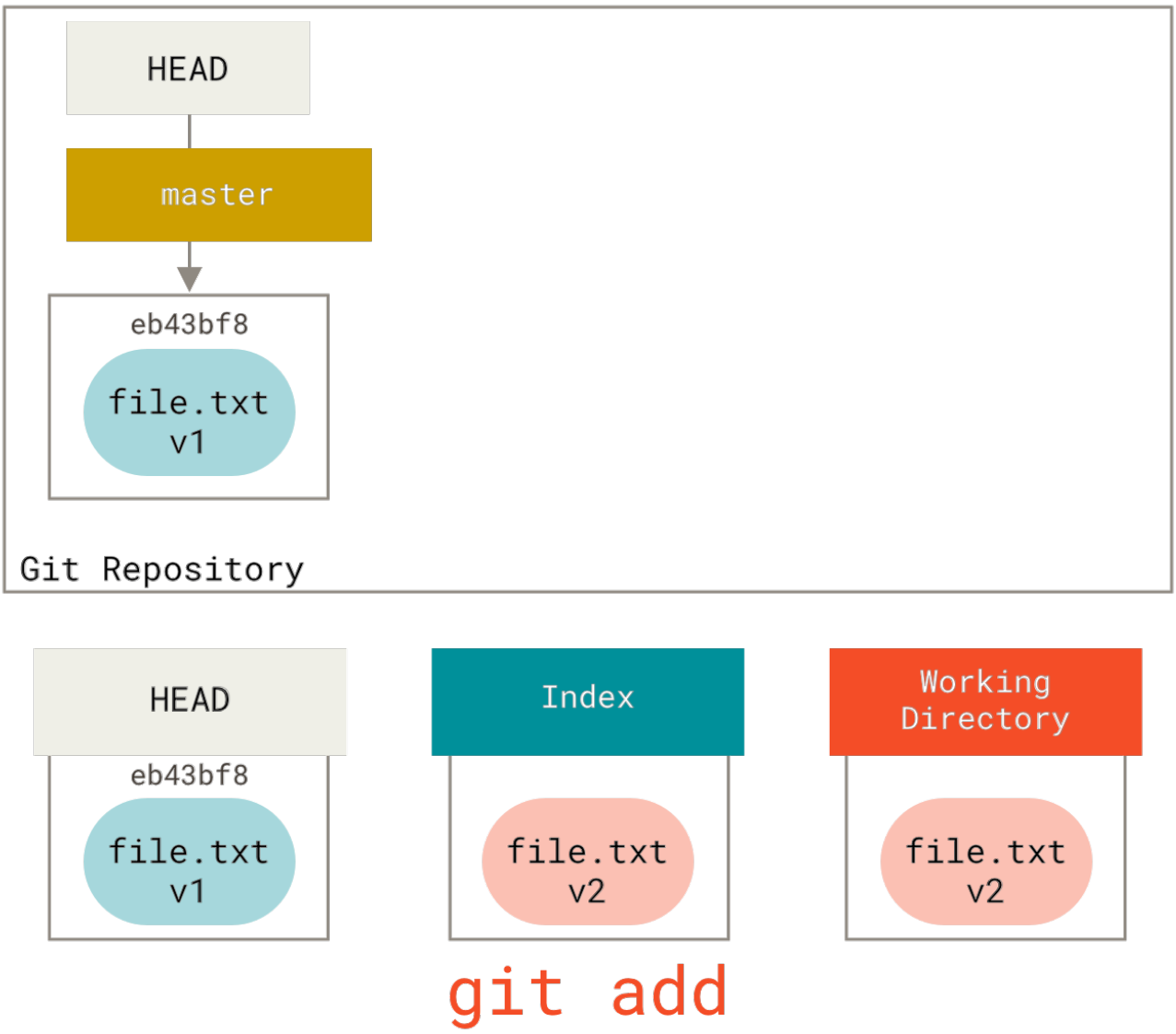


`git status`-u işə salsaq, heç bir dəyişiklik görməyəcəyik, çünki hər üç ağac eyni qalacaq.

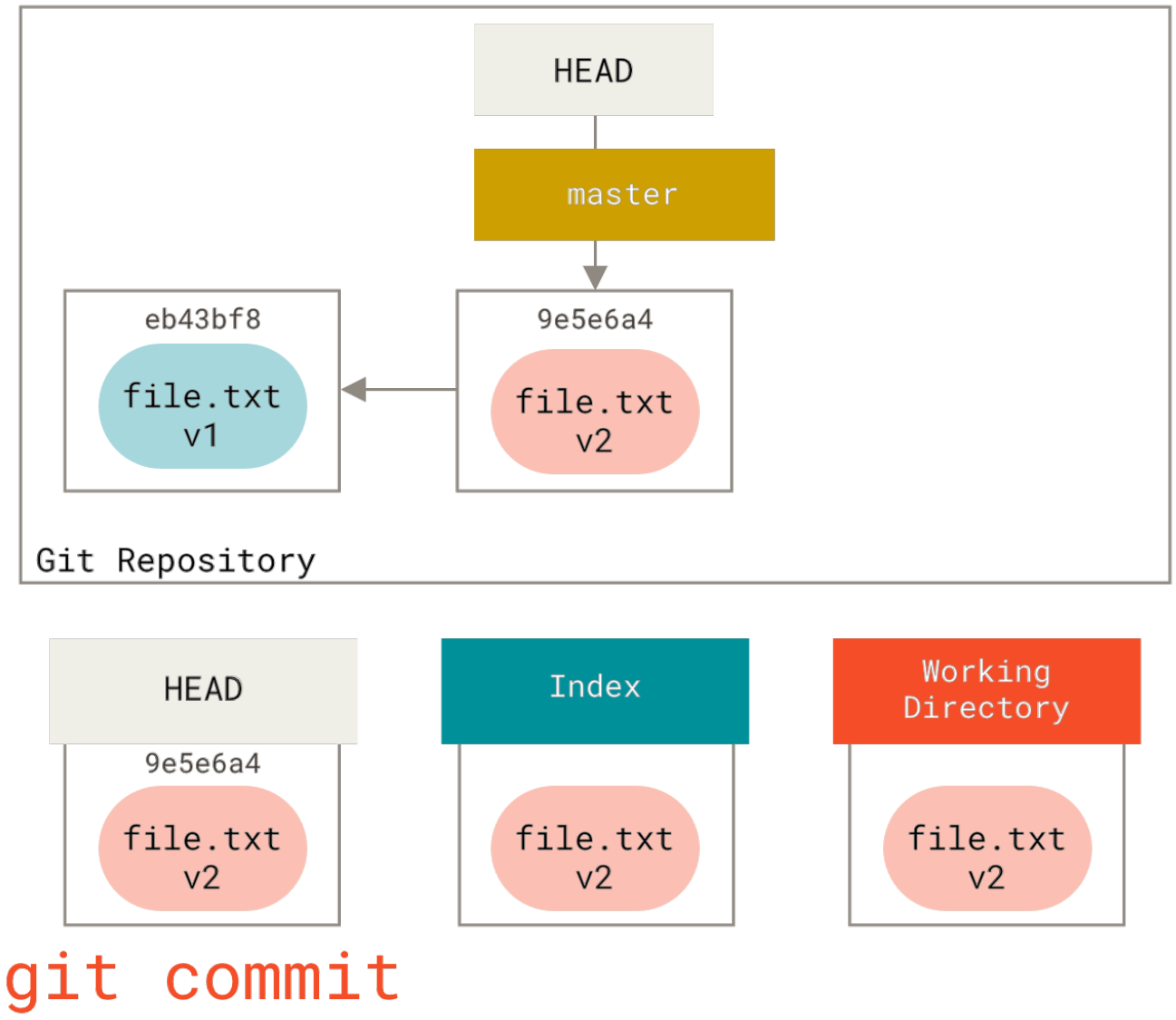
İndi həmin fayla bir dəyişiklik etmək və onu yerinə yetirmək istəyirik. Eyni prosesdən keçəcəyik; əvvəlcə iş sənədlərindəki faylı dəyişdiririk. Ona faylın **v2**-si deyək və qırmızı ilə göstərək.



Hal-hazırda `git status`-u işlədiriksə, faylı qırmızı rəngdə “Changes not staged for commit” şəklində görəcəyik, çünki bu giriş indeks və işçi qovluq arasında fərqlidir. Sonra biz onu indeksimizə daxil etmək üçün `git add`-ı işə salırıq.



Bu nöqtədə, əgər biz `git status`-u işlədiriksə, faylı “Changes to be committed” altında yaşıl rəngdə görəcəyik, çünki indeks və HEAD fərqlənir - yəni təklif olunan növbəti commit-imiz artıq son commit-mizdən fərqlənir. Nəhayət, commit-i yekunlaşdırmaq üçün `git commit`-i işlədirik.



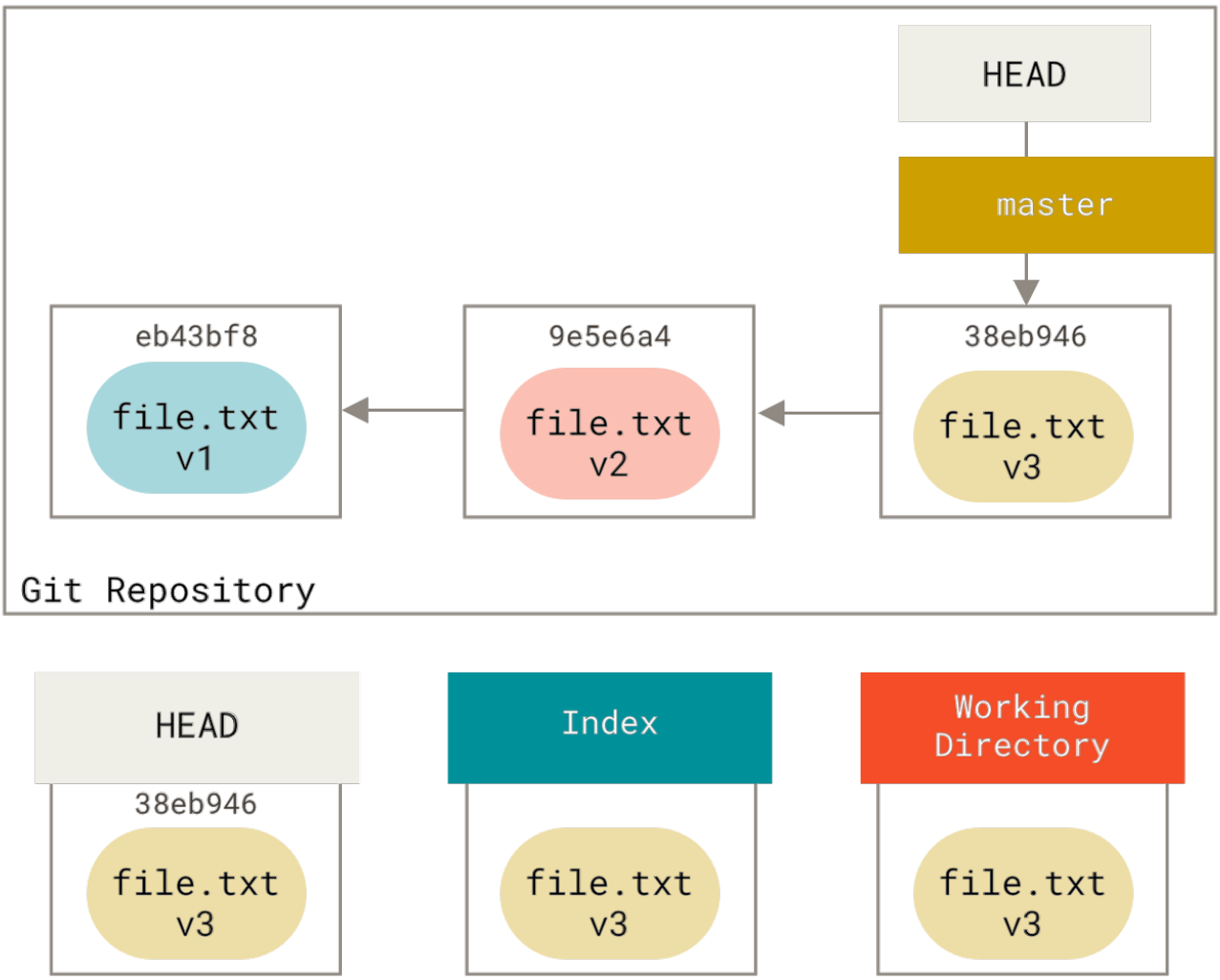
İndi `git status` bizə heç bir output verməyəcək, çünki hər üç ağac yenə eynidir.

Branch-ları dəyişdirmə və ya klonlama oxşar bir prosesdən keçir. Bir branch-ı çıxartdıqda, yeni branch-ı ref-ə işarələmək üçün **HEAD**-ı dəyişdirir, commit-in görüntüsü ilə **indeksinizi** doldurur, sonra indeksin məzmununu **iş qovluğunuza** kopyalayır.

Reset-in Rolu

`reset` əmri bu kontekstdə baxıldıqda daha mənalı olur.

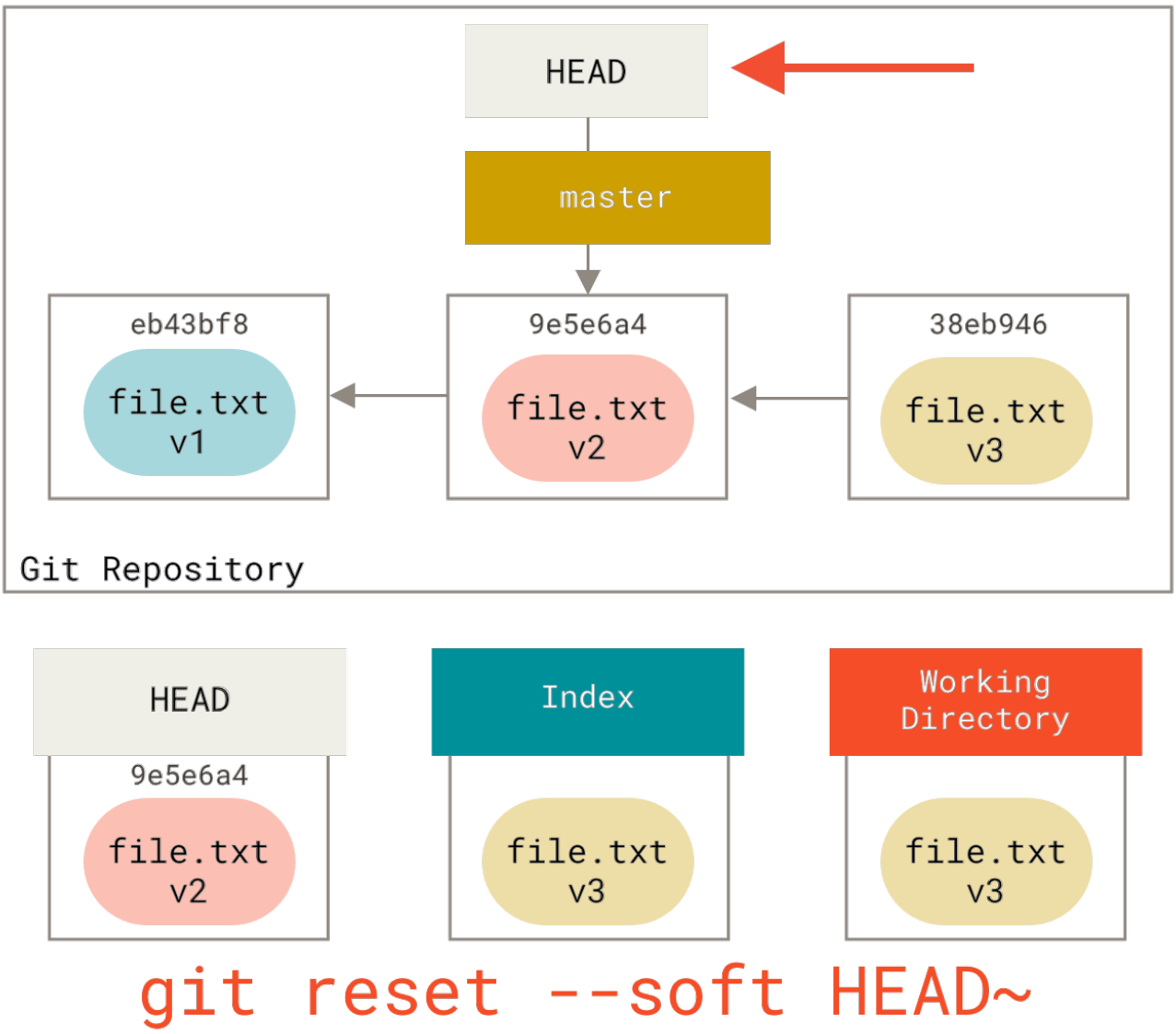
Bu nümunələrin məqsədləri üçün deyək ki, `file.txt`-ı yenidən dəyişdirdik və üçüncü dəfə tətbiq etdik. İndi tariximiz belə görünür:



İndi başlatdığınız zaman `reset`-in nə etdiyini araşdıraq. Bu üç ağacı birbaşa sadə və proqnozlaşdırılan şəkildə manipulyasiya edir. Üç əsas əməliyyatı yerinə yetirir.

Addım 1: Move HEAD

Yenidən qurmağın ilk işi, HEAD-in göstərdiyi şeyi hərəkət etdirməkdir. Bu, HEAD-ın özünü d əyişdirməklə eyni deyil (bu da `yoxlama` qaydasındadır); reset, rəhbərin işarə etdiyi branch-ı hərəkət etdirir. Bu, HEAD `master` branch-na (yəni hazırda `master` branch-a) təyin olunarsa, `9e5e6a4 git reset 9e5e6a4` səviyyəsinə `master` nöqtəsi ilə başlayacaq deməkdir.



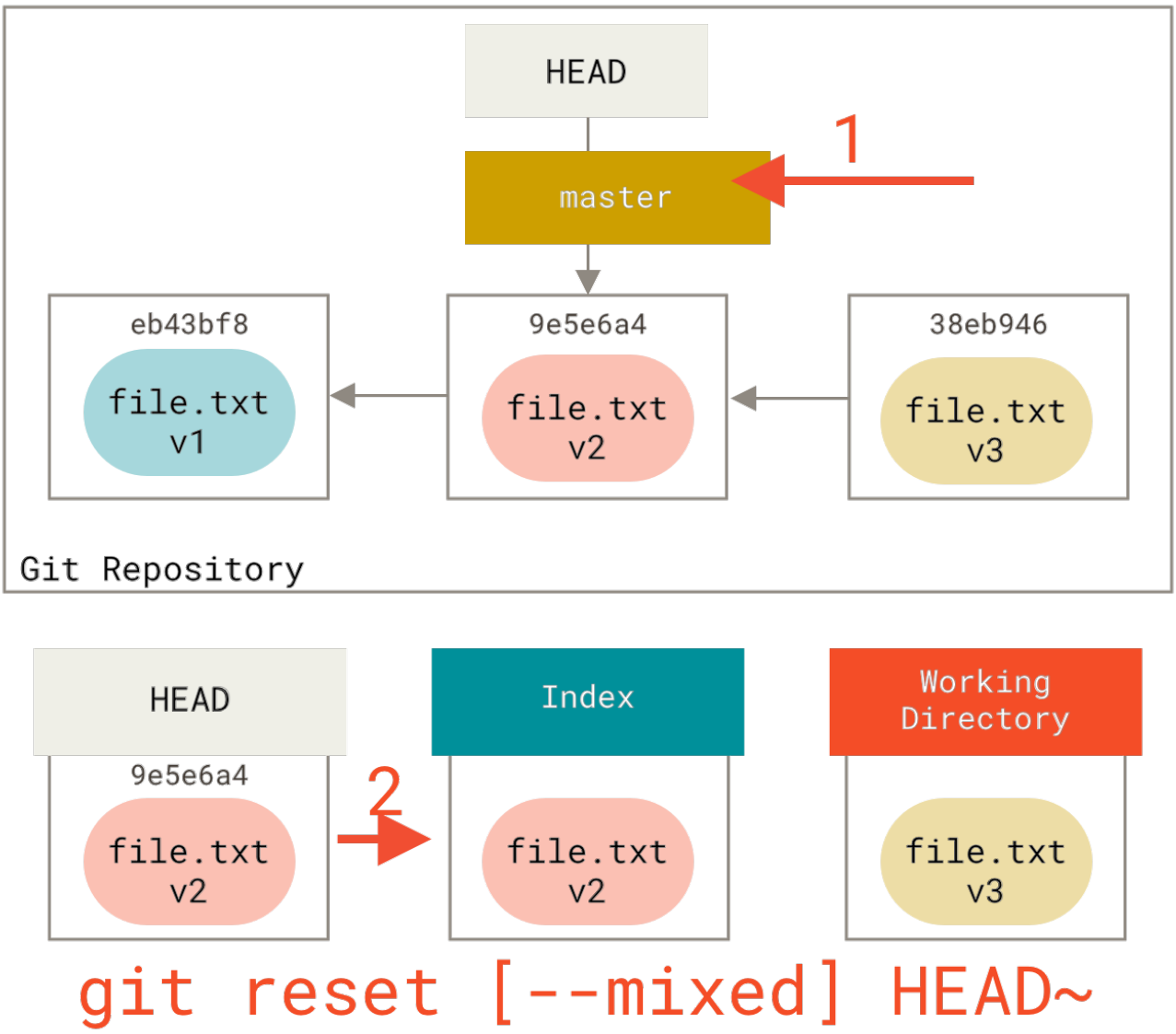
Commit ilə hər hansı bir `reset` formasından asılı olmayaraq, bu daima etməyə çalışacağı ilk şeydir. `reset --soft` ilə isə, sadəcə orada dayanacaq.

İndi bu diaqrama nəzər salmaq və nəyin baş verdiyini anlamaq üçün bir addım ataq: bu, son `git commit`-i ləğv etdi. `git commit`-i işə saldığınız zaman, Git yeni bir commit yaradır və rəhbərlik etdiyi branch-ı ona doğru istiqamətləndirir. `HEAD~`-a `reset` etdikdə (HEAD-ın valideyni), sənədi və ya iş qovluğunu dəyişdirmədən branch-ı olduğu yerə aparırsan. İndi indeksini yeniləyə bilər və `git commit --amend`-in nə edəcəyini yerinə yetirmək üçün yenidən `git commit` işlədə bilərsiniz (Son Commit Dəyişdirilməsi-a bax)

Addım 2: İndeksin yenilənməsi (-- mixed)

Qeyd edək ki, indi `git status`-u işləsəniz, indekslə yeni HEAD-in fərqini yaşıl rəngdə görəcəksiniz.

`Reset`-in sonrakı işi, indiyə kimi HEAD-ın qeyd etdiyi hər şeyi məzmunu ilə yeniləməkdir.

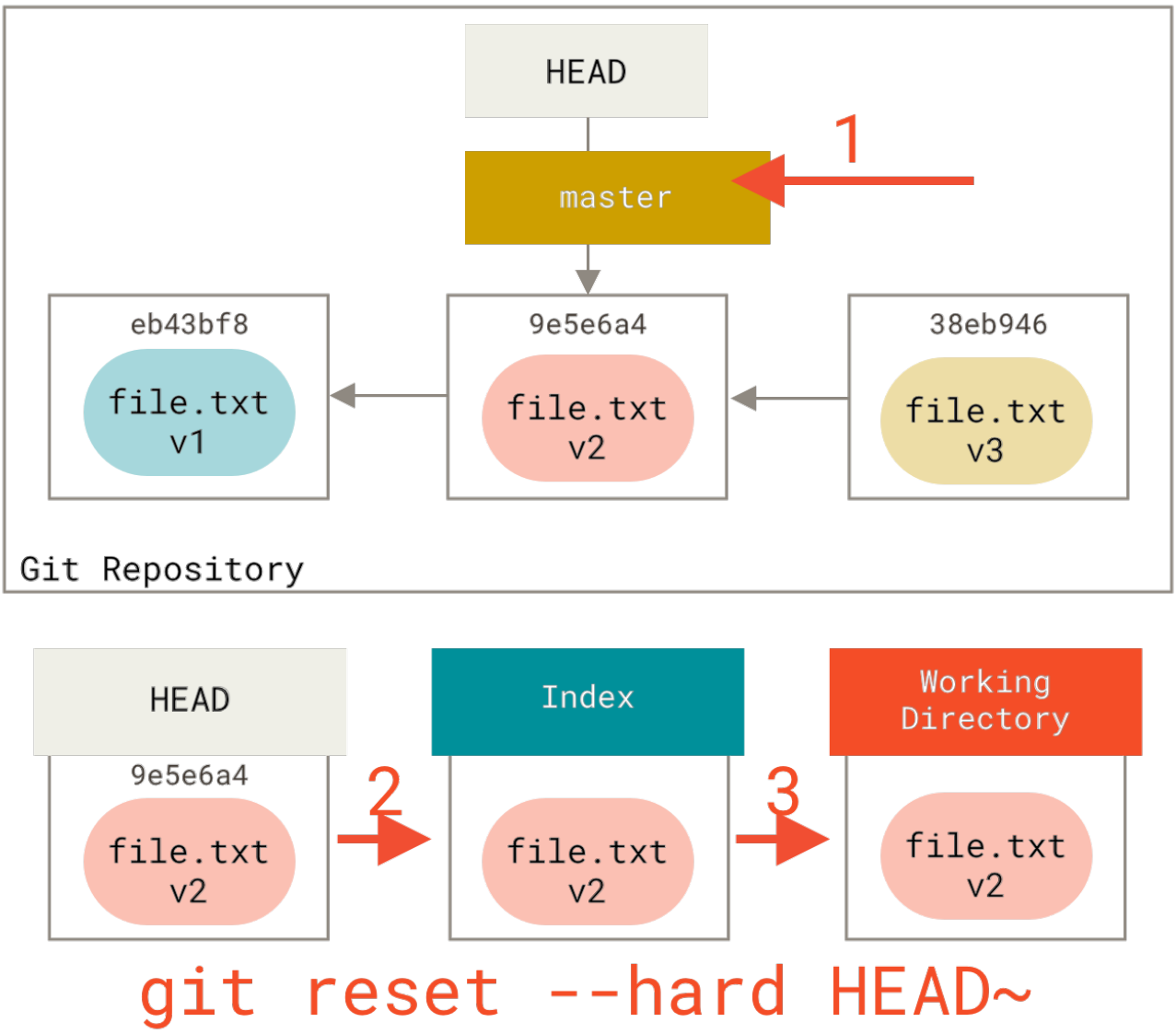


`--mixed` seçimi göstərsəniz, `reset` bu nöqtədə dayanacaq. Bu da standartdır, buna görə heç bir seçim etməmisinizsə (bu vəziyyətdə `git reset HEAD~` edin), əmr dayanacaq.

İndi bu diaqrama baxmaq və nəyin baş verdiyini anlamaq üçün başqa bir saniyə ayırın: bu hələ də son `commit`-nizi ləğv etdi, lakin, eyni zamanda hər şeyi dayandırdı. Bütün `git add` və `git commit` ə mrlərinizi yerinə yetirmədən əvvəl geri döndünüz.

Adım 3: İş Qovluğu Yeniləmə (--hard)

`reset`-in edəcəyi üçüncü şey, işləyən qovluğu indeksə bənzətməkdir. `--hard` seçimindən istifadə edirsinizsə, o, bu mərhələyə davam edəcəkdir.



Beləliklə, nə baş verdiyini düşünək. Sonuncu əmrinizi, `git add` və `git commit` əmrlərini **və** iş qovluğunuzda etdiyiniz bütün işləri ləğv etdiniz.

Qeyd etmək vacibdir ki, bu bayraq (`--hard`) `reset` əmrini təhlükəli hala gətirməyin yeganə yoludur və Git-in məlumatları məhv edəcəyi çox az hallardan biridir. `reset`-in hər hansı bir başqa çağırışı asanlıqla geri qaytarıla bilər, lakin `--hard` seçimi, işləyən qovluqdakı faylları çətinliklə yazacaq. Bu vəziyyətdə, hələ də Git DB-də commit-də **v3** versiyamız var və onu reflog-muza baxaraq geri ala bilərdik, amma əgər commit etməsəydik, Git yenə də faylın üstünə yazacaqdı və o bərpaedilməz olardı.

Recap

`reset` əmri bu üç ağacı müəyyən bir qaydada ləğv edir, bunu söylədikdə dayandırır:

1. Branch HEAD nöqtələrini hərəkətə gətirin *(`--soft` olduqda dayanın)*.
2. İndeksi HEAD kimi göstərin *(`--hard` olduqda dayandırın)*.
3. İş qovluğunu indeks kimi göstərin.

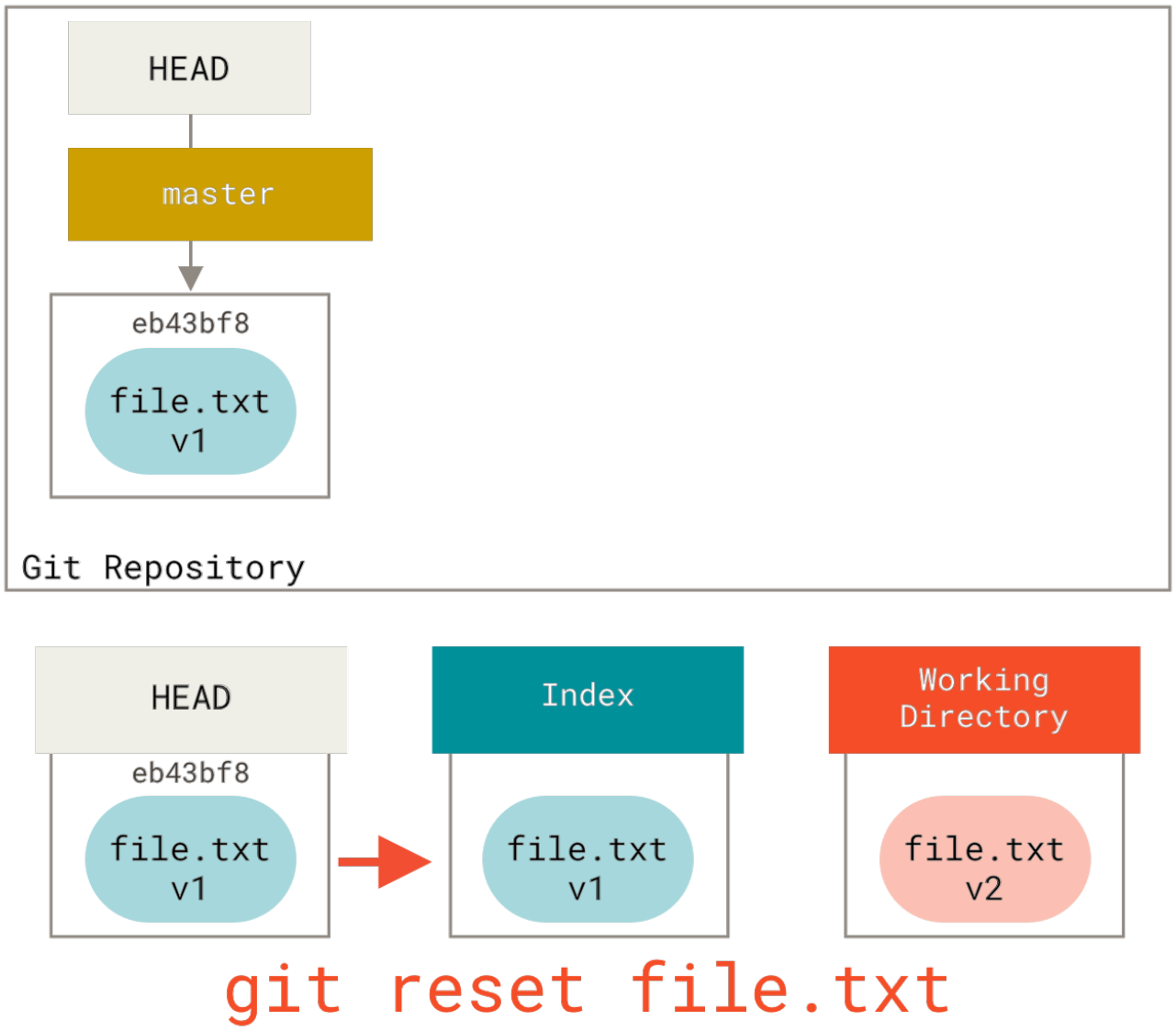
Path ilə Reset

Bu, əsas formada `reset`-in davranışını əhatə edir, eyni zamanda hərəkət etmək üçün path ilə təmin edə bilərsiniz. Path-i təyin etsəniz, `reset` 1-ci addımı atlayacaq və hərəkətlərinin qalan hissəsini müəyyən bir fayl və ya fayllar dəsti ilə məhdudlaşdıracaqdır. Bu, həqiqətən bir növ məna kəsb edir — HEAD sadəcə bir göstəricidir və bir commit-in bir hissəsini və digərinin hissəsini göstərə bilməzsiniz. Lakin indeks və işləyən qovluq qismən yenilənə *bilər*, buna görə reset 2 və 3 addımlarla davam edir.

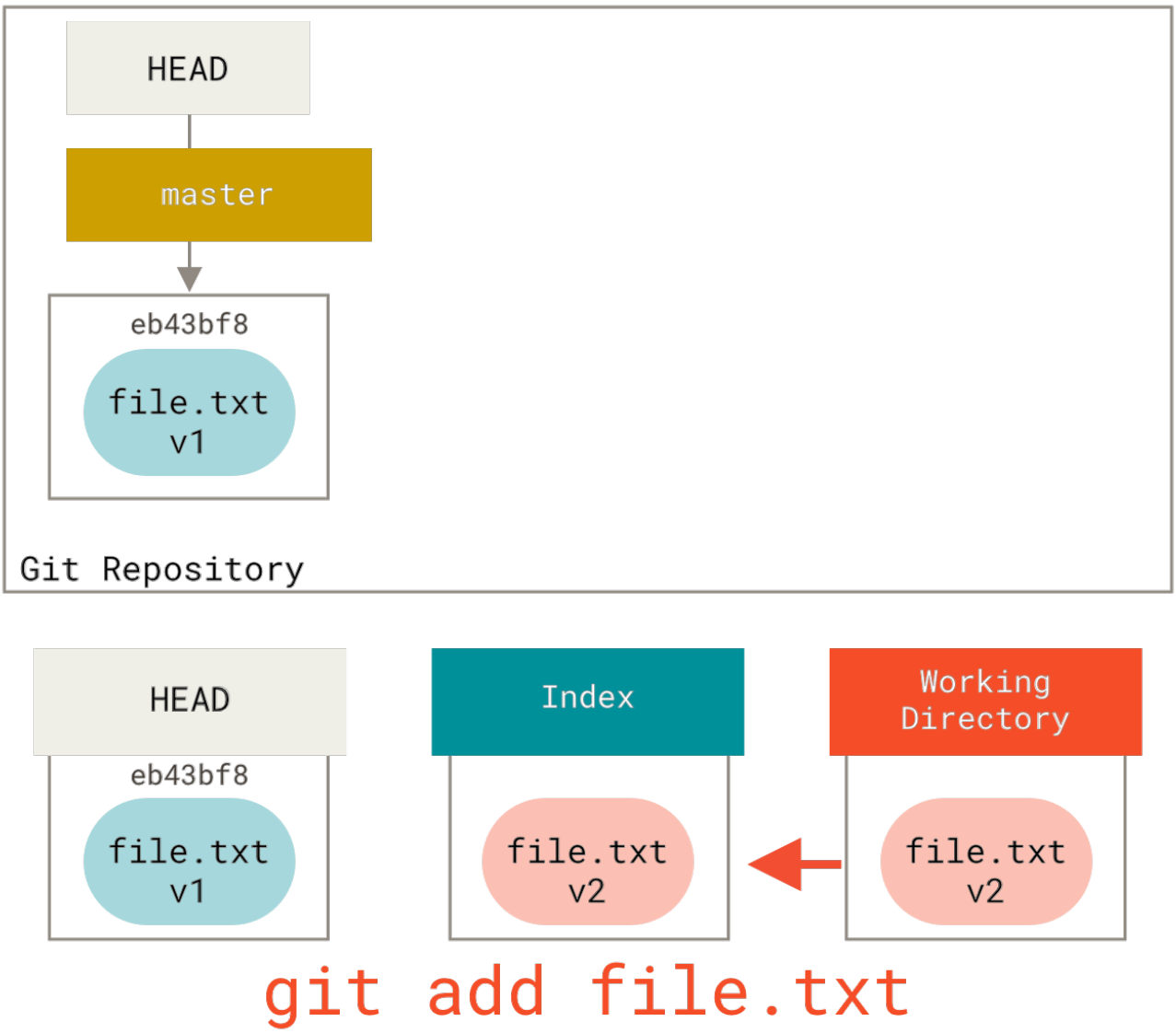
Beləliklə, `git reset file.txt`-u işlətdiyimizi fərz edək. Bu forma (bir SHA-1 və ya branch-ı göstərm ədiyinizə və `--soft` və ya `--hard` göstərmədiyinizə görə) `git reset --mixed HEAD file.txt` üçün stenoqramdır:

1. Branch-ı HEAD nöqtələrinə köçürün *(atıldı)*.
2. İndeksi HEAD kimi göstərin *(burada dayanın)*.

Beləliklə, bu, yalnız `file.txt`-ı HEAD-dan indeksə köçürür.

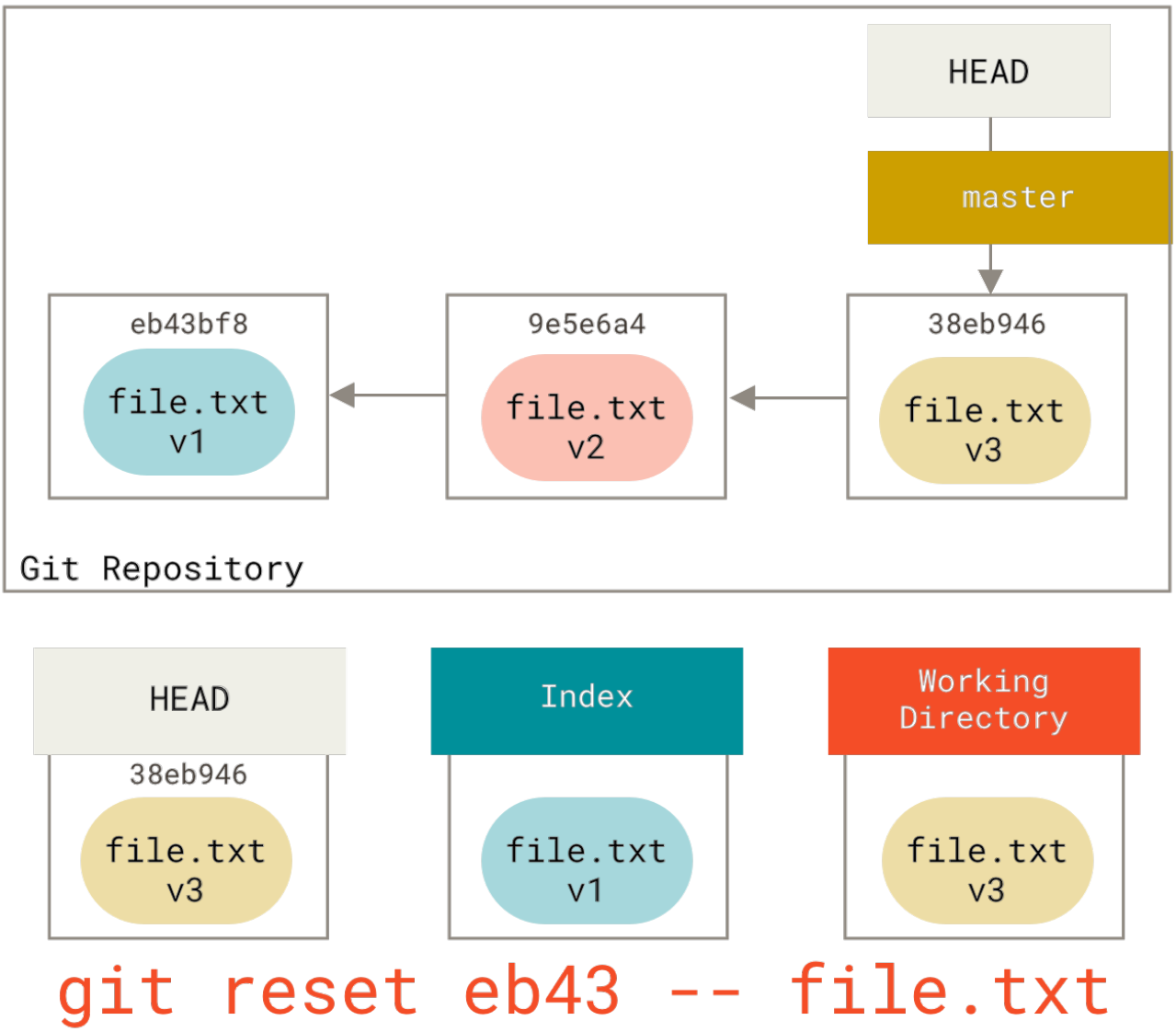


Bu, faylı *unstaging* etməyə təsir göstərir. Bu əmrin diaqramına baxsaq və `git add`-ın nə etdiyini düşünsək, onlar tam əkslərdir.



Buna görə `git status` əmrinin nəticəsi bir faylın açılmaması üçün işə başlamağınızı təklif edir (bu barədə daha çox məlumat üçün Mərhələli Bir Faylın Mərhələlərə Ayrılmaması-a baxın).

Git-in bu versiyasını çıxarmaq üçün müəyyən bir commit-i göstərərək “pull the data from HEAD” dediyini asanlıqla edə bilərik. Sadəcə `git reset eb43bf file.txt` kimi bir şey işlədərdik.



Bu, işin içindəki faylın məzmunu **v1**-ə qaytardığımız, üzərinə `git add` işlədib yenidən **v3**-ə qaytardığımız kimi eyni şeyi edir (həqiqətən bütün bu addımlardan keçmədən). İndi git commit-i işə salırıqsa, bu işi yenidən **v1**-ə qaytaran bir dəyişiklik qeyd edəcək, baxmayaraq ki, bu, heç işl ədiyimiz qovluqda heç olmamışdır.

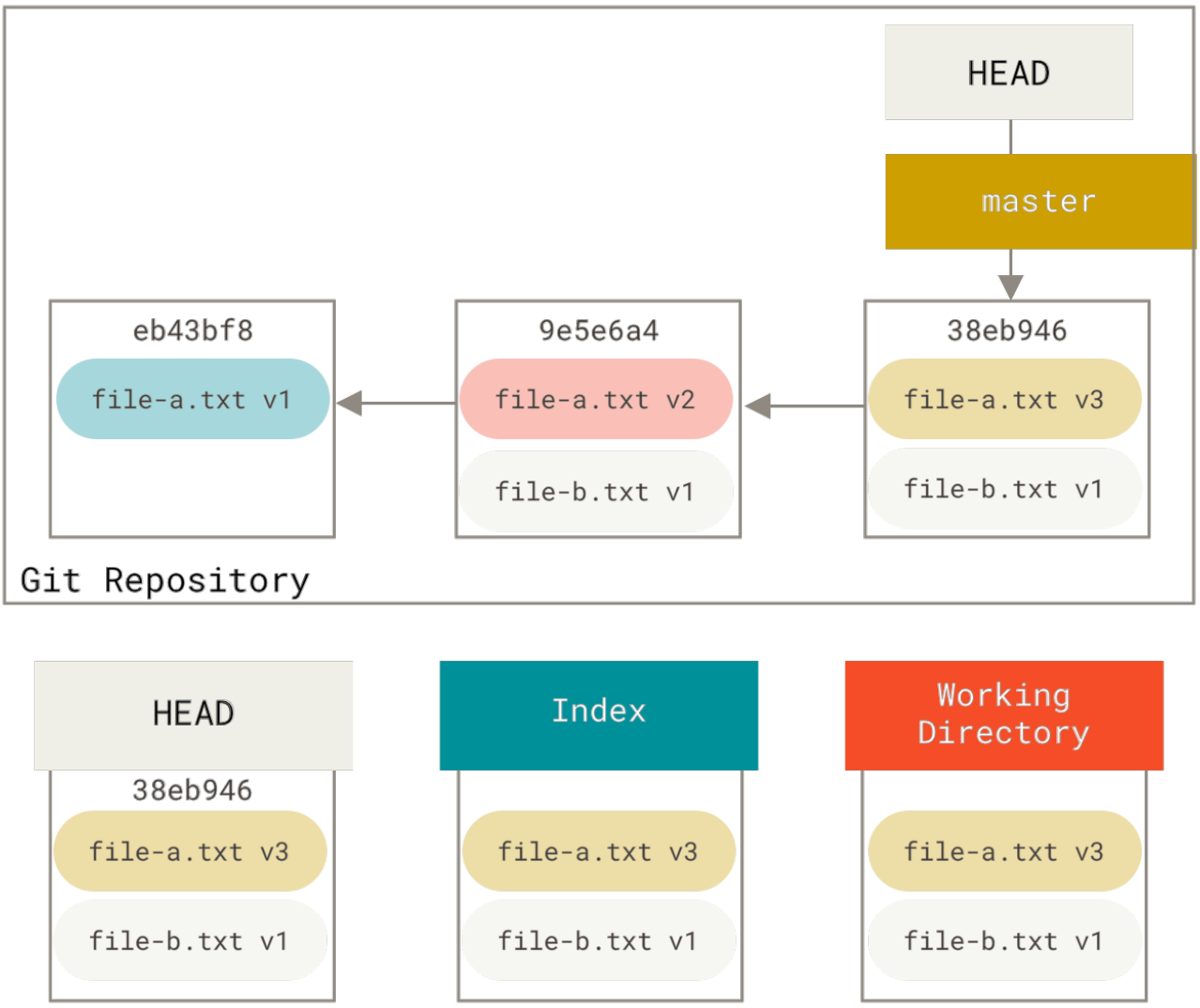
`git add` kimi, `reset` əmri, bir hunk-by-hunk əsasında məzmunu açmaq üçün `--patch` seçimini qəbul edəcəyi də maraqlıdır. Beləliklə, məzmunu seçmə şəkildə dayandıra və ya geri qaytara bilərsiniz.

Squashing

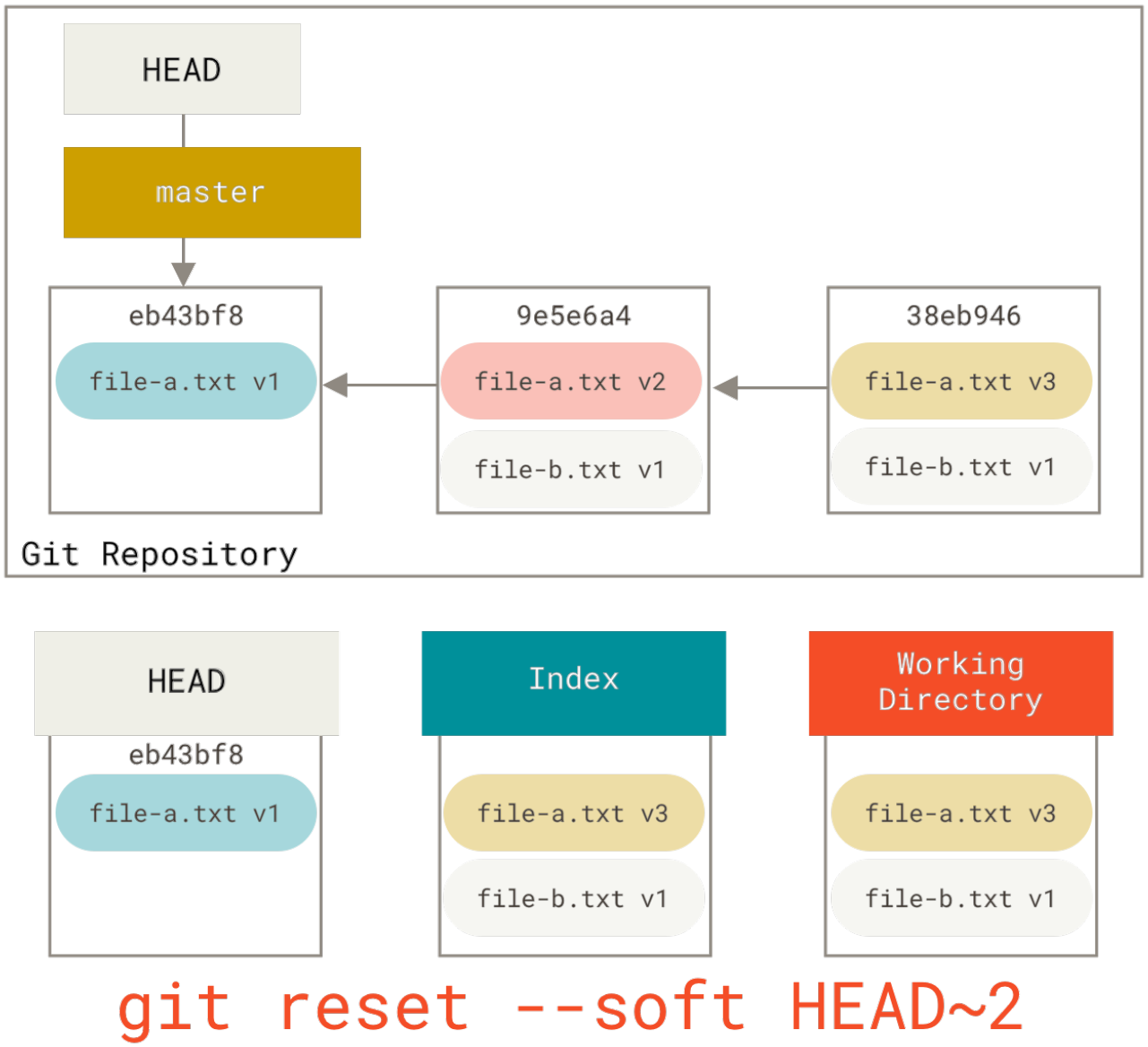
Gəlin bu yeni güc ilə necə maraqlı bir iş görəcəyimizə nəzər salaq.

“oops.”, “WIP” və “forgot this file” kimi bir sıra commit-niz olduğunu düşünək. Həqiqətən ağıllı görünməyinizi təmin edən `reset`-i tez və asanlıqla bir vahid commit halına gətirə bilərsiniz. Squashing Commits bunun başqa bir yolunu göstərir, lakin bu nümunədə `reset` tətbiq etmək daha asandır.

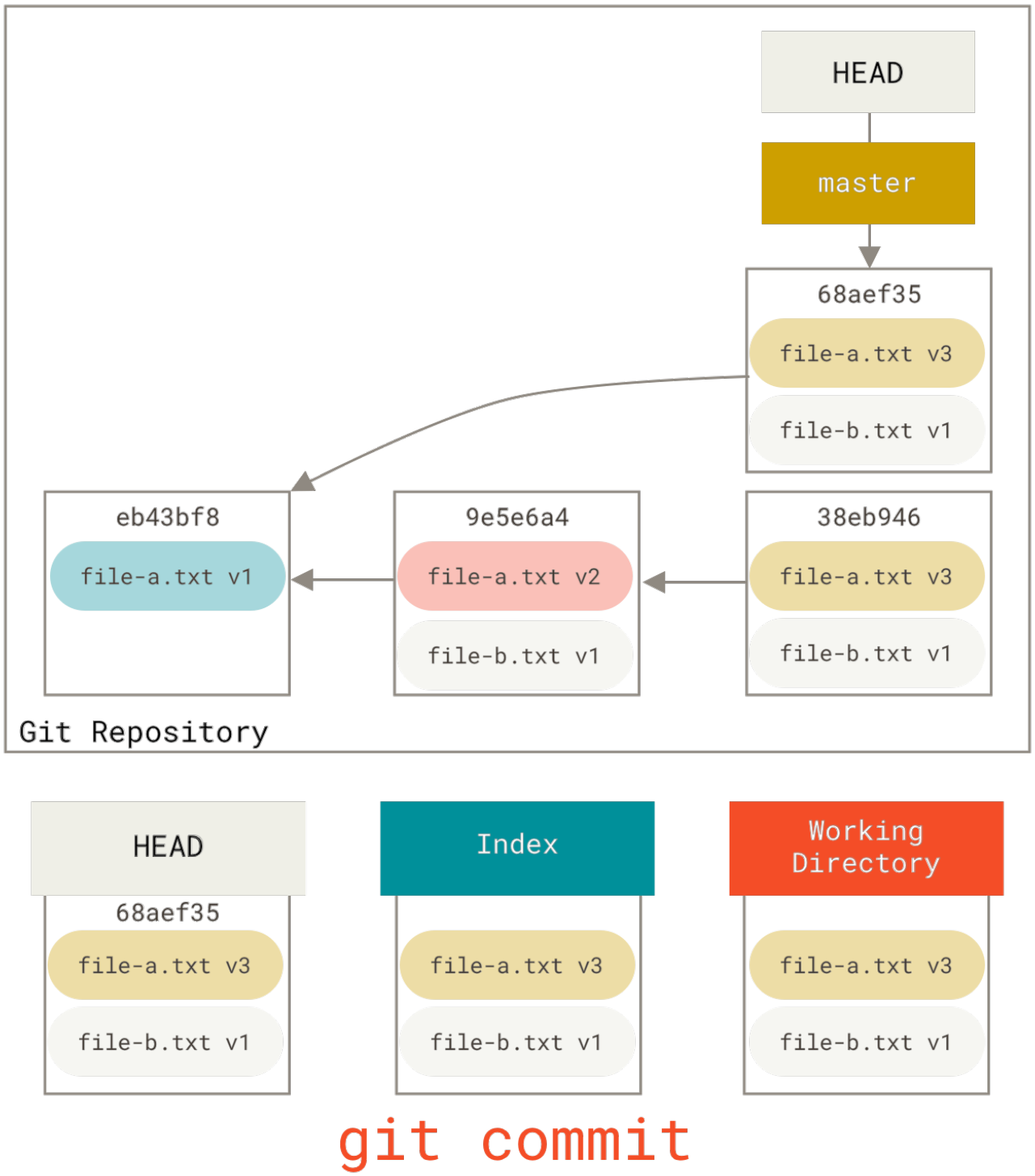
Deyək ki, birinci commit-in bir sənəd olduğu bir layihə var, ikincisi yeni bir fayl əlavə etdi və birincisini dəyişdirdi, üçüncü commit isə ilk sənədini yenidən dəyişdirdi. İkinci commit davam ed ən bir iş idi və siz onu squash istəyirsiniz.



HEAD branch-nı köhnə bir commit-ə (davam etdirmək istədiyiniz ən yeni commit-ə) köçürmək üçün `git reset --soft HEAD~2` tətbiq edə bilərsiniz.



Və sonra sadəcə yenidən `git commit`-i işə salın:



İndi əlçatan tarixçənizi, push edəcəyiniz tarixin indi bir `fayl-a.txt` v1 ilə iş görməyinizə, sonra da hər ikisinin `file-a.txt` faylını v3-ə dəyişdirib `fayl-b.txt` əlavə etməyinə bənzəyirsiniz. Faylın v2 versiyası ilə commit artıq tarixdə yoxdur.

Yoxlama

Nəhayət, `checkout` və `reset` arasındakı fərqin nə olduğunu düşünə bilərsiniz. `reset` kimi, `checkout` da üç ağacı idarə edir və əmrə bir fayl path verib verməməyinizə görə bir az fərqlidir.

Path-lar Olmadan

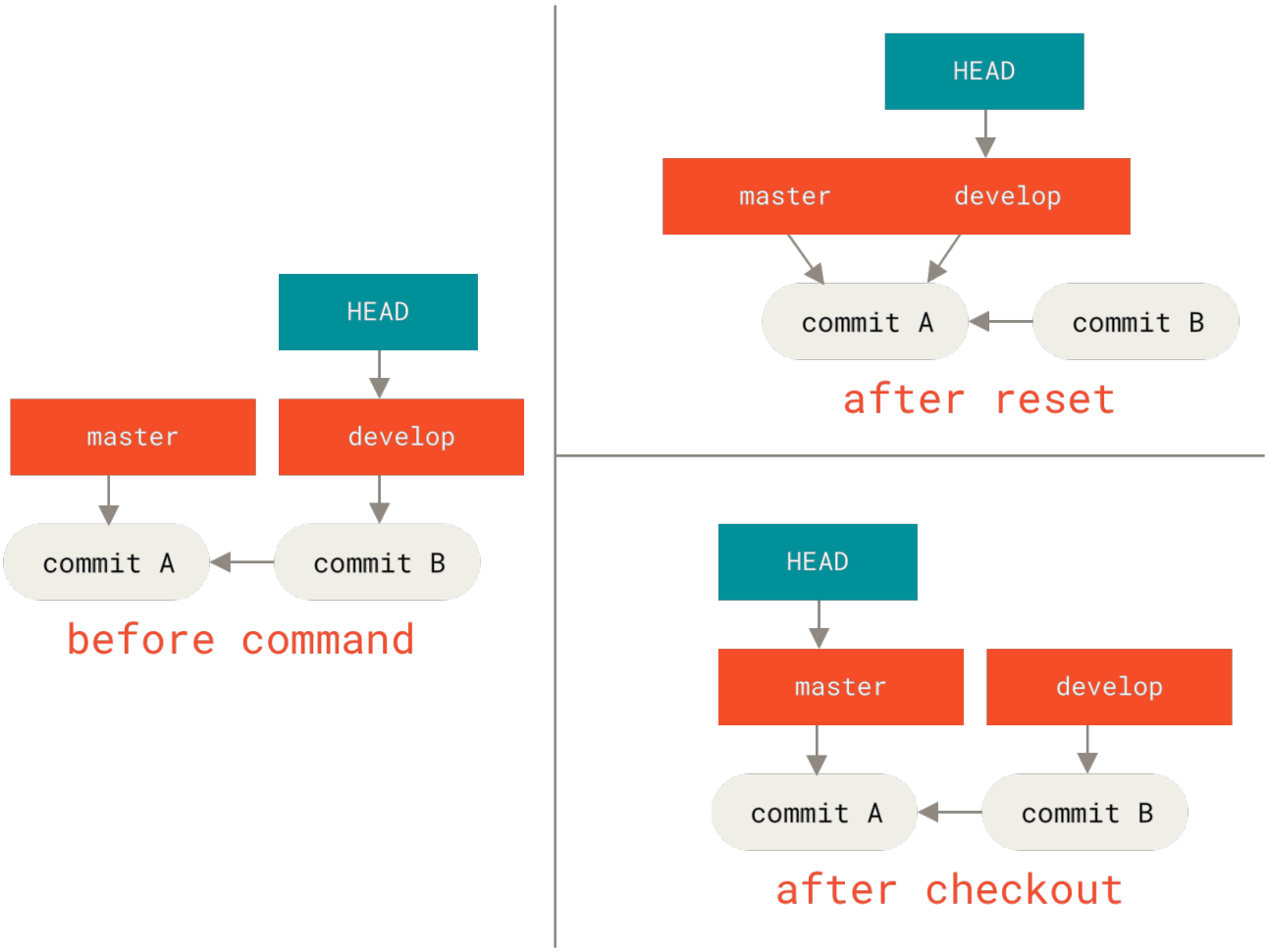
`git checkout [branch]` işlətməsi `git reset --hard [branch]` işlətməsinə bənzəyir, çünki `[branch]` kimi görünməyiniz üçün üç ağacı da yeniləyir, ancaq burada iki mühim fərq var.

Birincisi, `reset --hard`-dan fərqli olaraq, `checkout` iş üçün təhlükəsizdir; bu dəyişikliklər olan faylları üzə vurmadığından əmin olacaq. Əslində, o bundan bir az daha ağıllıdır - iş qovluğunda ə həmiyyətsiz bir birləşmə etməyə çalışır, belə ki dəyişdirmədiyiniz bütün fayllar yenilənəcəkdir. `reset --hard`, əksinə, lövhədə hər şeyi yoxlamadan əvəz edəcəkdir.

İkinci önəmli fərq, `checkout`-un HEAD-i necə yeniləməsidir. HEAD-ın göstərdiyi branch-ı yenidən h ərəkət etdirərkən, `checkout` başqa bir branch-a işarə etmək üçün HEAD-ı özünə aparacaqdır.

Məsələn, deyək ki, fərqli commit-lərə işarə edən `master` və `develop` branch-larımız var və hazırda `develop`-dayıq (ona görə də HEAD buna işarə edir). `git reset master` işləsək, `develop` özü indi `master` -in etdiyi eyni commit-ə işarə edəcəkdir. Bunun əvəzinə `git checkout master` işlətsək, `develop` hərə kət etmir, HEAD özü edir. HEAD indi `master`-ə işarə edəcəkdir.

Beləliklə, hər iki vəziyyətdə də HEAD-i A commit-ni göstərməyə yönəldirik, amma bunu *nec*ə etdiyimiz çox fərqlidir. `reset`, HEAD nöqtələrini hərəkətə gətirəcək, `checkout` HEAD-in özünü hərə kət etdirəcəkdir.



Path-larla Birlikdə

`checkout`-u işlətmənin başqa yolu, `reset` kimi HEAD hərəkət etməyən bir file path-dır. Eynilə `git reset [branch] file` faylına bənzəyir, indeksi həmin sənədlə həmin faylda yeniləyir, eyni zamanda iş qovluğundakı faylın üzərinə yazır. Tam olaraq `git reset --hard [branch] file` kimi olardı (`reset` onu işlətməyinizə imkan verərsə) - iş qovluğu təhlükəsiz deyil və HEAD hərəkət etmir.

Ayrıca, `git reset` və `git add` kimi, `checkout` seçilmiş şəkildə bir hunk-by-hunk əsasında fayl m əzmununu geri qaytarmağınız üçün bir `--patch` seçimi qəbul edəcəkdir.

Məzmun

Ümid edirik ki, indi `reset` əmri başa düşürsünüz və daha rahat hiss edirsiniz, amma bunun yəqin ki, `checkout`-dan nə dərəcədə fərqləndiyinə dair bir az qarışıq və bəlkə də fərqli çağırışların bütün qaydalarını xatırlaya bilmirsiniz.

Burada əmrlərin hansı ağaclara təsir etdiyini göstərən bir cheat-sheet var. “HEAD” sütununda, əmr, rəhbərin göstərdiyi istinadı (branch-ı) hərəkət etdirərsə, “HEAD” ifadəsini işlədiyi təqdirdə, “REF” və oxunuşunu özü idarə edərsə “HEAD” oxuyur. *WD Safe?* mövzusuna xüsusi diqqət yetirin. sütun - əgər **YOX** deyirsə, bu əmri işə salmadan əvvəl düşünün.

	HEAD	**Index**	**Workdir**	**WD Safe?**
Commit Level				
`reset --soft [commit]`	REF	NO	NO	YES
`reset [commit]`	REF	YES	NO	YES
`reset --hard [commit]`	REF	YES	YES	**NO**
`checkout <commit>`	HEAD	YES	YES	YES
File Level				
`reset [commit] <paths>`	NO	YES	NO	YES
`checkout [commit] <paths>`	NO	YES	YES	**NO**

İnkişaf etmiş Birləşmə

Git-də birləşdirmə əsasən olduqca asandır. Git başqa bir branch-ı dəfələrlə birləşdirməyi asanlaşdırdığından, uzunmüddətli bir branch-a sahib ola biləcəyinizi ifadə edir, ancaq irəlil ədiyiniz müddətdə onu saxlaya bilərsiniz, yəni, çox böyük bir konfliktə təəccüblənməkdənsə kiçik münaqişələri tez-tez həll etməklə, seriyanında sonunda böyük problemlərdən yan keçə bilərsiniz.

Ancaq bəzən çətin konfliktlər əmələ gəlir. Bəzi digər versiyalara nəzarət sistemlərindən fərqli olaraq, Git münaqişələrin həlli məsələsində həddən artıq ağıllı olmağa çalışmır. Git-in fəlsəfəsi birləşmə həllinin nə vaxt olacağını müəyyənləşdirməkdə ağıllı olmaqdır, ancaq konflikt varsa, avtomatik olaraq həll etmək üçün ağıllı olmağa çalışmır. Buna görə də, tez ayrılan iki branch-ı birl əşdirmək üçün çox gözləsəniz, bəzi problemlərlə qarşılaşa bilərsiniz.

Bu hissədə bəzi problemlərin nədən ibarət olacağını və Git-in bu daha çətin vəziyyətləri həll etmək üçün sizə hansı vasitələri verdiyini nəzərdən keçirəcəyik. Ayrıca edə biləcəyiniz fərqli, qeyri-

standart birləşmələrin bəzilərini də əhatə edəcəyik, həm də etdiyiniz birləşmələri necə geri çəkə biləcəyimizə baxacağıq.

Konfliktləri Birləşdirmə

Daha mürəkkəb konfliktlər üçün Əsas Birləşmə Konfiliktləri-də birləşmə konfliktlərinin həlli ilə bağlı bəzi əsasları izah etsək də, Git nəyin baş verdiyini və münaqişəni daha yaxşı necə həll edəc əyinizi anlamağa kömək edəcək bir neçə vasitə təqdim edir.

Əvvəlcə, əgər mümkündürsə, konfliktlər ola biləcək birləşməni etməzdən əvvəl işçi qovluğunun t əmiz olduğundan əmin olun. İşiniz davam edirsə, ya müvəqqəti bir branch-a verin və ya zibilə atın. Bu, burada etdiyiniz **hər şeyi** geri qaytara biləcəyiniz üçün edir. Birləşdirməyə çalışdığınız zaman iş qovluğunda qeyd olunmamış dəyişikliklər varsa, bu məsləhətlərdən bəziləri bu işi qorumağa kömək edə bilər.

Çox sadə bir misal verək. Üzərində *Hello world* yazdıran super sadə bir Ruby faylımız var.

```
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end

hello()
```

Qovluğumuzda `whitespace` adlı yeni bir branch yaradırıq və bütün Unix sətir sonlarını DOS xətt sonlarına dəyişdirərək sənədin hər sətirini ancaq whitespace ilə dəyişdiririk. Sonra “hello world” yazısını “hello mundo” olaraq dəyişdiririk.

```
$ git checkout -b whitespace
Switched to a new branch 'whitespace'

$ unix2dos hello.rb
unix2dos: converting file hello.rb to DOS format ...
$ git commit -am 'Convert hello.rb to DOS'
[whitespace 3270f76] Convert hello.rb to DOS
 1 file changed, 7 insertions(+), 7 deletions(-)

$ vim hello.rb
$ git diff -b
diff --git a/hello.rb b/hello.rb
index ac51efd..e85207e 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
 #! /usr/bin/env ruby

 def hello
-  puts 'hello world'
+  puts 'hello mundo'^M
 end

 hello()

$ git commit -am 'Use Spanish instead of English'
[whitespace 6d338d2] Use Spanish instead of English
 1 file changed, 1 insertion(+), 1 deletion(-)
```

İndi yenidən `master` branch-a qayıdırıq və funksiya üçün bəzi sənədlər əlavə edirik.

```
$ git checkout master
Switched to branch 'master'

$ vim hello.rb
$ git diff
diff --git a/hello.rb b/hello.rb
index ac51efd..36c06c8 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
   puts 'hello world'
 end

$ git commit -am 'Add comment documenting the function'
[master bec6336] Add comment documenting the function
 1 file changed, 1 insertion(+)
```

İndi isə `whitespace` branch-da birləşdirmə etməyə çalışırıq və whitespace dəyişikliklərinə görə konfliktlər görəcəyik.

```
$ git merge whitespace
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

Birləşməni Ləğv etmək

İndi bizim bir neçə seçimimiz var. Əvvəlcə bu vəziyyətdən necə çıxacağımızı izah edək. Əgər siz konfliktlərin olacağını gözləmirdinizsə və vəziyyətlə hələ çox işləmək istəmirsinizsə, sadəcə `git merge --abort` ilə birləşməni ləğv edə bilərsiniz.

```
$ git status -sb
## master
UU hello.rb

$ git merge --abort

$ git status -sb
## master
```

`git merge --abort` seçimi birləşmədən əvvəlki vəziyyətə qayıtmağa çalışır. Onun mükəmməl bir şəkildə işləyə bilməyəcəyi yeganə hallar, işlədiyiniz qovluqda açılmamış, buraxılmamış dəyişikliklərin olması ola bilər, əks halda yaxşı işləməlidir.

Hər hansı bir səbəbdən yenidən başlamaq istəsəniz, təkrar `git reset --hard HEAD` işlədə bilərsiniz və bu zaman depolarınız son vəziyyətinə qaytarılacaq. Yadda saxlayın ki, hər hansı bir iş itirilə bil ər, ona görə də dəyişikliklərinizdən heç birini istəmədiyinizdən əmin olun.

Whitespace-ə Məhəl Qoymamaq

Bu konkret vəziyyətdə, konfliktlər whitespace ilə əlaqədardır. Bunu bilirik, çünki məsələ sadədir, ancaq real vəziyyətlərdə konfliktə baxanda izah etmək çox asandır ki, hər xətt bir tərəfdən çıxarılır və digər tərəfdən yenidən əlavə olunur. Default olaraq, Git bu sətirlərin hamısının d əyişdirildiyini görür, buna görə də faylları birləşdirə bilmir.

Default birləşdirmə strategiyası arqumentlər götürə bilər və onlardan bir neçəsi whitespace d əyişikliklərinə məhəl qoymur. Birləşmədə çox sayda whitespace probleminizin olduğunu görsəniz, bu dəfə `-Xignore-all-space` və ya `-Xignore-space-change` ilə yenidən ləğv edib təkrar edə bilərsiniz. Birinci seçim xətləri müqayisə edərkən whitespace-i **tamamil**ə gözdən keçirir, ikincisi bir və ya daha çox whitespace simvollarının ardıcıllığını ekvivalent olaraq qəbul edir.

```
$ git merge -Xignore-space-change whitespace
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Bu vəziyyətdə, faktiki fayl dəyişiklikləri bir-birinə zidd olmadığına görə, whitespace dəyişikliklərini görməməzlikdən gəldikdə, hər şey yaxşı birləşir. Komandanızda bəzən boşluqlardan nişanlara reformat etməyi sevən və ya əksinə edən biri varsa, bu sizin üçün bir xilaskardır.

Manual Faylı Yenidən Birləşdirmə

Git whitespace-i əvvəlcədən emal etməyi bacarsa da, dəyişikliklərin digər növləri vardır ki, onlarda Git avtomatik idarə edə bilmir, lakin dəyişdirilə bilən düzəlişlərdir. Misal olaraq, Git'in whitespace dəyişikliyini həll edə bilmədiyini iddia edək və bunu manual şəkildə edək.

Həqiqətən etməli olduğumuz şey, faktiki faylı birləşdirməyə cəhd etmədən əvvəl `dos2unix` proqramı ilə birləşdirməyə çalışdığımız faylı işə salmaqdır. Bəs bunu necə edə bilərik?

Əvvəlcə birləşmə konflikti vəziyyətinə giririk. Sonra, mənim versiyamın nüsxələrini, onların versiyasını (birləşdirdiyimiz branch-dan) və ümumi versiyadan (hər iki tərəfin branch-dan çıxdığı yerdən) surətlərini almaq istəyirik. Sonra onların tərəfini və ya öz tərəfimizi düzəltmək istəyirik və yenidən bu tək fayl üçün yenidən birləşdirməyə çalışırıq.

Üç fayl versiyasını əldə etmək əslində olduqca asandır. Git, bu versiyaların hamısını əlaqəli nömrə ləri olan "stages" indeksində saxlayır. Stage 1 ortaq kökdür, stage 2 sizin versiyanızdır və stage 3 `MERGE_HEAD`-dan, birləşdiyiniz versiyadır ("theirs").

Konfliktli faylın bu versiyalarının hər birinin bir nüsxəsini git show əmri və xüsusi bir sintaksis ilə çıxara bilərsiniz.

```
$ git show :1:hello.rb > hello.common.rb
$ git show :2:hello.rb > hello.ours.rb
$ git show :3:hello.rb > hello.theirs.rb
```

Bir az daha sərt keçid əldə etmək istəyirsinizsə, bu faylların hər biri üçün Git bloklarının əsl SHA-1-lərini əldə etmək üçün `ls-files -u` Plumbing əmrindən də istifadə edə bilərsiniz.

```
$ git ls-files -u
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1   hello.rb
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2   hello.rb
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3   hello.rb
```

`:1:hello.rb` SHA-1 çubuğunu axtarmaq üçün sadəcə bir stenddir.

İndi işlədiyimiz qovluqda hər üç mərhələnin məzmunu olduğundan, whitespace məsələsini həll etmək üçün onları manual şəkildə düzəldə bilərik və yalnız bunu az tanınan `git merge-file` əmri ilə faylı yenidən birləşdirə bilərik.

```
$ dos2unix hello.theirs.rb
dos2unix: converting file hello.theirs.rb to Unix format ...

$ git merge-file -p \
    hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb

$ git diff -b
diff --cc hello.rb
index 36c06c8,e85207e..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,8 -1,7 +1,8 @@@
  #! /usr/bin/env ruby

 +# prints out a greeting
  def hello
-   puts 'hello world'
+   puts 'hello mundo'
  end

  hello()
```

Bu anda faylı qəşəng bir şəkildə birləşdirdik. Əslində, bu, `ignore-space-change` seçimindən daha yaxşı işləyir, çünki bu, sadəcə boş yerə dəyişiklik etmədən yerinə boşluq dəyişikliklərini düzəldir. `ignore-space-change` birləşməsində, işləri həqiqətən qarışıq hala gətirərək DOS xətti ucları ilə bir neçə xətt əldə edirik.

Əslində bir tərəf və ya digəri arasında dəyişdirilənin nə olduğu ilə əlaqədar bu tapşırığı bitirmədən əvvəl bir fikir əldə etmək istəyirsinizsə, `git diff`-dan bu mərhələlərdən hər hansı birində birl

əşdirmə nəticəsində iş qovluğunuzda nə əmri verdiyinizi müqayisə etməsini istəyə bilərsiniz. Hamısını izah edək.

Nəticəni birləşdirmədən əvvəl branch-nızda nə əldə etdiyinizi görmək, başqa sözlə birləşmənin n əyə tətbiq olunduğunu görmək üçün `git diff --ours` proqramını işlədə bilərsiniz.

```
$ git diff --ours
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index 36c06c8..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -2,7 +2,7 @@

 # prints out a greeting
 def hello
-  puts 'hello world'
+  puts 'hello mundo'
 end

 hello()
```

Beləliklə, burada asanlıqla görə bilərik ki, branch-mızda baş verənlər, əslində bu birləşmə ilə bu faylı tanıdığımız vahid xətti dəyişdirir.

Birləşmənin nəticəsinin onların tərəfindəki vəziyyətdən necə fərqləndiyini görmək istəyiriksə, `git diff --theirs` işlədə bilərik. Bu və aşağıdakı misalda biz təmizlənmiş `hello.theirs.rb` faylı ilə deyil, Git-də olanlarla müqayisə etdiyimiz üçün whitespace-dən çıxarmaq üçün `-b` istifadə etməliyik.

```
$ git diff --theirs -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index e85207e..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
   puts 'hello mundo'
 end
```

Sonda siz faylın `git diff --base` ilə hər iki tərəfdən necə dəyişdiyini görə bilərsiniz.

```
$ git diff --base -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index ac51efd..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,8 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
-  puts 'hello world'
+  puts 'hello mundo'
 end

 hello()
```

Bu zaman `git clean` əmrini manual şəkildə birləşdirmək üçün yaratdığımız, lakin artıq lazım olmayan əlavə sənədləri silmək üçün istifadə edə bilərik.

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

Konfliktləri Yoxlamaq

Ola bilsin ki, indi hansısa səbəbdən həll ilə razı deyilik və ya bir və ya hər iki tərəfdən manual şəkildə düzəltmək yenə də yaxşı işləmədi və bizim daha çox kontekstə ehtiyacımız var.

Gəlin nümunəni bir az dəyişək. Bu nümunə üçün, hər ikisində bir neçə əmr yerinə yetirən, lakin birləşdikdə qanuni məzmun konflikti yaradan iki daha uzun branch var.

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) Update README
* 9af9d3b Create README
* 694971d Update phrase to 'hola world'
| * e3eb223 (mundo) Add more tests
| * 7cff591 Create initial testing script
| * c3ffff1 Change text to 'hello mundo'
|/
* b7dcc89 Initial hello world code
```

İndi yalnız üç `master` branch-da və üç `mundo` branch-da yaşayan unikal əmrlər var. `mundo` branch-ı birləşdirməyə çalışarkən konflikt yaranır.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

Biz birləşdirmə konfliktinin nə olduğunu görmək istəyirik. Bu zaman faylı açsaq, bu kimi bir şey görərik:

```
#! /usr/bin/env ruby

def hello
<<<<<<< HEAD
  puts 'hola world'
=======
  puts 'hello mundo'
>>>>>>> mundo
end

hello()
```

Birləşmənin hər iki tərəfi bu fayla məzmun əlavə etdi, lakin bəziləri bu konfliktə səbəb olan yeri d əyişdirdi.

Bu konfliktin necə yarandığını müəyyənləşdirmək üçün indi əlinizdə olan bir neçə vasitəni araşdıraq. Bəlkə də bu konflikti dəqiq necə həll edə biləcəyiniz tam bəlli deyil. Daha çox kontekstə ehtiyacınız var.

Digər bir faydalı vasitə `--conflict` seçimi ilə `git checkout`-dur. Bu, faylı yenidən yoxlayır və konflikt birləşmə işarələrini dəyişdirir. O həm də, markerləri yenidən qurmaq və yenidən həll etmək istəs əniz də faydalı ola bilər.

Siz `--conflict` ya `diff3` ya da `merge` (standart olandır) keçirə bilərsiniz. Əgər siz onu `diff3`-ə keçirs əniz, bu zaman Git konflikt markerlərinin bir az fərqli versiyasını istifadə edəcək, nəinki “ours” və “theirs” versiyalarını, həm də daha çox kontekst vermək üçün “base” versiyasını əlavə edəcək.

```
$ git checkout --conflict=diff3 hello.rb
```

Bunu işlədikdən sonra əvəzində fayl belə görünəcək:

```
#! /usr/bin/env ruby

def hello
<<<<<<< ours
  puts 'hola world'
||||||| base
  puts 'hello world'
=======
  puts 'hello mundo'
>>>>>>> theirs
end

hello()
```

Bu formatı bəyənsəniz, `merge.conflictstyle` parametrini `diff3`-ə qoyaraq gələcək birləşdirmə konfliktləri üçün standart olaraq təyin edə bilərsiniz.

```
$ git config --global merge.conflictstyle diff3
```

`git checkout` əmri variantları birləşdirmədən yalnız bir tərəfi və ya digərini seçmək üçün həqiqət ən sürətli bir yol olan `--ours` və `--theirs` seçimlərini də istifadə edə bilərsiniz.

Bu, sadəcə bir tərəfi seçə biləcəyiniz və ya müəyyən bir faylları başqa bir branch-dan birləşdirmək istədiyiniz ikili sənədlərin ixtilafları üçün xüsusilə faydalı ola bilər. Yəni, birləşdirmədən və əmr vermədən əvvəl müəyyən faylları bir tərəfdən və ya digər tərəfdən yoxlamaq olar.

Birləşdirmə Log-u

Birləşmə konfliktlərink həll edərkən başqa bir faydalı vasitə isə `git log`-dur. Bu, konfliktlərə səbəb olan mövzularda kontekst əldə etməyə kömək edə bilər. İki inkişaf xəttinin eyni kod sahəsinə toxunduğunu xatırlamaq üçün bir az tarixə nəzər salmaq bəzən faydalı ola bilir.

Bu birləşmədə iştirak edən hər iki branch-a daxil olan bütün unikal əmrlərin tam siyahısını əldə etmək üçün Üçqat Nöqtə-da öyrəndiyimiz “triple dot” sintaksisindən istifadə edə bilərik.

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
< f1270f7 Update README
< 9af9d3b Create README
< 694971d Update phrase to 'hola world'
> e3eb223 Add more tests
> 7cff591 Create initial testing script
> c3ffff1 Change text to 'hello mundo'
```

Bu iştirak olunan altı ümumi tapşırığın, habelə hər bir inkişafın hansı xətt üzərində olmasının göz əl bir siyahısıdır.

Daha konkret kontekst vermək üçün bunu daha da asanlaşdıra bilərik. `git log`-a `--merge` seçimini

əlavə etsək, yalnız konflikt yaradan bir fayla toxunan birləşmənin hər iki tərəfindəki əmrləri göst ərəcəkdir.

```
$ git log --oneline --left-right --merge
< 694971d Update phrase to 'hola world'
> c3ffff1 Change text to 'hello mundo'
```

Bunun əvəzinə `-p` seçimi ilə işləsəniz, konfliktlə nəticələnən fayldan yalnız fərqi əldə edirsiniz. Bu, nəyə görə bir şeyin zidd olduğunu və onu daha ağıllı şəkildə həll etməyinizi başa düşməyinizə kömək etmək üçün lazım olan konteksti verməkdə **həqiqətən** faydalı ola bilər.

Kombinə olunmuş Diff Formatı

Git-in uğurlu olan birləşmə nəticələri mərhələli olduğundan, konflikt birləşdirmə vəziyyətində `git diff` işlətsəniz hələ də konflikdə olanı əldə edəcəksiniz. Bu hələ nəyi həll etməli olduğunuzu görm ək üçün faydalı ola bilər.

Birləşdirmə konfliktindən sonra birbaşa `git diff` işlətdiyiniz zaman sizə olduqca unikal fərqli f ərqli bir çıxış formatında məlumat verəcəkdir.

```
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,11 @@@
  #! /usr/bin/env ruby

  def hello
++<<<<<<< HEAD
 +  puts 'hola world'
++=======
+   puts 'hello mundo'
++>>>>>>> mundo
  end

  hello()
```

Format “Combined Diff” adlanır və hər bir sətrin yanında iki məlumat sütunu verir. Birinci sütun, bu sətir “ours” branch-mızla işləyən qovluğunuzdakı fayl arasında fərqli olduğunu (əlavə edilmiş və ya silinmiş) göstərir, ikinci sütun isə “theirs” branch ilə iş qovluğunun surətinin eyni olduğunu göstərir.

Beləliklə, bu misalda görə bilərsiniz ki, `<<<<<<<` and `>>>>>>>` xətləri işləmə nüsxəsindədir, lakin birl əşmənin hər iki tərəfində deyil. Bu o mənanı verir ki, birləşmə vasitəsi kontekstimiz üçün onları oraya yapışdırıb saxladı, lakin biz onları silməyi gözləyirdik.

Konflikti həll edib yenidən `git diff` tətbiq etsək, eyni şeyi görəcəyik, amma bu bir az daha

faydalıdır.

```
$ vim hello.rb
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
 -  puts 'hello mundo'
++  puts 'hola mundo'
  end

  hello()
```

Bu bizə göstərir ki, “hola world' işləmə nüsxəsində deyil, amma bizim tərəfimizdə idi, ``hello mundo” isə onların tərəfində idi, amma işləmə nüsxəsində deyildi və nəhayət “hello mundo” hər iki tərəfdə olmadığını halda işləyən nüsxədə idi. Bu qərar vermədən əvvəl nəzərdən keçirmək üçün faydalı ola bilər.

Siz bütün birləşdirmələrdə faktdan sonra məsələlərin necə həll olunduğunu görmək üçün `git log` -dan istifadə edə bilərsiniz. Birləşdirmə `git show`-u işlətdiyiniz təqdirdə və ya bir `git log -p`-ə `--cc` seçimi əlavə etsəniz, Git bu formatı çıxaracaqdır (bu standart olaraq yalnız birləşməmələr üçün patch-ları göstərir)

This shows us that “hola world” was in our side but not in the working copy, that “hello mundo” was in their side but not in the working copy and finally that “hola mundo” was not in either side but is now in the working copy. This can be useful to review before committing the resolution.

```
$ git log --cc -p -1
commit 14f41939956d80b9e17bb8721354c33f8d5b5a79
Merge: f1270f7 e3eb223
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Sep 19 18:14:49 2014 +0200

    Merge branch 'mundo'

    Conflicts:
        hello.rb

diff --cc hello.rb
index 0399cd5,59727f0..e1d0799
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
-  puts 'hello mundo'
++  puts 'hola mundo'
  end

  hello()
```

Birləşdirməni Ləğv Etmək

Birləşdirmə əmrini necə yaratmağı bildiyiniz üçün səhv etmə ehtimalınız var. Git ilə işləməyin ən yaxşı tərəflərindən biri də odur ki, səhv etmək olar, çünki onları düzəltmək mümkündür (və bir çox hallarda asandır).

Birləşmə əmrləri heç fərqlənmir. Deyək ki, bir mövzu branch-da işə başlamısınız, təsadüfən onu `master`-ə birləşdirdiniz və indi əmr tarixçəniz belə görünür:

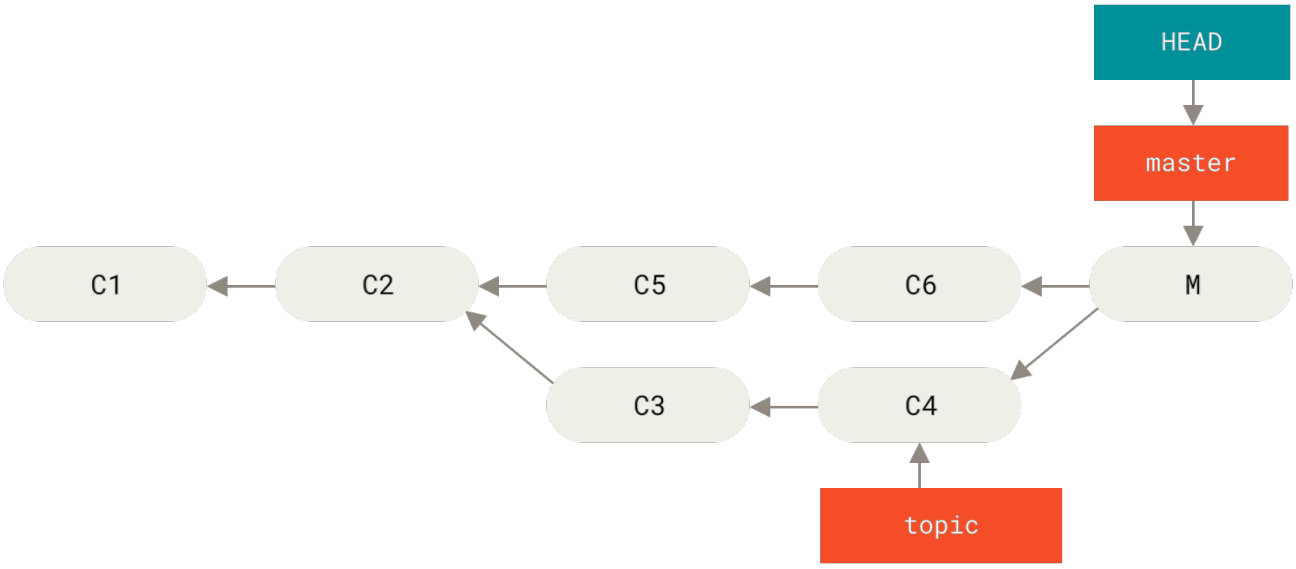


Figure 138. Təsadüfi birləşdirmə commit-i

İstədiyiniz nəticənin nə olduğundan asılı olaraq bu problemə yaxınlaşmanın iki yolu var.

Referansları Düzəltmək

İstənilməyən birləşmə əməliyyatı yalnız yerli depolarınızda varsa, ən asan və ən yaxşı həll yolu istədiyiniz yeri göstərmələri üçün branch-ları köçürməkdir. Əksər hallarda, `git reset --hard HEAD~` ilə səhv edilmiş `git merge`-i izləsəniz, bu branch göstəricilərini yenidən quracaq və buna görə də onlar belə görünəcəklər:

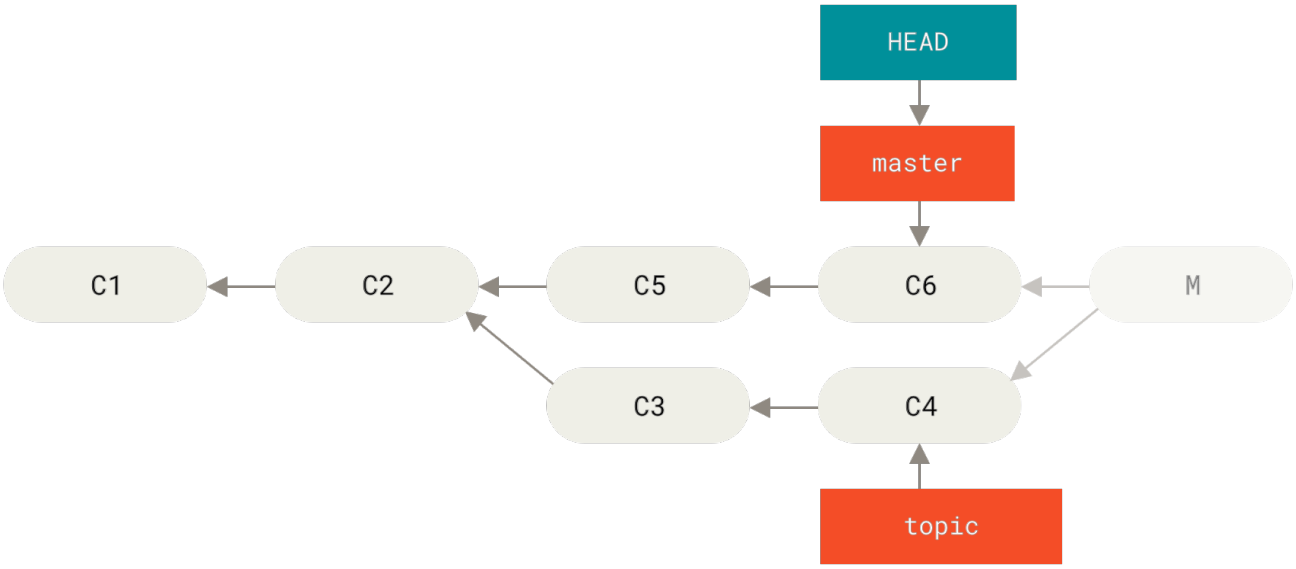


Figure 139. `git reset --hard HEAD~`-dan sonra tarixçə

Geri qayıtmağı Reset Demystified-də izah etdik, buna görə burada nələrin baş verdiyini anlamaq çox çətin olmayacaq. Sürətli bir xatırlatma: `reset --hard` ümumiyyətlə üç addımdan keçir:

1. Branch-ın HEAD nöqtələrini bu yerə köçürün. Bu vəziyyətdə, `master` birləşmədən (`C6`) əvvəl olduğu yerə köçürmək istəyirik.
2. İndeksi HEAD kimi göstərin.

3. İşləmə qovluğunu indeks kimi göstərin.

Bu yanaşmanın mənfi tərəfi ortaq bir depo ilə problem yarada bilən tarixin yenidən yazılmasıdır. Nə baş verə biləcəyi barədə daha çox məlumat almaq üçün Rebasing-in Təhlükələri-ə baxın; qısa versiyası odur ki, başqa insanların yazdığınız əmrləri varsa, yəqin ki, `reset` etməkdən çəkinm əlisiniz. Birləşdirmədən bəri başqa əmrlər yaradılıbsa bu yanaşma da işləməyəcək; refs hərəkət bu dəyişiklikləri itirəcəkdir.

Commit-ləri Tərs Çevirmək

Əgər ətrafdakı branch işarətçisini sizin üçün işləməyəcəksə, Git, hazırda mövcud olandan bütün d əyişiklikləri ləğv edən yeni bir commit vermək seçimi verir. Git ssenaridə bu əməliyyatı “revert” adlandırır və siz onu bu şəkildə çağıracaqsınız:

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

`-m 1` flag-ı hansı valideynin “mainline” olduğunu göstərir və saxlanılmalıdır. `HEAD`-a (`git merge topic`) birləşdirməyə çağırdığınız zaman, yeni əmrin iki valideyni olur: birincisi - `HEAD` (`C6`), ikincisi - (`C4`) birləşən branch-ın ucu. Bu vəziyyətdə, bütün məzmunu 1 saylı valideyndən (`C6`) qorumaqla, 2 saylı valideynə (`C4`) birləşdirməklə daxil edilmiş bütün dəyişiklikləri geri qaytarmaq istəyirik. Commiti geri alma tarixi belə görünür:

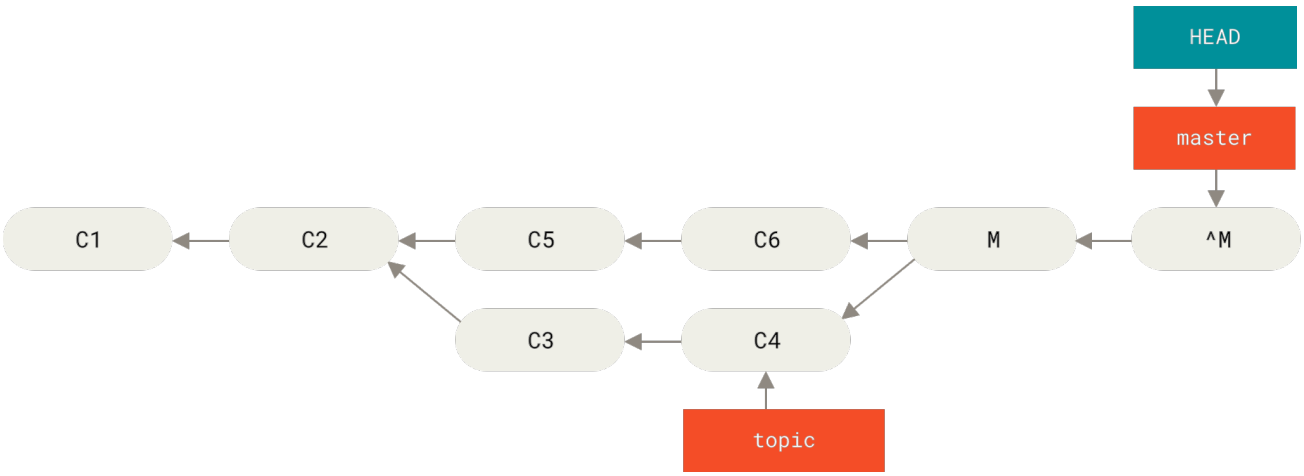


Figure 140. `git revert -m 1`-dən sonrakı tarixçə

Yeni commit `^M` `C6` ilə eyni məzmuna malikdir, buna görə də əgər buradan başlayaraq heç bir birl əşmə baş verməyibsə, hal hazırda başlamamış əmrlər hələ də `HEAD`-in tarixçəsində qalır. Siz `topic`-i yenidən `master`-ə birləşdirməyə çalışsanız, Git çaşacaq:

```
$ git merge topic
Already up-to-date.
```

Artıq `master`-də əlçatmaz bir `topic` yoxdur. Daha pisi isə, əgər `topic`-ə iş əlavə etsəniz və yenidən birləşdirsəniz, Git yalnız geri qaytarıldıqdan sonrakı dəyişiklikləri göstərəcək:

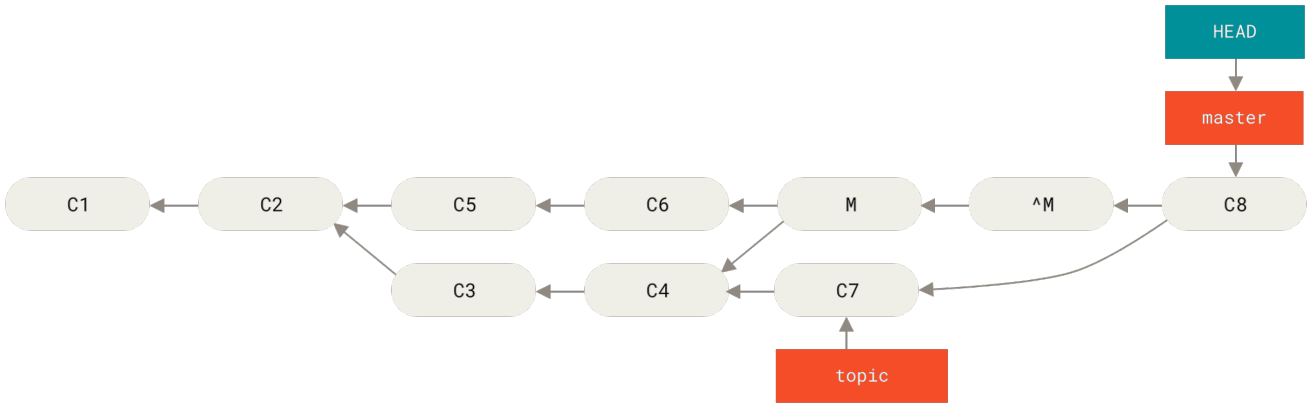


Figure 141. Yaxşı olmayan Birləşdirmə Tarixçəsi

Bunun ən yaxşı yolu orijinal birləşməni geri qaytarmaqdır, çünki indi geri qaytarılmış dəyişiklikl əri gətirmək, daha sonra isə yeni birləşmə commit-i yaratmaq istəyirsiniz:

```
$ git revert ^M
[master 09f0126] Revert "Revert "Merge branch 'topic'""
$ git merge topic
```

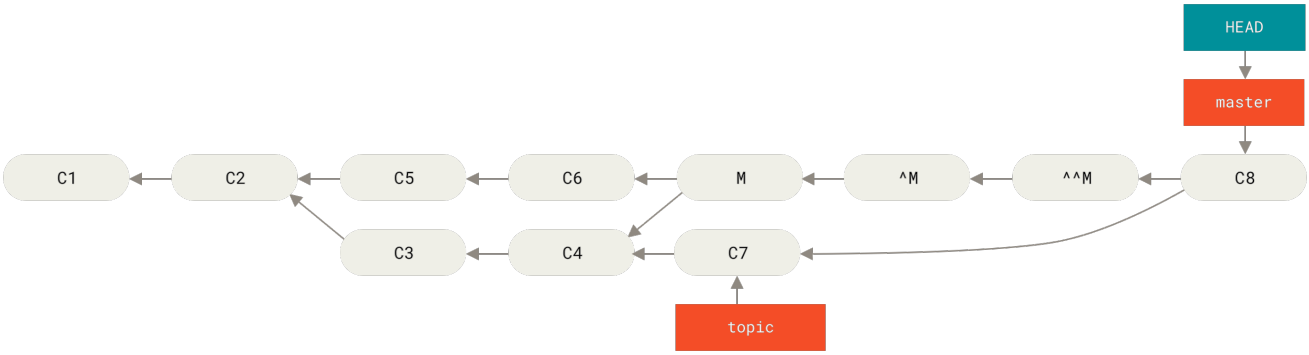


Figure 142. Geri çevrilmiş birləşməni yenidən birləşdirdikdən sonrakı tarixçə

Bu nümunədə,`M` və `^M`ləğv olunur. `^^M`, `C3` və `C4` dəyişiklikləri təsirli bir şəkildə birləşdirir və `C7` -nin dəyişikliklərindəki` C8 `birləşir, buna görə də `topic` tamamilə birləşdirilmişdir.

Birləşdirmənin Digər Tipləri

Biz indiyə qədər birləşmənin “recursive” strategiyası adlandırılan iki branch-ın normal birləşm əsini izah etdik. Bununla birlikdə branch-ları birləşdirməyin başqa yolları da var. Bir neçəsini sür ətlə izah edək.

Our və ya Theirs Üstünlükləri

Əvvəla, normal “recursive” birləşmə rejimi ilə edə biləcəyimiz başqa bir faydalı şey də var. `ignore-all-space` və `ignore-space-change` variantlarını `-X` ilə keçdiyini gördük, ancaq Git’in bir tərəfə və ya digərinə konflikt görəndə üstünlük verəcəyini söyləyə bilərik.

Standart olaraq, Git iki branch arasında bir konflikt görəndə, konflikt markerlərini kodunuza əlavə edəcək və faylı konflikt olaraq qeyd edəcək və onu həll etməyə imkan verəcəkdir. Git’in sadəcə konkret bir tərəf seçməsini və konflikti manual olaraq həll etməyinizin əvəzinə digər tərəfi görm

əməzliyə gəlməsini istəsəniz, birləşdirmə əmrini ya `-Xours` ya da `-Xtheirs`-ə verə bilərsiniz.

Git bunu görsə, konflikt markerlərini əlavə etməyəcəkdir. Birləşə bilən fərqlər birləşəcəkdir. Konfliktdə olan hər hansı bir fərq, sadəcə ikili sənədlər daxil olmaqla bütövlükdə göstərdiyiniz tər əfi seçəcəkdir.

Əvvəllər istifadə etdiyimiz "hello world" nümunəsinə qayıtsaq, branch-ımızda birləşməyin konfliktlərə səbəb olduğunu görə bilərik.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
```

Hər şəkildə əgər `-Xours` və ya `-Xtheirs`işlətsək bu alınmayacaq.

```
$ git merge -Xours mundo
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
 test.sh  | 2 ++
 2 files changed, 3 insertions(+), 1 deletion(-)
 create mode 100644 test.sh
```

Bu vəziyyətdə, o, bir tərəfdən "hello mundo" və digər tərəfdən "hola world" olan sənəddə konflikt nişanları almaq əvəzinə, sadəcə "hola world" olanı seçəcəkdir. Bununla birlikdə, bu branch-dakı digər konfliktli dəyişikliklər uğurla birləşdirilir.

Bu seçim ayrıca fərdi fayl birləşməsi üçün `git merge-file --ours` kimi bir şey işlədərək əvvəllər gördüyümüz `git merge-file` əmrinə keçə bilər.

Buna bənzər bir iş görmək istəyirsinizsə, lakin Git-in digər tərəfdən dəyişiklikləri birləşdirməyə c əhd etməməsini istəyirsinizsə, "ours" birləşmə *strategiyasından* daha sərt bir seçim var. Bu, "ours" rekursiv birləşmə *seçimindən* fərqlidir.

Bu, əsasən saxta bir birləşmə edəcəkdir. Hər iki branch-la birlikdə valideynlər kimi yeni bir birl əşmə qeyd edəcək, ancaq qoşulduğunuz branch-a belə baxmayacaq. Sadəcə cari branch-dakı dəqiq kodu birləşdirmə nəticəsilə qeyd edəcək.

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

Gördüyünüz kimi olduğumuz branch-la birləşmənin nəticəsi arasında heç bir fərq yoxdur.

Bu əsasən Git-ə branch-ı daha sonra birləşdirəcək ikən bir branch-ın artıq birləşdirildiyini düşündürmək üçün tez-tez faydalı ola bilər. Məsələn, bir `release` branch-ını dayandırdığınızı və bunun üçün bir anda yenidən `master` branch-nıza birləşdirmək istədiyinizi deyək. Bu vaxt `master` b əzi səhvlər `release` branch-na geri göndərilməlidir. Siz bugfix branch-ını `release` branch-na birl əşdirə bilərsiniz və eyni zamanda `merge -s ours` branch-nı master branch-a birləşdirə bilərsiniz (düzəliş artıq olsa da), belə ki, daha sonra yenidən branch-ı birləşdirdiyiniz zaman bugfix-də heç bir konflikt olmur.

Subtree Birləşdirməsi

Subtree birləşdirmə ideyası odur ki, iki layihəniz var və layihələrdən biri digərinin alt kateqoriyası ilə əlaqələndirildiyini əhatə edir. Bir subtree birləşməsi göstərdiyiniz zaman, Git həmin an birinin digərinin subtree-si olduğunu anlamaq və lazımi şəkildə birləşdirmək üçün kifayət qədər ağıllıdır.

Gəlin mövcud bir layihəyə ayrı bir layihə əlavə etmək və sonra ikincinin kodunu birincinin alt bölməsinə birləşdirmək nümunəsini göstərək.

Əvvəlcə Rack tətbiqini layihəmizə əlavə edəcəyik. Rack layihəsini öz layihəmizdə uzaq bir remote olaraq əlavə edəcəyik və sonra öz branch-ında yoxlayacağıq:

```
$ git remote add rack_remote https://github.com/rack/rack
$ git fetch rack_remote --no-tags
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From https://github.com/rack/rack
 * [new branch]      build      -> rack_remote/build
 * [new branch]      master     -> rack_remote/master
 * [new branch]      rack-0.4   -> rack_remote/rack-0.4
 * [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"
```

İndi `rack_branch` branch-mızda Rack layihəsinin kökü və `master` branch-mızda öz layihəmiz var. Birini, daha sonra isə digərini yoxlasanız, fərqli layihə köklərinə sahib olduqlarını görə bilərsiniz:

```
$ ls
AUTHORS         KNOWN-ISSUES   Rakefile      contrib         lib
COPYING         README         bin           example         test
$ git checkout master
Switched to branch "master"
$ ls
README
```

Bu qəribə bir anlayışdır. Depodakı branch-ların hamısı eyni layihənin branch-ları olmamalıdır. Çox yaygın deyil, çünki nadir hallarda faydalıdır, lakin branch-ların tamamilə fərqli tarixə sahib olmasını əldə etmək olduqca asandır.

Bu vəziyyətdə, Rack layihəsini alt layihə olaraq `master` proyektimizə çəkmək istəyirik. Bunu `git read-tree` ilə Git-də edə bilərik. `read-tree` və onun dostları haqqında Git'in Daxili İşləri-də daha çox məlumat əldə edəcəksiniz, ancaq indi onun bir branch-ın kök ağacını cari hazırlama sahənizdə və iş qovluğunda oxuduğunu bilin. Yenidən `master` branch-a qayıtdıq və `rack_branch` branch-nı əsas layihəmizin `master` branch-ın `rack` alt bölməsinə pull edirik:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

Commit zamanı, bütün alt Rack fayllarımız alt bölümdə olduğu kimi görünür - sanki biz onları tarball-dan köçürmüşük. Maraqlı olan budur ki, dəyişiklikləri branch-ların birindən digərinə qədər asanlıqla birləşdirə bilərik. Beləliklə, Rack layihəsi yenilənirsə, o branch-a keçərək üstdəki d əyişiklikləri pull edə bilərik:

```
$ git checkout rack_branch
$ git pull
```

Sonra biz bu dəyişiklikləri yenidən `master` branch-mıza birləşdirə bilərik. Dəyişiklikləri pull etmək və commit mesajını qabaqcadan hazırlamaq üçün `--squash` seçimini, həmçinin recursive birləşmə strategiyasının `-Xsubtree` seçimini istifadə edin. Rekursiv strategiya burada standartdır, lakin aydınlıq üçün onu da daxil edirik.

```
$ git checkout master
$ git merge --squash -s recursive -Xsubtree=rack rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

Rack layihəsindəki bütün dəyişikliklər birləşdirilmiş və yerli olaraq həyata keçirilməyə hazırdır. Siz bunun əksini də edə bilərsiniz - `master` branch-ınızın `rack\`subdirectory-sində dəyişikliklər aparın və sonra onları \`rack_branch` branch-a birləşdirin və onları maintainers-ə təqdim edin və ya yuxarı tərəfə push edin.

Bu, submodule-lərdən istifadə etmədən (biz Alt Modullar-də əhatə edəcəyik) istifadə etmədən submodule-un iş axınına bir qədər bənzəyən bir iş axınına sahib olmaq üçün bir imkan verir. Digər branch layihələri ilə branch-larımızı depolarımızda saxlaya bilərik və onları bəzən layihəmizə birləşdirə bilərik. Bəzi yollarla yaxşıdır, məsələn, bütün kodlar bir yerə sadiqdir. Bununla birlikdə, digər çatışmazlıqları var ki, dəyişiklikləri yenidən birləşdirmək və ya təsadüfən bir-birinə bağlı olmayan bir depozitə basaraq səhv etmək daha az mürəkkəbdir və səhv etmək daha asandır.

Başqa bir qəribəlik də `rack` alt qovluğunuzdakılar ilə `rack_branch` branch-nızdakı kod arasındakı f ərqi əldə etmək - onları birləşdirməyin lazım olub-olmadığını görməkdir - çünki, normal `diff` əmrindən istifadə edə bilməyəcəksiniz. Bunun əvəzinə müqayisə etmək istədiyiniz branch-la `git diff-tree` işlətməlisiniz:

```
$ git diff-tree -p rack_branch
```

Və ya, `rack` alt qovluğunuzda olanı, serverdəki `master` branch-ı sonuncu dəfə gətirdiyinizlə müqayisə etmək üçün bunu işlədə bilərsiniz:

```
$ git diff-tree -p rack_remote/master
```

Rerere

The `git rerere` functionality is a bit of a hidden feature. The name stands for “reuse recorded resolution” and, as the name implies, it allows you to ask Git to remember how you’ve resolved a hunk conflict so that the next time it sees the same conflict, Git can resolve it for you automatically.

There are a number of scenarios in which this functionality might be really handy. One of the examples that is mentioned in the documentation is when you want to make sure a long-lived topic branch will ultimately merge cleanly, but you don’t want to have a bunch of intermediate merge commits cluttering up your commit history. With `rerere` enabled, you can attempt the occasional merge, resolve the conflicts, then back out of the merge. If you do this continuously, then the final merge should be easy because `rerere` can just do everything for you automatically.

This same tactic can be used if you want to keep a branch rebased so you don’t have to deal with the same rebasing conflicts each time you do it. Or if you want to take a branch that you merged and fixed a bunch of conflicts and then decide to rebase it instead — you likely won’t have to do all the same conflicts again.

Another application of `rerere` is where you merge a bunch of evolving topic branches together into a testable head occasionally, as the Git project itself often does. If the tests fail, you can rewind the merges and re-do them without the topic branch that made the tests fail without having to re-resolve the conflicts again.

To enable `rerere` functionality, you simply have to run this config setting:

```
$ git config --global rerere.enabled true
```

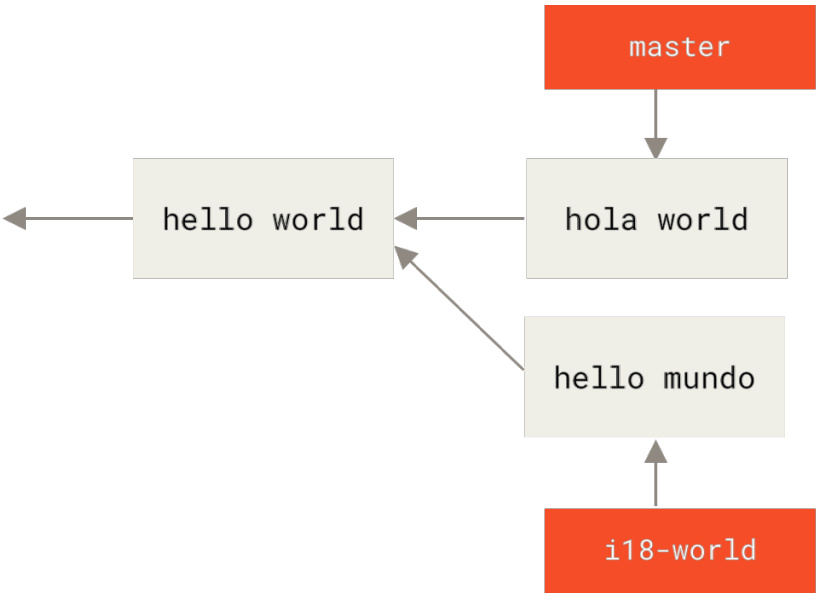
You can also turn it on by creating the `.git/rr-cache` directory in a specific repository, but the config setting is clearer and enables that feature globally for you.

Now let’s see a simple example, similar to our previous one. Let’s say we have a file named `hello.rb` that looks like this:

```
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end
```

In one branch we change the word “hello” to “hola”, then in another branch we change the “world” to “mundo”, just like before.



When we merge the two branches together, we’ll get a merge conflict:

```
$ git merge i18n-world
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Recorded preimage for 'hello.rb'
Automatic merge failed; fix conflicts and then commit the result.
```

You should notice the new line `Recorded preimage for FILE` in there. Otherwise it should look exactly like a normal merge conflict. At this point, `rerere` can tell us a few things. Normally, you might run `git status` at this point to see what all conflicted:

```
$ git status
# On branch master
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add <file>..." to mark resolution)
#
#   both modified:      hello.rb
#
```

However, `git rerere` will also tell you what it has recorded the pre-merge state for with `git rerere status`:

```
$ git rerere status
hello.rb
```

And `git rerere diff` will show the current state of the resolution — what you started with to resolve and what you've resolved it to.

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,11 @@
 #! /usr/bin/env ruby

 def hello
-<<<<<<<
-  puts 'hello mundo'
-=======
+<<<<<<< HEAD
   puts 'hola world'
->>>>>>>
+=======
+  puts 'hello mundo'
+>>>>>>> i18n-world
 end
```

Also (and this isn't really related to `rerere`), you can use `git ls-files -u` to see the conflicted files and the before, left and right versions:

```
$ git ls-files -u
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1   hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2   hello.rb
100644 54336ba847c3758ab604876419607e9443848474 3   hello.rb
```

Now you can resolve it to just be `puts 'hola mundo'` and you can run `git rerere diff` again to see what rerere will remember:

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,7 @@
 #! /usr/bin/env ruby

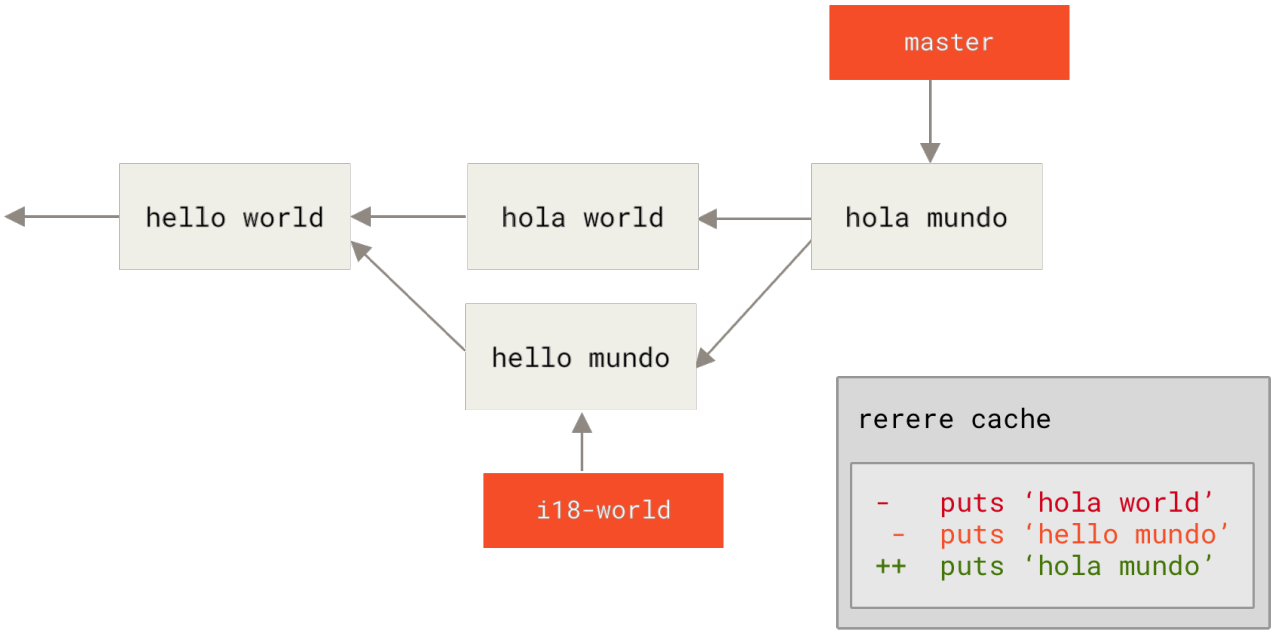
 def hello
-<<<<<<<
-  puts 'hello mundo'
-=======
-  puts 'hola world'
->>>>>>>
+  puts 'hola mundo'
 end
```

So that basically says, when Git sees a hunk conflict in a `hello.rb` file that has “hello mundo” on one side and “hola world” on the other, it will resolve it to “hola mundo”.

Now we can mark it as resolved and commit it:

```
$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'
```

You can see that it "Recorded resolution for FILE".



Now, let’s undo that merge and then rebase it on top of our `master` branch instead. We can move our branch back by using `git reset` as we saw in Reset Demystified.

```
$ git reset --hard HEAD^
HEAD is now at ad63f15 i18n the hello
```

Our merge is undone. Now let’s rebase the topic branch.

```
$ git checkout i18n-world
Switched to branch 'i18n-world'

$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: i18n one word
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Failed to merge in the changes.
Patch failed at 0001 i18n one word
```

Now, we got the same merge conflict like we expected, but take a look at the `Resolved FILE using previous resolution` line. If we look at the file, we'll see that it's already been resolved, there are no merge conflict markers in it.

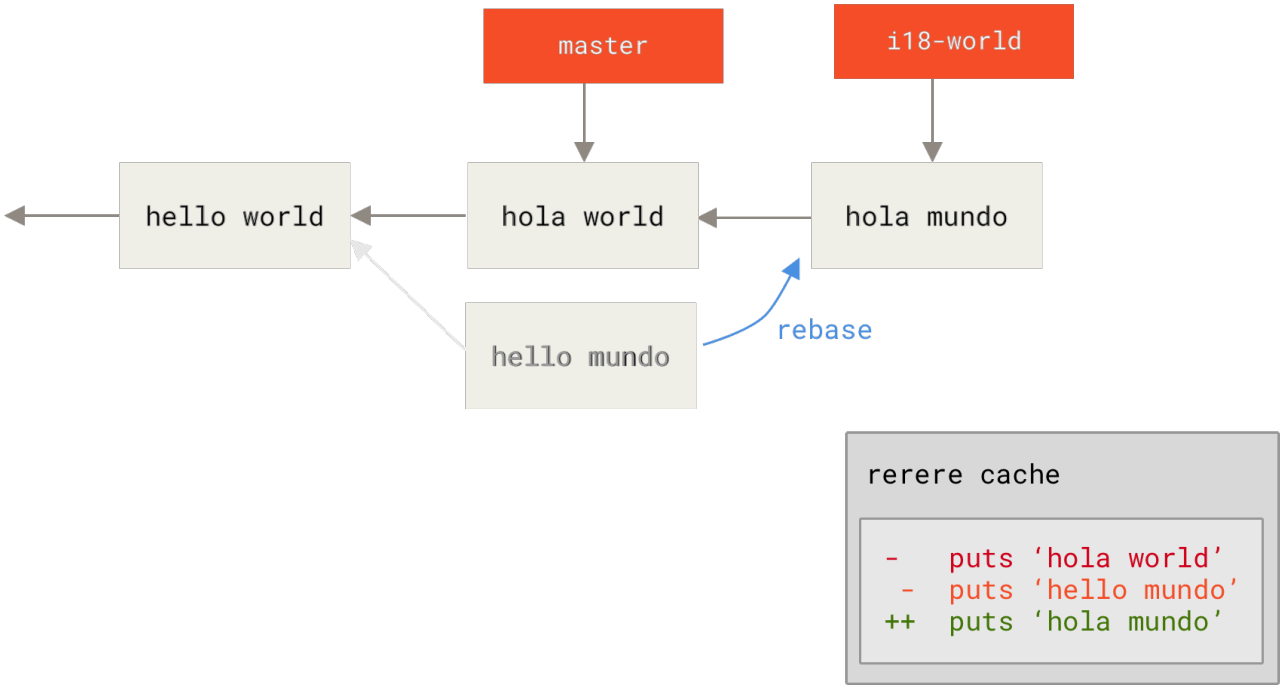
```
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

Also, `git diff` will show you how it was automatically re-resolved:

```
$ git diff
diff --cc hello.rb
index a440db6,54336ba..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
 -  puts 'hello mundo'
++  puts 'hola mundo'
  end
```



You can also recreate the conflicted file state with `git checkout`:

```
$ git checkout --conflict=merge hello.rb
$ cat hello.rb
#! /usr/bin/env ruby

def hello
<<<<<<< ours
  puts 'hola world'
=======
  puts 'hello mundo'
>>>>>>> theirs
end
```

We saw an example of this in İnkişaf etmiş Birləşmə. For now though, let's re-resolve it by just running `git rerere` again:

```
$ git rerere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

We have re-resolved the file automatically using the `rerere` cached resolution. You can now add and continue the rebase to complete it.

```
$ git add hello.rb
$ git rebase --continue
Applying: i18n one word
```

So, if you do a lot of re-merges, or want to keep a topic branch up to date with your `master` branch without a ton of merges, or you rebase often, you can turn on `rerere` to help your life out a bit.

Git ilə Debugging

Əsasən versiyaya nəzarət etməklə yanaşı, Git, source code layihələrinizi debug etməyə kömək etm ək üçün bir neçə əmr verir. Git, demək olar ki, hər hansı bir məzmun növünü idarə etmək üçün t ərtib olunduğundan, bu vasitələr ümumidir, lakin tez-tez işlər səhv olduqda bug və ya culprit üçün ovlanmağınıza kömək edə bilər.

Fayl Annotasiyası

Kodunuzdakı bir səhvi izləyirsinizsə və onun nə vaxt tətbiq olunduğunu və nə üçün olduğunu bilmək istəyirsinizsə, fayl annotasiyası çox vaxt ən yaxşı vasitədir. O sizə hər hansı bir faylın hər s ətirini dəyişdirmək üçün son commit-i göstərir. Aşağıdakı nümunə, üst səviyyəli Linux kernel `Makefile`-də və daha sonra hansı commit və committer-in cavabdeh olduğunu müəyyən etmək üçün `git balme`-dən istifadə edir, bundan əlavə loqotipin output-nu həmin sənədin 69-dan 82-yə qədər olan xətləri ilə məhdudlaşdırmaq üçün `-L `seçimindən istifadə edir:

```
$ git blame -L 69,82 Makefile
b8b0618cf6fab (Cheng Renquan  2009-05-26 16:03:07 +0800 69) ifeq ("$(origin V)",
"command line")
b8b0618cf6fab (Cheng Renquan  2009-05-26 16:03:07 +0800 70)   KBUILD_VERBOSE = $(V)
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 71) endif
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 72) ifndef KBUILD_VERBOSE
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 73)   KBUILD_VERBOSE = 0
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 74) endif
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 75)
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 76) ifeq ($(KBUILD_VERBOSE),1)
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 77)   quiet =
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 78)   Q =
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 79) else
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 80)   quiet=quiet_
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 81)   Q = @
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 82) endif
```

Diqqət yetirin ki, ilk sahə həmin xətti son dəyişdirən commit-in qismən SHA-1 hissəsidir. Növbəti iki sahə həmin commit-dən çıxarılan dəyərlərdir - müəllifin adı və həmin commit-in müəllif tarixi - bu xətti kimin və nə vaxt dəyişdirdiyini asanlıqla görə bilərsiniz. Bundan sonra xətt nömrəsi və faylın məzmunu gəlir. Həm də `^1da177e4c3f4` commit xətlərinə diqqət yetirin, burada `^` prefiksi deposunun ilkin commit-ə daxil edilmiş və indiyədək dəyişməmiş xətləri təyin edir. Bu çaşdırıcı bir şeydir, çünki indi Git-in SHA-1-i dəyişdirmək üçün `^`-dan istifadə etdiyi ən az üç fərqli üsulu gördünüz, amma burada da bunun mənası budur.

Git'in başqa bir xoş cəhəti, fayl adlarını açıq şəkildə izləməməsidir. O, bu anları qeyd edir və sonra gerçəkləşdirildikdən sonra adının dəyişdirildiyini anlamağa çalışır. Bunun maraqlı xüsusiyyətl ərindən biri odur ki, hər cür kod hərəkətini də anlamağını istəyə bilərsiniz. Əgər `–C`-ni `git blame`-ə ötürsəniz, Git yazdığınız faylı təhlil edir və başqa yerdən kopyalandığı təqdirdə içərisindəki kod parçalarının haradan gəldiyini anlamağa çalışır. Məsələn, `GITServerHandler.m` adlı bir faylı `GITPackUpload.m`olan birdən çox sənədə refactoring etdiyinizi fərz edək. `GITPackUpload.m`-u `-C` seçimi ilə blame edərək kodun bölmələrinin əvvəlcə haradan gəldiyini görə bilərsiniz:

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144)         //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 146)         NSString *parentSha;
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 147)         GITCommit *commit = [g
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 149)         //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151)         if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152)                 [refDict setOb
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

Bu həqiqətən faydalıdır. Normalda, kodun surətini çıxardığınız zaman orijinal commit-i götürdüyünüz üçün alırsınız, çünki bu sənəddəki bu sətirlərə ilk dəfə toxunursunuz. Git, başqa bir sənəddə olsa belə, bu sətirləri harada yazdığınızın orijinal commit-ni bildirir.

İkili Axtarış

Bir faylın qeyd edilməsi, əgər məsələnin nədən başlayacağını bilirsinizsə kömək edir. Əgər nəyin pozulduğunu bilmirsinizsə və kodun işlədiyini bildiyiniz son vəziyyətdən bəri onlarla və yüzlərlə iş görülmüşdürsə, çox güman ki, kömək üçün `git bisect`-ə müraciət edə bilərsiniz. `bisect` əmri bir problemi ortaya qoyduğunu mümkün qədər tez müəyyənləşdirməyə kömək etmək üçün commit tarixçənizlə ikili axtarış aparır.

Deyək ki, kodunuzu bir istehsal mühitinə push etdiniz, inkişaf mühitinizdə baş verməyən bir şey haqqında bug hesabatları alırsınız və kodun niyə belə etdiyini təsəvvür edə bilmirsiniz. Kodunuza qayıdırsınız və məsələni yenidən yarada biləcəyiniz ortaya çıxır, ancaq nəyin səhv olduğunu anlaya bilmirsiniz. Bunu bilmək üçün kodu *bisect* edə bilərsiniz. Əvvəlcə `git bisect`-i işə salın, sonra sistemin hazırkı commit-in pozulduğunu söyləmək üçün `git bisect` bad istifadə edin. Daha sonra, `git bisect good <good_commit>` istifadə edərək, son məlum olan yaxşı vəziyyətin nə vaxt olduğunu bisect-ə deməlisiniz.

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] Error handling on repo
```

Git anlayır ki, son yaxşı commit (v1.0) kimi qeyd edilmiş commit-lə hazırkı pis versiya arasında t əxminən 12 əmr gəlir və bu sizin üçün orta birini yoxlayır. Bu anda, problemin bu commit-ə görə olub olmadığını görmək üçün testinizi işə sala bilərsiniz. Bunu edərsə, onda bu orta commit-dən bir müddət əvvəl tətbiq edilmişdir; etmirsə, problem orta commit-dən bir müddət sonra ortaya çıxır. Buradan heç bir problem çıxmır və Git'ə `git bisect good` yazaraq işinizi davam etdirdiyinizi deyə bilərsiniz:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] Secure this thing
```

İndi siz sınadığınızla pis commit-in yarısının ardınca gələn başqa bir commit-dəsiniz. Yenidən testinizi həyata keçirirsiniz və bu commit-in pozulduğunu görürsünüz, buna görə `git bisect bad` ilə Git-ə deyirsiniz:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] Drop exceptions table
```

Bu commit yaxşıdır və indi Git məsələnin harada tətbiq olunduğunu müəyyənləşdirmək üçün lazım olan bütün məlumatlara sahibdir. Bu sizə ilk pis commit-ə aid SHA-1 deyir və commit m əlumatlarının bəzilərini göstərir və bu bug-da hansı hadisələrin baş verdiyini anlamaq üçün bu sə nəddə nəyin dəyişdirildiyini göstərir:

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date:   Tue Jan 27 14:48:32 2009 -0800

    Secure this thing

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M  config
```

Bitirdikdən sonra başlamazdan əvvəl olduğunuz yeri bərpa etmək üçün `git bisect reset` işlətm əlisiniz, yoxsa qəribə bir vəziyyətə düşəcəksiniz:

```
$ git bisect reset
```

Bu, bir neçə dəqiqə ərzində təqdim edilmiş bir bug üçün yüzlərlə commit-i yoxlamağa kömək edə biləcək güclü bir vasitədir. Əslində, 0-dan çıxacaq bir ssenariniz varsa, əgər layihə yaxşıdırsa və ya 0-dan pisdirsə, layihə ümumilikdə pisdirsə, `git bisect`-ini tam avtomatlaşdıra bilərsiniz. Birincisi, məlum pis və yaxşı əmrləri verməklə yenidən bisetin həcmini söyləyin. İstədiyiniz təqdirdə `bisect start` əmri ilə siyahıya alınmaqla, məlum pis commit-i birinci və məlum yaxşı commit-i ikinci sıralamaqla bunu edə bilərsiniz:

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

Bunu etmək Git ilk pozulmuş commit-ni tapana qədər hər test edilmiş `test-error.sh` üzərində avtomatik olaraq işə salır. Siz həmçinin testlər edə bilə və ya sizin üçün avtomatlaşdırılmış testləri işlədən hər hansı bir şeyi işlədə bilərsiniz.

Alt Modullar

Çox vaxt bir layihə üzərində işləyərkən içərisindən başqa bir layihəni istifadə etməli oluruq. Bu üçüncü tərəfin inkişaf etdirdiyi və ya ayrıca inkişaf etdirdiyiniz və bir çox əsas layihədə istifadə etdiyiniz bir kitabxanada ola bilər. Bu ssenarilərdə ortaq bir problem ortaya çıxır: iki layihəni ayrı bir hala gətirmək istəyərsən, amma yenə də birini digərindən istifadə edə bilmək istəyirsən.

Bir misal. Deyək ki, bir veb sayt hazırlayırsınız və Atom feed-lər yaradırsınız. Atom yaradan kodunuzu yazmaq əvəzinə bir kitabxanadan istifadə etməyi qərara alırsınız. Çox güman ki, bu kodu bir CPAN quraşdırması və ya Ruby gem kimi ortaq bir kitabxanadan daxil etməlisiniz və ya mənbə kodu öz layihə ağacınıza kopyalayırsınız. Kitabxananın daxil olması ilə əlaqədar problem, kitabxananı hər hansı bir şəkildə düzəltmək və tez-tez onu yerləşdirmək daha çətindir, çünki hər bir müştəridə bu kitabxananın mövcud olduğundan əmin olmalısınız. Kodun öz layihənizə kopyalanması ilə bağlı problem, yuxarı dəyişikliklər mövcud olduqda etdiyiniz hər hansı bir xüsusi dəyişikliklərin birləşməsi çətinləşməsidir.

Git bu məsələni submodullardan istifadə edərək həll edir. submodullar bir Git deponu başqa bir Git deponun alt kataloqu olaraq saxlamağa imkan verir. Bu, layihənizdə başqa bir depo klonlaşdırmağa və ayrılıqda vəzifələrinizi yerinə yetirməyə imkan verir.

Submodullarla başlayaq

Əsas bir layihəyə və bir neçə alt layihəyə bölünmüş sadə bir layihəni inkişaf etdirmək yolu ilə d əvam edəcəyik.

Mövcud Git deponu üzərində işlədiyimiz deponun submodulu olaraq əlavə etməklə başlayaq. Yeni bir submodul əlavə etmək üçün izləməyə başlamaq istədiyiniz layihənin mütləq və ya nisbi URL-i ilə `git submodule add` əmrindən istifadə edin. Bu nümunədə “DbConnector” adlı bir kitabxana əlavə edəcəyik.

```
$ git submodule add https://github.com/chaconinc/DbConnector
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

Varsayılan olaraq, submodullar alt layihəni "DbConnector" depo ilə eyni adlı bir qovluğa əlavə edə cəkdir. Əmrin sonunda başqa bir yerə getmək istəsəniz fərqli bir yol əlavə edə bilərsiniz.

Bu nöqtədə `git status` işlədirsinizsə, bir neçə şeyi görəcəksiniz.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   .gitmodules
    new file:   DbConnector
```

Əvvəlcə yeni `.gitmodules` faylını görməlisiniz. Bu, layihənin URL-si ilə çəkdiyiniz local subdirectory arasındakı datanı saxlayan bir konfiqurasiya faylıdır:

```
[submodule "DbConnector"]
    path = DbConnector
    url = https://github.com/chaconinc/DbConnector
```

Bir neçə submodulunuz varsa, bu faylda birdən çox giriş olacaqdır. Bu faylın `.gitignore` faylı kimi digər fayllarınızla da idarə olunduğunu qeyd etmək vacibdir. Layihənizin qalan hissəsi ilə push və pull edildi. Bu layihəni klonlayan digər insanlar submodul layihələrini haradan əldə edəcəyini bilirlər.

gitmodules faylındakı URL digər insanların əvvəlcə clone/fetch etməyə çalışdıqları bir şey olduqları üçün mümkün olduğundan əldə edə biləcəkləri bir URL istifadə etdiyinizd ən əmin olun.

Məsələn, başqalarından çəkmək üçün fərqli bir URL istifadə etsəniz, başqalarının istifadə edə biləcəyi URL-i istifadə edin. Öz istifadəniz üçün bu dəyəri local olaraq `git config submodule.DbConnector.url PRIVATE_URL` ilə yaza bilərsiniz. Tətbiq edildikdə, nisbi bir URL kömək edə bilər.

`git status` çıxışındakı digər siyahı, layihə qovluğuna girişdir. Bunun üzərinə `git diff` işləsəniz, maraqlı bir şey görəcəksiniz:

```
$ git diff --cached DbConnector
diff --git a/DbConnector b/DbConnector
new file mode 160000
index 0000000..c3f01dc
--- /dev/null
+++ b/DbConnector
@@ -0,0 +1 @@
+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc
```

`DbConnector` işlədiyiniz qovluğunda bir subdirectory olsa da, Git onu submodul kimi görür və bu qovluqda olmadığınız zaman onun konteksini izləmir. Bunun əvəzinə, Git bunu o depodan xüsusi bir commit kimi görür.

Biraz daha yaxşı fərqli çıxış istəyirsinizsə, `--submodule` seçimini` git diff`-ə əlavə edə bilərsiniz.

```
$ git diff --cached --submodule
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..71fc376
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "DbConnector"]
+       path = DbConnector
+       url = https://github.com/chaconinc/DbConnector
Submodule DbConnector 0000000...c3f01dc (new submodule)
```

Nə vaxt commit etsəniz, aşağıdakı kimi görünəcəkdir:

```
$ git commit -am 'Add DbConnector module'
[master fb9093c] Add DbConnector module
 2 files changed, 4 insertions(+)
 create mode 100644 .gitmodules
 create mode 160000 DbConnector
```

`DbConnector` girişi üçün `160000`rejiminə diqqət yetirin. Bu Git-də xüsusi bir rejimdir ki, bu, bir commiti bir subdirectory və ya bir fayl kimi deyil, bir qovluq girişi kimi qeyd etdiyiniz deməkdir.

Nəhayət, bu dəyişiklikləri push edin:

```
$ git push origin master
```

Bir Layihəni Submodullarla Klonlaşdırmaq

Burada bir submodul ilə bir layihə klonlayacağıq. Belə bir layihəni klonlaşdırdıqda, standart olaraq submodulları ehtiva edən qovluqları alırsınız, ancaq bunların içərisindəki faylların heç birini hələ

almırsınız:

```
$ git clone https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
$ cd MainProject
$ ls -la
total 16
drwxr-xr-x   9 schacon  staff  306 Sep 17 15:21 .
drwxr-xr-x   7 schacon  staff  238 Sep 17 15:21 ..
drwxr-xr-x  13 schacon  staff  442 Sep 17 15:21 .git
-rw-r--r--   1 schacon  staff   92 Sep 17 15:21 .gitmodules
drwxr-xr-x   2 schacon  staff   68 Sep 17 15:21 DbConnector
-rw-r--r--   1 schacon  staff  756 Sep 17 15:21 Makefile
drwxr-xr-x   3 schacon  staff  102 Sep 17 15:21 includes
drwxr-xr-x   4 schacon  staff  136 Sep 17 15:21 scripts
drwxr-xr-x   4 schacon  staff  136 Sep 17 15:21 src
$ cd DbConnector/
$ ls
$
```

`DbConnector` qovluğu var, lakin boşdur. İki əmri işlətməlisiniz: local konfiqurasiya sənədinizi başlatmaq üçün `git submodule init` və o layihədən bütün məlumatları almaq və layihənizdə sadalanan müvafiq commit-ləri yoxlamaq üçün `git submodule update`.

```
$ git submodule init
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path
'DbConnector'
$ git submodule update
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

İndi `DbConnector` subdirectory-niz əvvəllər commit etdiyiniz zaman olduğu vəziyyətdədir. Bunu etmək üçün bir az daha sadə başqa bir yol var. Əgər `--recurse-submodules` əmrini `git clone` əmrinə əlavə etsəniz, depodakı submodulların hər hansı birində submodullar varsa, avtomatik olaraq depodakı hər bir submodulu işə salır və yeniləyir.

```
$ git clone --recurse-submodules https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path
'DbConnector'
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Layihəni əvvəlcədən klonlaşdırmısınızsa və `--recurse-submodules` unutmusunuzsa, `git submodule update --init` əmrini işlədərək `git submodule init` və `git submodule update` addımlarını birləşdirə bilərsiniz. Hər hansı bir iç-içə keçmiş submodulu işə salmaq, almaq və yoxlamaq üçün `git submodule update --init --recursive` istifadə edə bilərsiniz.

Submodullar ilə bir Layihə üzərində İşləmək

İndi içərisində submodulları olan bir layihənin bir nüsxəsi var və həm əsas layihədə, həm də submodul layihədə komanda yoldaşlarımızla əməkdaşlıq edəcəyik.

Submodule Remote-dan Upstream Dəyişiklikləri Pulling etmək

Uzaqdan Layihədə submodulları istifadə etməyin ən sadə modeli, sadəcə bir layihə istifadə etm əyiniz və zaman zaman yeniləmələri almaq istəməyiniz, ancaq həqiqətən yoxlanışda bir şey d əyişdirməməyinizdir. Oradakı sadə bir nümunəyə baxaq.

Bir submodulda yeni işin olub olmadığını yoxlamaq istəyirsinizsə, qovluğa girib local kodu yenilə mək üçün yuxarıdakı branch-ı `git fetch` və `git merge` işlədə bilərsiniz.

```
$ git fetch
From https://github.com/chaconinc/DbConnector
   c3f01dc..d0354fc  master     -> origin/master
$ git merge origin/master
Updating c3f01dc..d0354fc
Fast-forward
 scripts/connect.sh | 1 +
 src/db.c           | 1 +
 2 files changed, 2 insertions(+)
```

İndi əsas layihəyə qayıdırsınız və `git diff --submodule` işlədirsinizsə, submodulun yeniləndiyini və ona əlavə olunanların siyahısını əldə etdiyini görə bilərsən. Hər dəfə `git diff` işlədikdə `--submodule`

yazmaq istəmirsinizsə, onu `diff.submodule` konfiqurasiya dəyərini “log”-a təyin edərək standart format kimi təyin edə bilərsiniz.

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
  > more efficient db routine
  > better connection routine
```

Bu anda commit etsəniz, submodulu digər insanlar yenilədikdə yeni kodun içərisinə bağlayacaqsınız.

Submodulda manual olaraq fetch edilib birləşməməyi istəsəniz, bunu etmək üçün daha asan bir yol var. `git submodule update --remote` işlədirsinizsə, Git submodullara daxil olacaq və sizin üçün yeniləyəcəkdir.

```
$ git submodule update --remote DbConnector
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   3f19983..d0354fc  master     -> origin/master
Submodule path 'DbConnector': checked out 'd0354fc054692d3906c85c3af05ddce39a1c0644'
```

Bu əmr standart olaraq, submodul deposunun `master` branch-na çıxışı yeniləmək istədiyinizi qəbul edəcəkdir. İstəsəniz, bunu fərqli bir şeyə təyin edə bilərsiniz. Məsələn, DbConnector submodulunun o deponun “stable” branch-nı izləməyini istəyirsinizsə, onu ya `.gitmodules` sən ədinizdə (beləcə hər kəs onu izləyər) və ya local `.git/config` faylında quraşdıra bilərsiniz. Gəlin onu `.gitmodules` sənədinə qoyaq:

```
$ git config -f .gitmodules submodule.DbConnector.branch stable

$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   27cf5d3..c87d55d  stable -> origin/stable
Submodule path 'DbConnector': checked out 'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae6687'
```

Əgər `-f .gitmodules` deaktiv etsəniz, bu yalnız sizin üçün dəyişikliyə səbəb olacaq, ancaq hər kəsin də olduğu kimi bu məlumatı depo ilə izləməyi daha mənalı olar.

Bu nöqtədə `git status` işlədikdə, Git submodulda “new commits” olduğunu göstərəcəkdir.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

  modified:   .gitmodules
  modified:   DbConnector (new commits)

no changes added to commit (use "git add" and/or "git commit -a")
```

Konfiqurasiya parametrini `status.submodulesummary` təyin etsəniz, Git sizə submodullarındakı d əyişikliklərin qısa xülasəsini də göstərəcəkdir:

```
$ git config status.submodulesummary 1

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   .gitmodules
    modified:   DbConnector (new commits)

Submodules changed but not updated:

* DbConnector c3f01dc...c87d55d (4):
  > catch non-null terminated lines
```

Bu nöqtədə `git diff` işlədirsinizsə, həm də `.gitmodules` faylımızı dəyişdirdiyimizi, həmçinin endirdiyimiz və submodul layihəmizin yerinə yetirməyə hazır olduğumuz bir sıra işlərin olduğunu görə bilərik.

```
$ git diff
diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
        path = DbConnector
        url = https://github.com/chaconinc/DbConnector
+       branch = stable
 Submodule DbConnector c3f01dc..c87d55d:
  > catch non-null terminated lines
  > more robust error handling
  > more efficient db routine
  > better connection routine
```

Bu, olduqca əlverişlidir, çünki həqiqətən submodulumuzda verəcəyimiz commit-lərin qeydini görə bilərik. Commit etdikdən sonra bu məlumatı `git log -p` işlədərkən də görə bilərsiniz.

```
$ git log -p --submodule
commit 0a24cfc121a8a3c118e0105ae4ae4c00281cf7ae
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Sep 17 16:37:02 2014 +0200

    updating DbConnector for bug fixes

diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
        path = DbConnector
        url = https://github.com/chaconinc/DbConnector
+       branch = stable
Submodule DbConnector c3f01dc..c87d55d:
  > catch non-null terminated lines
  > more robust error handling
  > more efficient db routine
  > better connection routine
```

Git standart olaraq `git submodule update --remote`-u işlətdiyiniz zaman submodullarınızın **hamısını** yeniləməyə çalışacaqdır. Əgər bunların çoxu varsa, yeniləməyə çalışmaq istədiyiniz yalnız submodulun adını ötürmək istəyə bilərsiniz.

Upstream Dəyişikliklərini Layihə Uzaqdan Pull edir

İndi MainProject depolarının öz local klonuna sahib olan işçinizin yerinə keçək. Yeni düzəlişlərinizi

əldə etmək üçün sadəcə `git pull` yerinə yetirmək kifayət etmir:

```
$ git pull
From https://github.com/chaconinc/MainProject
   fb9093c..0a24cfc  master     -> origin/master
Fetching submodule DbConnector
From https://github.com/chaconinc/DbConnector
   c3f01dc..c87d55d  stable     -> origin/stable
Updating fb9093c..0a24cfc
Fast-forward
 .gitmodules         | 2 +-
 DbConnector         | 2 +-
 2 files changed, 2 insertions(+), 2 deletions(-)

$ git status
 On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   DbConnector (new commits)

Submodules changed but not updated:

* DbConnector c87d55d...c3f01dc (4):
  < catch non-null terminated lines
  < more robust error handling
  < more efficient db routine
  < better connection routine

no changes added to commit (use "git add" and/or "git commit -a")
```

Varsayılan olaraq, `git pull` əmri yuxarıda göstərilən ilk əmrin nəticələrində gördüyümüz kimi, submodulları rekursiv şəkildə dəyişir. Ancaq submodulları **yenilə**mə**z**. Bu, submodulun "modified" olduğunu və "new commits" verdiyini göstərən `git status` əmrinin çıxışı ilə göstərilir. Üstəlik, yeni commit nöqtəsini göstərən mötərizələr (<), bu tapşırıqların MainProject-də qeydə alındığını, lakin local DbConnector yoxlanışında olmadığını göstərir. Yeniləməni yekunlaşdırmaq üçün `git submodule update` işləməlisiniz:

```
$ git submodule update --init --recursive
Submodule path 'vendor/plugins/demo': checked out
'48679c6302815f6c76f1fe30625d795d9e55fc56'

$ git status
 On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

Etibarlı tərəfdə olmaq üçün MainProject yeni submodulları əlavə etməyi commit etdiyi submodullar varsa, `--init` və `--recursive` flag-larından istifadə edərək `git submodule update`-ni işə salmalısınız.

Bu prosesi avtomatlaşdırmaq istəyirsinizsə, `--recurse-submodules` flag-nı `git pull` əmrinə əlavə edə bilərsiniz (Git 2.14-dən bəri). Bu, Git pull etməkdən dərhal sonra `git submodule update`-ni işə salacaq və submodulları düzgün vəziyyətə gətirəcəkdir. Üstəlik, Git'i həmişə `--recurse-submodules` ilə pull etmək istəsəniz, `submodule.recurse` konfiqurasiya seçimini "true" olaraq təyin edə bilərsiniz (Git 2.15-dən bəri `git pull` üçün işləyir). Bu seçim Git-in onu dəstəkləyən bütün əmrlər üçün `--recurse-submodules` flag-ından istifadə etməsini təmin edəcəkdir (`clone` xaricində).

Super layihə yeniləmələrini pulling edərkən baş verə biləcək xüsusi bir vəziyyət var: ola bilər ki, yuxarıdakı depo, pull etdiyiniz commit-lərin birində `.gitmodules` faylındakı submodulun URL-sini dəyiştirmiş ola bilər. Bu, məsələn submodul layihəsi hosting platformasını dəyişdirərsə, baş verə bilər. Bu vəziyyətdə, super layihə deposunda local olaraq tənzimlənmiş submodulun uzaq yerində tapılmayan bir submodul əməlinə istinad edərsə, `git pull --recurse-submodules` və ya `git submodule update` üçün uğursuz ola bilər. Bu vəziyyəti düzəltmək üçün `git submodule sync` əmri təl əb olunur:

```
# copy the new URL to your local config
$ git submodule sync --recursive
# update the submodule from the new URL
$ git submodule update --init --recursive
```

Submodul üzərində İşləmək

Çox güman ki, submodullardan istifadə edirsinizsə, bunu əsas layihədə (və ya bir neçə submodulda) kod üzərində işləyərkən eyni zamanda submoduldakı kod üzərində işləmək ist ədiyiniz üçün edirsiniz. Əks təqdirdə bunun əvəzinə daha sadə bir asılılıq idarəetmə sistemindən istifadə edərdiniz (məsələn Maven və ya Rubygems).

Beləliklə, indi əsas layihə ilə eyni vaxtda submodulda dəyişiklik etmək və eyni zamanda bu d əyişiklikləri commit etmək və yayımlamaq nümunəsinə baxaq.

İndiyə qədər, submodul depolarından dəyişiklik almaq üçün `git submodule update` əmrini işlə tdikdə Git dəyişiklikləri əldə edib subdirectory-da faylları yeniləyəcək, lakin sub-repository-i "detached HEAD" vəziyyəti adlanacaq. Bu o deməkdir ki, dəyişiklikləri izləyən local (məsələn, `master` kimi) branch yoxdur. Heç bir işləyən branch izləmə dəyişiklikləri olmadan o deməkdir ki, submodula dəyişiklik etsəniz də, növbəti dəfə `git submodule update` işlədikdə bu dəyişikliklər tamamilə itiriləcəkdir. Bir submoduldakı dəyişikliklərin izlənməsini istəyirsinizsə, əlavə addımlar atmalısınız.

Giriş və hack etmək daha asan olması üçün alt modulunuzu qurmaq üçün iki şey etməlisiniz. Hər bir submodula daxil olmalı və işləmək üçün bir branch-ı yoxlamalısınız. Sonra dəyişikliklər etdikd ən sonra Git'ə nə etməli olduğunu izah etməlisiniz və sonra `git submodule update --remote` yuxarıdan yeni işə pull ediləcəkdir. Seçimlər bunlardır ki, onları local işinizə birləşdirə bilərsiniz və ya local işinizi yeni dəyişikliklərin başına qaytarmağa cəhd edə bilərsiniz.

Əvvəlcə submodul qovluğumuza daxil olub, bir branch-ı yoxlayaq.

```
$ cd DbConnector/
$ git checkout stable
Switched to branch 'stable'
```

Submodulumuzu “merge” seçimi ilə yeniləməyə çalışaq. Manual müəyyənləşdirmək üçün `update` çağırışımıza `--merge` seçimini əlavə edə bilərik. Burada bu submodul üçün serverdə bir dəyişiklik olduğunu və birləşdirildiyini görərik.

```
$ cd ..
$ git submodule update --remote --merge
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   c87d55d..92c7337  stable     -> origin/stable
Updating c87d55d..92c7337
Fast-forward
 src/main.c | 1 +
 1 file changed, 1 insertion(+)
Submodule path 'DbConnector': merged in '92c7337b30ef9e0893e758dac2459d07362ab5ea'
```

DbConnector qovluğuna daxil olsaq, dəyişikliklər local `stable` branch-da göstərilmişdir. İndi gəlin görək kitabxanaya öz local dəyişikliklərimizi edərik və başqası eyni zamanda başqa bir dəyişikliyi yuxarıya push edəndə nə olacağına baxaq.

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'Unicode support'
[stable f906e16] Unicode support
 1 file changed, 1 insertion(+)
```

İndi submodulumuzu yeniləsək, local bir dəyişiklik etdikdə və yuxarıda əlavə etməli olduğumuz bir dəyişikliyə sahib olduğumuzu görə bilərik.

```
$ cd ..
$ git submodule update --remote --rebase
First, rewinding head to replay your work on top of it...
Applying: Unicode support
Submodule path 'DbConnector': rebased into '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

`--rebase` və ya `--merge` unudarsanız, Git yalnız submodulu serverdəki hər hansı bir şeyə yeniləyəcək və layihənizi ayrı bir HEAD vəziyyətinə qaytaracaqdır.

```
$ git submodule update --remote
Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Əgər bu baş verərsə, narahat olmayın, sadəcə qovluğuna qayıda bilər və yenidən branch-nızı yoxlaya bilərsiniz (bu hələ də işinizi özündə saxlayacaq) `origin/stable` (və ya istədiyiniz hər hansı bir branch-ı) manual olaraq birləşdirə və ya dəyişdirə bilərsiniz.

Submoduldakı dəyişikliklərinizi etməmisinizsə və problem yarada biləcək bir submodul yeniləm əsini işlədirsinizsə, Git dəyişiklikləri alır, ancaq submodul qovluğunuzda qeyd olunmamış işləri yazmır.

```
$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 4 (delta 0)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   5d60ef9..c75e92a  stable     -> origin/stable
error: Your local changes to the following files would be overwritten by checkout:
    scripts/setup.sh
Please, commit your changes or stash them before you can switch branches.
Aborting
Unable to checkout 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path
'DbConnector'
```

Yuxarıda dəyişdirilmiş bir şey ilə ziddiyyətli dəyişikliklər etmisinizsə, Git yeniləməni işlədiyiniz zaman sizə xəbər verəcəkdir.

```
$ git submodule update --remote --merge
Auto-merging scripts/setup.sh
CONFLICT (content): Merge conflict in scripts/setup.sh
Recorded preimage for 'scripts/setup.sh'
Automatic merge failed; fix conflicts and then commit the result.
Unable to merge 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path
'DbConnector'
```

Submodul qovluğuna girib konflikti normal qaydada istədiyiniz kimi düzəldə bilərsiniz.

Submodul Dəyişikliklərini Yayımlamaq

İndi submodul qovluğunda bəzi dəyişikliklər var. Bunlardan bəziləri yeniləmələrimiz tərəfindən yuxarıdan gətirilib, digərləri local olaraq hazırlanıb və hələ də onları açmamışıq.

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
  > Merge from origin/stable
  > Update setup script
  > Unicode support
  > Remove unnecessary method
  > Add new option for conn pooling
```

Əsas layihədə iştirak etsək və submodula dəyişikliklərini də pushing etmədən push etsək, d əyişikliklərimizi yoxlamağa çalışan digər insanlara problemə yarnacaq, çünki asılı olan submodul dəyişikliklərini əldə etmək üçün bir yol qalmayacaqlar. Bu dəyişikliklər yalnız local nüsxəmizdə olacaq.

Bunun baş verməməsinə əmin olmaq üçün Git-dən əsas layihəni push etmədən əvvəl bütün submodullarınızın düzgün şəkildə push edildiyini yoxlamağı xahiş edə bilərsiniz. `git push` əmri, “check” və ya “on-demand” olaraq təyin edilə bilən `--recurse-submodules` argumentini alır. “check” seçimi, submodul dəyişikliklərindən hər hansı birinə push edilmədiyi təqdirdə, `push` etməyi asanlaşdıracaq.

```
$ git push --recurse-submodules=check
The following submodule paths contain changes that can
not be found on any remote:
  DbConnector

Please try

    git push --recurse-submodules=on-demand

or cd to the path and use

    git push

to push them to a remote.
```

Gördüyünüz kimi, bundan sonra da nə etmək istədiyimiz barədə bəzi faydalı məsləhətlər verir. Sadə seçim odur ki, hər bir submodula daxil olub uzaqdan mövcud olduqlarına əmin olmaq üçün manual olaraq uzaqdan idarə etməkdir və sonra bu push etməyi yenidən sınamaqdır. Bütün push etməklər üçün yoxlama davranışının olmasını istəyirsinizsə,`git config push.recurseSubmodules check` əməliyyatı edərək bu davranışı standart hala gətirə bilərsiniz.

Digər seçim, “on-demand” dəyərini istifadə etməkdir, bu sizin üçün bunu etməyə çalışacaqdır.

```
$ git push --recurse-submodules=on-demand
Pushing submodule 'DbConnector'
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 3), reused 0 (delta 0)
To https://github.com/chaconinc/DbConnector
   c75e92a..82d2ad3  stable -> stable
Counting objects: 2, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 266 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
To https://github.com/chaconinc/MainProject
   3d6d338..9a377d1  master -> master
```

Gördüyünüz kimi, Git DbConnector moduluna girdi və əsas layihəyə keçməmişdən əvvəl push etdi. Bu submodul push etmək nədənsə uğursuz olarsa, əsas layihə push etmə də uğursuz olacaq. Bu davranışı standart olaraq `git config push.recurseSubmodules on-demand` edərək edə bilərsiniz.

Submodul Dəyişikliklərini Birləşdirmək

Bir submodul arayışını başqası ilə eyni vaxtda dəyişdirsəniz, bəzi problemlərlə qarşılaşa bilərsiniz. Yəni, submodul tarixləri bir-birindən ayrılıbsa və bir super layihədə branch-ları ayırmaq öhd əliyindədirsə, düzəltmək üçün bir az iş tələb oluna bilər.

Təqdim olunanlardan biri digərinin birbaşa əcdadıdırsa (sürətli birləşmə), onda Git birləşmə üçün sadəcə sonuncunu seçəcəkdir ki, beləcə yaxşı işləyəcək.

Git sizin üçün mənasız birləşməyə belə cəhd etməyəcək. Submodul bir-birindən fərqli commit edirsə və birləşdirilməlidirsə, buna bənzər bir şey alacaqsınız:

```
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1)
Unpacking objects: 100% (2/2), done.
From https://github.com/chaconinc/MainProject
   9a377d1..eb974f8  master     -> origin/master
Fetching submodule DbConnector
warning: Failed to merge submodule DbConnector (merge following commits not found)
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

Beləliklə, əsasən burada nə baş verən budur ki, Git iki branch-ın submodulun tarixində bir-birind ən fərqli olan və birləşdirilməli olduğunu nöqtələri qeyd etdi. Bunu çaşdırıcı olan “merge following

commits not found” kimi izah edir, amma bunun niyə olduğunu birazdan izah edəcəyik.

Problemi həll etmək üçün submodulun hansı vəziyyətdə olduğunu müəyyənləşdirməlisiniz. Qərib ədir ki, Git buraya kömək etmək üçün çox məlumat vermir, hətta tarixin hər iki tərəfinin verdiyi SHA-1-lərdən də məlumat vermir. Xoşbəxtlikdən, bunu anlamaq çox sadədir. `git diff` işlədirsinizsə, birləşdirməyə çalışdığınız hər iki branch-da qeyd olunan əmsalların SHA-1-lərini əldə edə bilərsiniz.

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```

Beləliklə, bu vəziyyətdə, `eb41d76` submodulumuzda **biz** sahip olduğumuz commit və `c771610` upstream-in sahib olduğu commitdir. Submodule qovluğumuza daxil olsaq, birləşmə ona toxunmayacağı üçün artıq `eb41d76` üzərində olmalıdır. Hansı səbəbdən olursa olsun, sadəcə ona işarə edən bir branch yarada və nəzarət edə bilərsiniz.

Önəmli olan qarşı tərəfin commit-nin SHA-1 olmasıdır. Bu birləşdirib və həll etmək məcburiyyətində olduğunuz bir şeydir. Sadəcə birbaşa SHA-1 ilə birləşməyə cəhd edə bilərsiniz və ya bunun üçün bir branch yarada və sonra birləşməyə cəhd edə bilərsiniz. Sonuncunu, daha yaxşı birləşmə commiti verəcəyi üçün tövsiyə edirik.

Beləliklə, submodul qovluğumuza daxil olacağıq, `git diff`-dən ikinci SHA-1 əsasında “try-merge” adlı bir branch yaradacağıq və manual olaraq birləşdirəcəyik.

```
$ cd DbConnector

$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610

$ git merge try-merge
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Recorded preimage for 'src/main.c'
Automatic merge failed; fix conflicts and then commit the result.
```

Burada faktiki olaraq birləşmə konflikti var, buna görə də bu problemi həll etsək və öhdəsinə götürsək, əsas layihəni nəticə ilə yeniləyə bilərik.

```
$ vim src/main.c ①
$ git add src/main.c
$ git commit -am 'merged our changes'
Recorded resolution for 'src/main.c'.
[master 9fd905e] merged our changes

$ cd .. ②
$ git diff ③
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
@@@ -1,1 -1,1 +1,1 @@@
- Subproject commit eb41d764bccf88be77aced643c13a7fa86714135
 -Subproject commit c77161012afbbe1f58b5053316ead08f4b7e6d1d
++Subproject commit 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
$ git add DbConnector ④

$ git commit -m "Merge Tom's Changes" ⑤
[master 10d2c60] Merge Tom's Changes
```

① Əvvəlcə konflikti həll edirik.

② Sonra əsas layihə qovluğuna qayıdırıq.

③ SHA-1-ləri yenidən yoxlaya bilərik.

④ Konflikt submodul girişini həll edin.

⑤ Birləşməyimizi commit edin.

Bir az qarışıq ola bilər, amma həqiqətən çox çətin deyil.

Maraqlıdır ki, Git-in ələ keçirdiyi başqa bir hadisə də var. Birləşdirmə əməliyyatı tarixində **hər ikisini** ehtiva edən submodule qovluğunda varsa, Git bunu sizə mümkün bir həll yolu kimi təklif edəcəkdir. Submodule layihəsinin bir nöqtəsində kimsə bu iki əmrdən ibarət branch-ları birl əşdirdiyini görür, bəlkə buna görə bunlardan birini istəyərsən.

Buna görə əvvəllər edilən xəta mesajı “merge following commits not found” idi, çünki **bunu** edə bilmədi. Çaşdırıcıdır, çünki bunu etmək üçün kimin **cəhd etməsini** gözləyəcək ki?

Bir məqbul birləşmə commiti taparsa, bu kimi bir şey görəcəksiniz:

```
$ git merge origin/master
warning: Failed to merge submodule DbConnector (not fast-forward)
Found a possible merge resolution for the submodule:
 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a: > merged our changes
If this is correct simply add it to the index for example
by using:

  git update-index --cacheinfo 160000 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
"DbConnector"

which will accept this suggestion.
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

Git-in təklif etdiyi əmr indeksin yenilənməsini təmin edir, sanki `git add` (münaqişəni təmizləyən) kimi işlədin, sonra commit edin. Yəqin ki, bunu etməməlisiniz. Asanlıqla submodul qovluğuna girib fərqin nə olduğunu görə bilərsiniz, bu tapşırığa sürətlə irəliləyin, düzgün sınaqdan keçirin və sonra commit edin.

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward

$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forward to a common submodule child'
```

Bu eyni şeyi yerinə yetirir, amma heç olmasa bu şəkildə bunun işlədiyini və tamamlandığınızda submodul qovluğunda kodunuz olduğunu yoxlaya bilərsiniz.

Submodul Göstərişləri

Submodullarla işləməyi bir az asanlaşdırmaq üçün edə biləcəyiniz bir neçə iş var.

Submodule Foreach

Hər submodulda bəzi ixtiyari əmrləri işlətmək üçün `foreach` submodul əmr var. Eyni layihədə bir neçə submodul varsa, bu həqiqətən faydalı ola bilər.

Məsələn, deyək ki, yeni bir xüsusiyyətə başlamaq və ya səhv düzəltmək istəyirik və bir neçə submodulda işimiz var. Bütün işləri bütün submodullarımızda asanlıqla yığa bilərik.

```
$ git submodule foreach 'git stash'
Entering 'CryptoLibrary'
No local changes to save
Entering 'DbConnector'
Saved working directory and index state WIP on stable: 82d2ad3 Merge from
origin/stable
HEAD is now at 82d2ad3 Merge from origin/stable
```

Sonra yeni bir branch yarada və bütün alt modullarımızda ona keçə bilərik.

```
$ git submodule foreach 'git checkout -b featureA'
Entering 'CryptoLibrary'
Switched to a new branch 'featureA'
Entering 'DbConnector'
Switched to a new branch 'featureA'
```

Fikir aldın. Həqiqətən edə biləcəyiniz bir şey, əsas layihənizdə və bütün alt layihələrinizdə d əyişdirilənlərdən gözəl birləşmiş fərq yaratmaqdır.

```
$ git diff; git submodule foreach 'git diff'
Submodule DbConnector contains modified content
diff --git a/src/main.c b/src/main.c
index 210f1ae..1f0acdc 100644
--- a/src/main.c
+++ b/src/main.c
@@ -245,6 +245,8 @@ static int handle_alias(int *argcp, const char ***argv)

      commit_pager_choice();

+     url = url_decode(url_orig);
+
      /* build alias_argv */
      alias_argv = xmalloc(sizeof(*alias_argv) * (argc + 1));
      alias_argv[0] = alias_string + 1;
Entering 'DbConnector'
diff --git a/src/db.c b/src/db.c
index 1aaefb6..5297645 100644
--- a/src/db.c
+++ b/src/db.c
@@ -93,6 +93,11 @@ char *url_decode_mem(const char *url, int len)
        return url_decode_internal(&url, len, NULL, &out, 0);
 }

+char *url_decode(const char *url)
+{
+       return url_decode_mem(url, strlen(url));
+}
+
 char *url_decode_parameter_name(const char **query)
 {
        struct strbuf out = STRBUF_INIT;
```

Burada submodulda bir funksiya təyin etdiyimizi və əsas layihədə adlandırdığımızı görə bilərik. Bu açıqca sadələşdirilmiş bir nümunədir, amma ümid edirik bunun necə faydalı ola biləcəyi barədə sizə bir fikir verəcək.

Faydalı Alias-lar

Bu əmrlərdən bəziləri üçün bəzi alias-lar qurmaq istəyə bilərsiniz, çünki onlar kifayət qədər uzun ola bilər və əksəriyyətini defolt halına gətirmək üçün konfiqurasiya seçimlərini təyin edə bilm əzsiniz. Git ləqəblərinin qurulmasını Git Alias'lar -də qbaa bilərsiniz, ancaq Git-də submodullarla çox işləməyi planlaşdırırsınızsa, qurmaq istədiyiniz bir nümunəni burada tapa bilərsiniz.

```
$ git config alias.sdiff '!'"git diff && git submodule foreach 'git diff'"
$ git config alias.spush 'push --recurse-submodules=on-demand'
$ git config alias.supdate 'submodule update --remote --merge'
```

Bu yolla submodullarınızı yeniləmək istədikdə `git supdate` və ya submodula asılılığını yoxlamaqla basmaq üçün `git spush` işlədə bilərsiniz.

Submodullarla Bağlı Məsələlər

Submodullardan istifadə hiccups olmadan da mümkün deyil.

Branch-ları Dəyişdirmək

Məsələn, branch-larında submodulları olan branch-ları dəyişdirmək Git 2.13-dən daha köhnə Git versiyaları ilə də çətin ola bilər. Yeni bir branch yaradırsınızsa, orada bir submodul əlavə edin və sonra bu submodul olmadan yenidən bir branch-a keçin, submodul qovluğu hələ də yığılmamış bir qovluq olaraq qalır:

```
$ git --version
git version 2.12.2

$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...

$ git commit -am 'Add crypto library'
[add-crypto 4445836] Add crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    CryptoLibrary/

nothing added to commit but untracked files present (use "git add" to track)
```

Qovluqda silmək çətin deyil, amma orada olması bir az çaşdırıcı ola bilər. Əgər onu çıxarıb sonra yenidən həmin submodulu olan branch-a qayıtsanız, onu təkrarlamaq üçün `submodule update --init` işlətməlisiniz.

```
$ git clean -ffdx
Removing CryptoLibrary/

$ git checkout add-crypto
Switched to branch 'add-crypto'

$ ls CryptoLibrary/

$ git submodule update --init
Submodule path 'CryptoLibrary': checked out 'b8dda6aa182ea4464f3f3264b11e0268545172af'

$ ls CryptoLibrary/
Makefile    includes    scripts     src
```

Yenə də həqiqətən çox çətin deyil, ancaq bir az qarışıq ola bilər.

Yeni Git versiyaları (Git >= 2.13), keçid etdiyimiz branch üçün submodulları düzgün vəziyyətdə yerləşdirməyin qayğısına qalan `git checkout` əmrinə `--recurse-submodules` flagı əlavə etməklə bütün bunları asanlaşdırır.

```
$ git --version
git version 2.13.3

$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...

$ git commit -am 'Add crypto library'
[add-crypto 4445836] Add crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout --recurse-submodules master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean
```

`git checkout`-nın `--recurse-submodules` flag-dan istifadə, super layihədə bir neçə branch-da işləyə rkən faydalı ola bilər. Həqiqətən, submodulunu fərqli commit-lərdə qeyd edən branch-lar arasında keçsəniz, `git status` yerinə yetirildikdə, alt modul ‘modified" olaraq görünəcək və “new commits”

göstəriləcədir. Bu, submodule vəziyyətinin branch-ları dəyişdirərkən bir qayda olaraq aparılmamasıdır.

Bu, həqiqətən çaşdırıcı ola bilər, buna görə də layihənizdə submodullar olduqda həmişə `git checkout --recurse-submodules` etmək yaxşı bir fikirdir. `--recurse-submodules` flag-ı olmayan köhnə Git versiyaları üçün, yoxlanışdan sonra submodulları düzgün vəziyyətə gətirmək üçün `git submodule update --init --recursive` istifadə edə bilərsiniz.

Xoşbəxtlikdən, Git (>= 2.14) konfiqurasiya seçimini `submodule.recurse` qoyaraq həmişə `--recurse-submodules` flag-ından istifadə etməyi söyləyə bilərsiniz: `git config submodule.recurse true`. Yuxarıda qeyd edildiyi kimi, bu da Git-in `--recurse-submodules` seçimi olan hər bir əmr üçün submodullara təkrarlanmasını təmin edəcəkdir (`git clone` xaricində).

Subdirectory-lərdən submodul-lara keçid

Bir çox insanın işlətdiyi digər əsas xəbərdarlıq subdirectory-dən submodullara keçməkdir. Layih ənizdəki faylları izləmisinizsə və onları submodula köçürmək istəyirsinizsə, diqqətli olmalısınız və ya Git sizə qəzəblənəcəkdir. Layihənizin alt alt bölməsində fayllarınız olduğunu düşünün və onu bir subdirectory-ə keçirmək istəyirsiniz. Əgər subdirectory-ni silib sonra `submodule add` işlə dirsinizsə, Git sizə qışqırır:

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

Əvvəlcə `CryptoLibrary` qovluğunu çıxartmalısınız. Sonra submodulu əlavə edə bilərsiniz:

```
$ git rm -r CryptoLibrary
$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

İndi bir branch-da etdiyinizi düşünün. Bu sənədlərin submodula deyil, həqiqi ağacda olduğu bir branch-a geri dönməyə çalışarsanız, bu səhvinizi alırsınız:

```
$ git checkout master
error: The following untracked working tree files would be overwritten by checkout:
  CryptoLibrary/Makefile
  CryptoLibrary/includes/crypto.h
  ...
Please move or remove them before you can switch branches.
Aborting
```

Onu `checkout -f` ilə dəyişdirməyə məcbur edə bilərsiniz, ancaq bu əmrlə yenidən yazıla biləcəyiniz üçün orada saxlanmamış dəyişikliklərin olmadığından ehtiyatlı olun.

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

Sonra geri dönəndə nədənsə boş bir `CryptoLibrary` qovluğu alırsınız və `git submodule update` onu da düzəldə bilməz. Bütün sənədlərinizi geri qaytarmaq üçün submodul qovluğuna daxil olmalı və `git checkout .` başlatmalısınız. Bunu bir neçə submodul üçün işlətmək üçün `submodule foreach` skriptində işlədə bilərsiniz.

Qeyd etmək vacibdir ki, submodullar bu gün bütün Git məlumatlarını ən yaxşı layihənin `.git` qovluğunda saxlayırlar, buna görə Git-in çox köhnə versiyasından fərqli olaraq, submodule qovluğunu məhv etmək heç bir commiti və ya branch-nı itirməyəcəkdir.

Bu vasitələrlə submodullar eyni vaxtda bir neçə əlaqəli, lakin ayrıca layihələr üzərində inkişaf etdirmək üçün olduqca sadə və təsirli bir metod ola bilər.

Bundling

Git məlumatlarını şəbəkə (HTTP, SSH və s.) üzərindən ötürməyin ümumi yollarını nəzərdən keçirs ək də, ümumilikdə istifadə olunmayan, lakin olduqca faydalı ola biləcək daha bir yol var.

Git öz məlumatlarını vahid bir faylda “bundling” etməyə qadirdir. Bu müxtəlif ssenarilərdə faydalı ola bilər. Bəlkə şəbəkəniz itdi və iş yoldaşlarınıza dəyişikliklər göndərmək istəyirsiniz. Ola bilsin ki, başqa yerdə işləyirsiniz və təhlükəsizlik səbəbi ilə yerli şəbəkəyə qoşula bilmirsiniz. Bəlkə wireless / ethernet kartınız sadəcə qırılıb. Bəlkə bu anda ortaq bir serverə girmə imkanı yoxdur, kiməsə yeniləmələri e-poçtla göndərmək istəyirsiniz və 40 əmri `format-patch` vasitəsilə ötürmək ist əmirsiniz.

`git bundle` əmrinin kömək edə biləcəyi yer məhz budur. `bundle` əmri, adətən `git push` əmri ilə telin üzərindən atılacaq hər şeyi, kiməsə e-poçt göndərə biləcəyiniz və ya bir flash sürücünüzə qoya bil əcəyiniz və sonra başqa bir depoya daxil olaraq göndərə biləcəyiniz ikili sənədə yığacaqdır.

Sadə bir misala baxaq. Deyək ki, iki commit olan bir depo var:

```
$ git log
commit 9a466c572fe88b195efd356c3f2bbeccdb504102
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:10 2010 -0800

    Second commit

commit b1ec3248f39900d2a406049d762aa68e9641be25
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:01 2010 -0800

    First commit
```

Bu deponukiməsə göndərmək istəsəniz və push etmək üçün bir depo əldə edə bilmirsinizsə və ya sadəcə onu quraşdırmaq istəmirsinizsə, onu `git bundle create` ilə bağlaya bilərsiniz.

```
$ git bundle create repo.bundle HEAD master
Counting objects: 6, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 441 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
```

İndi `repo.bundle` adlı bir faylınız var ki, içində deponun əsas branch-ını yenidən yaratmaq üçün lazım olan bütün məlumatlar var. Bundle əmri ilə daxil olmasını istədiyiniz hər bir arayışı və ya bir sıra əmrləri siyahıya salmalısınız. Bunun başqa bir yerdə də klonlanmasını istəsəniz, burada etdiyimiz kimi bir məlumat olaraq HEAD əlavə etməlisiniz.

Bu `repo.bundle` faylını başqasına göndərə və ya USB sürücüsünə qoyub üzərindən keçə bilərsiniz.

Digər tərəfdən, bu `repo.bundle` faylı göndərdiyinizk və layihə üzərində işləmək istədiyinizi söyləyin. İkili fayldan bir URL-ə müraciət edə biləcək kimi bir qovluğu klonlaşdıra bilərsiniz.

```
$ git clone repo.bundle repo
Cloning into 'repo'...
...
$ cd repo
$ git log --oneline
9a466c5 Second commit
b1ec324 First commit
```

Əgər arayışlara HEAD daxil etmirsinizsə, `-b master` və ya hansı branch daxil edildiyini də göstərm əlisiniz, çünki əks halda hansı branch-ın yoxlanılacağını bilməyəcək.

İndi deyək ki, üç tapşırığı yerinə yetirmisiniz və yeni əmrləri bir USB çubuğuna və ya e-poçtunuzdakı bir dəstə vasitəsilə geri göndərmək istəyirsiniz.

```
$ git log --oneline
71b84da Last commit - second repo
c99cf5b Fourth commit - second repo
7011d3d Third commit - second repo
9a466c5 Second commit
b1ec324 First commit
```

Əvvəlcə paketə daxil etmək istədiyimiz tapşırığı müəyyənləşdirməliyik. Şəbəkə üzərindən ötürülməsi üçün minimum məlumat toplusunu təyin edən şəbəkə protokollarından fərqli olaraq, bunu manual olaraq anlamalıyıq. İndi eyni şeyi edə bilər və işləyəcək bütün deponu bundle edə bilərsiniz, ancaq fərqləri yəni, yalnız local olaraq hazırladığınız üç əmri bundle etmək daha yaxşıdır.

Bunu etmək üçün fərqi hesablamalı olacaqsınız. Commit Aralıqları-da təsvir etdiyimiz kimi, bir sıra yollarla bir sıra tapşırıqları təyin edə bilərsiniz. Əvvəlcə klonlaşdırdığımız branch-da olmayan `master` branch-mızdakı üç əmri əldə etmək üçün `origin/master..master` və ya `master ^origin/master` kimi bir şeydən istifadə edə bilərik. Bunu `log` əmri ilə sınaya bilərsiniz.

```
$ git log --oneline master ^origin/master
71b84da Last commit - second repo
c99cf5b Fourth commit - second repo
7011d3d Third commit - second repo
```

Beləliklə, bundle-a daxil etmək istədiyimiz tapşırıqların siyahısı olduqdan sonra onları yığırıq. Bunu `git bundle create` əmri ilə edirik və ona bundle-mızın olmasını istədiyimiz bir ad verərək, daxil olmağımız lazım olan əmrlər sıra təqdim edirik.

```
$ git bundle create commits.bundle master ^9a466c5
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (9/9), 775 bytes, done.
Total 9 (delta 0), reused 0 (delta 0)
```

İndi qovluğumuzda bir `commits.bundle` faylı var. Bunu götürüb ortağımıza göndərsək, o vaxt orada daha çox iş görülmüş olsa belə, onu orijinal depo içərisinə idxal edə bilər.

Bundle-ı aldıqda, onu depo içərisinə gətirmədən əvvəl nə olduğunu bilmək üçün yoxlaya bilər. Birinci əmr, sənədin doğrudan Git paketi olduğuna və onu düzgün şəkildə yenidən qurması üçün bütün lazımi soylarınıza sahib olduğundan əmin olacaq `bundle verify` əmridir.

```
$ git bundle verify ../commits.bundle
The bundle contains 1 ref
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
The bundle requires these 1 ref
9a466c572fe88b195efd356c3f2bbeccdb504102 second commit
../commits.bundle is okay
```

Bundler, hər üçü deyil, yalnız son iki əmrdən ibarət bir dəstə yaratmış olsaydı, orijinal depo, lazımlı tarix olmadığı üçün onu idxal edə bilməzdi. `Verify` əmri bunun yerinə belə görünərdi:

```
$ git bundle verify ../commits-bad.bundle
error: Repository lacks these prerequisite commits:
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 Third commit - second repo
```

Hər şəkildə ilk bundle-mız etibarlıdır, ona görə də ondan əmr götürə bilərik. Bundle-da hansı branch-ların idxal edilə biləcəyini görmək istəyirsinizsə, yalnız head-ları sadalamaq üçün bir əmr var:

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

`verify` alt komandası sizə head-ləri də xəbər verəcəkdir. Məsələ budur ki, nəyin pull edilə biləc əyini görmək üçün bu bundle-dan `fetch` və ya `pull` əmrlərini idxal əmrləri kimi istifadə edə bil ərsiniz. Budur, bundle-ın `master` branch-nı depomuzdakı `other-master` adlı bir branch-a gətirəcəyik:

```
$ git fetch ../commits.bundle master:other-master
From ../commits.bundle
 * [new branch]      master     -> other-master
```

İndi görürük ki, idxal olunmuş əmrlərini `other-master` branch-ı ilə eyni zamanda öz `master` branch-mızda etdiyimiz hər hansı bir əmr var.

```
$ git log --oneline --decorate --graph --all
* 8255d41 (HEAD, master) Third commit - first repo
| * 71b84da (other-master) Last commit - second repo
| * c99cf5b Fourth commit - second repo
| * 7011d3d Third commit - second repo
|/
* 9a466c5 Second commit
* b1ec324 First commit
```

Beləliklə, `git bundle` sizin üçün müvafiq şəbəkə və ya bölüşdürülmüş depolarınız olmadıqda bölüşmək və ya şəbəkə tipli əməliyyatlar aparmaq üçün həqiqətən faydalı ola bilər.

Dəyişdirmək

Daha əvvəl vurğuladığımız kimi, Git-in obyektlər bazasındakı obyektlər dəyişilməzdir, lakin Git verilənlər bazasında olan obyektləri digər obyektlərlə əvəz etmək üçün maraqlı bir yol təqdim edir.

`replace` əmri Git-də bir obyekti göstərməyə və "hər dəfə *this* obyektə istinad etdikdə, onu *fərqli_* obyekt" kimi göstərməyə imkan verir. Bu, ümumiyyətlə `git filter-branch` ilə bütün tarixi yenidən qurmadan, tarixinizdəki bir commit-i digəri ilə əvəz etmək üçün çox faydalıdır.

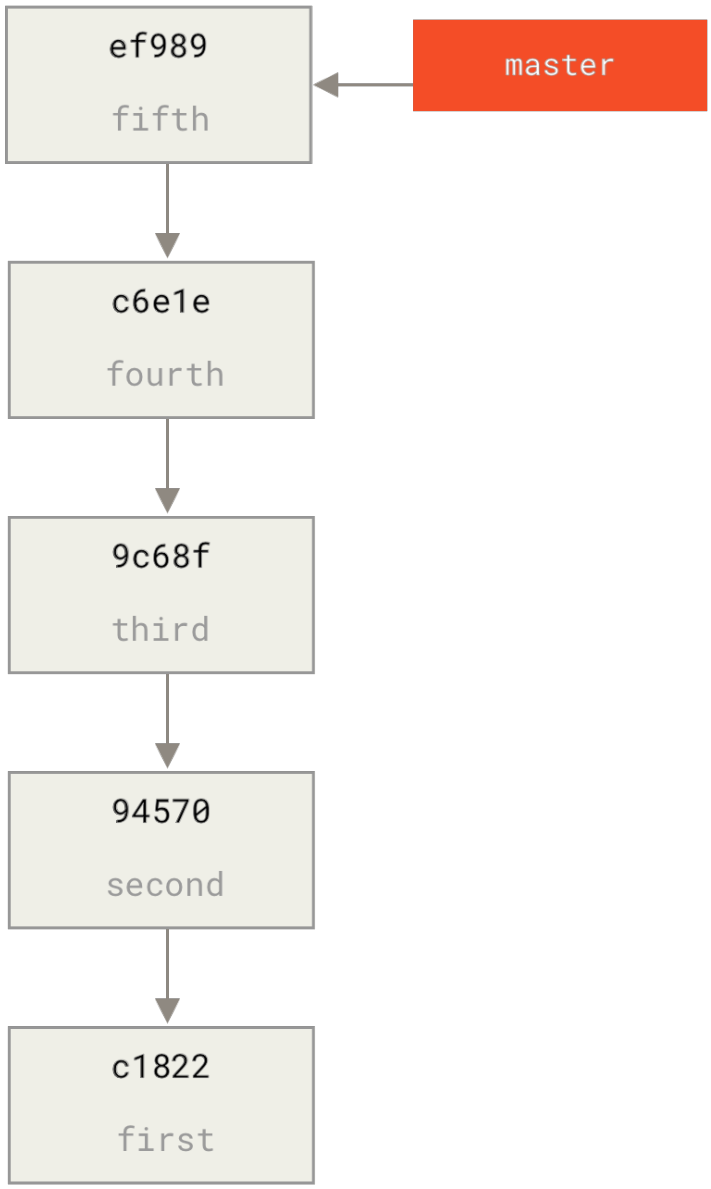
Məsələn, deyək ki, böyü bir kod tarixiniz var və depolarınızı yeni developerlər üçün qısa bir tarixə və məlumat hasilatı ilə maraqlanan insanlar üçün daha uzun və daha böyük bir tarixə bölmək ist əyirsiniz. Yeni sətrdəki ən köhnə commit-i, köhnə olanın ən son commit-i ilə "replacing" etməklə bağlaya bilərsiniz Bu gözəldir, çünki adətən onları birləşdirmək üçün etməli olduğunuz kimi (çünki valideynlik SHA-1-lərə təsir göstərir) yeni tarixdəki hər bir commit-i yenidən yazmaq məcburiyy ətində olmadığınız anlamına gəlir.

Bunu sınayaq. Mövcud bir deponu götürək, onu biri yeni digəri keçmiş olmaq üzrə iki depoya ayırın və sonra son depoların SHA-1 dəyərlərini `replace` yolu ilə dəyişdirmədən onları necə birl əşdirəcəyimizi görəcəyik.

Biz beş sadə commit ilə sadə bir depo istifadə edək:

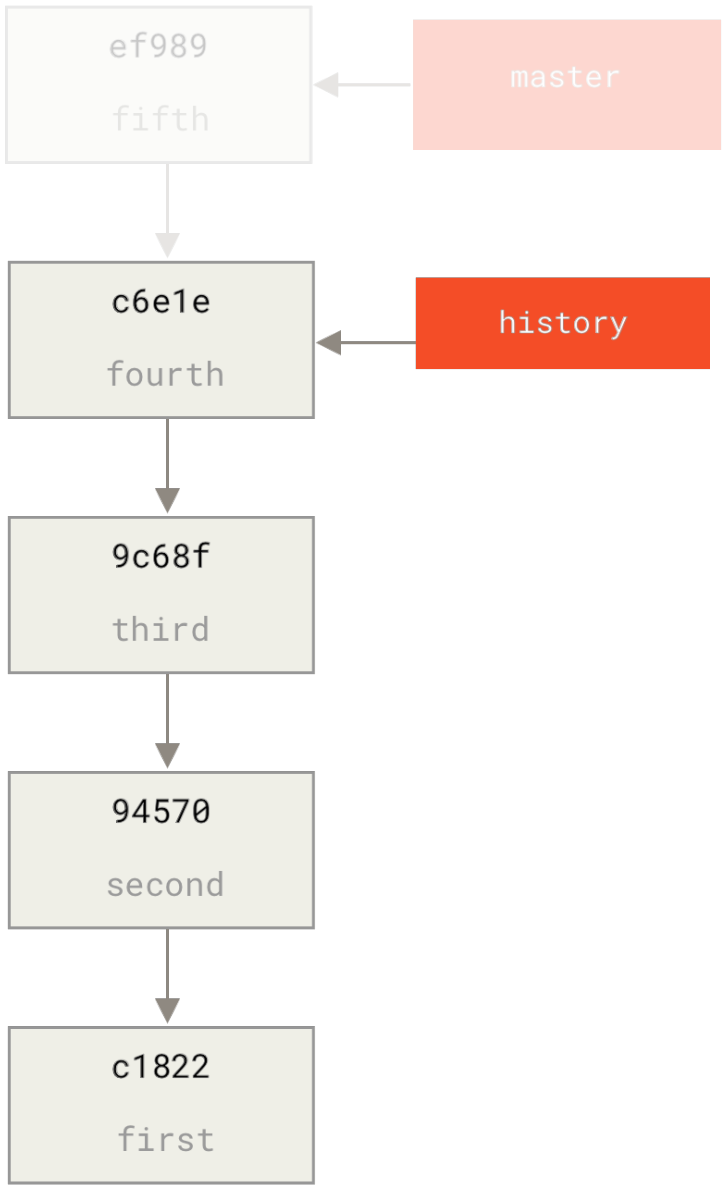
```
$ git log --oneline
ef989d8 Fifth commit
c6e1e95 Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```

Bunu tarixin iki xəttinə bölmək istəyirik. Birinci sətrdə commit-lər birdən dördə qədər davam edə cək - bu tarixi bir xətt olacaqdır. İkinci sətr sadəcə dörd və beşinci commit-lərdən ibarət olacaq - bu yaxın tarix olacaq.



Yaxşı, tarixi bir tarix yaratmaq asandır, sadəcə tarixə bir branch əlavə edə bilərik və sonra bu branch-ı yeni uzaq bir deponun `master` branch-na push edə bilərik.

```
$ git branch history c6e1e95
$ git log --oneline --decorate
ef989d8 (HEAD, master) Fifth commit
c6e1e95 (history) Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```



İndi yeni `history` branch-ını yeni depomuzun `master` branch-na push edə bilərik:

```
$ git remote add project-history https://github.com/schacon/project-history
$ git push project-history history:master
Counting objects: 12, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (12/12), 907 bytes, done.
Total 12 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
To git@github.com:schacon/project-history.git
 * [new branch]      history -> master
```

OK, buna görə tariximiz dərc olunur. İndi daha çətin hissə yaxın tariximizi qısaldıb,

balacalaşdırmaqdır. Birində olan bir commiti digərində olan ekvivalent bir commit ilə əvəz edə biləcəyimiz üçün üst-üstə düşməyimiz lazımdır, buna görə dörd və beş commit-i yerinə yetirmək üçün (dörd üst-üstə düşmək üçün) kəsilmək fikrindəyik.

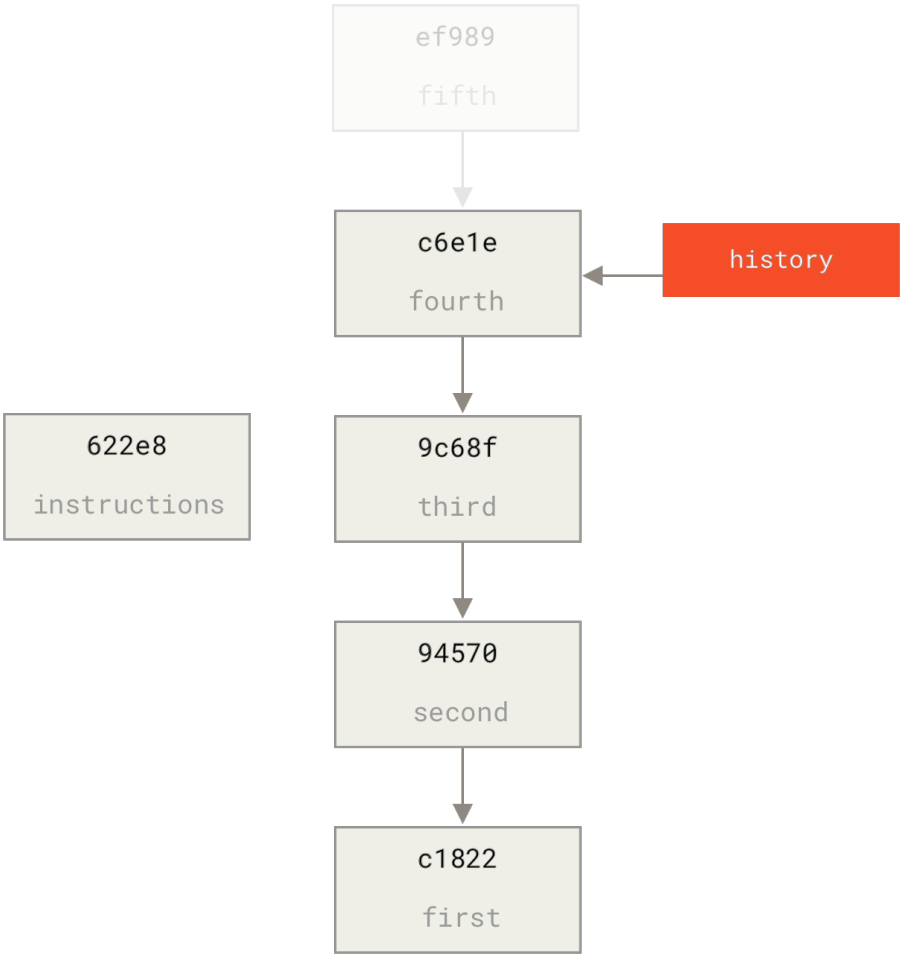
```
$ git log --oneline --decorate
ef989d8 (HEAD, master) Fifth commit
c6e1e95 (history) Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```

Bu vəziyyətdə tarixin genişləndirilməsi barədə təlimatları olan baza commit-i yaratmaq faydalıdır, buna görə digər developerlər kəsilmiş tarixdə ilk commit-i vurduqda və daha çoxuna ehtiyac duyduqda nə edəcəyini bilirlər. Beləliklə, edəcəyimiz şey təlimatla təməl nöqtəmiz kimi ilkin commit-i obyekti yaratmaq, sonra qalan əmrləri (dörd və beş) yenidən əvəz etməkdir.

Bunu etmək üçün, bölüşmək üçün bir nöqtə seçməliyik ki, bu da bizim üçüncü commit-mizdir, yəni SHA-speak-də `9c68fdc`-dir. Beləliklə, baza commit-imiz həmin ağacdan asılı olacaq. Əsas commit-mizi `commit-tree` əmrindən istifadə edərək yarada bilərik, bu sadəcə bir ağac alır və bizə yeni, valideynsiz bir commit obyekti SHA-1-i verir.

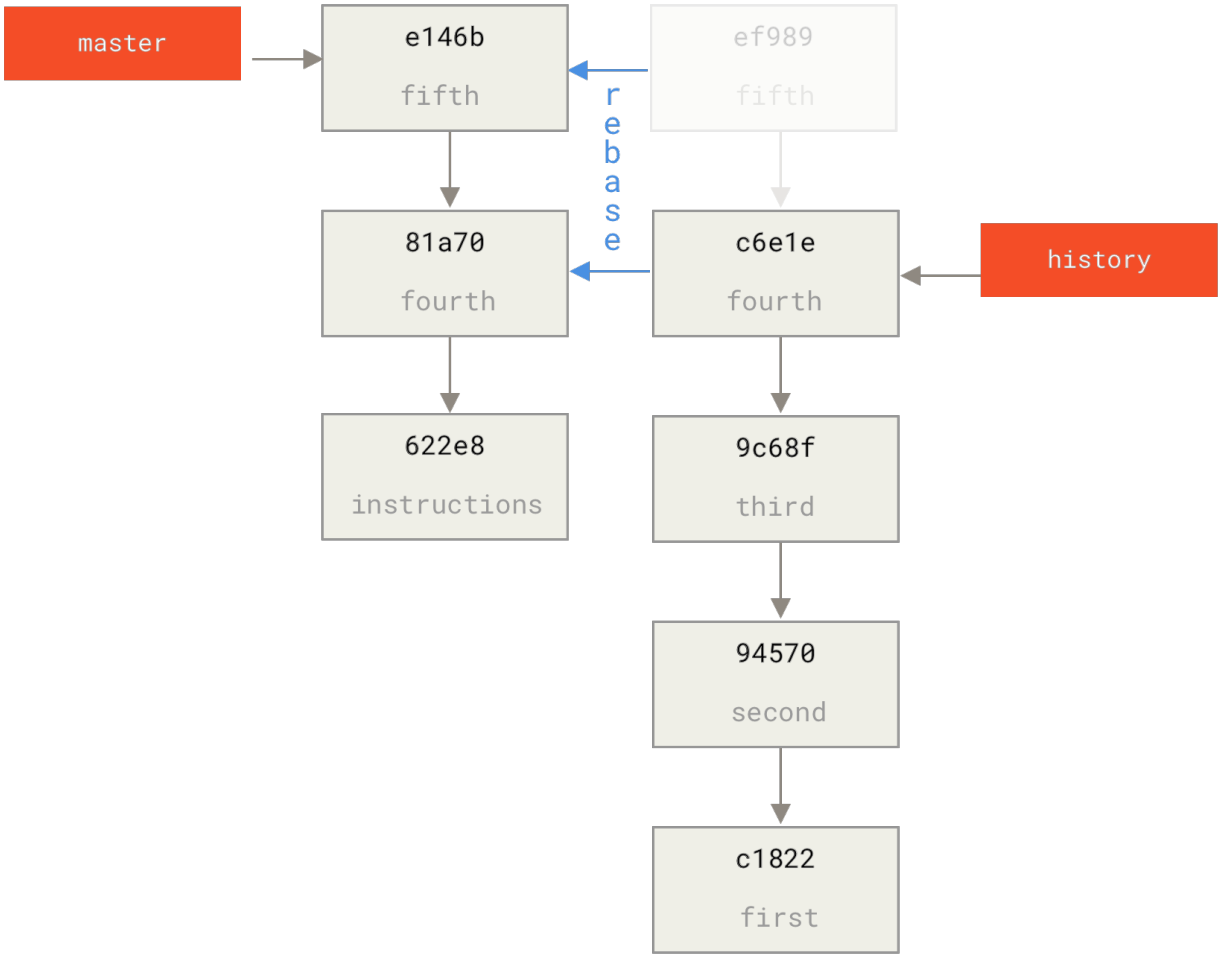
```
$ echo 'Get history from blah blah blah' | git commit-tree 9c68fdc^{tree}
622e88e9cbfbacfb75b5279245b9fb38dfea10cf
```

`commit-tree` əmri, ümumiyyətlə *plumbing* əmrləri adlandırılan bir sıra əmrlərdən biridir. Bunlar ümumiyyətlə birbaşa istifadə üçün nəzərdə tutulmayan əmrlərdir, əksinə daha kiçik işləri görmək üçün **başqa** Git əmrləri tərəfindən istifadə olunur. Bu kimi səliqəli şeylər etdiyimiz hallarda bizə həqiqətən aşağı səviyyəli işlər görməyə imkan verir, lakin gündəlik istifadə üçün nəzərdə tutulmur. Santexnika əmrləri haqqında daha çox məlumatı Plumbing və Porcelain-dən oxuya bilərsiniz.



OK, buna görə bir əsas commit-miz olduğuna görə tariximizin qalan hissəsini `git rebase --onto` ilə əvəz edə bilərik. `--onto` argumenti,`commit-tree`-dan geri aldığımız SHA-1 olacaq və yenidən yerinə yetirmə nöqtəsi üçüncü commit olacaqdır (saxlamaq istədiyimiz ilk ommit-in valideyni, `9c68fdc`):

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
```



Ok, buna görə yaxın keçmişimizi atmaqdan çəkinən əsas commit-ə yazmaq istədikdə bütün tarixi yenidən yaratmağımız barədə təlimatları yazdıq. Bu yeni tarixi yeni bir layihəyə köçürə bilərik və indi insanlar bu deponu klonladıqda, göstərişlərlə əsas commit-in ardınca son iki commit-i görəcə klər.

İndi bütün rolları ilk dəfə layihəni klonlayan birinə dəyişdirək. Bu kəsilmiş depo klonlaşdırıldıqdan sonra tarix məlumatlarını əldə etmək üçün, tarixi depo üçün ikinci bir məsafəni əlavə etmək lazımdır.

```
$ git clone https://github.com/schacon/project
$ cd project

$ git log --oneline master
e146b5f Fifth commit
81a708d Fourth commit
622e88e Get history from blah blah blah

$ git remote add project-history https://github.com/schacon/project-history
$ git fetch project-history
From https://github.com/schacon/project-history
 * [new branch]      master     -> project-history/master
```

İndi işçi `master` branch-ındakı son vəzifələrini və `project-history/master` branch-ında tarixi commit-ləri yerinə yetirəcəkdir.

```
$ git log --oneline master
e146b5f Fifth commit
81a708d Fourth commit
622e88e Get history from blah blah blah

$ git log --oneline project-history/master
c6e1e95 Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```

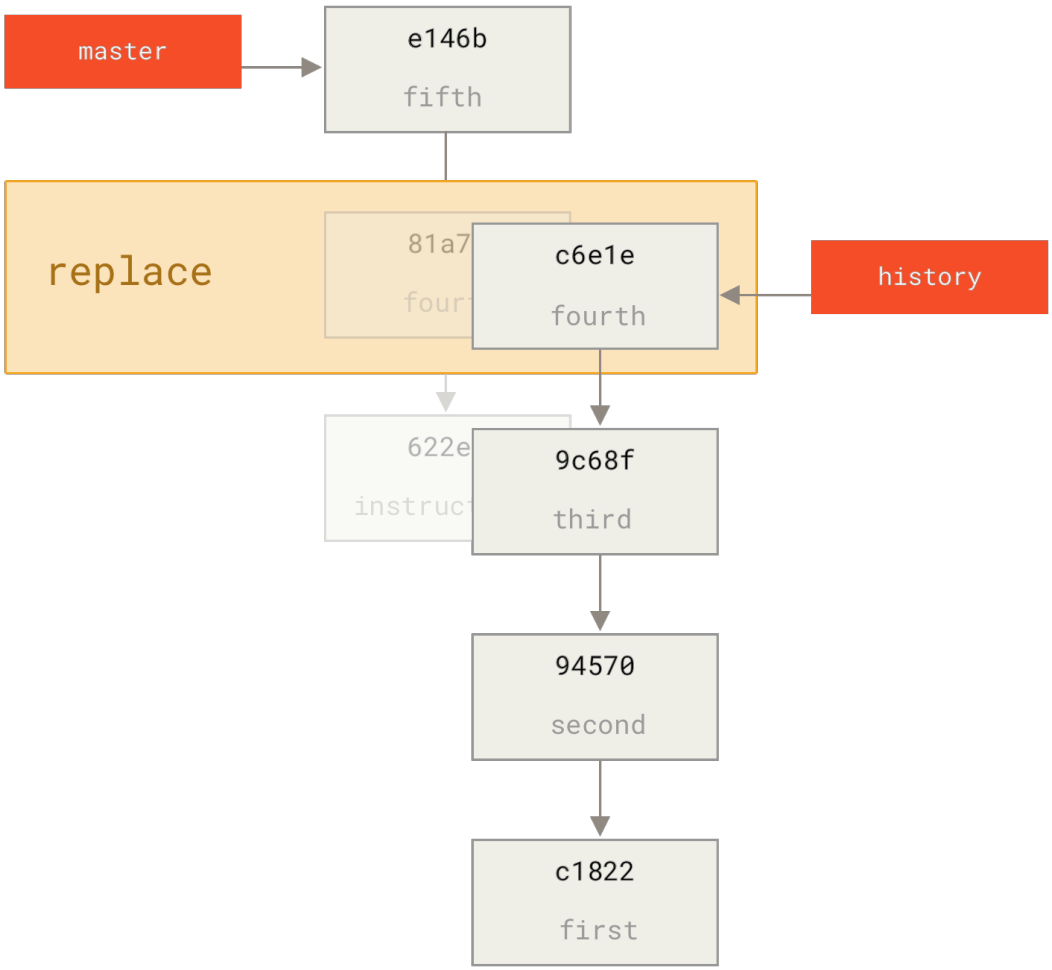
Bunları birləşdirmək üçün əvəz etmək istədiyiniz commit-i ilə sadəcə `git replace` çağıra bilərsiniz. Beləliklə, `master` branch-ındakı "fourth" commit-i `project-history/master` branch-ındakı "fourth" commit-i ilə əvəz etmək istəyirik:

```
$ git replace 81a708d c6e1e95
```

İndi `master` branch-nın tarixinə baxsanız, belə görünür:

```
$ git log --oneline master
e146b5f Fifth commit
81a708d Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```

Əladır, elə deyil? Bütün SHA-1-ləri yuxarıdan dəyişdirmədən, tariximizdəki bir commiti tamamilə fərqli bir commit ilə əvəz edə bildik və bütün normal vasitələr (`bisect`, `blame` və s.) onlardan gözl ədiyimiz şəkildə işləyəcək.



Maraqlısı odur ki, əvəz etdiyimiz `c6e1e95` commit məlumatlarından istifadə etməsinə baxmayaraq, hələ də `81a708d`-ni SHA-1 olaraq göstərir. `cat-file` kimi bir əmr işləsəniz də əvəz edilmiş m əlumatları göstərəcəkdir:

```
$ git cat-file -p 81a708d
tree 7bc544cf438903b65ca9104a1e30345eee6c083d
parent 9c68fdceee073230f19ebb8b5e7fc71b479c0252
author Scott Chacon <schacon@gmail.com> 1268712581 -0700
committer Scott Chacon <schacon@gmail.com> 1268712581 -0700

fourth commit
```

`81a708d`-in həqiqi valideyni, burada qeyd edildiyi kimi `9c68fdce` deyil, əvəzedicimiz (`622e88e`) olduğunu unutmayın ki.

Başqa bir maraqlı məqam bu məlumatların istinadlarımızda saxlanmasıdır:

```
$ git for-each-ref
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/heads/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/remotes/history/master
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/HEAD
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit
refs/replace/81a708dd0e167a3f691541c7a6463343bc457040
```

Bu o deməkdir ki, dəyişdirəcəyimizi başqaları ilə bölüşmək asandır, çünki bunu serverimizə push edə bilərik və digər insanlar asanlıqla yükləyə bilər. Bu keçdiyimiz tarixin paylanması ssenarisində o qədər də faydalı deyil (çünki hər kəs hər iki tarix də yükləyəcək, buna görə niyə onları ayırasız ki?), lakin digər hallarda faydalı ola bilər.

Etibarlı Yaddaş

Uzaqdan əlaqə qurmaq üçün SSH transport-dan istifadə etsəniz, istifadəçi adınızı və şifrənizi yazmadan məlumatları etibarlı şəkildə ötürməyinizə imkan verən keçidsiz bir key-in olması mümkündür. Bununla birlikdə, HTTP protokolları ilə bu mümkün deyil - hər əlaqə üçün istifadəçi adı və şifrə lazımdır. Parol üçün istifadə etdiyiniz işarələrin təsadüfi yaradılmış və açıqlanmayan olması iki faktor identifikasiyası olan sistemlər üçün daha da çətinləşir.

Xoşbəxtlikdən, Git bu işdə kömək edə biləcək creditantals sisteminə sahibdir. Git qutusunda bir neçə seçim vardır:

- Standart olan heç bir şey cache deyil. Hər bir əlaqə istifadəçi adınızı və şifrənizi tələb edəc əkdir.
- “cache” rejimi creditental-ları müəyyən müddət yaddaşlarda saxlayır. Parolların heç biri diskdə saxlanılmır və 15 dəqiqədən sonra yaddaşdan silinir.
- “store” rejimi creditental-ları diskdəki düz mətnli bir faylda saxlayır və heç vaxt bitmir. Bu o deməkdir ki, Git host üçün parolunuzu dəyişməyincə, yenidən creditental-nızı təkrar yazmağa məcbur olmayacaqsınız. Bu yanaşmanın mənfi tərəfi odur ki, parollarınız mətndə ev qovluğunuzdakı düz bir sənəddə saxlanılır.
- Əgər Mac istifadə edirsinizsə, Git sistem hesabınıza daxil edilmiş etibarlı key zəncirində creditental məlumatlarını saxlayan bir “osxkeychain” rejimi ilə gəlir. Bu üsul creditental-larını diskdə saxlayır və heç vaxt bitmir, lakin onlar HTTPS sertifikatları və Safari auto-fill-ləri saxlayan eyni sistemlə şifrələnirlər.
- Windows istifadə edirsinizsə, “Git Credential Manager for Windows” adlı bir köməkçi quraşdıra bilərsiniz. Onu https://github.com/Microsoft/Git-Credential-Manager-for-Windows-da tapa bil ərsiniz.

Git konfiqurasiya dəyərini təyin edərək bu üsullardan birini seçə bilərsiniz:

```
$ git config --global credential.helper cache
```

Bu köməkçilərin bəzilərinin də seçimləri var. “store” köməkçisi, plain-text faylının harada

saxlandığını təyin edən (standart olan `~/.git-credentials`) `--file <path>` arqumenti götürə bilər. “cache” köməkçisi, demonunun işlədilmə müddətini dəyişdirən `--timeout <seconds>` seçimi qəbul edir (standart olaraq “900” və ya 15 dəqiqə). “store” köməkçisini xüsusi bir fayl adı ilə necə düzəld əcəyinizə dair bir nümunə:

```
$ git config --global credential.helper 'store --file ~/.my-credentials'
```

Git hətta sizə bir neçə köməkçini konfiqurasiya etməyə imkan verir. Müəyyən bir host üçün creditental-ı axtararkən Git bunları qaydasında soruşacaq və ilk cavab verildikdən sonra dayanacaq. Creditental-nı saxlayarkən Git, siyahıdakı köməkçilərin **hamısına** istifadəçi adı və şifr ənizi göndərəcək və sizin onlarla nə edəcəyinizi seçə bilərlər. Budur `.gitconfig`, thumb drive-da creditental sənədləri faylı olsaydı, ancaq drive qoşulmadığı təqdirdə bəzi yazmaları saxlamaq üçün yaddaş cache-dan istifadə etmək istərdi:

```
[credential]
    helper = store --file /mnt/thumbdrive/.git-credentials
    helper = cache --timeout 30000
```

Hood Altında

Bəs bütün bunlar necə işləyir? Git’in creditental-köməkçi sistemi üçün kök əmri, arqument olaraq bir əmr götürən və sonra stdin vasitəsilə daha çox giriş olan `git creditental`-dır.

Bir nümunə ilə başa düşmək daha asan ola bilər. Tutaq ki, creditental köməkçisi quruldu və köm əkçi `mygithost` üçün creditental-ı saxladı. Budur, Git bir host üçün creditentals-ı tapmaq istəyərkən səslənən “fill” əmrini istifadə edən bir sessiya:

```
$ git credential fill ①
protocol=https ②
host=mygithost
③
protocol=https ④
host=mygithost
username=bob
password=s3cre7
$ git credential fill ⑤
protocol=https
host=unknownhost

Username for 'https://unknownhost': bob
Password for 'https://bob@unknownhost':
protocol=https
host=unknownhost
username=bob
password=s3cre7
```

① Bu qarşılıqlı fəaliyyətə başlayan əmr sətridir.

② Git-creditental sonra stdin-də giriş gözləyir. Biz onu bildiyimiz şeylərlə təmin edirik: protokol və host adı.

③ Boş bir xətt girişin tamamlandığını göstərir və creditental sistemi bildikləri ilə cavab verməlidir.

④ Git-creditental sonra hər şeyi toplayır və tapdığı məlumatlarla stdout yazır.

⑤ Creditental-lar tapılmadıqda, Git istifadəçi adını və şifrəsini soruşur və onları yenidən çağıran stdouta təqdim edir (burada eyni console-lara əlavə olunur).

Creditental sistemi, həqiqətən Git-dən ayrı bir proqramı işə salır; hansı və necə olmalı olduğu creditental.helper konfiqurasiya dəyərindən asılıdır. Onun qəbul edə biləcəyi bir neçə forma var:

Configuration Value	Behavior
`foo`	Runs `git-credential-foo`
`foo -a --opt=bcd`	Runs `git-credential-foo -a --opt=bcd`
`/absolute/path/foo -xyz`	Runs `/absolute/path/foo -xyz`
`!f() { echo "password=s3cre7"; }; f`	Code after `!` evaluated in shell

Beləliklə, yuxarıda təsvir olunan köməkçilər əslində `git-credential-cache`, `git-credential-store` və sair adlandırılmışdır və onları əmr sətri arqumentlərini götürmək üçün konfiqurasiya edə bilərik. Bunun üçün ümumi forma “git-credential-foo [args] <action>.” Stdin/stdout protokolu git-creditental ilə eynidır, lakin onlar bir az fərqli tədbirlər toplusundan istifadə edirlər:

- `get` bir istifadəçi adı/parol cütü üçün bir tələbdir.
- `store`, bu köməkçinin yaddaşında bir sıra creditentals saxlamaq istəyidir.
- `erase` verilmiş xüsusiyyətlərin creditentals-nı bu köməkçinin yaddaşından təmizləmək üçündür.

`store` və `erase` hərəkətləri üçün heç bir cavab tələb olunmur (Git buna onsuz da məhəl qoymur). Lakin, `get` hərəkəti üçün Git köməkçinin söylədiyi fikirlərlə çox maraqlanır. Köməkçi faydalı bir şey bilmirsə, heç bir output olmadan çıxa bilər, ancaq bilirsə, verdiyi məlumatı store-dakı m əlumatlarla artırmalıdır. Output bir sıra tapşırıq bəyanatları kimi qəbul edilir; təqdim olunan hər şey Git-in artıq bildiyini əvəz edəcəkdir.git-creditentals

Budur yuxarıdakı misal kimi, ancaq -ı atlayaraq və git-credential-store-a üçün birbaşa getmək:

```
$ git credential-store --file ~/git.store store ①
protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential-store --file ~/git.store get ②
protocol=https
host=mygithost

username=bob ③
password=s3cre7
```

① Burada bəzi creditentals-ı saxlamaq üçün `git-credential-store`-a deyirik: istifadəçi adı “bob” və parol “s3cre7” `https://mygithost`-a daxil olduqda istifadə edilməlidir.

② ndi bu creditentals-ı geri alacağıq. Artıq tanıdığımız əlaqə hissələrini (`https://mygithost`) və boş bir xətt provide edirik.

③ `git-credential-store` yuxarıda saxladığımız istifadəçi adı və şifrə ilə cavablayır.

Budur `~/git.store` faylının görünüşü:

```
https://bob:s3cre7@mygithost
```

Bunların hər biri özündə creditental ilə bəzədilmiş bir URL olan bir sıra xətlərdir. Bu `osxkeychain` və `wincred` köməkçiləri öz backing store-larının yerli formatını istifadə edirlər, cache isə öz yaddaş formatını (başqa heç bir proses oxuya bilməz) istifadə edir.

Xüsusi Creditental Cache

`git-creditental-store` və dostların Git-dən ayrı bir proqram olduğunu nəzərə alsaq, hər hansı bir proqramın Git creditental-ın köməkçisi ola biləcəyini başa düşmək çox çətin deyil. Git tərəfindən t əmin olunan köməkçilər bir çox ümumi istifadə hallarını əhatə edir, lakin hamısını deyil. Məsələn, deyək ki, komandanızın, bəlkə də yerləşdirilmə üçün bütün komanda ilə paylaşılan bəzi creditentals-ı var. Bunlar ortaq bir qovluqda saxlanılır, ancaq tez-tez dəyişdikləri üçün onları öz creditental store-a kopyalamaq istəmirsiniz. Mövcud köməkçilərdən heç biri bu işi əhatə etmir; buna görə də özünüzə yazmaq üçün nə lazım olduğuna baxaq. Bu proqramın ehtiyac duyduğu bir neçə əsas xüsusiyyət var:

1. Diqqət etməli olduğumuz yeganə hərəkət `get`-dir; `store` və `erase` yazma əməliyyatlarıdır, buna görə də, o qəbul olunduqda yalnız təmiz şəkildə çıxacağıq.
2. Paylaşılan creditentals sənədinin fayl formatı `git-credential-store`-da istifadə edilənə bənz ərdir.
3. Həmin faylın yeri kifayət qədər standartdır, lakin istifadəçinin yalnız bir halda custom path-i keçməsinə icazə verməliyik.

Bu genişlənməni bird aha Ruby-də yazacağıq, ancaq Git hazır məhsulu işləyə bilənə qədər istənilən dil işləyəcəkdir. Budur yeni creditentals köməkçimizin tam mənbə kodu:

```
#!/usr/bin/env ruby

require 'optparse'

path = File.expand_path '~/.git-credentials' ①
OptionParser.new do |opts|
    opts.banner = 'USAGE: git-credential-read-only [options] <action>'
    opts.on('-f', '--file PATH', 'Specify path for backing store') do |argpath|
        path = File.expand_path argpath
    end
end.parse!

exit(0) unless ARGV[0].downcase == 'get' ②
exit(0) unless File.exists? path

known = {} ③
while line = STDIN.gets
    break if line.strip == ''
    k,v = line.strip.split '=', 2
    known[k] = v
end

File.readlines(path).each do |fileline| ④
    prot,user,pass,host = fileline.scan(/^(.*?):\/\/(.*?):(.*?)@(.*)$/).first
    if prot == known['protocol'] and host == known['host'] and user ==
known['username'] then
        puts "protocol=#{prot}"
        puts "host=#{host}"
        puts "username=#{user}"
        puts "password=#{pass}"
        exit(0)
    end
end
```

① Burada command-line seçimlərini təhlil edirik, istifadəçiyə giriş faylı göstərməyə imkan verir. Standart bir `~/.git-credentials`-dır.

② . Bu proqram yalnız hərəkət `get` olduqda və backing store faylı mövcud olduqda cavab verir.

③ Bu loop stdin-dən ilk boş sətrə çatana qədər oxunur. Girişlər sonrakı istinad üçün known hash-də saxlanılır.

④ Bu loop, storage faylının məzmununu oxuyur, uyğunluq axtarır. `Known`-dan olan protokol və host bu xətlə uyğunlaşırsa, proqram nəticələri stdout-a yazır və çıxış edir.

Köməkçimizi `git-credential-read-only` olaraq saxlayacağıq, onu PATH-a bir yerə qoyub icra edilə bilən kimi işarələyəcəyik. İnteraktiv seans isə bu tip görünür:

```
$ git credential-read-only --file=/mnt/shared/creds get
protocol=https
host=mygithost

protocol=https
host=mygithost
username=bob
password=s3cre7
```

Adı “git-” ilə başladığından, konfiqurasiya dəyəri üçün sadə sintaksisdən istifadə edə bilərik:

```
$ git config --global credential.helper 'read-only --file /mnt/shared/creds'
```

Gördüyünüz kimi, bu sistemin uzanması olduqca sadədir və eyni zamanda sizin və komandanız üçün bəzi ümumi problemləri həll edə bilər.

Qısa Məzmun

Comit-lərinizi və quruluş sahənizi daha dəqiq idarə etməyə imkan verən bir sıra inkişaf etmiş alə tlər gördünüz. Məsələləri gördükdə, onları nəyin, nə vaxt və kim tərəfindən təqdim etdiyini asanlıqla anlamalısınız. Layihənizdə alt layihələrdən istifadə etmək istəyirsinizsə, artıq bu ehtiyacları necə qarşılayacağınızı öyrənmisiniz. Bu nöqtədə, Git-də komanda xəttində hər gün ehtiyac duyacağınız və bunu etməkdə rahat olduğunuz şeylərin çoxunu edə bilməlisiniz.

Git'i Fərdiləşdirmək

İndiyə qədər Git'in necə işlədiyini və necə istifadə ediləcəyini izah etdik və asanlıqla və səmərəli istifadə etməyiniz üçün Git'in təqdim etdiyi bir sıra vasitələri təqdim etdik. Bu fəsildə bir neçə vacib konfiqurasiya parametrlərini və hook'lar sistemini təqdim edərək Git'i daha çox fərdi qaydada necə işlədə biləcəyinizi görəcəyik. Bu vasitələrlə Git'in sizin, şirkətinizin və ya qrupunuzun ehtiyac duyduğu şəkildə işləməsini təmin etmək asandır.

Git Konfiqurasiyası

Qısa şəkildə Başlanğıc-da oxuduğumuz kimi, Git konfiqurasiyasını `git config` əmri ilə tənzimləyə bilərsiniz. Etdiyiniz ilk işlərdən biri adınızı və e-poçt adresinizi qurmaq idi:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

İndi Git istifadənizi fərdiləşdirmək üçün bu şəkildə təyin edə biləcəyiniz daha maraqlı variantlardan bir neçəsini öyrənəcəksiniz.

Birincisi, sürətli bir nəzərdən keçirmə: Git, istəyə biləcəyiniz qeyri-standart davranışı təyin etmək üçün bir sıra konfiqurasiya fayllarından istifadə edir.

Git-in bu dəyərləri axtardığı ilk yer sistemdəki bütün istifadəçilərə və onların bütün depolarına t ətbiq olunan parametrləri ehtiva edən sistem səviyyəsində `/etc/gitconfig` faylındadır. `--system` seçimini `git config`-ə pass etsəniz, o xüsusilə bu fayldan oxuyur və yazır.

Git'in növbəti göründüyü yer hər bir istifadəçiyə xas olan `~/.gitconfig` (və ya `~/.config/git/config`) faylıdır. Git-i bu `--global` seçiminə pass edərək bu faylı oxumağa və yazmağa məcbur edə bil ərsiniz.

Nəhayət, Git, istifadə etdiyiniz hər hansı bir deponun Git qovluğundakı (`.git/config`) konfiqurasiya faylındakı konfiqurasiya dəyərlərini axtarır. Bu dəyərlər həmin tək olan depoya xasdır və `--local` seçiminin `git config`-ə pass edilməsini təmsil edir. Hansı səviyyə ilə işləmək istədiyinizi müəyyənl əşdirməmisinizsə, bu standartdır.

Bu "levels" (sistem, qlobal, yerli) hər biri əvvəlki səviyyə üzərində yazır, buna görə `.git/config`-d əki dəyərlər, məsələn, `/etc/gitconfig` dəki şeyləri qazanır.

Git-in konfiqurasiya sənədləri sadə mətndir, buna görə də bu dəyərləri faylı manual olaraq düzəldərək və düzgün sintaksisini əlavə edərək təyin edə bil ərsiniz. Hərçənd, `git config` əmrini çalışdırmaq ümumiyyətlə daha asandır.

Sadə Müştəri Konfiqurasiyası

Git tərəfindən tanınan konfiqurasiya seçimləri iki kateqoriyaya bölünür: müştəri və server tərəfi. Seçimlərin əksəriyyəti müştəri tərəfli — configurating yəni şəxsi iş seçimlərinizdir. Bir çox, *çox* konfiqurasiya variantı dəstəklənir, lakin bunların böyük bir hissəsi yalnız müəyyən kənar hallarda

faydalıdır; burada ən çox yayılmış və faydalı variantları nəzərdən keçirəcəyik. Git versiyanızın tanıdığı bütün seçimlərin siyahısını görmək istəyirsinizsə, aşağıdakıları edə bilərsiniz:

```
$ man git-config
```

Bu əmrdə mövcud olan bütün seçimlər bir az ətraflı şəkildə sadalanır. Bu istinad materialını https://git-scm.com/docs/git-config səhifəsində də tapa bilərsiniz.

`core.editor`

Standart olaraq, Git, vizual mətn redaktoru olaraq təyin etdiyiniz hər şeyi `VISUAL` və ya `EDITOR` shell mühiti dəyişənlərindən biri vasitəsilə istifadə edir və ya commit-lərinizi düzəltmək və etiketləmək üçün `vi` redaktoruna qayıdır. Standart olanı başqa bir şeyə dəyişdirmək üçün `core.editor` ayarını istifadə edə bilərsiniz:

```
$ git config --global core.editor emacs
```

İndi, standart shell redaktorunuz kimi təyin olunmasından asılı olmayaraq, Git mesajları düzəltm ək üçün Emacs-ı işə salacaq.

`commit.template`

Bunu sisteminizdəki bir fayl path-a düzəltmisinizsə, Git bu fayldan commit götürdüyünüzdə ilkin mesaj olaraq istifadə edəcəkdir. Xüsusi bir commit şablonu yaratmağın dəyəri, bir commit mesajı yaradarkən özünüzə (və ya başqalarına) uyğun format və üslubu xatırlatmaq üçün istifadə edə bil ərsiniz.

Məsələn, `~/.gitmessage.txt` ünvanındakı bir şablon faylını nəzərdən keçirin:

```
Subject line (try to keep under 50 characters)

Multi-line description of commit,
feel free to be detailed.

[Ticket: X]
```

Bu commit şablonunun vəzifələndiriciyə mövzu sətrini qısa saxlamağı (`git log --oneline` çıxışı naminə), bunun altına daha çox təfərrüat əlavə etməsini və mövcud olduqda bir problemə və ya s əhv izləyici bilet nömrəsinə istinad etməsini necə xatırlatdığına diqqət yetirin.

Git-ə, `git commit`-i işə saldığınız zaman redaktorunuzda görünən standart mesaj kimi istifadə etm əsini söyləmək üçün, `commit.template` konfiqurasiya dəyərini təyin edin:

```
$ git config --global commit.template ~/.gitmessage.txt
$ git commit
```

Bundan sonra, redaktorunuz, commit zamanı placeholder commit mesajı üçün belə bir şey açacaq:

```
Subject line (try to keep under 50 characters)

Multi-line description of commit,
feel free to be detailed.

[Ticket: X]
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   lib/test.rb
#
~
~
".git/COMMIT_EDITMSG" 14L, 297C
```

Komandanızın bir commit mesajı siyasəti varsa, bu siyasət üçün bir şablon sisteminizə qoymaq və Git'i standart olaraq istifadə etmək üçün konfiqurasiya etmək, bu siyasətin mütəmadi olaraq izl ənmə şansını artırmağa kömək edə bilər.

`core.pager`

Bu parametr, Git səhifələrinin `log` və `diff` kimi output-u zamanı hansı pager cihazının istifadə olunduğunu təyin edir. Onu `more` ya da sevdiyiniz pager cihazına qura bilərsiniz (standart olaraq, daha `less`-dir) və ya boş bir sətir quraraq söndürə bilərsiniz:

```
$ git config --global core.pager ''
```

Bunu işləsəniz, Git, nə qədər olmasına baxmayaraq bütün əmrlərin bütün output-nu səhifələndirə cəkdir.

`user.signingkey`

İmzalı şərhli etiketlər düzəldirsinizsə (İşinizin İmzalanması də müzakirə edildiyi kimi), GPG imzalama key-nizi konfiqurasiya ayarı olaraq təyin etmək işləri asanlaşdırır. Key ID-nizi belə ayarlayın:

```
$ git config --global user.signingkey <gpg-key-id>
```

İndi hər dəfə `git tag` əmri ilə key-nizi göstərmədən etiketlər imzalaya bilərsiniz:

```
$ git tag -s <tag-name>
```

`core.excludesfile`

Layihənizin `.gitignore` sənədinə nümunələr qoya bilərsiniz ki, Git onları izlənilməmiş fayllar kimi görməsin və ya Nəzərə Alınmayan Fayllar də deyildiyi kimi onlara `git add` işlədərkən səhnələşdirməyə çalışın.

Ancaq bəzən işlədiyiniz bütün depolar üçün müəyyən sənədləri görməməzlikdən gəlmək istəyərsiniz. Kompüterinizdə macOS işləyirsə, ehtimal ki, `.DS_Store` faylları ilə tanışsınız. Tercih etdiyiniz redaktor Emacs və ya Vim-dirsə, `~` və ya `.swp` ilə bitən fayl adlarını da bilirsiniz.

Bu parametr bir növ qlobal `.gitignore` faylı yazmağınıza imkan verir. Bu məzmunu olan bir `~/.gitignore_global` faylı yaratsanız:

```
*~
.*.swp
.DS_Store
```

...və `git config --global core.excludesfile ~/.gitignore_global` işə salsanız, Git bir daha bu fayllarla sizi narahat etməyəcəkdir.

`help.autocorrect`

Əgər əmri səhv yazsanız, sizə belə bir şey göstərəcək:

```
$ git chekcout master
git: 'chekcout' is not a git command. See 'git --help'.

Buna ən bənzər əmr checkout-dur.
```

Git köməkliklə nə demək istədiyinizi anlamağa çalışır, amma yenə də bunu etməkdən imtina edir. `help.autocorrect`-i 1 olaraq təyin etsəniz, Git bu əmri həqiqətən sizin üçün işlədəcək:

```
$ git chekcout master
DİQQƏT: Siz mövcud olmayan 'chekcout' adlı bir Git əmri çağırmısınız.
'Checkout' demək istədiyiniz fərziyyəsi ilə 0.1 saniyədə davam edin...
```

Qeyd edək ki, “0.1 seconds” işi `help.autocorrect` əslində saniyənin onda birini təmsil edən bir tam rəqəmdir. Buna görə 50-yə qoysanız, Git, avtomatik düzəliş əmrini yerinə yetirmədən əvvəl fikrinizi dəyişdirmək üçün 5 saniyə verəcəkdir.

Git-də Rənglər

Git, komanda çıxışını tez və asanlıqla vizual olaraq təhlil etməyə kömək edən rəngli terminal çıxışını tamamilə dəstəkləyir. Bir sıra seçimlər rənglənməni seçiminizə uyğunlaşdırmağa kömək edə bilər.

`color.ui`

Git output-un çox hissəsini avtomatik olaraq rəngləndirir, ancaq bunu bəyənməsəniz, master-ə keçid var. Bütün Gitin rəngli terminal output-nu söndürmək üçün bunu edin:

```
$ git config --global color.ui false
```

Standart ayar `auto`-dur, hansı rənglər output-dursa, onda birbaşa terminala gedir, amma output bir pipe və ya bir fayla yönəldildikdə o zaman rəng nəzarət kodlarını nəzərdən buraxır.

Terminallar və pipe-lar arasındakı fərqi görməməzlikdən gəlmək üçün onu `always` olaraq da qura bilərsiniz.

Bunu nadir hallarda istəyəcəksiniz; əksər ssenarilərdə, yönləndirilmiş output-unuzda rəng kodları istəsəniz, bunun əvəzinə rəng kodlarını istifadə etməyə məcbur etmək üçün Git əmrinə bir `--color` flag-ını ötürə bilərsiniz. Standart ayar demək olar ki, həmişə istədiyiniz şeydir.

`color.*`

Hansı əmrlərin və necə rəngləndiyinə dair daha konkret olmaq istəyirsinizsə, Git verb-spesific r əngləmə parametrlərini təqdim edir. Bunların hər biri `true`, `false` və ya `always` olaraq təyin edilə bilər:

```
color.branch
color.diff
color.interactive
color.status
```

Bundan əlavə, bunların hər birində, hər bir rəngin üstündən keçmək istəsəniz, output hissələri üçün xüsusi rənglər təyin etmək üçün istifadə edə biləcəyiniz alt parametrlər var. Məsələn, fərqli çıxışınızdakı meta məlumatını mavi ön planda, qara fonda və qalın mətn olaraq təyin etmək üçün aşağıdakıları edə bilərsiniz:

```
$ git config --global color.diff.meta "blue black bold"
```

Rəngi aşağıdakı dəyərlərdən hər hansı birinə qura bilərsiniz: `normal`, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, və ya `white`. Əvvəlki nümunədəki kimi qalın bir atribut istəsəniz, `bold`, `dim`, `ul` (underline), `blink`, and `reverse` (swap foreground və background) seçə bilərsiniz.

Xaricdən Birləşdirmə və Diff Vasitələri

Git'in bu kitabda göstərdiyimiz fərqli bir diff tətbiqinə sahib olmasına baxmayaraq bunun əvəzinə xarici bir vasitə qura bilərsiniz. Konfliktləri manual şəkildə həll etmək əvəzinə qrafik merge-conflict-resolution vasitəsini də qura bilərsiniz. Diff-lərinizi yerinə yetirmək və nəticələrinizi birl əşdirmək üçün Perforce Visual Merge Tool (P4Merge) qurmağı nümayiş etdirəcəyik, çünki gözəl bir qrafik vasitədir və ödənişsizdir. Bunu sınamaq istəyirsinizsə, P4Merge bütün əsas platformalarda işləyir, buna görə də bunu bacaracaqsınız. MacOS və Linux sistemlərində işləyən nümunələrdə

path adlarını istifadə edəcəyik; Windows üçün mühitinizdə `/usr/local/bin`-i icra oluna bilən bir path-a dəyişdirməlisiniz.

Başlamaq üçün, download P4Merge from Perforce. Sonra, əmrlərinizi işə salmaq üçün xarici bağlama skriptlərini quracaqsınız. İştifadə oluna bilənlər üçün macOS yolunu istifadə edəcəyik; digər sistemlərdə, `p4merge` ikilinizin quraşdırıldığı yer olacaq. Verilən bütün arqumentlərlə ikili çağıran `extMerge` adlı birləşdirmə wrapper skriptini qurun:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

Diff wrapper yeddi arqumentin verildiyini yoxlayır və onlardan ikisini birləşmə ssenarinizə ötürür. Standart olaraq, Git diff proqramına aşağıdakı arqumentləri ötürür:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Yalnız `old-file` və `new-file` arqumentlərini istədiyiniz üçün ehtiyac duyduqlarınızı ötürmək üçün wrapper skriptindən istifadə edirsiniz.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Həm də bu alətlərin icra oluna biləcəyinə əmin olmalısınız:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

İndi konfiqurasiya sənədinizi xüsusi birləşdirmə qərarından və diff alətlərindən istifadə etmək üçün qura bilərsiniz. Bunun üçün bir sıra xüsusi ayarlar lazımdır: Git-ə hansı strategiyanı istifadə edəcəyini söyləmək üçün `merge.tool`, `mergetool.<tool>.cmd` əmrinin necə işlədiləcəyini göstərmək üçün, `mergetool.<tool>.trustExitCode`, Git-ə deyin. Bu proqramın çıxış kodu xüsusi birləşmə q ərarını göstərir və ya göstərmirsə və Git-ə difflər üçün hansı əmri çalıştıracağını söyləmək üçün `diff.external` yoxlayın. Beləliklə, dörd konfiqurasiya əmrini işə sala bilərsiniz:

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
  'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'
$ git config --global mergetool.extMerge.trustExitCode false
$ git config --global diff.external extDiff
```

Və ya bu sətirləri əlavə etmək üçün `~/.gitconfig` faylını yoxlaya bilərsiniz:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
  trustExitCode = false
[diff]
  external = extDiff
```

Bütün bunlar qurulduqdan sonra, bu kimi diff əmrləri işlədirsinizsə:

```
$ git diff 32d1776b1^ 32d1776b1
```

Əmr sətrində diff output-u əldə etmək əvəzinə, Git P4Merge-i işə salır, buna bənzər bir şey görünür:

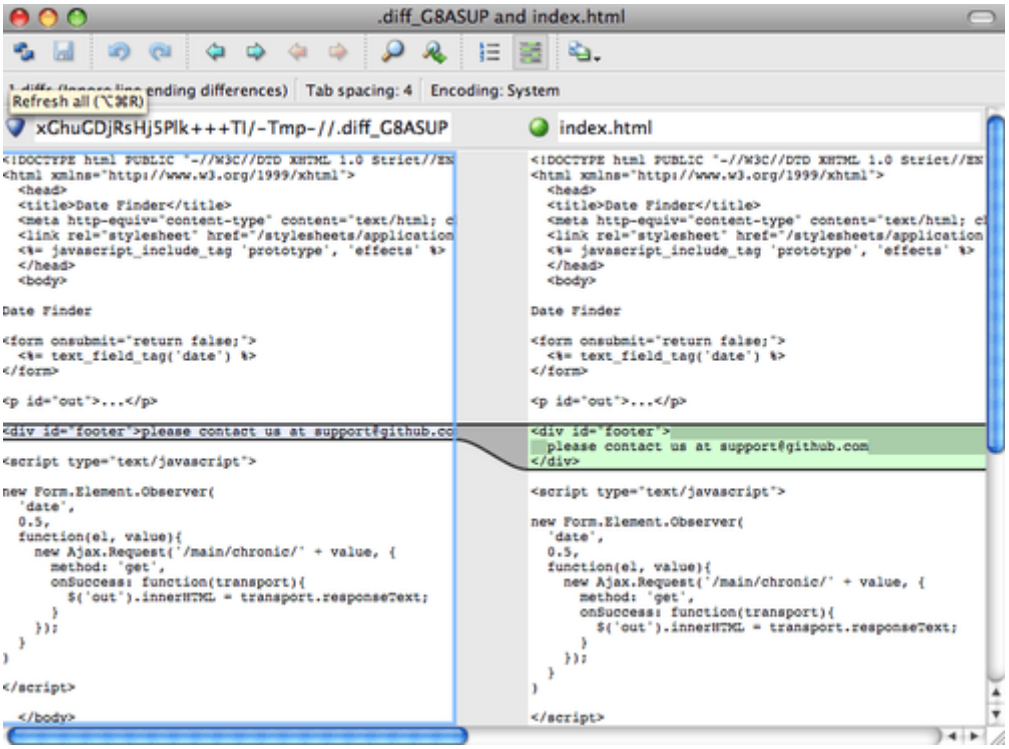


Figure 143. P4Merge

İki branch-ı birləşdirməyə və daha sonra birləşmə konfliktlərinə sahib olsanız, `git mergetool` əmrini işə sala bilərsiniz; bu GUI vasitəsi ilə konfliktləri həll etməyiniz üçün P4Merge başladır.

Bu wrapper quruluşunun ən yaxşı tərəfi odur ki, diff-lərinizi dəyişə və alətlərinizi asanlıqla birl əşdirə bilərsiniz. Məsələn, bunun əvəzinə KDiff3 alətini işə salmaq üçün `extDiff` və `extMerge` alətl ərinizi dəyişdirmək üçün yalnız `extMerge` sənədinizi düzəltməlisiniz:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

İndi Git, diff qərarını və konflikt həllini birləşdirmək üçün KDiff3 alətini istifadə edəcəkdir.

Git, cmd konfiqurasiyasını qurmadan bir sıra digər birləşmə qərarı alətlərindən istifadə etmək üçün əvvəlcədən hazırlanmışdır. Dəstəklədiyi alətlərin siyahısını görmək üçün bunu sınayın:

```
$ git mergetool --tool-help
'git mergetool --tool=<tool>' may be set to one of the following:
        emerge
        gvimdiff
        gvimdiff2
        opendiff
        p4merge
        vimdiff
        vimdiff2

The following tools are valid, but not currently available:
        araxis
        bc3
        codecompare
        deltawalker
        diffmerge
        diffuse
        ecmerge
        kdiff3
        meld
        tkdiff
        tortoisemerge
        xxdiff

Some of the tools listed above only work in a windowed
environment. If run in a terminal-only session, they will fail.
```

KDiff3-ü diff üçün istifadə etməklə maraqlanmırsınız, əksinə onu yalnız birləşdirmə həlli üçün istifadə etmək istəyirsinizsə və kdiff3 əmri sizin path-dadırsa, o zaman işə sala bilərsiniz:

```
$ git config --global merge.tool kdiff3
```

Bunu `extMerge` və `extDiff` fayllarını qurmaq əvəzinə işlətsəniz, Git birləşmə qərarı üçün KDiff3'ü və difflər üçün normal Git diff alətini istifadə edəcəkdir.

Formatlama və Whitespace

Formatlama və whitespace məsələləri, bir çox developer-in, xüsusilə cross-platform-da əməkdaşlıq edərkən qarşılaşdıqları daha əsəb pozucu və incə problemlərdən biridir. Patch-lar və ya digər işl ərdə incə whitespace dəyişikliklərini tətbiq etmək çox asandır, çünki redaktorlar səssizcə təqdim edirlər və sənədləriniz hər hansı bir Windows sisteminə toxunarsa, xətt uçları dəyişdirilə bilər. Git-in bu məsələlərdə kömək edəcək bir neçə konfiqurasiya variantı var.

`core.autocrlf`

Windows-da proqram hazırlayırsınızsa və olmayan insanlarla işləyirsinizsə (və ya əksinə), ehtimal ki, bir nöqtədə sətir sonuna çatanda problemlərlə qarşılaşacaqsınız. Bunun səbəbi, Windows-un fayllarında həm satır qayıtma xarakteri, həm də yeni xətlər üçün xətt xarakteri istifadə etməsi, MacOS və Linux sistemlərində isə yalnız xətt xarakteri istifadə etməsidir. Bu, cross-platform işinin incə, lakin inanılmaz dərəcədə cansıxıcı bir həqiqətidir; Windows-dakı bir çox redaktor səssizcə mövcud LF stili sətir uclarını CRLF ilə əvəz edir və ya istifadəçi giriş düyməsini vurduqda hər iki s ətir işarəsini əlavə edir.

Git, indeksə bir fayl əlavə etdiyiniz zaman CRLF sətir sonlarını avtomatik olaraq LF-ə çevirməklə və əksinə fayl sisteminizə kodu yoxladıqda bunu edə bilər. Bu funksiyanı `core.autocrlf` ayarı ilə aça bilərsiniz. Bir Windows qurğusundasınızsa, onu `true` olaraq ayarlayın - bu kodu yoxladığınız zaman LF sonluqlarını CRLF-ə çevirir:

```
$ git config --global core.autocrlf true
```

Əgər LF sətir sonluqlarından istifadə edən bir Linux və ya macOS sistemindəsinizsə, faylları yoxlayarkən Git-in onları avtomatik olaraq çevirməsini istəmirsiniz; Bununla birlikdə, CRLF sonlu bir fayl təsadüfən təqdim edilərsə, Git-in düzəltməsini istəyə bilərsiniz. Git-ə, `core.autocrlf`-ni girişə ayarlayaraq CRLF-i commit götürdükdə LF-yə çevirməsini söyləyə bilərsiniz:

```
$ git config --global core.autocrlf input
```

Bu quraşdırma sizi Windows checkout-larında CRLF sonları ilə qoymalıdır, lakin macOS və Linux sistemlərində və depoda LF sonluqlarıdır.

Yalnız bir Windows layihəsi həyata keçirən bir Windows proqramçısınızsa, konfiqurasiya dəyərini `false` olaraq təyin edərək daşıma ehtiyatını depoda qeyd edərək bu funksiyanı söndürə bilərsiniz:

```
$ git config --global core.autocrlf false
```

`core.whitespace`

Git, bəzi whitespace problemlərini aşkarlamaq və düzəltmək üçün əvvəlcədən hazırlanmışdır. O, altı əsas whitespace məsələsinə baxa bilər - üçü standart olaraq aktivdir və söndürülə bilər və üçü standart olaraq deaktivdir, lakin aktivləşdirilə bilər. Standart olaraq açılan üçü bir sətrin sonunda boşluq axtaran `blank-at-eol`; bir sənədin sonunda boş sətirləri görən `blank-at-eof`; və bir sətrin ə vvəlindəki nişanlardan əvvəl boşluqlar axtaran `space-before-tab`.

Standart olaraq deaktiv edilmiş, lakin açıla bilən üç nişanlar əvəzinə boşluqlarla başlayan sətirləri axtaran (və `tabwidth` seçimi ilə idarə olunan) olan `indent-with-non-tab`-dir; bir sətrin girinti hiss əsindəki nişanları izləyən `tab-in-indent`; və Git-ə sətirlərin sonunda carriage qayıtmalarının yaxşı olduğunu bildirən `cr-at-eol`.

Bunlardan hansının işə salınmasını istədiyinizi vergüllə ayrılmış və ya söndürmək istədiyiniz dəy ərlərə `core.whitespace` ayarlayaraq söyləyə bilərsiniz. Bir seçimi adının önünə `-` yazaraq söndürə

və ya tamamilə ayar sətrindən kənarda qoyaraq standart dəyəri istifadə edə bilərsiniz.

Məsələn, `space-before-tab` başqa hamısının qurulmasını istəyirsinizsə, bunu edə bilərsiniz (`trailing-space` həm `blank-at-eol` həm də `blank-at-eof` əhatə edəcək short-hand-dir):

```
$ git config --global core.whitespace \
    trailing-space,-space-before-tab,indent-with-non-tab,tab-in-indent,cr-at-eol
```

Və ya yalnız özəlləşdirmə hissəsini təyin edə bilərsiniz:

```
$ git config --global core.whitespace \
    -space-before-tab,indent-with-non-tab,tab-in-indent,cr-at-eol
```

Git, `git diff` əmri işlədərkən bu problemləri aşkarlayacaq və onları rəngləməyə çalışacaq, belə ki, siz onları etmədən əvvəl düzəldə bilərsiniz. Ayrıca, `git apply` ilə patch-lar tətbiq etdikdə sizə köm ək etmək üçün bu dəyərlərdən istifadə edəcəkdir.

Patch-lar tətbiq edərkən, Git-dən göstərilən whitespace məsələləri ilə patch-lar tətbiq edərsə sizi x əbərdar etməsini istəyə bilərsiniz:

```
$ git apply --whitespace=warn <patch>
```

Ya da Git patch tətbiq etməzdən əvvəl problemi avtomatik olaraq həll etməyə çalışa bilərsiniz:

```
$ git apply --whitespace=fix <patch>
```

Bu seçimlər `git rebase` əmrinə də aiddir. Əgər whitespace problemi yaratmısınız, lakin hələ də yuxarıya doğru push etməmisinizsə, Git-in boşluqları yenidən yazdığı üçün whitespace problemini avtomatik olaraq düzəltməsini təmin etmək üçün `git rebase --whitespace=fix` düyməsini işə sala bilərsiniz.

Server Konfiqurasiyası

Git-in server tərəfi üçün o qədər də konfiqurasiya seçimi mövcud deyil, ancaq qeyd etmək istəyə biləcəyiniz bir neçə maraqlı seçim var.

`receive.fsckObjects`

Git, push zamanı alınan hər bir obyektin hələ də SHA-1 hesablama məbləğinə uyğun gəldiyini və etibarlı obyektləri göstərdiyini təmin edə bilər. Ancaq bunu standart olaraq etmir; bu olduqca bahalı bir əməliyyatdır və xüsusən böyük depolarda və ya push-larda əməliyyatı ləngidə bilər. Git-in hər push üzərində obyekt uyğunluğunu yoxlamasını istəyirsinizsə, bunu `receive.fsckObjects`-i true olaraq təyin edərək məcbur edə bilərsiniz:

```
$ git config --system receive.fsckObjects true
```

İndi, Git, hər bir push qəbul edilməzdən əvvəl səhvli (və ya zərərli) müştərilərin pozulmuş m əlumatları təqdim etməməsinə əmin olmaq üçün deponuzun bütövlüyünü yoxlayacaqdır.

`receive.denyNonFastForwards`

Əgər siz artıq push etdiyiniz commit-nizi geri qaytarsanız və sonra yenidən push etməyə çalışsanız və ya başqa bir şəkildə uzaq branch-ın göstərdiyi commit-i ehtiva etməyən bir remote branch-a push etməyə çalışarsanız, rədd ediləcəksiniz. Bu ümumiyyətlə yaxşı siyasətdir; lakin geri qaytarma vəziyyətində nə etdiyinizi bildiyinizi və remote branch-ı `-f` flag-ı ilə push əmrinizə məcburi şəkildə yeniləyə biləcəyinizi müəyyən edə bilərsiniz.

Git-ə güc tətbiqetməsindən imtina etməsini söyləmək üçün `receive.denyNonFastForwards` seçin:

```
$ git config --system receive.denyNonFastForwards true
```

Bunu edə biləcəyiniz başqa bir yol, server tərəfdən qəbul hook-larıdır və onları indi bir az əhatə edəcəyik. Bu yanaşma, müəyyən bir istifadəçi qrupuna sürətli olmayan hücumları inkar etmək kimi daha mürəkkəb şeylər etməyə imkan verir.

`receive.denyDeletes`

`denyNonFastForwards` siyasətinin həll yollarından biri də istifadəçinin branch-ı silmək və sonra yeni istinadla geri push etməsidir. Bunun qarşısını almaq üçün, `receive.denyDeletes`-i true olaraq seçin:

```
$ git config --system receive.denyDeletes true
```

Bu, branch və ya etiketlərin silinməsini inkar edir - heç bir istifadəçi bunu edə bilməz. Remote branch-ları silmək üçün ref fayllarını serverdən manual şəkildə çıxarmalısınız. Bunu ACL-lər vasit əsilə hər istifadəçi bazasında etmək üçün daha maraqlı yollar var, hansı ki Git-Enforced Siyasət Nümunəsi-də öyrənəcəksiniz.

Git Atributları

Bu parametrlərdən bəziləri bir path üçün də göstərilə bilər, belə ki Git bu parametrləri yalnız bir subdirectory və ya bir fayl dəsti üçün tətbiq edir. Bu path-a xas olan parametrlərə Git atributları deyilir və ya dizinlərinizdən birindəki `.gitattributes` sənədində (normal olaraq layihənizin root-u) və ya `.git/info/attributes` faylında quraşdırılır. Layihənizlə əlaqəli xüsusiyyətlər sənədini istəyin.

Atributlardan istifadə edərək layihənizdə ayrı-ayrı fayllar və ya qovluqlar üçün ayrı birləşmə strategiyaları müəyyənləşdirmək, Git-ə mətndən kənar faylları necə fərqləndirəcəyini söyləmək və ya Git-ə daxil olub-çıxmadan əvvəl Git filtri məzmununa sahib olmaq kimi şeylər edə bilərsiniz. Bu bölmədə Git layihənizdə yollarınıza qura biləcəyiniz bəzi xüsusiyyətlər haqqında məlumat əldə edəcək və bu xüsusiyyətdən praktikada istifadə etmək üçün bir neçə nümunə əldə edə bilərsiniz.

Binary Fayllar

Git atributlarından istifadə edə biləcəyiniz yaxşı fəndlərdən biri, Git-ə hansı faylların binary olduğunu söyləmək (əks halda bunu başa düşməyə bilər) və Git-ə bu sənədləri necə idarə etməsi barədə xüsusi təlimatlar verməkdir. Məsələn, bəzi mətn faylları maşın şəklində yaradıla bilər və f ərqlilik göstərməyə bilər, bəzi binary fayllar isə fərqli ola bilər. Git-ə hansının necə olduğunu necə izah edəcəyini görəcəksiz.

Binary Faylların Müəyyənləşdirilməsi

Bəzi fayllar mətn fayllarına bənzəyir, lakin bütün məqsədlər üçün binary məlumat kimi qəbul edilməlidir. Məsələn, macOS-dakı Xcode layihələri, `.pbxproj` ilə bitən, əsasən IDE tərəfindən diskə yazılmış bir JSON (düz mətnli JavaScript məlumat formatı) verilənlər bazası olan bir sənəd ehtiva edir və s. Texniki cəhətdən bir mətn faylı olmasına baxmayaraq (hamısı UTF-8 olduğu üçün), hə qiqətən lightweight bir verilənlər bazası olduğu üçün bu şəkildə davranmaq istəmirsiniz - iki nəfər dəyişsə məzmunu birləşdirə bilməzsiniz və fərqlər ümumiyyətlə faydalı deyil. Fayl bir maşın tər əfindən istehlak edilməlidir. Əslində, binary bir fayl kimi davranmaq istəyirsiz. Git-ə bütün `pbxproj` fayllarını binary məlumatlar kimi qəbul etməsini söyləmək üçün `.gitattributes` faylınıza aşağıdakı sətri əlavə edin:

```
*.pbxproj binary
```

İndi Git, CRLF problemlərini çevirməyə və ya düzəltməyə çalışmayacaq; proyektinizdə `git show` və ya `git diff` işlətdiyiniz zaman bu fayldakı dəyişikliklər üçün bir fərqi hesablamağa və ya çap etm əyə çalışmayacaq.

Binary Faylları Fərqləndirmək

Binary faylları effektiv şəkildə ayırmaq üçün Git atributları funksiyasından da istifadə edə bil ərsiniz. Bunu Git-ə binary məlumatlarınızı normal fərq ilə müqayisə edilə bilən bir mətn formatına necə çevirəcəyinizi söyləyərək edirsiniz.

Birincisi, bu texnikanı insana məlum olan ən əsəbi problemlərdən birini həll etmək üçün istifadə edəcəksiniz: Microsoft Word sənədlərinin versiyaya nəzarəti. Hər kəs Word-un ətrafındakı ən qorxunc redaktor olduğunu bilir, amma qəribədir ki, hər kəs hələ də onu istifadə edir. Word sənə dlərini versiya ilə idarə etmək istəyirsinizsə, onları Git deposuna saxlaya və bir dəfəyə edə bil ərsiniz; bəs bunun nə faydası var? Normal olaraq `git diff` işlədirsinizsə, yalnız belə bir şey görürsünüz:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 88839c4..4afcb7c 100644
Binary files a/chapter1.docx and b/chapter1.docx differ
```

Əgər yoxlamamısınız və manual olaraq skan etməmisinizsə, iki versiyanı birbaşa müqayisə edə bilməzsiniz, düzdür? Bunu Git atributlarından istifadə edərək kifayət qədər yaxşı edə biləcəyiniz ortaya çıxdı. Aşağıdakı sətri `.gitattributes` faylınıza əlavə edin:

```
*.docx diff=word
```

Bu, Git-ə, bu pattern ilə (`.docx`) uyğun gələn hər hansı bir faylın dəyişiklikləri ehtiva edən bir fərqə baxmağa çalışdığınız zaman “word” filterindən istifadə etməsi lazım olduğunu bildirir. “word” filteri nədir? Siz bunu qurmalısınız. Burada Git’i Word sənədlərini oxunaqlı mətn sənədlərinə çevirmək üçün `docx2txt` proqramından istifadə etmək üçün konfiqurasiya edəcəksiniz, daha sonra düzgün şəkildə fərqlənəcəkdir.

Əvvəlcə `docx2txt` yükləməlisiniz; https://sourceforge.net/projects/docx2txt saytından yükləyə bil ərsiniz. Shell-nizin tapa biləcəyi bir yerə qoymaq üçün `INSTALL` faylındakı təlimatları izləyin. Sonra çıxışı Git-in gözlədiyi formata çevirmək üçün bir wrapper skript yazacaqsınız. Yolunuzda bir yerdə `docx2txt` adlı bir sənəd yaradın və bu məzmunu əlavə edin:

```
#!/bin/bash
docx2txt.pl "$1" -
```

Bu faylı `chmod a+x` etməyi unutmayın. Nəhayət, Git’i bu skriptdən istifadə etmək üçün konfiqurasiya edə bilərsiniz:

```
$ git config diff.word.textconv docx2txt
```

İndi Git bilir ki, iki snapshot arasında bir fərq yaratmağa çalışarsa və hər hansı bir sənəd `.docx` ilə bitərsə, bu sənədləri `docx2txt` proqramı olaraq təyin olunan “word” filterindən keçirməlidir. Bu, onları fərqləndirməyə çalışmadan əvvəl Word sənədlərinizin gözəl mətn əsaslı versiyalarını effektiv şəkildə hazırlayır.

Nümunə: Bu kitabın 1-ci fəsli Word formatına çevrildi və Git deposunda hazırlandı. Sonra yeni bir abzas əlavə edildi. `git diff` nəyi göstərdiyinə baxaq:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 0b013ca..ba25db5 100644
--- a/chapter1.docx
+++ b/chapter1.docx
@@ -2,6 +2,7 @@
 This chapter will be about getting started with Git. We will begin at the beginning
by explaining some background on version control tools, then move on to how to get Git
running on your system and finally how to get it setup to start working with. At the
end of this chapter you should understand why Git is around, why you should use it and
you should be all setup to do so.
 1.1. About Version Control
 What is "version control", and why should you care? Version control is a system that
records changes to a file or set of files over time so that you can recall specific
versions later. For the examples in this book you will use software source code as the
files being version controlled, though in reality you can do this with nearly any type
of file on a computer.
+Testing: 1, 2, 3.
 If you are a graphic or web designer and want to keep every version of an image or
layout (which you would most certainly want to), a Version Control System (VCS) is a
very wise thing to use. It allows you to revert files back to a previous state, revert
the entire project back to a previous state, compare changes over time, see who last
modified something that might be causing a problem, who introduced an issue and when,
and more. Using a VCS also generally means that if you screw things up or lose files,
you can easily recover. In addition, you get all this for very little overhead.
 1.1.1. Local Version Control Systems
 Many people's version-control method of choice is to copy files into another
directory (perhaps a time-stamped directory, if they're clever). This approach is very
common because it is so simple, but it is also incredibly error prone. It is easy to
forget which directory you're in and accidentally write to the wrong file or copy over
files you don't mean to.
```

Git uğurla və qısaca olaraq bizə “Testing: 1, 2, 3.” string-i əlavə etdiyimizi söyləyir, bu düzgündür. Bu təbii ki mükəmməl deyil — formatlaşdırma dəyişiklikləri burada görünməyəcək - amma əlbəttə ki, işləyir.

Bu şəkildə həll edə biləcəyiniz bir başqa maraqlı problem, şəkil sənədlərinin diffing edilməsidir. Bunu etmək üçün bir üsul, şəkil sənədlərini EXIF məlumatlarını çıxaran bir filtrdən keçirməkdir - əksər şəkil formatları ilə qeyd olunan metadata. `exiftool` proqramını yüklədiyiniz və yüklədiyiniz təqdirdə, şəkillərinizi metadata aid mətnə çevirmək üçün istifadə edə bilərsiniz, beləliklə ən azı f ərq sizə baş verən bütün dəyişikliklərin mətn şəklində göstərilməsini göstərəcəkdir. Aşağıdakı sətri `.gitattributes` faylınıza qoyun:

```
*.png diff=exif
```

Bu aləti istifadə etmək üçün Git’i konfiqurasiya edin:

```
$ git config diff.exif.textconv exiftool
```

Layihənizdəki bir şəkli əvəz edib `git diff` işlədirsinizsə, belə bir şey görürsünüz:

```
diff --git a/image.png b/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
 ExifTool Version Number         : 7.74
-File Size                       : 70 kB
-File Modification Date/Time     : 2009:04:21 07:02:45-07:00
+File Size                       : 94 kB
+File Modification Date/Time     : 2009:04:21 07:02:43-07:00
 File Type                       : PNG
 MIME Type                       : image/png
-Image Width                     : 1058
-Image Height                    : 889
+Image Width                     : 1056
+Image Height                    : 827
 Bit Depth                       : 8
 Color Type                      : RGB with Alpha
```

Fayl ölçüsünün və şəkil ölçülərinin hər ikisinin dəyişdiyini asanlıqla görə bilərsiniz.

Keyword Expansion

SVN və ya CVS üslubunda keyword genişləndirilməsi çox vaxt bu sistemlərdə istifadə olunan developerlər tərəfindən tələb olunur. Git-də bunun əsas problemi, commit etdikdən sonra commit barədə məlumatı olan bir faylı dəyişdirə bilməməyinizdir, çünki Git əvvəlcə bu faylı yoxlayır. Bununla birlikdə, mətni yoxlandıqda bir fayla daxil edə və commit-ə əlavə olunmadan yenidən silə bilərsiniz. Git atributları sizə bunun üçün iki yol təklif edir.

Əvvəlcə, bir blobun SHA-1 hesablama məbləğini avtomatik olaraq fayldakı `$Id$` sahəsinə daxil edə bilərsiniz. Bu xüsusiyyəti bir faylda və ya fayl dəstində qursanız, bu branch-ı növbəti dəfə yoxladığınız zaman Git bu sahəni blokun SHA-1 ilə əvəz edəcəkdir. Commit-in SHA-1-nin deyil, blob-un özünün olduğuna diqqət yetirmək vacibdir. Aşağıdakı sətri `.gitattributes` faylınıza qoyun:

```
*.txt ident
```

Bir test sənədinə `$Id$` referansına əlavə edin:

```
$ echo '$Id$' > test.txt
```

Növbəti dəfə bu fayla baxanda Git blobun SHA-1-ini vurur:

```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

Ancaq bu nəticə məhdud istifadə üçündür. CVS-də və ya Subversion-da keyword əvəzetməsindən istifadə etmisinizsə, bir məlumat datestamp əlavə edə bilərsiniz - SHA-1 o qədər də faydalı deyil, çünki kifayət qədər təsadüfi və yalnız onlara baxmaqla bir SHA-1-in digərindən daha yaşlı və ya daha yeni olduğunu deyə bilməzsiniz.

commit/checkout fayllarında dəyişiklik etmək üçün öz filtrlərinizi yaza biləcəyiniz ortaya çıxdı. Bunlara “clean” və “smudge” filtrləri deyilir.

`.gitattributes` faylında, müəyyən yollar üçün bir filtr qura və sonra sənədləri yoxlanılmadan d ərhal əvvəl işləyəcək skriptlər qura bilərsiniz (“smudge”, bax “smudge” filtri checkout zamanı işl əyir) və bunlardan bir az əvvəl. səhnələşdirilmişdir (“clean”, bax Fayllar səhnələşdirildikdə “clean” filtr işə salınır). Bu filtrlər hər cür əyləncəli şeylər etmək üçün qurula bilər.

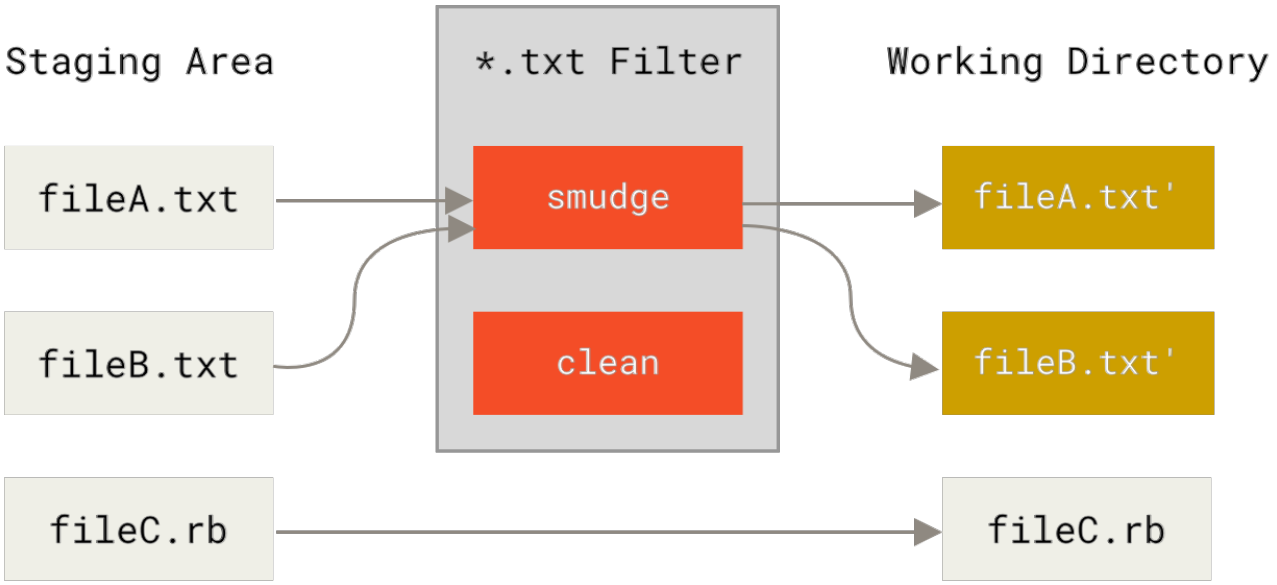


Figure 144. “smudge” filtri checkout zamanı işləyir

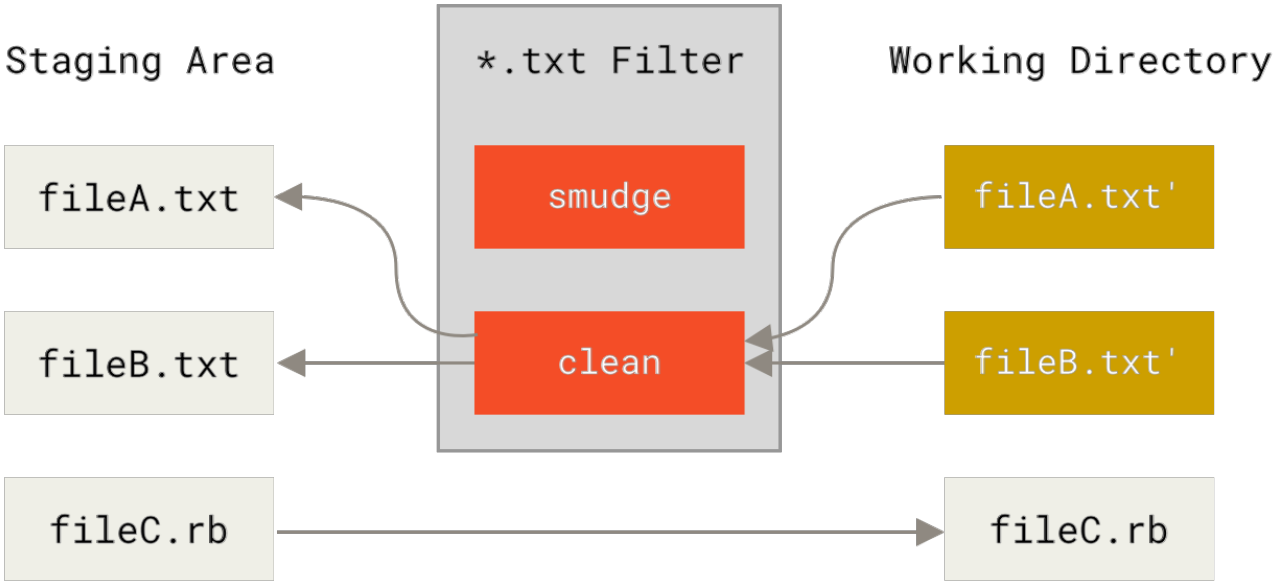


Figure 145. Fayllar səhnələşdirildikdə "clean" filtr işə salınır

Bu xüsusiyyət üçün orijinal commit mesajı, bütün C qaynaq kodunuzu işləmədən əvvəl `indent` proqramı vasitəsilə işlədən sadə bir nümunə verir. Bunu `.gitattributes` faylında `*.c` faylarını "indent" filtri ilə filtrləmək üçün filtr atributunu quraraq edə bilərsiniz:

```
*.c filter=indent
```

Sonra Git-ə "indent" filtrinin ləkə və təmizləmədə nə etdiyini söyləyin:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

Bu vəziyyətdə, `*.c`-yə uyğun fayllar işlədiyiniz zaman Git onları mərhələləndirmədən əvvəl indent proqramı vasitəsilə işə salacaq və onları diskə qaytarmadan əvvəl `cat` proqramı vasitəsilə işə salacaqdır. `cat` proqramı əslində heç bir şey etmir: daxil olduğu məlumatları verir. Bu birləşmə bütün C mənbə kodu fayllarını committing etmədən əvvəl `indent` ilə effektiv şəkildə filtrləyir.

Digər maraqlı bir nümunə RCS stili olan `$Date$` keyword expansion alır. Bunu düzgün şəkildə yerinə yetirmək üçün bir fayl adı götürən, bu layihə üçün son commit tarixini müəyyənləşdirən və tarixi fayla əlavə edən kiçik bir skriptə ehtiyacınız var. Bunu edən kiçik bir Ruby skriptidir:

```
#! /usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

Bütün skriptlər `git log` əmrindən ən son commit tarixini almaq, stdin-də gördüyü hər hansı bir `$Date$` sətrinə yapışdırmaq və nəticələri yazdırmaqdır - istədiyiniz dildə etmək sadə olmalıdır. ən rahat. Bu fayla `expand_date` adını verib path-nıza qoya bilərsiniz. İndi Git-də bir filtr qurmalısınız

(onu `dater` adlandırın) və kassadakı sənədləri ləkələmək üçün `expand_date` filtrinizi istifadə etm əsini söyləməlisiniz. Bunu başa çatdırmaq üçün Perl ifadəsini istifadə edəcəksiniz:

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\\\$Date[^\\\$]*\\\$/\\\$Date\\\$/"'
```

Bu Perl snippet-i başladığınız yerə qayıtmaq üçün bir `$Date$` sətrində gördüyü hər şeyi silir. Artıq filtriniz hazır olduqda yeni filtrlə əlaqəli bir fayl üçün bir Git atributu quraraq və `$Date$` keyword ilə bir fayl yaradaraq test edə bilərsiniz:

```
date*.txt filter=dater
```

```
$ echo '# $Date$' > date_test.txt
```

Bu dəyişiklikləri edib yenidən faylı yoxlayırsınızsa, keyword düzgün şəkildə əvəzləndiyini görürsünüz:

```
$ git add date_test.txt .gitattributes
$ git commit -m "Test date expansion in Git"
$ rm date_test.txt
$ git checkout date_test.txt
$ cat date_test.txt
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

Bu texnikanın xüsusi tətbiqetmələr üçün nə qədər güclü ola biləcəyini görə bilərsiniz. Diqqətli olmalısınız, çünki `.gitattributes` faylları layihə ilə əlaqələndirilir və ötürülür, lakin driver (bu halda `dater`) belə deyil, buna görə də hər yerdə işləməyəcəkdir. Bu filtrləri hazırladığınız zaman, onlar problemsiz bir şəkildə uğursuz olmalı və layihənin hələ də düzgün işləməsini təmin etm əlidirlər.

Deponuzu İxrac Edin

Git atribut məlumatlarıl ayihənizin bir arxivini ixrac edərkən bəzi maraqlı şeylər etməyə imkan verir.

`export-ignore`

Arxiv yaradan zaman Git-ə müəyyən faylları və ya qovluqları ixrac etməməsini deyə bilərsiniz. Arxiv sənədinizə daxil etmək istəmədiyiniz, ancaq layihənizə baxılmasını istədiyiniz bir subdirectory və ya bir sənəd varsa, bu faylları `export-ignore` atributu ilə təyin edə bilərsiniz.

Məsələn, bir `test/` subdirectory-da bəzi test sənədləriniz olduğunu və bunları layihənizin tarball ixracatına daxil etməyin mənasız olduğunu söyləyin. Git atributları sənədinizə aşağıdakı sətri əlavə edə bilərsiniz:

```
test/ export-ignore
```

İndi layihənizin bir tarball-unu yaratmaq üçün `git archive` işə saldığınız zaman, bu qovluq arxivə daxil edilməyəcəkdir.

`export-subst`

Deployment üçün faylları ixrac edərkən, `export-subst` atributu ilə işarələnmiş seçilmiş fayl hissəl ərinə `git log` formatlaşdırma və keyword-expansion işləmə tətbiq edə bilərsiniz.

Məsələn, layihənizə `LAST_COMMIT` adlı bir sənəd əlavə etmək və `git archive` işə salındıqda avtomatik olaraq ona enjekte edilmiş son commit barədə metadata sahib olmaq istəyirsinizsə, məsələn, `.gitattributes` and `LAST_COMMIT` qura bilərsiniz:

```
LAST_COMMIT export-subst
```

```
$ echo 'Last commit date: $Format:%cd by %aN$' > LAST_COMMIT
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

`git archive` işlədiyiniz zaman arxivləşdirilmiş sənədin məzmunu belə görünür:

```
$ git archive HEAD | tar xCf ../deployment-testing -
$ cat ../deployment-testing/LAST_COMMIT
Last commit date: Tue Apr 21 08:38:48 2009 -0700 by Scott Chacon
```

Əvəzetmələrə, məsələn, commit mesajı və hər hansı bir `git notes` daxil ola bilər və `git log` sadə word wrapping edə bilər:

```
$ echo '$Format:Last commit: %h by %aN at %cd%n%+w(76,6,9)%B$' > LAST_COMMIT
$ git commit -am 'export-subst uses git log'\''s custom formatter

git archive uses git log'\''s `pretty=format:` processor
directly, and strips the surrounding `$Format:` and `$`
markup from the output.
'
$ git archive @ | tar xfO - LAST_COMMIT
Last commit: 312ccc8 by Jim Hill at Fri May 8 09:14:04 2015 -0700
       export-subst uses git log's custom formatter

         git archive uses git log's `pretty=format:` processor directly, and
         strips the surrounding `$Format:` and `$` markup from the output.
```

Nəticədə yaradılan arxiv yerləşdirmə işləri üçün əlverişlidir, lakin ixrac olunan hər hansı bir arxiv kimi uyğundur, daha da inkişaf etdirmək üçün uyğun deyil.

Merge Strategies

Ayrıca, Git-ə layihənizdə müəyyən sənədlər üçün fərqli birləşmə strategiyaları istifadə etməsini söyləmək üçün Git atributlarından istifadə edə bilərsiniz. Çox faydalı seçimlərdən biri, Git-ə müə yyən faylları bir-birinə konflikt olduqda birləşdirməyə çalışmamasını, əksinə birləşmə tərəfinizi başqasının üzərindən istifadə etməsini söyləməkdir.

Bu, dəyişiklikləri yenidən birləşdirmək və müəyyən sənədləri görməməzlikdən gəlmək istəsəniz, layihənizdəki bir branch ayrıldığı və ya ixtisaslaşmış olduğu halda faydalıdır. İki verilənlər bazasında fərqli olan `database.xml` adlı verilənlər bazası parametrləri sənədinizin olduğunu və verilənlər bazası faylını qarışdırmadan digər branch-nızda birləşmək istədiyinizi söyləyin. Belə bir atribut qura bilərsiniz:

```
database.xml merge=ours
```

Və sonra bir dummy `ours` birləşmə strategiyasını müəyyənləşdirin:

```
$ git config --global merge.ours.driver true
```

Digər verilənlər bazasında birləşirsinizsə, `database.xml` faylı ilə birləşmə konfliktləri yerinə, buna bənzər bir şey görürsünüz:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

Bu vəziyyətdə, `database.xml` əvvəldə olduğunuz hər hansı bir versiyada qalır.

Git Hook'ları

Bir çox digər Versiya İdarəetmə Sistemləri kimi, Git də müəyyən vacib tədbirlər baş verdikdə xüsusi skriptləri söndürmək üçün bir yola malikdir. Bu hook-ların iki qrupu var: müştəri tərəfi və server tərəfi. Müştəri tərəfindəki hook-lar committing və birləşmə kimi əməliyyatlarla başlayır, server tərəfindəki hook-lar, push edilmiş commit-lərin alınması kimi şəbəkə əməliyyatlarında işl əyir. Bu hook-lardan hər cür səbəbə görə istifadə edə bilərsiniz.

Hook'un Qurulması

Hook-lar hamısı Git qovluğunun `hooks` subdirectory-da saxlanılır. Əksər layihələrdə bu `.git/hooks` -dur. `git init` ilə yeni bir depo qurduğunuzda, Git, hook-lar qovluğunu bir çox nümunə skript ilə doldurur, bir çoxu bunlar üçün yararlıdır; eyni zamanda hər bir skriptin giriş dəyərlərini sənədl əşdirirlər. Bütün nümunələr bir hissəsi Perl-in atıldığı shell skriptləri kimi yazılır, lakin düzgün adlandırılan hər hansı bir icra olunan skript yaxşı işləyəcək - onları Ruby və ya Python-da və ya tanış olduğunuz hər hansı bir dildə yaza bilərsiniz. Birləşdirilmiş hook skriptlərindən istifadə etm ək istəyirsinizsə, adını dəyişdirməlisiniz; fayl adlarının hamısı `.sample` ilə bitir.

Bir hook skriptini işə salmaq üçün, müvafiq olaraq adlandırılan (heç bir extension olmadan) və icra oluna bilən .git qovluğunuzun `hooks` subdirectory-ə bir fayl qoyun. O nöqtədən irəli çağırılmalıdır. Burada əsas hook fayl adlarının əksəriyyətini əhatə edəcəyik.

Müştəri Tərəfindəki Hook'lar

Bir çox müştəri-tərəfi hook var. Bu bölmə onları committing-workflow hook-larına, email-workflow skriptlərinə və hər şeyə bölür.

Müştəri tərəfindəki hook-ların bir depo klonlandığında **deyil** kopyalandığını qeyd etmək vacibdir. Bu skriptlərlə niyyətiniz bir policy-ni tətbiq etməkdirsə, bunu server tərəfində etmək istərdiniz; Git-Enforced Siyasət Nümunəsi-də nümunəyə baxın.

Committing-Workflow Hook'ları

İlk dörd hook, işləmə prosesi ilə əlaqəlidir.

`pre-commit` hook-nu bir commit mesajı yazmadan əvvəl işə salın. Tətbiq ediləcək snapshot-u yoxlamaq, bir şeyi unutduğunuzu yoxlamaq, testlərin işlədiyinə əmin olmaq və ya kodda yoxlamaq üçün lazım olan hər şeyi araşdırmaq üçün istifadə olunur. Bu hook-dan non-zero-dan çıxmaq, commit-i ləğv edir, baxmayaraq ki, `git commit --no-verify` ilə atlaya bilərsiniz. Kod tərzini yoxlamaq (`lint` və ya buna bənzər bir şey işlətmək), whitespace boşluğunu yoxlamaq (standart hook məhz bunu edir) və ya yeni metodlarla uyğun sənədlərin olmasını yoxlamaq kimi işləri edə bilərsiniz.

Commit mesaj redaktoru işə düşməmişdən əvvəl, lakin default mesajı yaradıldıqdan əvvəl `prepare-commit-msg` hook-unu işə salın. Commit müəllifi görməmişdən əvvəl standart mesajı düzəltməyə imkan verir. Bu hook bir neçə parametr götürür: indiyə qədər commit mesajını tutan faylın yolu, commit-in növü və bu düzəliş edilmiş bir commit-dirsə, SHA-1 commit-i.

Bu hook ümumiyyətlə normal commit-lər üçün faydalı deyildir; daha doğrusu, şablon mesajlar, birləşmə commit-ləri, düzəliş edilmiş commit-lər kimi standart mesajın avtomatik olaraq yaradıldığı commit-lər üçün yaxşıdır. Proqramı daxil etmək üçün bir commit şablonu ilə birlikdə istifadə edə bilərsiniz.

`commit-msg` hook-u bir parametr alır, bu da yenə developerlər tərəfindən yazılan commit mesajını alan keçici bir faylın yoludur. Bu skript sıfırdan kənar olarsa, Git commit götürmə prosesini ləğv edir, beləliklə proyekt vəziyyətinizi təsdiqləmək və ya commit-in yerinə yetirilməsinə icazə vermə dən əvvəl mesaj göndərmək üçün istifadə edə bilərsiniz. Bu fəslin son hissəsində, commit mesajınızın tələb olunan bir nümunəyə uyğun olub olmadığını yoxlamaq üçün bu hook-dan istifadə edəcəyik.

Bütün commit prosesi başa çatdıqdan sonra `post-commit` hook-u işləyir. Heç bir parametr tələb etmir, ancaq `git log -1 HEAD`-u işə salmaqla sonuncu commit-i asanlıqla əldə edə bilərsiniz. Ümumiyyətlə, bu skript bildiriş və ya oxşar bir şey üçün istifadə olunur.

E-poçt İş Axını Hook'ları

E-poçt əsaslı workflow üçün üç müştəri tərəfində hook qura bilərsiniz. Hamısı `git am` əmri ilə çağırılır, buna görə workflow əmrinizdə bu əmrdən istifadə etmirsinizsə, növbəti hissəyə etibarlı şəkildə keçə bilərsiniz. Əgər `git format-patch` tərəfindən hazırlanmış e-poçt üzərindən patch-lar alırsınızsa, bunlardan bəziləri sizin üçün faydalı ola bilər.

İşlədilən ilk hook `applypatch-msg`-dir. Tək bir arqument tələb olunur: təklif edilən commit mesajını ehtiva edən müvəqqəti sənədin adı. Bu skript non-zero-dan çıxsa Git patch-ı ləğv edir. Bunu bir commit mesajının düzgün şəkildə formatlandığından əmin olmaq və ya skriptin yerində düzəldər ək mesajı normallaşdırmaq üçün istifadə edə bilərsiniz.

`git am` vasitəsilə patch-lar tətbiq edilərkən işə salınacaq növbəti hook `pre-applypatch`-dir. Bu bir az qarışıqdır, patch tətbiq olunduqdan *sonra* işlənir, lakin bir commit götürülmədən əvvəl, snapshotunu yoxlamaq üçün istifadə edə bilərsiniz.

Bu skriptlə testlər apara və ya işləyən ağacı yoxlaya bilərsiniz. Bir şey əskikdirsə və ya testlər keçmirsə, non-zero-dan çıxmaq patch etmədən `git am` skriptini ləğv edir.

Bir `git am` əməliyyatı zamanı işləyən son hook, commit götürüldükdən sonra işləyən `post-applypatch`-dir. Bunu bir qrupa və ya pull etdiyiniz patch-ın müəllifinə bunu etdiyinizi bildirmək üçün istifadə edə bilərsiniz. Bu skriptlə patching prosesini dayandıra bilməzsiniz.

Başqa Müştəri Hook'ları

`pre-rebase` hook-u hər hansı bir şeyi təzələmədən əvvəl işləyir və non-zero-dan çıxaraq prosesi dayandıra bilər. Bu hook-dan artıq push edilmiş commit-lərin azaldılmasına icazə verməmək üçün istifadə edə bilərsiniz. Git'in quraşdırdığı `pre-rebase` hook nümunəsi bunu edir, baxmayaraq ki, workflow uyğun gəlməyəcək bəzi fərziyyələr irəli sürür.

`post-rewrite` hook-u `git commit --amend` və `git rebase` (`git filter-branch` tərəfindən olmasa da) kimi commit-ləri əvəz edən əmrlər tərəfindən idarə olunur. Tək arqumenti, yenidən yazmağı hansı əmrlə işə salmasıdır və `stdin`-də yenidən yazılanların siyahısını alır. Bu hook-un `post-checkout` və `post-merge` hook-larla eyni istifadəsi çoxdur.

Uğurlu bir `git checkout` etdikdən sonra `post-checkout` hook-u işləyir; iş qovluğunu layihə mühitini düzgün qurmaq üçün istifadə edə bilərsiniz. Bu, mənbədən idarə olunan avtomatik olaraq yaradan sənədləri və ya bu sətirdə bir şey istəmədiyiniz böyük ikili sənədlərdə hərəkət etmək deməkdir.

`post-merge` hook-u uğurlu `merge` əmrindən sonra işləyir. Bunu Git-in izləyə bilmədiyi iş ağacındakı icazə məlumatları kimi məlumatları bərpa etmək üçün istifadə edə bilərsiniz. Bu hook eyni şəkildə işləyən ağac dəyişdikdə kopyalamaq istədiyiniz Git nəzarətindən kənar sənədlərin mövcudluğunu təsdiqləyə bilər.

Uzaqdakı istinadlar yeniləndikdən sonra, lakin hər hansı bir obyekt köçürülməmişdən əvvəl, `pre-push` hook-u `git push` zamanı işləyir. Remote-un adını və yerini parametr olaraq alır və `stdin` vasit əsilə yenilənəcək reflərin siyahısını alır. Bir push etmə meydana gəlməzdən əvvəl bir sıra ref yeniləmələrini təsdiqləmək üçün istifadə edə bilərsiniz (non-zero bir çıxış kodu push etməyi ləğv edəcək).

Git, bəzən normal fəaliyyətinin bir hissəsi olaraq garbage collection-u, `git gc --auto` əmrini işlədə rək edir. `pre-auto-gc` hook garbage collection-dan bir az əvvəl çağırılır və bunun baş verdiyini xəb ərdar etmək və ya yaxşı bir vaxt olmadığı təqdirdə collection-u ləğv etmək üçün istifadə edilə bilər.

Server Tərəfindəki Hook'lar

Müştəri tərəfindəki hook-lara əlavə olaraq, layihə üçün hər cür siyasəti tətbiq etmək üçün bir sistem administratoru kimi bir neçə vacib server tərəfli hook-dan istifadə edə bilərsiniz. Pre hook-lar push etməni rədd etmək və müştəriyə bir səhv mesajı yazdırmaq üçün istənilən vaxt non-zero-dan çıxa bilər; istədiyiniz qədər kompleks bir push etmə siyasəti yarada bilərsiniz.

`pre-receive`

Müştəri tərəfdən bir push etmə işləyərkən işləyən ilk skript `pre-receive`-dır. Stdin-dən pushed edil ən istinadların siyahısını alır; non-zero-dan çıxsa, heç biri qəbul edilmir.

Bu hook-dan istifadə edərək güncəllənmiş istinadların heç birinin sürətli göndərilmədiyinə əmin olmaq və ya push etməklə dəyişdirdikləri bütün istinadlar və sənədlər üçün giriş nəzarəti etmək üçün istifadə edə bilərsiniz.

`update`

`update` skripti `pre-receive` skriptinə çox oxşayır, yalnız push etmə yeniləməyə çalışdığı hər branch üçün bir dəfə işlədilir. Pusher birdən çox branch-a push etməyə çalışırsa, `pre-receive` yalnız bir də fə işləyir, yeniləmə push etdikləri hər branch üçün bir dəfə işləyir. Stdin-dən oxumaq əvəzinə bu skript üç arqument götürür: istinadın adı (branch), istinaddan əvvəl göstərilən SHA-1 və istifadəçi push etməyə çalışdığı SHA-1. Yeniləmə skripti non-zero-dan kənar olarsa, yalnız həmin müraciət r ədd edilir; digər istinadlar yenilənə bilər.

`post-receive`

`post-receive` hook bütün proses başa çatdıqdan sonra işləyir və digər xidmətləri yeniləmək və ya istifadəçiləri xəbərdar etmək üçün istifadə edilə bilər. `pre-receive` hook-la eyni stdin məlumatlarını alır. Nümunələr siyahının elektron poçtla göndərilməsi, davamlı inteqrasiya serverinə bildiriş verilməsi və ya ticket-tracking sisteminin yenilənməsi daxildir - hətta biletlərin açılmalı, d əyişdirilməli və ya bağlanmalı olub olmadığını görmək üçün commit mesajlarını təhlil edə bil ərsiniz. Bu skript push etmə prosesini dayandıra bilməz, lakin müştəri tamamlanana qədər əlaq əsini kəsmir, buna görə uzun müddət ala biləcək bir şey etməyə çalışsanız ehtiyatlı olun.

Git-Enforced Siyasət Nümunəsi

Bu hissədə, xüsusi bir commit mesajı formatını yoxlayan və yalnız müəyyən istifadəçilərin bir layihədəki müəyyən alt qovluğu dəyişdirməsinə imkan verən bir Git workflow qurmaq üçün öyr əndiklərinizi istifadə edəcəksiniz. Developer-in push-nunun rədd ediləcəyini və siyasətləri həqiqə tən tətbiq edən server skriptlərini bilməsinə kömək edən müştəri skriptləri quracaqsınız.

Göstərəcəyimiz ssenarilər Ruby ilə yazılmışdır; qismən intellektual hərəkətsizliyimizə görə, həm də Ruby-ni mütləq yaza bilməsəniz də oxumaq asandır. Bununla birlikdə, hər hansı bir dil işləyəc əkdir - Git ilə paylanan bütün nümunə hook skriptləri ya Perl ya da Bash-dadır, buna görə nümunə lərə baxaraq bu dillərdə çox sayda hook nümunəsi görə bilərsiniz.

Server Tərəf Hook'u

Bütün server tərəfli işlər `hooks` qovluğunuzdakı `update` faylına daxil olacaq. `update` hook-u hər bir push üçün bir dəfə işləyir və üç arqument götürür:

- push olunan istinadın adı
- Bu branch-ın olduğu köhnə düzəliş
- push olunan yeni versiya

Push SSH üzərində işləyirsə push edən istifadəçiyə də girişiniz var. Hər kəsin ümumi key identifikasiyası yolu ilə tək bir istifadəçi ilə ("git" kimi) əlaqə qurmasına icazə verdiyiniz təqdirdə, bu istifadəçiyə açıq key-ə əsasən hansı istifadəçinin birləşdiyini təyin edən shell wrapper verm əlisiniz və müvafiq olaraq bu mühit dəyişəndir. Burada əlaqə istifadəçisinin `$USER` mühit dəyiş ənində olduğunu düşünəcəyik, buna görə də yeniləmə skriptiniz lazım olan bütün məlumatları toplamaqla başlayır:

```
#!/usr/bin/env ruby

$refname = ARGV[0]
$oldrev  = ARGV[1]
$newrev  = ARGV[2]
$user    = ENV['USER']

puts "Enforcing Policies..."
puts "(#{$refname}) (#{$oldrev[0,6]}) (#{$newrev[0,6]})"
```

Bəli, bunlar qlobal dəyişənlərdir. Mühakimə etməyin - bu şəkildə nümayiş etdirmək daha asandır.

Xüsusi bir Commit-Mesaj Formatının Tətbiq Edilməsi

İlk probleminiz hər bir commit mesajının müəyyən bir formata riayət etməsini təmin etməkdir. Yalnız bir hədəfə sahib olmaq üçün hər bir mesajın "ref: 1234" kimi görünən bir simli daxil etdiyini fərz edin, çünki hər bir commit-in bilet sisteminizdəki bir iş elementi ilə əlaqələndirilməsini ist əyirsiniz. Hər bir push-un yuxarı qaldırıldığına baxmalısınız, bu sətirin commit mesajında olub olmadığını görməlisiniz və əgər sətir heç bir commit-də yoxdursa, non-zero-dan çıxın, beləliklə push rədd edilir.

`$newrev` və `$oldrev` dəyərlərini götürərək `git rev-list` adlı Git plumbing əmrinə ötürərək push edil ən bütün commit-lərin SHA-1 dəyərlərinin siyahısını əldə edə bilərsiniz. Bu, əsasən `git log` əmridir, lakin standart olaraq yalnız SHA-1 dəyərlərini yazdırır və başqa heç bir məlumat yoxdur. Beləliklə, bir SHA-1 əmri ilə digər biri arasında tətbiq olunan bütün SHA-1-lərin siyahısını almaq üçün belə bir şey edə bilərsiniz:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

Bu nəticəni götürə bilər, SHA-1-ləri işləyənlərin hər birindən keçə bilər, bunun üçün mesajı götürə və bu mesajı nümunə axtaran normal bir ifadəyə qarşı test edə bilərsiniz.

Bu commit-lərin hər birindən test etmək üçün commit mesajını necə alacağınızı bilməlisiniz. Raw commit məlumatlarını almaq üçün, `git cat-file` adlı başqa bir plumbing əmrindən istifadə edə bilərsiniz. Plumbing əmrlərinə daha detallı Git'in Daxili İşləri-dan baxacağıq; ancaq hələlik bu əmrin sizə verdiyi budur:

```
$ git cat-file commit ca82a6
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

Change the version number
```

SHA-1 dəyərinə sahib olduqda commit mesajını commit-dən əldə etməyin sadə bir yolu ilk boş sətirə keçmək və bundan sonra hər şeyi götürməkdir. Bunu Unix sistemlərindəki `sed` əmri ilə edə bilərsiniz:

```
$ git cat-file commit ca82a6 | sed '1,/^$/d'
Change the version number
```

Uyğun olmayan bir şey görsəniz, push etməyə və çıxmağa çalışan hər bir commit-dən commit mesajını almaq üçün bu tilsimi istifadə edə bilərsiniz. Ssenaridən çıxmaq və push-u rədd etmək üçün non-zero'dan çıxın. Bütün metod belə görünür:

```
$regex = /\[ref: (\d+)\]/

# enforced custom commit message format
def check_message_format
  missed_revs = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  missed_revs.each do |rev|
    message = `git cat-file commit #{rev} | sed '1,/^$/d'`
    if !$regex.match(message)
      puts "[POLICY] Your message is not formatted correctly"
      exit 1
    end
  end
end
check_message_format
```

Bunu `update` skriptinə qoymağınız qaydanıza əməl etməyən mesajları olan commit-ləri olan yenil əmələri rədd edəcəkdir.

User-Based ACL Sisteminin Tətbiqi

Fərz edək ki, hansı istifadəçilərə layihələrinizin hansı hissələrində dəyişikliklər etməyə icazə verildiyini göstərən bir giriş nəzarəti siyahısı (ACL) istifadə edən bir mexanizm əlavə etmək ist əyirsiniz. Bəzi insanlar tam giriş hüququna malikdirlər, bəziləri isə yalnız müəyyən alt qovluqlara və ya müəyyən fayllara dəyişiklik edə bilər. Bunu tətbiq etmək üçün bu qaydaları serverdəki boş Git deposunda yaşayan `acl` adlı bir fayla yazacaqsınız. Bu qaydalara baxaraq `update` hook-na baxacaqsınız, push etdiyiniz bütün commit-lər üçün hansı faylların təqdim olunduğuna baxın və push edən istifadəçinin bütün bu faylları yeniləməyə giriş imkanının olub olmadığını müəyyənl əşdirəcəksiniz.

Edəcəyiniz ilk şey ACL yazmaqdır. Burada CVS ACL mexanizmi kimi bir formatdan çox istifadə ed əcəksiniz: birinci sahənin `avail` və ya `unavail` olduğu bir sıra sətirlərdən istifadə edir, növbəti sahə istifadəçilərin vergüllə ayrılmış siyahısıdır,sonra qayda tətbiq olunur və son sahə qaydanın tətbiq olunduğu path-dir (blank açıq giriş deməkdir). Bu sahələrin hamısı boru (`|`) işarəsi ilə ayrılmışdır.

Bu vəziyyətdə, bir neçə idarəçiniz var, bəzi sənəd yazarlarına `doc` qovluğuna giriş imkanı var və yalnız `lib` və `tests` qovluqlarına çıxışı olan bir developer var və ACL dosyanız belə görünür:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

Bu məlumatları istifadə edə biləcəyiniz bir quruluşu oxumaqla başlayırsınız. Bu vəziyyətdə, nümunəni sadə saxlamaq üçün yalnız `avail` direktivlərini yerinə yetirəcəksiniz. Burada key-in istifadəçi adı olduğu və dəyərin istifadəçinin yazma girişinə sahib olduğu bir sıranın olduğu assosiativ bir sıra verən metod var:

```
def get_acl_access_data(acl_file)
  # read in ACL data
  acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
  access = {}
  acl_file.each do |line|
    avail, users, path = line.split('|')
    next unless avail == 'avail'
    users.split(',').each do |user|
      access[user] ||= []
      access[user] << path
    end
  end
  access
end
```

Daha əvvəl baxdığınız ACL faylında bu `get_acl_access_data` metodu aşağıdakı kimi bir məlumat strukturunu qaytarır:

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

Artıq icazələrinizi çeşidlədikdən sonra, push etdiyiniz commit-lərin hansı path-ların d əyişdirildiyini təyin etməlisiniz, beləliklə push edən istifadəçinin hər yerə girişinə əmin ola bil ərsiniz.

`git log` əmrinə `--name-only` seçimi ilə təkcə bir işdə hansı faylların dəyişdirildiyini olduqca asanlıqla görə bilərsiniz (Git'in Əsasları-də qısaca qeyd olunur):

```
$ git log -1 --name-only --pretty=format:'' 9f585d

README
lib/test.rb
```

`get_acl_access_data` metodundan qaytarılan ACL quruluşundan istifadə edirsinizsə və hər bir siyahıda sadalanan fayllarla müqayisə etsəniz, istifadəçinin bütün commit-lərini push etmək imkanının olub olmadığını müəyyən edə bilərsiniz:

```
# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('acl')

  # see if anyone is trying to push something they can't
  new_commits = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  new_commits.each do |rev|
    files_modified = `git log -1 --name-only --pretty=format:'' #{rev}`.split("\n")
    files_modified.each do |path|
      next if path.size == 0
      has_file_access = false
      access[$user].each do |access_path|
        if !access_path  # user has access to everything
           || (path.start_with? access_path) # access to this path
          has_file_access = true
        end
      end
      if !has_file_access
        puts "[POLICY] You do not have access to push to #{path}"
        exit 1
      end
    end
  end
end

check_directory_perms
```

Serverinizə `git rev-list` ilə göndərilən yeni commit-lərin siyahısını alırsınız. Sonra, bu commit-lərin hər biri üçün hansı faylların dəyişdirildiyini tapır və push edən istifadəçinin dəyişdirilən bütün path-lara girişinə əmin olun.

Artıq istifadəçiləriniz pis qurulmuş mesajlarla və ya təyin olunmuş path-ların xaricində dəyişdirilmiş fayllarla heç bir commit götürə bilməzlər.

Testdən Keçirmək

Bütün bu kodu qoymağınız lazım olan fayl olan `chmod u+x .git/hooks/update` işlədirsinizsə və uyğun olmayan bir mesajla bir commit götürməyə çalışsanız, buna bənzər bir şey əldə edirsiniz.

```
$ git push -f origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Burada bir-iki maraqlı şey var. Birincisi, bunu hook işləməyə başladığı yerdə görürsən.

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Yeniləmə skriptinizin əvvəlində onu çap etdiyinizi unutmayın. Ssenarinizin `stdout` ilə əks olunduğu hər hansı bir şey müştəriyə ötürüləcəkdir.

Növbəti şey error mesajıdır.

```
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

Birinci sətir sizin tərəfinizdən çap olundu, digər ikisi Git, yeniləmə skriptinin sıfırdan çıxdığını və bununla push etmənizin azaldığını söylədi. Son olaraq:

```
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Hook-un rədd etdiyi hər bir müraciət üçün remote rədd edilmiş bir mesaj görəcəksiniz və o, bunun hook çatışmazlığı səbəbindən xüsusi olaraq rədd edildiyini bildirir. Bundan əlavə, kimsə girişi olmayan bir faylı düzəltməyə və içərisində olan bir commit-i push etməyə çalışsa, oxşar bir şey görəcəkdir. Məsələn, bir fayl müəllifi `lib` qovluğunda bir şey dəyişdirərək bir commit-i push etmə yə çalışırsa, belə görürür:

```
[POLICY] You do not have access to push to lib/test.rb
```

Bundan sonra, o `update` skripti olduğu və icra edilə biləcəyi müddətdə, deponuzda heç vaxt sizin

nümunəniz olmayan bir commit mesajı olmayacaq və istifadəçiləriniz sandbox altında qalacaq.

Müştəri Tərəf Hook-lar

Bu yanaşmanın mənfi tərəfi, istifadəçilərinizin push-ları rədd edildikdə qaçılmaz olaraq nəticələn əcək. Diqqətlə hazırlanmış işlərinin son anda rədd edilməsi son dərəcə məyus və qarışıq ola bilər; və bundan əlavə tarixlərini düzəltmək üçün edit etməli olacaqlar ki, bu da həmişə ürək qırıqlığıdır.

Bu çıxılmaz vəziyyətin cavabı, istifadəçilərin serverin rədd edə biləcəyi bir şey etdikləri zaman xə bərdar etmək üçün istifadə edə biləcəyi bəzi müştəri tərəfindəki hook-ları təmin etməkdir. Belə liklə, hər hansı bir problemi commit etməzdən əvvəl və bu problemləri həll etmək çətinləşmə mişdən əvvəl düzəldə bilərlər. Hook-lar bir layihənin klonu ilə köçürülmədiyi üçün bu skriptləri başqa bir şəkildə paylamalı və sonra istifadəçilərinizdən bunları `.git/hooks` qovluğuna kopyalayıb icraya hazır etməlisiniz. Bu hook-ları proyekt daxilində və ya ayrı bir layihədə paylaya bilərsiniz, lakin Git onları avtomatik olaraq qurmayacaqdır. Başlamaq üçün, hər bir commit qeydə alınmazdan əvvəl commit mesajınızı yoxlamalısınız, belə ki, serverin pis formatlanmış commit mesajlarına görə dəyişikliklərinizi rədd etməyəcəyini bilirsiniz. Bunu etmək üçün `commit-msg` hook-u əlavə edə bilərsiniz. Birinci arqument kimi ötürülən fayldan mesajı oxudunuzsa və bunu nümunə ilə müqayisə etsəniz, uyğunlaşma olmadığı təqdirdə Git-i commit-i ləğv etməyə məcbur edə bilərsiniz:

```
#!/usr/bin/env ruby
message_file = ARGV[0]
message = File.read(message_file)

$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
  puts "[POLICY] Your message is not formatted correctly"
  exit 1
end
```

Əgər həmin skript yerindədirsə (`.git/hooks/commit-msg`-də) və işlədilə bilərsə və düzgün hazırlanmamış bir mesajla məşğul olursunuzsa, bunu görürsünüz:

```
$ git commit -am 'Test'
[POLICY] Your message is not formatted correctly
```

Bu vəziyyətdə heç bir commit tamamlanmamışdır. Bununla birlikdə, mesajınız uyğun bir pattern-i ehtiva edirsə, Git sizə imkan verir:

```
$ git commit -am 'Test [ref: 132]'
[master e05c914] Test [ref: 132]
 1 file changed, 1 insertions(+), 0 deletions(-)
```

Sonra ACL əhatənizdən kənar faylları dəyişdirmədiyinizə əmin olmaq istəyirsiniz. Layihənizin `.git`

qovluğunda əvvəllər istifadə etdiyiniz ACL sənədinin bir nüsxəsi varsa, aşağıdakı `pre-commit` ssenarisi sizin üçün bu məhdudiyyətləri tətbiq edəcəkdir:

```
#!/usr/bin/env ruby

$user    = ENV['USER']

# [ insert acl_access_data method from above ]

# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('.git/acl')

  files_modified = `git diff-index --cached --name-only HEAD`.split("\n")
  files_modified.each do |path|
    next if path.size == 0
    has_file_access = false
    access[$user].each do |access_path|
    if !access_path || (path.index(access_path) == 0)
      has_file_access = true
    end
    if !has_file_access
      puts "[POLICY] You do not have access to push to #{path}"
      exit 1
    end
  end
end

check_directory_perms
```

Bu, server tərəfindəki hissə ilə təxminən eyni skriptdir, lakin iki mühüm fərq var. Birincisi, ACL faylı başqa bir yerdədir, çünki bu ssenari `.git` qovluğundan deyil, iş qovluğundan işləyir. ACL faylının yolunu buradan dəyişdirməlisiniz:

```
access = get_acl_access_data('acl')
```

to this:

```
access = get_acl_access_data('.git/acl')
```

Digər vacib fərq, dəyişdirilmiş faylların siyahısını əldə etmək üsuludur. Server tərəfindəki metod, tapşırıqların jurnalına baxdığından və bu anda commit hələ qeyd olunmadığından, bunun əvəzinə fayl siyahısını səhnələşdirmə sahəsindən almalısınız. Bunun əvəzinə:

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

you have to use:

```
files_modified = `git diff-index --cached --name-only HEAD`
```

Lakin, sadəcə iki fərq bunlardır - əks halda, ssenari eyni şəkildə işləyir. Bir xəbərdarlıq budur ki, remote machine-a push etdiyiniz eyni istifadəçi ilə yerli olaraq işləməyinizi gözləyir. Fərqlidirsə, `$user` dəyişənini manual olaraq təyin etməlisiniz.

Burada edə biləcəyimiz başqa bir şey də istifadəçinin sürətli göndərilməyən istinadları push etm əsindən əmin olmaqdır. Sürətli olmayan bir istinad almaq üçün ya əvvəldən push etdiyiniz bir commit-i geri qaytarmalısınız, ya da fərqli bir yerli branch-ı eyni remote branch-a push etməyə çalışmalısınız.

Çox güman ki, server bu siyasəti tətbiq etmək üçün artıq `receive.denyDeletes` və `receive.denyNonFastForwards` ilə konfiqurasiya olunmuşdur, buna görə tutmağa çalışa biləcəyiniz t ək təsadüfi şey, artıq push olunmuş commit-lərin geri qaytarılmasıdır.

Budur bunu yoxlayan bir pre-rebase skriptinin nümunəsi. Yenidən yazmaq istədiyiniz bütün commit-lərin siyahısını alır və remote istinadlarınızdan birində olub olmadığını yoxlayır. Remote istinadlarınızdan birinin əlçatan olduğunu görsə, geri qaytarmanı ləğv edir.

```
#!/usr/bin/env ruby

base_branch = ARGV[0]
if ARGV[1]
  topic_branch = ARGV[1]
else
  topic_branch = "HEAD"
end

target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
remote_refs = `git branch -r`.split("\n").map { |r| r.strip }

target_shas.each do |sha|
  remote_refs.each do |remote_ref|
    shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
    if shas_pushed.split("\n").include?(sha)
      puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"
      exit 1
    end
  end
end
```

Bu skript Reviziya Seçimi-də əhatə olunmayan bir sintaksisdən istifadə edir. Bunu işə salmaqla ə vvəlcədən push edilmiş commit-lərin siyahısını alırsınız:

```
`git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
```

`SHA^@` sintaksisi, bu vəzifəni yerinə yetirən bütün valideynləri həll edir. Remote-dakı son işdən əldə edilə bilən və push etməyə çalışdığınız SHA-1-lərin hər hansı bir valideynindən əlçatmaz olan hər hansı bir commit axtarırsınız - yəni bu irəliləyişdir.

Bu yanaşmanın əsas çatışmazlığı çox yavaş ola biləcəyi və çox vaxt lazımsız olmasıdır - itələməni `-f` ilə məcbur etməyə çalışmasanız, server sizi xəbərdar edəcək və push qəbul etməyəcək. Bununla birlikdə, bu maraqlı bir məşqdir və nəzəri cəhətdən daha sonra geri qayıtmaq və düzəltmək m əcburiyyətində qalmağınızın qarşısını almağa kömək edə bilər.

Qısa Məzmun

Git müştəri və serverinizi iş axınınıza və layihələrinizə ən yaxşı şəkildə uyğunlaşdırmağın əsas yollarının əksəriyyətini nəzərdən keçirdik. Hər cür konfiqurasiya parametrləri, fayl əsaslı atributlar və event hook'ları haqqında məlumat əldə etdiniz və və bir nümunə policy-enforcing bir server qurdunuz.

İndi Git'i xəyal edə biləcəyiniz hər hansı bir iş axınına uyğunlaşdırmağı bacarmalısınız.

Git və Digər Sistemlər

Dünya mükəmməl deyil. Ümumiyyətlə, təmasda olduğunuz hər bir layihəni dərhal Git'ə keçirə bilməzsiniz. Bəzən başqa bir VNS istifadə edərək bir layihədə qalırsınız və bunun Git istəyirsiniz. Bu fəslin birinci hissəsini üzərində çalışdığınız layihə fərqli bir sistemdə yerləşdikdə Git'in müştəri kimi istifadəsi yollarını öyrənməyə sərf edəcəyik.

Bir nöqtədə mövcud layihənizi Git'ə çevirmək istəyə bilərsiniz. Bu fəslin ikinci hissəsi, layihənizi bir neçə xüsusi sistemdən Git'ə necə köçürəcəyinizi və əvvəlcədən qurulmuş idxal vasitəsi olmadığı təqdirdə işləyəcək bir üsulu əhatə edir.

Git Müştəri kimi

Git, developer'lər üçün o qədər gözəl bir təcrübə təmin edir ki, komandasının qalan hissəsi tamamilə fərqli bir VNS istifadə etsə də bir çox insan iş yerində necə istifadə ediləcəyini anladı. "Bridges" adlanan bu adapterlərdən bir neçəsi mövcuddur. Burada vəhşi təbəqədə qarşılaşma ehtimalı olanları əhatə edəcəyik.

Git və Subversion

Açıq mənbəli development layihələrinin böyük bir hissəsi və çox sayda korporativ layihə mənbə kodlarını idarə etmək üçün Subversiondan istifadə edir.

On ildən çoxdur ki, mövcuddur və əksər vaxt açıq mənbəli layihələr üçün *de facto* VNS seçimi idi. Bundan başqa, bir çox cəhətdən ondan əvvəl mənbə nəzarəti dünyasının böyük oğlu olan CVS-ə çox oxşayır.

Git-in ən böyük xüsusiyyətlərindən biri də, `git svn` adlanan Subversion-a iki yönlü körpüdür. Bu vasitə, Git'i bir Subversion serverinə etibarlı bir müştəri olaraq istifadə etməyə imkan verir, bel əliklə Git'in bütün lokal xüsusiyyətlərindən istifadə edə bilərsiniz və sonra Subversion'u lokal istifadə etdiyiniz kimi bir Subversion serverinə push edə bilərsiniz. Bu, həmkarlarınız qaranlıq və qədim yollarla işləməyə davam edərkən local branching və birləşmə, səhnə sahəsindən istifadə, rebasing və cherry-picking və s edə bilərsiniz. Git-i korporativ mühitdə gizlətmək və developer-l ərinizin daha effektiv olmasına kömək etmək üçün yaxşı bir yoldur, Git-i tam dəstəkləmək üçün infrastrukturun dəyişdirilməsini təmin edin. Subversion körpüsü DVCS dünyasına açılan bir d ərmandır.

`git svn`

Bütün Subversion körpü əmrləri üçün Git-dəki əsas əmr `git svn`-dir. Bunun üçün kifayət qədər bir neçə əmr lazımdır, buna görə bir neçə sadə iş axınından keçərkən ən çox yayılmışları göstərəcəyik. Qeyd etmək vacibdir ki, `git svn` istifadə edərkən Git-dən çox fərqli işləyən bir sistem olan Subversion ilə qarşılıqlı əlaqə qurursunuz. Local branching və birləşmə işlərini edə bilsəniz də, işinizi bərpa edərək və eyni zamanda bir Git remote deposu ilə qarşılıqlı əlaqə qurmaq kimi şeyl ərdən çəkinərək tarixinizi mümkün qədər xətti tutmaq yaxşıdır. Tarixinizi yenidən yazmayın və t əkrar push etməyə çalışmayın və eyni zamanda Git developerləri ilə əməkdaşlıq etmək üçün paralel bir Git deposuna push etməyin.

Subversion yalnız bir xətti tarixçəyə sahib ola bilər və onu qarışdırmaq çox asandır. Bir komanda ilə işləyirsinizsə, bəziləri SVN, bəziləri Git istifadə edirsinizsə, hər kəsin əməkdaşlıq etmək üçün SVN serverindən istifadə etdiyinə əmin olun - bunu etmək həyatınızı asanlaşdıracaq.

Ayarlamaq

Bu funksiyanı nümayiş etdirmək üçün yazmaq üçün daxil olduğunuz tipik bir SVN deposuna ehtiyacınız var. Bu nümunələri kopyalamaq istəyirsinizsə, bir SVN test deposunun yazılı surətini çıxarmalısınız. Bunu asanlıqla etmək üçün Subversion ilə gələn `svnsync` adlı bir vasitə istifadə edə bilərsiniz.

İzləmək üçün əvvəlcə yeni bir lokal Subversion deposu yaratmalısınız:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Sonra bütün istifadəçilərə revpropları dəyişdirməyə imkan verin - asan yol, həmişə 0-dan çıxan bir `pre-revprop-change` skriptini əlavə etməkdir:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

Artıq bu proyekti depolardan və `svnsync init` çağıraraq lokal maşınınızla sinxronizasiya edə bil ərsiniz.

```
$ svnsync init file:///tmp/test-svn \
  http://your-svn-server.example.org/svn/
```

Bu, sinxronizasiyanı işə salmaq üçün xüsusiyyətləri qurur. Daha sonra kodu işə salmaqla klonlaya bilərsiniz:

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Transmitting file data .............................[...]
Committed revision 2.
Copied properties for revision 2.
[…]
```

Bu əməliyyat yalnız bir neçə dəqiqə çəkə bilsə də, orijinal deponu lokal əvəzinə başqa bir uzaq depoya köçürməyə çalışarsanız, 100-dən az commit olmasına baxmayaraq proses təxminən bir saat çəkəcəkdir. Subversion bir dəfəyə bir revizyonu klonlamalı və sonra başqa bir depoya qaytarmalıdır - gülünc dərəcədə səmərəsizdir, amma bunu etmək üçün yeganə asan yoldur.

Başlayırıq

İndi yazma girişiniz olan bir Subversion deposuna sahib olduğunuz üçün tipik bir iş axınından keçə bilərsiniz. Bütün Subversion deposunu yerli Git deposuna idxal edən `git svn clone` əmri ilə başlayacaqsınız. Unutmayın ki, həqiqi bir yerləşdirilmiş Subversion deposundan idxal edirsinizsə, buradakı `file:///tmp/test-svn`-ni Subversion deposunuzun URL'si ilə əvəz etməlisiniz:

Artıq yazma girişiniz olan bir Subversion deposuna sahib olduğunuz üçün tipik bir iş axınından keçə bilərsiniz. Bütün Subversion deposunu yerli Git deposuna idxal edən `git svn clone` əmri ilə başlayacaqsınız. Unutmayın ki, həqiqi bir yerləşdirilmiş Subversion deposundan idxal edirsinizsə, buradakı `file:///tmp/test-svn`-ni Subversion deponuzun URL'si ilə əvəz etməlisiniz:

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /private/tmp/progit/test-svn/.git/
r1 = dcbfb5891860124cc2e8cc616cded42624897125 (refs/remotes/origin/trunk)
    A   m4/acx_pthread.m4
    A   m4/stl_hash.m4
    A   java/src/test/java/com/google/protobuf/UnknownFieldSetTest.java
    A   java/src/test/java/com/google/protobuf/WireFormatTest.java
…
r75 = 556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae (refs/remotes/origin/trunk)
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-
svn/branches/my-calc-branch, 75
Found branch parent: (refs/remotes/origin/my-calc-branch)
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae
Following parent with do_switch
Successfully followed parent
r76 = 0fb585761df569eaecd8146c71e58d70147460a2 (refs/remotes/origin/my-calc-branch)
Checked out HEAD:
  file:///tmp/test-svn/trunk r75
```

Bu, təqdim etdiyiniz URL-də iki əmr - `git svn init` və ardından `git svn fetch` kimi ekvivalentdir. Bu bir müddət çəkə bilər.

Məsələn, test layihəsində yalnız 75 iş görülürsə və kod bazası o qədər böyük deyilsə, Git buna baxmayaraq hər versiyanı bir-bir yoxlamalı və fərdi olaraq həyata keçirməlidir. Yüzlərlə və ya minlərlə commit-i olan bir layihə üçün bunun sözün əsl mənasında bitməsi saatlarla, hətta günlə rlə davam edə bilər.

`-T trunk -b branches -t tags` hissəsi Git-ə bu Subversion deposunun əsas branching və etiketləmə şərtlərini izlədiyini bildirir. Trunk-nıza, branch-larınıza və ya etiketlərinizə fərqli ad verirsinizsə, bu seçimləri dəyişə bilərsiniz. Bu çox yayılmış olduğundan, bütün hissəni standart düzən mənasını verən və bütün bu variantları nəzərdə tutan `-s` ilə əvəz edə bilərsiniz. Aşağıdakı əmr bərabərdir:

```
$ git svn clone file:///tmp/test-svn -s
```

Bu nöqtədə branch-larınızı və etiketlərinizi idxal edən etibarlı bir Git deposuna sahib olmalısınız:

```
$ git branch -a
* master
  remotes/origin/my-calc-branch
  remotes/origin/tags/2.0.2
  remotes/origin/tags/release-2.0.1
  remotes/origin/tags/release-2.0.2
  remotes/origin/tags/release-2.0.2rc1
  remotes/origin/trunk
```

Bu vasitənin Subversion etiketlərini uzaqdan idarəedici olaraq necə idarə etdiyinə diqqət yetirin. Git plumbing əmri ilə `show-ref` ilə daha yaxından tanış olaq:

```
$ git show-ref
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/heads/master
0fb585761df569eaecd8146c71e58d70147460a2 refs/remotes/origin/my-calc-branch
bfd2d79303166789fc73af4046651a4b35c12f0b refs/remotes/origin/tags/2.0.2
285c2b2e36e467dd4d91c8e3c0c0e1750b3fe8ca refs/remotes/origin/tags/release-2.0.1
cbda99cb45d9abcb9793db1d4f70ae562a969f1e refs/remotes/origin/tags/release-2.0.2
a9f074aa89e826d6f9d30808ce5ae3ffe711feda refs/remotes/origin/tags/release-2.0.2rc1
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/remotes/origin/trunk
```

Git bir Git serverindən klonladıqda bunu etmir; etiketli bir deponun təzə bir klondan sonra necə göründüyü:

```
$ git show-ref
c3dcbe8488c6240392e8a5d7553bbffcb0f94ef0 refs/remotes/origin/master
32ef1d1c7cc8c603ab78416262cc421b80a8c2df refs/remotes/origin/branch-1
75f703a3580a9b81ead89fe1138e6da858c5ba18 refs/remotes/origin/branch-2
23f8588dde934e8f33c263c6d8359b2ae095f863 refs/tags/v0.1.0
7064938bd5e7ef47bfd79a685a62c1e2649e2ce7 refs/tags/v0.2.0
6dcb09b5b57875f334f61aebed695e2e4193db5e refs/tags/v1.0.0
```

Git etiketi uzaq branch-lara müalicə etmək əvəzinə birbaşa `refs/tags`-ə gətirir.

Committing Back to Subversion

Artıq işləyən bir qovluğa sahib olduğunuz üçün Git-dən bir SVN müştərisi kimi effektiv istifadə ed ərək, proyektdə bir az iş görə bilər və commit-lərinizi yuxarı axın istiqamətində push edə bil ərsiniz. Fayllardan birini redaktə edib onu yerinə yetirirsinizsə, Subversion serverində olmayan Git-də yerli olaraq mövcud olan bir commit-iniz var:

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 4af61fd] Adding git-svn instructions to the README
 1 file changed, 5 insertions(+)
```

Bundan sonra dəyişikliklərinizi yuxarıya doğru push etməlisiniz . Bunun Subversion ilə işləmə t

ərzinizi necə dəyişdirdiyinə diqqət yetirin - bir neçə commit-i oflayn edə və sonra hamısını birdən Subversion serverinə köçürə bilərsiniz. Subversion serverinə keçmək üçün `git svn dcommit` əmrini yerinə yetirirsiniz:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   README.txt
Committed r77
    M   README.txt
r77 = 95e0222ba6399739834380eb10afcd73e0670bc5 (refs/remotes/origin/trunk)
No changes between 4af61fd05045e07598c553167e0f31c84fd6ffe1 and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Bu, Subversion server kodunun üstündə etdiyiniz bütün commit-ləri götürür, hər biri üçün bir Subversion commit-i götürür və sonra yerli Git unikal identifikatoru əlavə etmək üçün yenidən yazır.

Bu vacibdir, çünki commit-ləriniz üçün bütün SHA-1 hesablama cəmlərinin dəyişməsi deməkdir. Qismən bu səbəbdən bir Subversion server ilə eyni vaxtda layihələrinizin Git əsaslı uzaq versiyaları ilə işləmək yaxşı bir fikir deyil. Son commit-ə baxsanız, əlavə edilmiş yeni `git-svn-id`-i görə bilərsiniz:

```
$ git log -1
commit 95e0222ba6399739834380eb10afcd73e0670bc5
Author: ben <ben@0b684db3-b064-4277-89d1-21af03df0a68>
Date:   Thu Jul 24 03:08:36 2014 +0000

    Adding git-svn instructions to the README

    git-svn-id: file:///tmp/test-svn/trunk@77 0b684db3-b064-4277-89d1-21af03df0a68
```

Diqqət yetirin ki, commit etdiyiniz zaman əvvəlcə `4af61fd` ilə başlayan SHA-1 hesablama cəmi `95e0222` ilə başlayır. Həm Git serverinə, həm də bir Subversion serverinə push etmək istəyirsinizsə, əvvəlcə Subversion serverinə (`dcommit`) push etməyə, çünki bu hərəkət commit məlumatınızı d əyişdirir.

Yeni Dəyişikliklərdə Pulling Etmək

Digər developerlərlə işləyirsinizsə, o zaman biriniz birini push edəcəksiniz, sonra digəri konflikt bir dəyişikliyə push etməyə çalışacaq. İşlərinizdə birləşənə qədər bu dəyişiklik rədd ediləcək. `git svn`-də belə görünür:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: d5837c4b461b7c0e018b49d12398769d2bfc240a and refs/remotes/origin/trunk differ,
using rebase:
:100644 100644 f414c433af0fd6734428cf9d2a9fd8ba00ada145
c80b6127dd04f5fcda218730ddf3a2da4eb39138 M  README.txt
Current branch master is up to date.
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

Bu vəziyyəti həll etmək üçün serverdə hələ olmadığınız hər hansı bir dəyişiklikləri aşağı salan və serverdə olanların üstündə etdiyiniz hər hansı bir işi yenidən bərpa edən `git svn rebase`-i işlədə bilirsiniz:

```
$ git svn rebase
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: eaa029d99f87c5c822c5c29039d19111ff32ef46 and refs/remotes/origin/trunk differ,
using rebase:
:100644 100644 65536c6e30d263495c17d781962cfff12422693a
b34372b25ccf4945fe5658fa381b075045e7702a M  README.txt
First, rewinding head to replay your work on top of it...
Applying: update foo
Using index info to reconstruct a base tree...
M   README.txt
Falling back to patching base and 3-way merge...
Auto-merging README.txt
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

İndi bütün işləriniz Subversion serverindəki işlərin üstündədir, beləliklə uğurla `dcommit` edə bil ərsiniz:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   README.txt
Committed r85
    M   README.txt
r85 = 9c29704cc0bbbed7bd58160cfb66cb9191835cd8 (refs/remotes/origin/trunk)
No changes between 5762f56732a958d6cfda681b661d2a239cc53ef5 and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Diqqət yetirmədən əvvəl local işinizi birləşdirməyinizi tələb edən Gitdən fərqli olaraq, `git svn` bunu yalnız dəyişikliklər konflikt təşkil edərsə (Subversionun necə işlədiyi kimi) edir. Başqası bir dəyişikliyi bir fayla push edirsə, sonra başqa bir fayla dəyişiklik göndərirsinizsə, `dcommi`-iniz yaxşı işləyəcək:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   configure.ac
Committed r87
    M   autogen.sh
r86 = d8450bab8a77228a644b7dc0e95977ffc61adff7 (refs/remotes/origin/trunk)
    M   configure.ac
r87 = f3653ea40cb4e26b6281cec102e35dcba1fe17c4 (refs/remotes/origin/trunk)
W: a0253d06732169107aa020390d9fefd2b1d92806 and refs/remotes/origin/trunk differ,
using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18
e757b59a9439312d80d5d43bb65d4a7d0389ed6d M  autogen.sh
First, rewinding head to replay your work on top of it...
```

Bunu unutmamaq vacibdir, çünki nəticə siz push etdiyiniz zaman hər iki kompüterinizdə olmayan bir layihə vəziyyətidir. Dəyişikliklər bir-birinə uyğun deyilsə, konflikt təşkil etmirsə, diaqnozu çətin olan problemlərlə qarşılaşa bilərsiniz. Bu, bir Git serverindən istifadə etməkdən fərqlidir - Git-də, vəziyyəti dərc etmədən əvvəl müştəri sisteminizdə tam olaraq yoxlaya bilərsiniz, halbuki SVN-də, işdən əvvəl və commit götürdükdən sonra vəziyyətlərin eyni olduğuna əmin ola bilməzsiniz.

Özünüzü commit etməyə hazır olmasanız da, Subversion serverindən dəyişikliklər etmək üçün bu əmri də işləməlisiniz. Yeni məlumatları əldə etmək üçün `git svn fetch`-i işə sala bilərsiniz, lakin `git svn rebase` gətirir və sonra local commit-lərinizi yeniləyir.

```
$ git svn rebase
    M   autogen.sh
r88 = c9c5f83c64bd755368784b444bc7a0216cc1e17b (refs/remotes/origin/trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/origin/trunk.
```

Hər dəfə `git svn rebase` işlətmək kodunuzun daima yeniləndiyinə əmin olur. Buna baxmayaraq işləyərkən iş qovluğunuzun təmiz olduğundan əmin olmalısınız.

Local dəyişikliklərniz varsa, işinizi gizlətməlisiniz və ya `git svn rebase`-i işə salmadan əvvəl müv əqqəti olaraq etməlisiniz - əks təqdirdə, rebase-in birləşmə konflikti ilə nəticələnəcəyini görsə əmr dayanacaq.

Git Branching Issues

Bir Git iş axını ilə rahat olduqda, ehtimal ki, mövzu branch-ları yaradacaq, üzərində işləyəcək və sonra onları birləşdirəcəksiniz. `git svn` vasitəsi ilə Subversion serverinə müraciət edirsinizsə, branch-larınızı birləşdirmək əvəzinə hər dəfə işinizi tək branch-da bərpa etmək istəyə bilərsiniz.

Rebasing seçiminə üstünlük vermə səbəbi, Subversionun xətti bir tarixçəyə sahib olması və Git kimi birləşmələrlə məşğul olmamasıdır, buna görə də `git svn` snapshotları Subversion commit-l ərinə çevirərkən yalnız ilk valideynə commit edir.

Tutaq ki, tarixiniz aşağıdakı kimidir: bir `experiment` branch-ı yaratdınız, iki commit etdiniz və sonra onları yenidən `master`-ə çevirdiniz. `dcommit` etdiyiniz zaman, nəticəni belə görürsünüz:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   CHANGES.txt
Committed r89
    M   CHANGES.txt
r89 = 89d492c884ea7c834353563d5d913c6adf933981 (refs/remotes/origin/trunk)
    M   COPYING.txt
    M   INSTALL.txt
Committed r90
    M   INSTALL.txt
    M   COPYING.txt
r90 = cb522197870e61467473391799148f6721bcf9a0 (refs/remotes/origin/trunk)
No changes between 71af502c214ba13123992338569f4669877f55fd and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Birləşdirilmiş tarixə malik bir branch-da `dcommit` çalışması yaxşı işləyir, ancaq Git layihə tarixçə nizə baxdığınız zaman `experiment` branch-da etdiyiniz commit-lərin heç birini yenidən yazmadı - bunun əvəzinə bütün bu dəyişikliklər tək birləşmə commit-inin SVN versiyası.

Başqa birisi işləyən klonlaşdırdıqda, gördükləri bütün işin içərisinə yığılan birləşmə əmridir, sanki `git merge --squash` işlətmisiniz; haradan gəldiyi və nə vaxt edildiyi barədə commit məlumatlarını görmürlər.

Subversion Branching

Subversionda branching, Git-də branching ilə eyni deyil; istifadə etməkdən çox çəkinə bilsəniz, ən yaxşısı budur. Bununla birlikdə, `git svn` istifadə edərək Subversion-dakı branch-ları yarada və commit edə bilərsiniz.

Yeni SVN Branch-ının Yaradılması

Subversion-da yeni bir branch yaratmaq üçün `git svn branch [new-branch]` işlədin:

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r90 to file:///tmp/test-svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-
svn/branches/opera, 90
Found branch parent: (refs/remotes/origin/opera)
cb522197870e61467473391799148f6721bcf9a0
Following parent with do_switch
Successfully followed parent
r91 = f1b64a3855d3c8dd84ee0ef10fa89d27f1584302 (refs/remotes/origin/opera)
```

Bu, Subversiondakı `svn copy trunk branches/opera` əmrinin ekvivalentini edir və Subversion serverində işləyir. Qeyd etmək vacibdir ki, sizi bu branch-a daxil etmir; Əgər bu anda commit götürsəniz, bu əməliyyat serverdəki `opera`-a yox, `trunk`-a gedəcək.

Aktiv Branche-ların Dəyişdirilməsi

Git, tarixinizdəki Subversion branch-larınızın hər hansı birinin ucunu axtararaq dcommits-nizin hansı branch-a getdiyini müəyyənləşdirir - yalnız birinə sahib olmalısınız və mövcud branch-ınızda bir "git-svn-id" olan sonuncusu olmalıdır tarix.

Git, tarixinizdəki Subversion branch-larınızın hər hansı birinin ucunu axtararaq dcommitslərinizin hansı qola getdiyini müəyyənləşdirir - yalnız birinə sahib olmalısınız və mövcud filialınızda bir `git-svn-id` olan sonuncusu olmalıdır.

Eyni anda birdən çox branch üzərində işləmək istəyirsinizsə, bu branch üçün idxal olunan Subversion commit-indən başlayaraq müəyyən Subversion branch-larına `dcommit` etmək üçün local branch-lar qura bilərsiniz. Ayrı-ayrılıqda işləyə biləcəyiniz bir `opera` branch-ı istəsəniz, çalışdıra bilərsiniz:

```
$ git branch opera remotes/origin/opera
```

İndi `opera` branch-ınızı `trunk`-a (`master` branch-nızın) birləşdirmək istəyirsinizsə, bunu normal bir git birləşmə ilə edə bilərsiniz. Ancaq təsviri bir commit mesajı verməlisiniz (`-m` vasitəsilə), əks halda birləşmə faydalı bir şey əvəzinə “Merge branch opera” deyəcəkdir.

Unutmayın ki, bu əməliyyatı yerinə yetirmək üçün `git merge` istifadə etsəniz də birləşmə ehtimalı Subversionda olduğundan daha asan olacaq (çünki Git sizin üçün uyğun birləşmə bazasını avtomatik olaraq aşkar edəcək), Git birləşmə commiti normal deyil. Bu məlumatları birdən çox valideynə baxan bir commit-i yerinə yetirə bilməyən bir Subversion serverinə qaytarmalısınız; belə ki, onu push etdikdən sonra, başqa bir branch-ın bütün işlərində tək bir commit altında əzilən tək bir commit-ə bənzəyir. Bir branch-ı digərinə birləşdirdikdən sonra asanlıqla geri qaytara bilm əzsiniz və normal olaraq Gitdə olduğu kimi bu branch üzərində işləməyə davam edə bilməzsiniz. İşlətdiyiniz `dcommit` əmri hansı branch-ın birləşdirildiyini bildirən hər hansı bir məlumatı silir, bu səbəbdən sonrakı birləşmə bazası hesablamaları səhv olacaq -`dcommit git merge` nəticənizi `git`

`merge --squash` kimi işlədir. Təəssüf ki, bu vəziyyətdən qaçınmaq üçün yaxşı bir yol yoxdur - Subversion bu məlumatı saxlaya bilməz, buna görə də onu server olaraq istifadə edərkən m əhdudiyyətləri ilə əlil olacaqsınız. Problemlərin qarşısını almaq üçün trunk-a birləşdirdikdən sonra local branch-ı (bu halda, `opera`) silməlisiniz.

Subversion Əmrləri

`git svn` alətlər dəsti Subversion-dakılara bənzər bəzi funksionallıq təmin edərək Git-ə keçidi asanlaşdırmağa kömək edəcək bir sıra əmrlər təqdim edir. Budur sizə Subversionun istifadə etdiyini verən bir neçə əmr.

SVN Style Tarixi

Subversiona alışmışsınızsa və tarixinizi SVN çıxış tərzində görmək istəyirsinizsə, SVN formatında commit tarixçənizə baxmaq üçün `git svn log`-nu işə sala bilərsiniz:

```
$ git svn log
------------------------------------------------------------------------
r87 | schacon | 2014-05-02 16:07:37 -0700 (Sat, 02 May 2014) | 2 lines

autogen change

------------------------------------------------------------------------
r86 | schacon | 2014-05-02 16:00:21 -0700 (Sat, 02 May 2014) | 2 lines

Merge branch 'experiment'

------------------------------------------------------------------------
r85 | schacon | 2014-05-02 16:00:09 -0700 (Sat, 02 May 2014) | 2 lines

updated the changelog
```

`git svn log` haqqında iki vacib şeyi bilməlisiniz. Birincisi, Subversion serverindən məlumat istəyən həqiqi `svn log` əmrindən fərqli olaraq oflayn işləyir. İkincisi, yalnız Subversion serverinə sadiq qaldığınızı göstərir. İcazə vermədiyiniz Local Git commit-ləri görünmür; bu müddətdə insanların Subversion serverinə verdiyi commit-lər də yoxdur. Subversion serverindəki commit-lərin son bilinən vəziyyətinə daha çox bənzəyir.

SVN Annotasiyası

`git svn log` əmri, `svn log` əmrini oflayn olaraq simulyasiya etdiyi kimi `git svn blame [FILE]` işlədə rək `svn annotate` bərabərliyini ala bilərsiniz. Nəticə belə görünür:

```
$ git svn blame README.txt
 2   temporal Protocol Buffers - Google's data interchange format
 2   temporal Copyright 2008 Google Inc.
 2   temporal http://code.google.com/apis/protocolbuffers/
 2   temporal
22   temporal C++ Installation - Unix
22   temporal =======================
 2   temporal
79    schacon Committing in git-svn.
78    schacon
 2   temporal To build and install the C++ Protocol Buffer runtime and the Protocol
 2   temporal Buffer compiler (protoc) execute the following:
 2   temporal
```

Yenə də yerli olaraq Gitdə etdiyiniz və ya bu müddətdə Subversion-a itələdiyiniz commit-ləri göst ərmir.

SVN Server Məlumatı

Siz də `git svn info`-nu işə salıb `svn info`-un sizə verdiyi məlumatları əldə edə bilərsiniz:

```
$ git svn info
Path: .
URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
Node Kind: directory
Schedule: normal
Last Changed Author: schacon
Last Changed Rev: 87
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)
```

Bu, `blame` və `log` kimi bir şeydir ki, oflayn işləyir və yalnız Subversion server ilə sonuncu dəfə ünsiyyət qurduğunuz günə qədərdir.

Subversion Ignore-larını Nə İgnore Etdi

Hər hansı bir yerdə qurulmuş `svn:ignore` xüsusiyyətlərinə sahib olan bir Subversion deposunu klonlaşdırırsınızsa, ehtimal ki, commit etməməli olduğunuz faylları təsadüfən etməməyiniz üçün müvafiq `.gitignore` fayllarını təyin etmək istərdiniz. `git svn`-in bu məsələdə kömək edəcək iki əmri var. Birincisi, avtomatik olaraq sizin üçün müvafiq `.gitignore` fayllarını yaradan `git svn create-ignore`, növbəti işinizdə bunları daxil edə bilər. İkinci əmr, `.gitignore` faylına qoymağınız lazım olan sətirləri düzəltmək üçün yazdıran `git svn show-ignore`-dir, nəticədə proyektinizə çıxarılan faylı yönləndirə bilərsiniz:

```
$ git svn show-ignore > .git/info/exclude
```

Bu şəkildə layihəni `.gitignore` sənədləri ilə zibil etmirsiniz. Subversion komandasındakı tək Git istifadəçisisinizsə və komanda yoldaşlarınız layihədəki `.gitignore` sənədlərini istəmirsinizsə, bu yaxşı bir seçimdir.

Git-Svn-nin Qısa Məzmunu

`git svn` alətləri bir Subversion serverinə qapıldığınızda və ya başqa bir şəkildə Subversion serverinin işlədilməsini zəruri edən inkişaf mühitində olduğunuzda faydalıdır. Bununla birlikdə şikəst Git olduğunu düşünməlisiniz, yoxsa tərcümədə sizi və həmkarlarınızı qarışdıra biləcək məs ələlərə toxunacaqsınız. Problemlərdən uzaq olmaq üçün aşağıdakı qaydalara əməl etməyə çalışın:

- `git merge` tərəfindən birləşdirmə commit-lərini ehtiva etməyən xətti bir Git tarixçəsi saxlayın. Əsas xətt branch-ınız xaricində etdiyiniz hər hansı bir işi yenidən üzərinə qaytarın; birləşdirm əyin.
- Ayrı bir Git server qurmayın və əməkdaşlıq etməyin. Yeni developerlər üçün klonları sürətl əndirmək üçün birinə sahib ola bilərsiniz, ancaq ona `git-svn-id` girişi olmayan bir şey push etməyin. Hətta hər bir gediş mesajını bir `git-svn-id` üçün yoxlayan və onsuz commit-ləri olan itkiləri rədd edən bir `pre-receive` hook-nu əlavə etmək istəyə bilərsiniz.

Bu təlimatlara əməl etsəniz, Subversion server ilə işləmək daha dözümlü ola bilər. Ancaq həqiqi bir Git serverinə keçmək mümkündürsə, bunu etmək komandanızı daha çox qazana bilər.

Git və Mercurial

DVCS kainatı yalnız Git-dən daha böyükdür. Əslində, bu məkanda çox sayda sistem var, hər biri paylanmış versiya nəzarətinin düzgün şəkildə necə ediləcəyinə dair öz bucağına malikdir. Git xaricində ən populyar Mercurialdır və ikisi bir çox cəhətdən çox oxşardır.

Yaxşı bir xəbər, Git'in müştəri tərəfindəki davranışını üstün tutsanız, lakin mənbə kodu Mercurial ilə idarə olunan bir layihə ilə işləyirsinizsə, Git'i bir Mercurial-a yerləşdirilən bir depo üçün müşt əri olaraq istifadə etməyin bir yolu olmasıdır. Gitin server depoları ilə danışıq üsulu uzaqdan idar əedici olduğundan bu körpünün uzaq bir köməkçi kimi həyata keçirilməsinə təəccüblənməməliyik. Layihənin adı git-remote-hg-dir və https://github.com/felipec/git-remote-hg saytında tapa bilərsiniz.

git-remote-hg

Əvvəlcə git-remote-hg yükləməlisiniz. Bu, əsasən faylı path-nıza bir yerə atmağa səbəb olur:

```
$ curl -o ~/bin/git-remote-hg \
  https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-remote-hg
$ chmod +x ~/bin/git-remote-hg
```

...`~/bin`-in `$PATH`-da olduğunuzu düşünürük. Git-remote-hg-nin başqa bir asılılığı var: Python üçün `mercurial` kitabxanası. Python quraşdırılıbsa, bu qədər sadədir:

```
$ pip install mercurial
```

Python'unuz yoxdursa, https://www.python.org/ səhifəsini ziyarət edin və əvvəl əldə edin.

Sizə lazım olan son şey Mercurial müştərisidir. https://www.mercurial-scm.org/ saytına gedin və hə lə də yoxdursa quraşdırın.

İndi isə rock etməyə hazırsınız. Sadəcə, push etməyə bir Mercurial deposuna ehtiyacınız var. Xoşb əxtlikdən, hər bir Mercurial deposu bu şəkildə hərəkət edə bilər, buna görə Mercurial-ı öyrənmək üçün hər kəsin istifadə etdiyi "hello world" deposundan istifadə edəcəyik:

```
$ hg clone http://selenic.com/repo/hello /tmp/hello
```

Başlangıc

Artıq uyğun bir "server-tərəf" depomuz olduğuna görə tipik bir iş axınından keçə bilərik. Gördüyünüz kimi, bu iki sistem kifayət qədər sürtünmə olmadan oxşayır.

Həmişə olduğu kimi Git ilə əvvəlcə klonlaşdırırıq:

```
$ git clone hg::/tmp/hello /tmp/hello-git
$ cd /tmp/hello-git
$ git log --oneline --graph --decorate
* ac7955c (HEAD, origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master, master) Create a
makefile
* 65bb417 Create a standard 'hello, world' program
```

Bir Mercurial deposu ilə işləmək üçün standart `git clone` əmrini istifadə etdiyini görəcəksiniz. Git-in HTTP/S protokolunun necə tətbiq olunduğuna bənzər bir mexanizm istifadə edərək (remote köməkçilər) git-remote-hg-nin kifayət qədər aşağı səviyyədə işləməsidir. Git və Mercurial, hər bir müştərinin anbar tarixinin tam bir nüsxəsinə sahib olması üçün hazırlandığından, bu əmr bütün layihə tarixi daxil olmaqla tam bir klon yaradır və bunu kifayət qədər tez edir.

Log əmri iki commit göstərir, ən sonuncusu isə bir çox refer tərəfindən göstərilmişdir. Bunlardan bəzilərinin əslində orada olmadığı ortaya çıxdı. Gəlin `.git` qovluğundakılara nəzər salaq:

```
$ tree .git/refs
.git/refs
├──── heads
│    └──── master
├──── hg
│    └──── origin
│        ├──── bookmarks
│        │    └──── master
│        └──── branches
│            └──── default
├──── notes
│    └──── hg
├──── remotes
│    └──── origin
│        └──── HEAD
└──── tags

9 directories, 5 files
```

Git-remote-hg şeyləri daha idiomatik olaraq Git-esque etməyə çalışır, ancaq başlıq altında iki fərqli sistem arasındakı konseptual xəritələşməni idarə edir. `refs/hg` qovluğu faktiki uzaq reflərin saxlanıldığı yerdir. Məsələn, `refs/hg/origin/branches/default`, `master`-in işarə etdiyi "ac7955c" ilə başlayan SHA-1-i ehtiva edən bir Git ref sənədidir.

Beləliklə, `refs/hg` qovluğu bir növ saxta `refs/remotes/origin` kimidir, lakin bookmark-lar və branch-lar arasında əlavə bir fərq var.

`notes/hg` faylı, git-remote-hg xəritələrinin Gitin Mercurial dəyişən ID-lərinə necə hash olduğunun üçün başlanğıc nöqtəsidir. Gəlin bir az araşdıraq:

```
$ cat notes/hg
d4c10386...

$ git cat-file -p d4c10386...
tree 1781c96...
author remote-hg <> 1408066400 -0800
committer remote-hg <> 1408066400 -0800

Notes for master

$ git ls-tree 1781c96...
100644 blob ac9117f...  65bb417...
100644 blob 485e178...  ac7955c...

$ git cat-file -p ac9117f
0a04b987be5ae354b710cefeba0e2d9de7ad41a9
```

Beləliklə, `refs/notes/hg`, Git obyekt bazasında adları olan digər obyektlərin siyahısı olan bir ağaca

işarə edir. `git ls-tree` bir ağacın içindəki əşyalar üçün rejimi, növü, obyekt hash-ı və fayl adını çıxarır.

Ağac item-larından birinə endikdən sonra içərisində "0a04b98" konteksi ilə (bu, `default` branch-nın ucundakı Mercurial dəyişikliyinin identifikatorudur) "ac9117f" (`master` tərəfindən göstərilən commit-in SHA-1 hashı) adlı bir blob olduğunu görürük.

Yaxşı xəbər odur ki, bunların hamısı üçün narahat olmaq lazım deyil. Tipik iş axını, Git remote ilə işləməkdən çox fərqli olmayacaq.

Davam etməzdən əvvəl iştirak etməli olduğumuz bir şey daha var: ignores. Mercurial və Git bunun üçün çox bənzər bir mexanizm istifadə edirlər, lakin çox güman ki, bir `.gitignore` sənədini Mercurial deposuna daxil etmək istəməyəcəksiniz. Xoşbəxtlikdən, Git'in diskdəki bir depo üçün lokal olan faylları ignore etməyin bir yolu var və Mercurial formatı Git ilə uyğundur, buna görə onu köçürməlisiniz:

```
$ cp .hgignore .git/info/exclude
```

The `.git/info/exclude` file acts just like a `.gitignore`, but isn't included in commits.

Workflow

Fərz edək ki, `master` branch-ında bir az iş gördük və bir neçə commit götürdük və onu uzaq depoya push etməyə hazırsınız. Depomuz hazırda belə görünür:

```
$ git log --oneline --graph --decorate
* ba04a2a (HEAD, master) Update makefile
* d25d16f Goodbye
* ac7955c (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Create a makefile
* 65bb417 Create a standard 'hello, world' program
```

`master` branch-ımız `origin/master`-dan iki commit-dir, lakin bu iki vəzifə yalnız local maşınımızda mövcuddur. Başqa birinin eyni zamanda mühüm bir iş görüb görmədiyini baxaq:

```
$ git fetch
From hg::/tmp/hello
   ac7955c..df85e87  master     -> origin/master
   ac7955c..df85e87  branches/default -> origin/branches/default
$ git log --oneline --graph --decorate --all
* 7b07969 (refs/notes/hg) Notes for default
* d4c1038 Notes for master
* df85e87 (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Add some
documentation
| * ba04a2a (HEAD, master) Update makefile
| * d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard 'hello, world' program
```

`--all` flag-ından istifadə etdiyimiz üçün git-remote-hg tərəfindən daxili istifadə edilən “qeydlər” istinadlarını görürük, lakin onları ignore edə bilərik. Qalan nə gözlədiyimizdir; `origin/master` bir commit-lə irəli getdi və tariximiz indi fərqlidir. Bu fəsildə işlədiyimiz digər sistemlərdən fərqli olaraq, Mercurial birləşmələri idarə edə bilir, buna görə də zərif bir şey etməyəcəyik.

```
$ git merge origin/master
Auto-merging hello.c
Merge made by the 'recursive' strategy.
 hello.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git log --oneline --graph --decorate
*   0c64627 (HEAD, master) Merge remote-tracking branch 'origin/master'
|\
| * df85e87 (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Add some
documentation
* | ba04a2a Update makefile
* | d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard 'hello, world' program
```

Əla. Testləri aparırıq və hər şey uğurlu keçir, buna görə işimizi komandanın qalan üzvləri ilə bölüşməyə hazırıq:

```
$ git push
To hg::/tmp/hello
   df85e87..0c64627  master -> master
```

Bu qədər! Merkuri deposuna nəzər yetirsəniz, bunun gözlədiyimizi etdiyini görərsiniz:

```
$ hg log -G --style compact
o    5[tip]:4,2   dc8fa4f932b8   2014-08-14 19:33 -0700   ben
|\     Merge remote-tracking branch 'origin/master'
| |
| o  4   64f27bcefc35   2014-08-14 19:27 -0700   ben
| |    Update makefile
| |
| o  3:1   4256fc29598f   2014-08-14 19:27 -0700   ben
| |    Goodbye
| |
@ |  2   7db0b4848b3c   2014-08-14 19:30 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard 'hello, world' program
```

2 nömrəli dəyişikliklər Mercurial tərəfindən, *3* və *4* nömrəli dəyişikliklər git-remote-hg tərəfindən Git ilə edilən commit-ləri push etməklə edilmişdir.

Branch-lar və Bookmark-lar

Gitin yalnız bir növ branch-ı var: commit-lər qoyulduqda hərəkət edən bir istinad. Mercurial-da bu növ bir istinad “bookmark” adlanır və Git branch-ı ilə eyni şəkildə davranır.

Mercurial-ın “branch” konsepsiyası daha ağırdır. Dəyişikliklərin edildiyi branch *dəyişikliklərlə* yazılır, yəni həmişə depo tarixində olacaqdır.

Budur `develop` branch-ında edilən bir commit-in nümunəsi:

```
$ hg log -l 1
changeset:   6:8f65e5e02793
branch:      develop
tag:         tip
user:        Ben Straub <ben@straub.cc>
date:        Thu Aug 14 20:06:38 2014 -0700
summary:     More documentation
```

“branch” ilə başlayan sətrə diqqət yetirin. Git bunu həqiqətən təkrarlaya bilməz (və buna ehtiyac yoxdur; hər iki branch növü də Git ref kimi təmsil edilə bilər), lakin git-remote-hg fərqi anlamalıdır, çünki Mercurial qayğı göstərir.

Mercurial bookmark-lar yaratmaq, Git branch-ları yaratmaq qədər asandır. Git tərəfində:

```
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ git push origin featureA
To hg::/tmp/hello
 * [new branch]      featureA -> featureA
```

Bunun üçün hər şey var. Mercurial tərəfdən bu belə görünür:

```
$ hg bookmarks
   featureA                  5:bd5ac26f11f9
$ hg log --style compact -G
@  6[tip]   8f65e5e02793   2014-08-14 20:06 -0700   ben
|    More documentation
|
o    5[featureA]:4,2   bd5ac26f11f9   2014-08-14 20:02 -0700   ben
|\     Merge remote-tracking branch 'origin/master'
| |
| o  4   0434aaa6b91f   2014-08-14 20:01 -0700   ben
| |    update makefile
| |
| o  3:1   318914536c86   2014-08-14 20:00 -0700   ben
| |    goodbye
| |
o |  2   f098c7f45c4f   2014-08-14 20:01 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard 'hello, world' program
```

Reviziya 5-də yeni `[featureA]` etiketinə diqqət yetirin. Bunlar bir istisna olmaqla, Git tərəfindəki Git branch-ları kimi hərəkət edirlər: bir bookmark-ı Git tərəfdən silə bilməzsiniz (bu, uzaq köməkçil ərin məhdudiyyətidir).

“heavyweight” bir Mercurial branch-da da işləyə bilərsiniz: sadəcə `branches` namespace-nə bir branch qoyun:

```
$ git checkout -b branches/permanent
Switched to a new branch 'branches/permanent'
$ vi Makefile
$ git commit -am 'A permanent change'
$ git push origin branches/permanent
To hg::/tmp/hello
 * [new branch]      branches/permanent -> branches/permanent
```

Mercurial tərəfdə görünən budur:

```
$ hg branches
permanent                      7:a4529d07aad4
develop                        6:8f65e5e02793
default                        5:bd5ac26f11f9 (inactive)
$ hg log -G
o  changeset:   7:a4529d07aad4
|  branch:      permanent
|  tag:         tip
|  parent:      5:bd5ac26f11f9
|  user:        Ben Straub <ben@straub.cc>
|  date:        Thu Aug 14 20:21:09 2014 -0700
|  summary:     A permanent change
|
| @  changeset:   6:8f65e5e02793
|/   branch:      develop
|    user:        Ben Straub <ben@straub.cc>
|    date:        Thu Aug 14 20:06:38 2014 -0700
|    summary:     More documentation
|
o    changeset:   5:bd5ac26f11f9
|\   bookmark:    featureA
| |  parent:      4:0434aaa6b91f
| |  parent:      2:f098c7f45c4f
| |  user:        Ben Straub <ben@straub.cc>
| |  date:        Thu Aug 14 20:02:21 2014 -0700
| |  summary:     Merge remote-tracking branch 'origin/master'
[...]
```

“permanent” branch adı *7* işarəsi ilə dəyişiklik dəsti ilə qeyd edildi.

Git tərəfdən, bu branch üslublarından hər hansı biri ilə işləmək eynidir: normalda istədiyiniz kimi checkout, commit, fetch, merge, pull, və push etmək. Bilməli olduğunuz bir şey, Mercurial-ın tarixin yenidən yazılmasını dəstəkləməməsi, yalnız tarixə əlavə etməsidir. Mercurial depomuzun interaktiv rebase və force-push tətbiqindən sonra belə görünəcək:

```
$ hg log --style compact -G
o  10[tip]   99611176cbc9   2014-08-14 20:21 -0700   ben
|    A permanent change
|
o  9   f23e12f939c3   2014-08-14 20:01 -0700   ben
|    Add some documentation
|
o  8:1   c16971d33922   2014-08-14 20:00 -0700   ben
|    goodbye
|
| o  7:5   a4529d07aad4   2014-08-14 20:21 -0700   ben
| |    A permanent change
| |
| | @  6   8f65e5e02793   2014-08-14 20:06 -0700   ben
| |/     More documentation
| |
| o    5[featureA]:4,2   bd5ac26f11f9   2014-08-14 20:02 -0700   ben
| |\     Merge remote-tracking branch 'origin/master'
| | |
| | o  4   0434aaa6b91f   2014-08-14 20:01 -0700   ben
| | |    update makefile
| | |
+---o  3:1   318914536c86   2014-08-14 20:00 -0700   ben
| |      goodbye
| |
| o  2   f098c7f45c4f   2014-08-14 20:01 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard "hello, world" program
```

8, *9* və *10* dəyişikliklər yaradıldı və `permanent` branch-na aiddir, lakin köhnə dəyişikliklər hələ də mövcuddur. Bu, Mercurial istifadə edən komanda yoldaşlarınız üçün **çox** qarışıq ola bilər, buna görə də bundan çəkinməyə çalışın.

Mercurial-ın Qısa Məzmunu

Git və Mercurial kifayət qədər oxşardır və sərhəd boyunca işləmək kifayət qədər ağrısızdır. Maşınınızda qalan tarixi dəyişdirməkdən çəkinsəniz (ümumiyyətlə tövsiyə olunduğu kimi), digər ucunun Mercurial olduğunu da bilməyəcəksiniz.

Git və Bazaar

DVCS arasında daha bir məşhur Bazaar-dır. Bazaar pulsuz və açıq mənbəlidir və GNU Project-nin bir hissəsidir. Bu Git-dən çox fərqli davranır. Bəzən Git ilə eyni şeyi etmək üçün fərqli bir keyword istifadə etməlisiz və ümumi olan bəzi keyword-lər eyni məna daşımır. Xüsusilə branch idarəetməsi

çox fərqlidir və xüsusən kimsə Git'in kainatından gələndə qarışıqlığa səbəb ola bilər. Buna baxmayaraq bir Git deposundan Bazaar deposu üzərində işləmək mümkündür.

Git'i Bazaar müştərisi kimi istifadə etməyə imkan verən bir çox layihə var. Burada https://github.com/felipec/git-remote-bzr saytında tapa biləcəyiniz Felipe Contreras layihəsini istifadə edəcəyik. Qurmaq üçün yalnız git-remote-bzr faylını `$PATH`-nızdakı bir qovluğa yükləm əlisiniz:

```
$ wget https://raw.github.com/felipec/git-remote-bzr/master/git-remote-bzr -O
~/bin/git-remote-bzr
$ chmod +x ~/bin/git-remote-bzr
```

Bazaarın quraşdırılmasına da ehtiyacınız var. Bu qədər!

Bir Bazaar deposundan bir Git deposu yaradın

İstifadəsi sadədir. Bir Bazaar deposunu əvvəldən əlavə edən `bzr::` ilə klonlaşdırmaq kifayətdir. Git və Bazaar hər ikisi də cihazınıza tam klonlar verdiyindən local Bazaar klonuna bir Git klonu əlavə etmək mümkündür, lakin tövsiyə edilmir. Git klonunuzu birbaşa Bazaar klonunuzun bağlandığı yerə - mərkəzi depoya əlavə etmək daha asandır.

Fərz edək ki, `bzr+ssh://developer@mybazaarserver:myproject` ünvanında olan uzaq bir depo ilə işl əmisiniz. Sonra aşağıdakı şəkildə klonlamalısınız:

```
$ git clone bzr::bzr+ssh://developer@mybazaarserver:myproject myProject-Git
$ cd myProject-Git
```

Bu nöqtədə Git deposunuz yaradılır, lakin optimal disk istifadəsi üçün sıxılmır. Bu səbəbdən Git deposunuzu təmizləmək və yığmaq lazımdır, xüsusən də böyükdürsə:

```
$ git gc --aggressive
```

Bazaar Branch'ları

Bazaar yalnız branch-ları klonlamağa imkan verir, ancaq bir depo bir neçə branch-dan ibarət ola bilər və `git-remote-bzr` hər ikisini də klonlaya bilər. Məsələn, branch-ı klonlamaq üçün:

```
$ git clone bzr::bzr://bzr.savannah.gnu.org/emacs/trunk emacs-trunk
```

Və bütün deponu klonlamaq üçün:

```
$ git clone bzr::bzr://bzr.savannah.gnu.org/emacs emacs
```

İkinci komanda, emacs deposundakı bütün branch-ları klonlaşdırır; buna baxmayaraq, bəzi

branch-lara işarə etmək mümkündür:

```
$ git config remote-bzr.branches 'trunk, xwindow'
```

Bəzi uzaq depolar branch-larını siyahıya almağa imkan vermir, bu halda onları manual olaraq t əyin etməlisiniz və klonlama əmrində konfiqurasiyanı təyin edə bilsəniz də, bunu daha asan tapa bilərsiniz:

```
$ git init emacs
$ git remote add origin bzr::bzr://bzr.savannah.gnu.org/emacs
$ git config remote-bzr.branches 'trunk, xwindow'
$ git fetch
```

.bzrignore ilə ignore edilənləri ignore edin

Bazaar ilə idarə olunan bir layihə üzərində çalışdığınız üçün `.gitignore` faylı yaratmamalısınız, çünki təsadüfən onu versiya nəzarəti altına olacaqsınız və Bazaar ilə işləyən digər insanlar narahat olacaqlar. Çözüm, `.git/info/exclude` faylını simvolik bir link və ya adi bir fayl olaraq yaratmaqdır. Bu sualın necə həll ediləcəyini daha sonra görəcəyik.

Bazaar, faylları ignore etmək üçün Git ilə eyni modeli istifadə edir, eyni zamanda Git-ə ekvivalenti olmayan iki xüsusiyyəti var. Tam təsviri the documentation-də tapa bilərsiniz. İki xüsusiyyət bunlardır:

1. "!!" "!" qaydada göstərilsə də, müəyyən fayl nümunələrini ignore etməyə imkan verir.
2. Bir sətrin əvvəlindəki "RE:" bir Python regular expression təyin etməyə imkan verir (Git yalnız shell glob-lara icazə verir).

Nəticə olaraq, nəzərə alınacaq iki fərqli vəziyyət var:

1. Əgər `.bzrignore` faylında bu iki xüsusi prefixes-dən heç biri yoxdursa, onda depoda sadəcə simvolik bir əlaqə qura bilərsiniz: `ln -s .bzrignore .git/info/exclude`.
2. Əks təqdirdə, `.git/info/exclude` faylını yaratmalı və `.bzrignore`-dakı tam eyni faylları ignore etməlisiniz.

Nə olursa olsun, `.git/info/exclude` faylının hər zaman `.bzrignore` yansıtdığından əmin olmaq üçün `.bzrignore` dəyişikliyinə qarşı xəbərdar olmalısınız. Həqiqətən, `.bzrignore` faylı dəyişdirilərsə və "!!" ilə başlayan bir və ya daha çox sətir var idi və ya "RE:", Git bu sətirləri şərh edə bilmirsə, `.git/info/exclude` faylınızı `.bzrignore` ilə ignore olunanlarla eyni faylları ignore etmək üçün uyğunlaşdırmalısınız. Üstəlik, `.git/info/exclude` faylı simvolik bir əlaqə olsaydı, əvvəlcə simvolik linki silməli, `.bzrignore` `.git/info/exclude` şəklində kopyalayıb sonra ikincisini uyğunlaşdırmalısınız. Bununla birlikdə, onun yaradılmasında diqqətli olun, çünki Git ilə bu faylın bir ana qovluğu xaric edildikdə bir faylı yenidən daxil etmək mümkün deyil.

Uzaq deponun dəyişikliklərini fetch edin

Remote-un dəyişikliklərini əldə etmək üçün Git əmrlərindən istifadə edərək dəyişiklikləri adətən

fetch edirsiniz. Dəyişikliklərinizin `master` branch-ında olduğunu düşünərək, `origin/master` branch-ında işlərinizi birləşdirir və ya yenidən artırırsınız:

```
$ git pull --rebase origin
```

Uzaqdakı depoda işinizi push edin

Bazaar da birləşmə commit-i konsepsiyasına sahib olduğundan birləşdirmə commit-ini irəli sürs əniz, heç bir problem olmayacaqdır. Beləliklə, bir branch-da işləyə, dəyişiklikləri `master` halına g ətirə və işinizi push edə bilərsiniz. Sonra, branch-larınızı yaradırsınız, işinizi həmişəki kimi sınaqdan keçirirsiniz. Nəhayət işinizi Bazaar deposuna push edirsiniz:

```
$ git push origin master
```

Xəbərdarlıqlar

Git'in uzaqdan köməkçilər çərçivəsinin tətbiq olunan bəzi məhdudiyyətləri var. Xüsusilə, bu əmrl ər işləmir:

- git push origin :branch-to-delete (Bazaar bu şəkildə ref silmələrini qəbul edə bilməz)
- git push origin old:new (`old`-u push edəcək)
- git push --dry-run origin branch (push edəcək)

Qısa Məzmun

Git və Bazaar modelləri bir-birinə bənzədiyi üçün sərhəd xaricində işləyərkən çox müqavimət göstərmir. Məhdudiyyətlərə diqqət yetirdiyiniz və uzaqdakı deponun local olaraq Git olmadığını bildiyiniz müddətdə yaxşı olacaqsınız.

Git və Perforce

Perforce korporativ mühitlərdə çox populyar bir versiya nəzarət sistemidir. 1995-ci ildən bəri mövcuddur, bu da bu bölmədə yer alan ən qədim sistemdir. Beləliklə, gününün məhdudiyyətləri ilə dizayn edilmişdir; hər zaman tək bir mərkəzi serverə bağlı olduğunuzu və yalnız bir versiyanın yerli diskdə saxlandığını düşünür. Əmin olmaq üçün xüsusiyyətləri və məhdudiyyətləri bir neçə spesifik problemə çox uyğundur, lakin Perforce istifadə edərək Git-in daha yaxşı işləyəcəyi bir çox layihə var.

Perforce və Git istifadəsini qarışdırmaq istəsəniz iki seçim var. Əhatə edəcəyimiz ilk şey, Perforce depolarınızın alt ağaclarını Git read-write depoları kimi göstərməyə imkan verən Perforce istehsalçılarının "Git Fusion" körpüsüdür. İkincisi, Git'i Perforce serverinin hər hansı bir yenidən konfiqurasiyasına ehtiyac olmadan Perforce müştərisi kimi istifadə etməyə imkan verən müştəri t ərəfi olan git-p4-dür.

Git Fusion

Perforce, bir Perforce serverini server tərəfindəki Git depoları ilə sinxronizasiya edən Git Fusion adlı bir məhsul təqdim edir (http://www.perforce.com/git-fusion).

Quraşdırılması

Nümunələrimiz üçün Perforce demonunu və Git Fusion-u işləyən bir virtual maşın yükləyən Git Fusion üçün ən asan quraşdırma metodundan istifadə edəcəyik. Virtual maşın şəklini http://www.perforce.com/downloads/Perforce/20-User saytından əldə edə bilərsiniz və yüklədikdən sonra onu ən sevdiyiniz virtualizasiya proqramına daxil edin (VirtualBox istifadə edəcəyik).

Maşın ilk işə salındıqdan sonra üç Linux istifadəçisi üçün şifrənizi (`root`, `perforce` və `git`) fərdil əşdirməyinizi və bu quraşdırmanı digərlərindən fərqləndirmək üçün istifadə edilə bilən bir nümunə adını eyni şəbəkədə verməyinizi xahiş edir. Hamısı tamamlandıqda bunları görəcəksiniz:

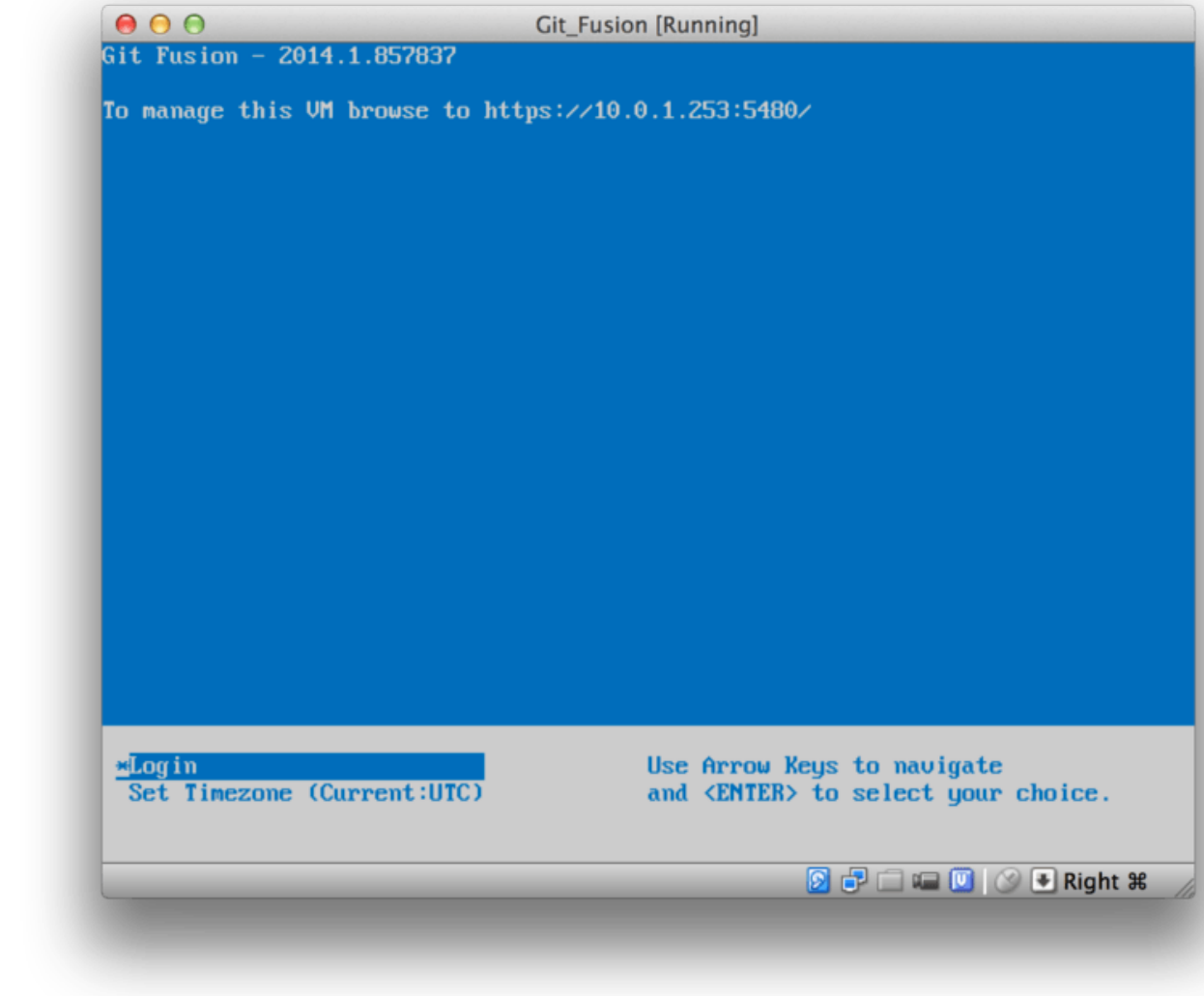


Figure 146. The Git Fusion virtual machine boot screen

Burada göstərilən IP ünvanını qeyd etməlisiniz, daha sonra istifadə edəcəyik. Sonra bir Perforce istifadəçisi yaradacağıq. Aşağıdakı “Login” seçimini seçin və enter düyməsini basın (və ya maşın üçün SSH) və `root` olaraq daxil olun. Sonra bir istifadəçi yaratmaq üçün bu əmrləri istifadə edin:

```
$ p4 -p localhost:1666 -u super user -f john
$ p4 -p localhost:1666 -u john passwd
$ exit
```

Birincisi, istifadəçini fərdiləşdirmək üçün VI redaktoru açacaq, ancaq `:wq` yazaraq enter düyməsini basaraq standartları qəbul edə bilərsiniz. İkincisi, iki dəfə bir parol daxil etməyinizi istər. Bir shell prompt ilə etməli olduğumuz şey budur, buna görə sessiyadan çıxın.

İzləmək üçün etməyiniz lazım olan növbəti şey, Git-ə SSL sertifikatlarının təsdiqlənməməsini söyl əməkdir. Git Fusion görüntüsü sertifikatla gəlir, ancaq virtual maşınınızın IP ünvanına uyğun gə lməyən bir domen üçündür, buna görə Git HTTPS bağlantısını rədd edəcəkdir. Qalıcı bir quraşdırma olacaqsa, fərqli bir sertifikat quraşdırmaq üçün Perforce Git Fusion təlimatına müraci ət edin; nümunə məqsədlərimiz üçün bu kifayət edəcəkdir:

```
$ export GIT_SSL_NO_VERIFY=true
```

İndi hər şeyin işlədiyini test edə bilərik.

```
$ git clone https://10.0.1.254/Talkhouse
Cloning into 'Talkhouse'...
Username for 'https://10.0.1.254': john
Password for 'https://john@10.0.1.254':
remote: Counting objects: 630, done.
remote: Compressing objects: 100% (581/581), done.
remote: Total 630 (delta 172), reused 0 (delta 0)
Receiving objects: 100% (630/630), 1.22 MiB | 0 bytes/s, done.
Resolving deltas: 100% (172/172), done.
Checking connectivity... done.
```

Virtual maşın şəkli klonlaya biləcəyiniz bir nümunə layihə ilə təchiz olunmuşdur. Budur yuxarıda yaratdığımız `john` istifadəçisi ilə HTTPS üzərindən klonlaşdırırıq; Git bu əlaqə üçün etimadnam əsini tələb edir, lakin etimadnamə cache-i, sonrakı istəklər üçün bu addımı atlamağımıza imkan verəcəkdir.

Fusion Konfiqurasiyası

Git Fusion quraşdırıldıqdan sonra konfiqurasiyanı düzəltmək lazımdır. Ən sevdiyiniz Perforce müştərisini istifadə edərək bunu etmək olduqca asan; yalnız Perforce serverindəki `//.git-fusion` qovluğunu iş sahənizə uyğunlaşdırın. Fayl quruluşu belə görünür:

```
$ tree
.
├──── objects
│    ├──── repos
│    │    └──── [...]
│    └──── trees
│         └──── [...]
│
├──── p4gf_config
├──── repos
│    └──── Talkhouse
│         └──── p4gf_config
└──── users
     └──── p4gf_usermap

498 directories, 287 files
```

`objects` qovluğuna Perforce obyektlərini Git-ə uyğunlaşdırmaq üçün Git Fusion tərəfindən daxili olaraq istifadə olunur və əksinə, orada heç bir şey ilə məşğul olmaq lazım deyil. The `objects` directory is used internally by Git Fusion to map Perforce objects to Git and vice versa, you won't have to mess with anything in there. Bu qovluqda qlobal bir `p4gf_config` faylı və hər bir depo üçün biri var - bunlar Git Fusion-un necə davrandığını təyin edən konfiqurasiya sənədləridir. Root-dakı fayla nəzər salaq:

```
[repo-creation]
charset = utf8

[git-to-perforce]
change-owner = author
enable-git-branch-creation = yes
enable-swarm-reviews = yes
enable-git-merge-commits = yes
enable-git-submodules = yes
preflight-commit = none
ignore-author-permissions = no
read-permission-check = none
git-merge-avoidance-after-change-num = 12107

[perforce-to-git]
http-url = none
ssh-url = none

[@features]
imports = False
chunked-push = False
matrix2 = False
parallel-push = False

[authentication]
email-case-sensitivity = no
```

Buradakı bu flag-ların mənalarına toxunmayacağıq, ancaq bunun Git-in konfiqurasiya üçün istifadə etdiyi kimi yalnız INI formatlı bir mətn faylı olduğunu unutmayın. Bu fayl daha sonra `repos/Talkhouse/p4gf_config` kimi depoya məxsus konfiqurasiya sənədləri ilə ləğv edilə bilən qlobal seçimləri müəyyənləşdirir. Bu faylı açsanız, qlobal standartlardan fərqli bəzi parametrləri olan bir `[@repo]` bölməsini görəcəksiniz. Buna bənzər bölmələri də görəcəksiniz:

```
[Talkhouse-master]
git-branch-name = master
view = //depot/Talkhouse/main-dev/... ...
```

Bu, Perforce branch-ı ilə Git branch-ı arasındakı bir mapping-dir. Bölmə istədiyiniz hər hansı bir ad verilə bilər, ad özünəməxsus olduğu təqdirdə. `git-branch-name`, Git altında çətin olacaq bir depo yolunu daha səmimi bir ada çevirməyə imkan verir. `View` ayarı, standart görünüş uyğunlaşdırma sintaksisindən istifadə edərək Perforce fayllarının Git deposuna necə uyğunlaşdırıldığını idarə edir. Bu nümunədə olduğu kimi birdən çox uyğunlaşma müəyyən edilə bilər:

```
[multi-project-mapping]
git-branch-name = master
view = //depot/project1/main/... project1/...
       //depot/project2/mainline/... project2/...
```

Bu yolla normal iş sahəsi xəritənizə qovluqların strukturundakı dəyişikliklər daxildirsə, bunu Git deposu ilə təkrarlaya bilərsiniz.

Müzakirə edəcəyimiz son sənəd Perforce istifadəçilərini Git istifadəçiləri ilə əlaqələndirən və hətta ehtiyacınız olmaya biləcək `users/p4gf_usermap`dır. Bir Perforce dəyişikliyindən Git commit-inə çevrildikdə Git Fusion-un standart davranışı Perforce istifadəçisini axtarmaq və Git-d əauthor/committer sahəsi üçün orada saxlanılan e-poçt adresini və tam adından istifadə etməkdir. Başqa bir şəkildə konvertasiya edərkən default Perforce istifadəçisini Git commit-inin müəllif sah əsində saxlanan e-poçt ünvanı ilə axtarmaq və dəyişiklikləri həmin istifadəçi kimi təqdim etməkdir (icazələrin tətbiqi ilə). Əksər hallarda, bu davranış çox yaxşı olacaq, lakin aşağıdakı uyğunlaşdırma sənədini nəzərdən keçirin:

```
john john@example.com "John Doe"
john johnny@appleseed.net "John Doe"
bob employeeX@example.com "Anon X. Mouse"
joe employeeY@example.com "Anon Y. Mouse"
```

Hər sətir `<user> <email> "<full name>"` formatındadır və vahid istifadəçi xəritəsini yaradır. İlk iki sətirdə iki fərqli e-poçt ünvanı eyni Perforce istifadəçi hesabına uyğunlaşdırılır. Bu, Git'i bir neçə f ərqli e-poçt ünvanı altında yaratdığınız (və ya e-poçt adreslərini dəyişdirdiyiniz), lakin eyni Perforce istifadəçisinə uyğunlaşdırılmasını istəməyiniz üçün faydalıdır. Bir Perforce dəyişikliyind ən bir Git commit-i yaradarkən, Git müəllif məlumatları üçün Perforce istifadəçisinə uyğun gələn ilk sətir istifadə olunur.

Son iki sətir, Bob və Joe-nun əsl adlarını və Git-dən yaradılan e-poçt adreslərini gizlədir. Daxili bir layihə mənbəyi açmaq istəsəniz də yaxşıdır, ancaq işçilərinizin qovluğunu bütün dünyaya yayımlamaq istəmirsiniz. Bütün Git commit-lərinin tək bir uydurma müəllifə aid edilməsini istəm ədiyiniz təqdirdə e-poçt adresləri və tam adların unikal olması lazım olduğunu unutmayın.

Workflow

Perforce Git Fusion, Perforce və Git versiyası nəzarəti arasında iki tərəfli bir körpüdür. Git tərəfdən işləməyin nə hiss etdiyinə bir nəzər salaq. Yuxarıda göstərildiyi kimi bir konfiqurasiya faylı istifadə edərək "Jam" layihəsində uyğunlaşdırdığımızı güman edəcəyik, bu şəkildə klonlaya bilərik:

```
$ git clone https://10.0.1.254/Jam
Cloning into 'Jam'...
Username for 'https://10.0.1.254': john
Password for 'https://john@10.0.1.254':
remote: Counting objects: 2070, done.
remote: Compressing objects: 100% (1704/1704), done.
Receiving objects: 100% (2070/2070), 1.21 MiB | 0 bytes/s, done.
remote: Total 2070 (delta 1242), reused 0 (delta 0)
Resolving deltas: 100% (1242/1242), done.
Checking connectivity... done.
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/rel2.1
$ git log --oneline --decorate --graph --all
* 0a38c33 (origin/rel2.1) Create Jam 2.1 release branch.
| * d254865 (HEAD, origin/master, origin/HEAD, master) Upgrade to latest metrowerks on
Beos -- the Intel one.
| * bd2f54a Put in fix for jam's NT handle leak.
| * c0f29e7 Fix URL in a jam doc
| * cc644ac Radstone's lynx port.
[...]
```

Bunu ilk dəfə etdiyiniz zaman biraz çəkə bilər. Olan şey, Git Fusion Perforce tarixindəki bütün d əyişiklikləri Git commit-lərinə çevirir. Bu, serverdə yerli olaraq baş verir, buna görə nisbətən sür ətlidir, ancaq bir çox tarixiniz varsa, hələ bir az vaxt ala bilər. Sonrakı gətirmələr artan dönüşüm edir, beləliklə Gitin yerli sürətinə daha çox bənzəyir.

Gördüyünüz kimi, depomuz işləyə biləcəyiniz digər Git deposuna bənzəyir. Üç branch var və Git köməyi ilə `origin/master`-ni izləyən yerli bir `master` branch-ı yaratdı. Gəlin bir az iş görək və bir neçə yeni commit yaradaq:

```
# ...
$ git log --oneline --decorate --graph --all
* cfd46ab (HEAD, master) Add documentation for new feature
* a730d77 Whitespace
* d254865 (origin/master, origin/HEAD) Upgrade to latest metrowerks on Beos -- the
Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

İki yeni vəzifəmiz var. İndi başqasının işlədiyini yoxlayaq:

```
$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://10.0.1.254/Jam
   d254865..6afeb15  master     -> origin/master
$ git log --oneline --decorate --graph --all
* 6afeb15 (origin/master, origin/HEAD) Update copyright
| * cfd46ab (HEAD, master) Add documentation for new feature
| * a730d77 Whitespace
|/
* d254865 Upgrade to latest metrowerks on Beos -- the Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

Deyəsən kimsə olub! Bu baxışdan bunu bilmirsən, amma `6afeb15` commit-i əslində bir Perforce müştərisindən istifadə edərək yaradılmışdır. Sadəcə Git baxımından başqa bir commit-ə bənzəyir, bu da tam nöqtədir. Gəlin Perforce serverinin birləşdirmə commit-i ilə necə məşğul olduğunu gör ək:

```
$ git merge origin/master
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 6), reused 0 (delta 0)
remote: Perforce: 100% (3/3) Loading commit tree into memory...
remote: Perforce: 100% (5/5) Finding child commits...
remote: Perforce: Running git fast-export...
remote: Perforce: 100% (3/3) Checking commits...
remote: Processing will continue even if connection is closed.
remote: Perforce: 100% (3/3) Copying changelists...
remote: Perforce: Submitting new Git commit objects to Perforce: 4
To https://10.0.1.254/Jam
   6afeb15..89cba2b  master -> master
```

Git düşünür ki, bu işləyir. Gəlin `p4v`-nin düzəliş qrafiki xüsusiyyətindən istifadə edərək Perforce baxımından `README` faylının tarixinə nəzər salaq:

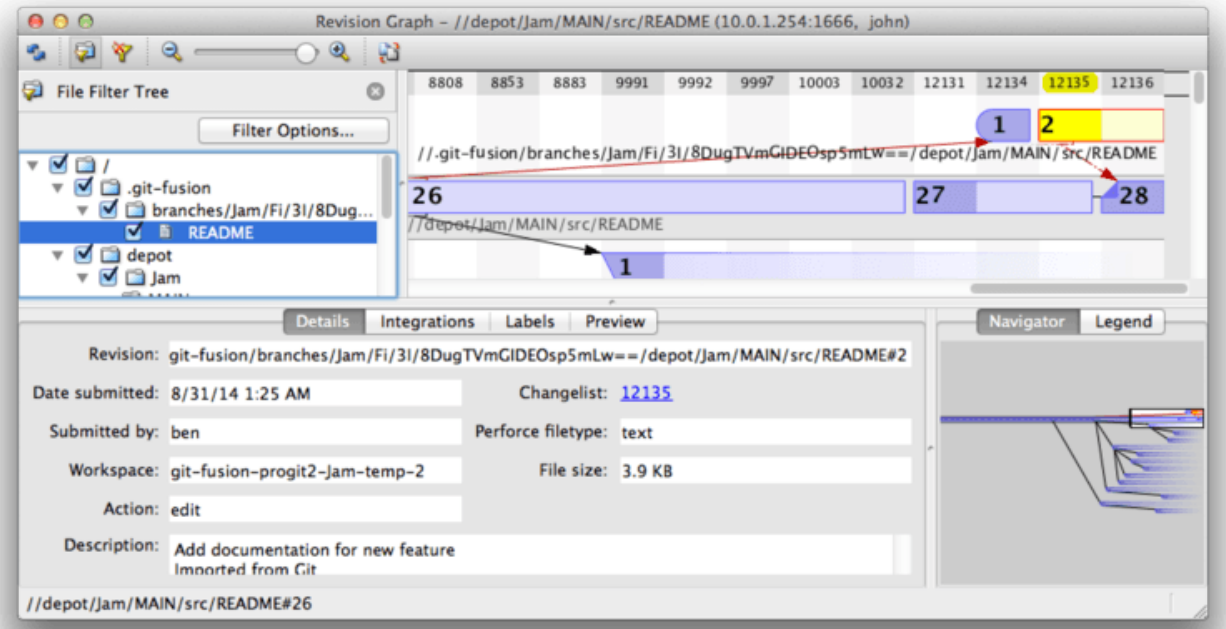


Figure 147. Perforce revision graph resulting from Git push

Bu görünüşü əvvəllər heç görməmisinizsə, qarışıq görünə bilər, ancaq Git tarixçəsi üçün qrafik görüntüləyici ilə eyni anlayışları göstərir. `README` faylının tarixçəsinə baxırıq, buna görə sol üstdəki qovluq ağacı yalnız müxtəlif branch-larda səthdə olduğu halda bu faylı göstərir. Sağ üstdə, faylın müxtəlif düzəlişlərinin bir-birilə əlaqəli olduğuna dair əyani bir qrafik var və bu qrafın böyük şəkil görünüşü sağ alt hissədədir. Görünüşün qalan hissəsi seçilmiş düzəliş üçün detallar görünüşünə verilir (bu vəziyyətdə 2).

Diqqət çəkən bir şey qrafikin Git tarixindəki şəkilə bənzəməsidir. Perforce-nin 1 və 2 commit-lərini saxlamaq üçün adlanan bir branch-ı yox idi, ona görə də `.git-fusion` qovluğunda “anonymous” bir branch yaratdı. Bu, adı verilən bir Perforce branch-ına uyğun gəlməyən Git branch-ları üçün də baş verəcəkdir (və daha sonra onları konfiqurasiya sənədini istifadə edərək Perforce branch-ına uyğunlaşdıra bilərsiniz).

Bunların əksəriyyəti pərdə arxasında baş verir, amma nəticə budur ki, komandadakı bir nəfər Git, digəri Perforce istifadə edə bilər və heç biri digərinin seçimini bilmiyəcək.

Git-Fusion Nəticəsi

Perforce serverinizə girişiniz varsa (və ya əldə edə bilsəniz), Git Fusion, Git və Perforce’u bir-biri ilə danışdırmaq üçün əla bir yoldur. Bir az konfiqurasiya var, amma öyrənmə əyrisi çox dik deyil. Bu, Git-in tam gücündən istifadə ilə bağlı xəbərdarlıqların görünməyəcəyi bu hissədəki bəzi hissələrd ən biridir. Bu, Perforceun atdığınız hər şeydən məmnun qalacağı demək deyil - onsuz da push edilmiş bir keçmişi yenidən yazmağa çalışsanız, Git Fusion bunu rədd edəcək - ancaq Git Fusion t əbii hiss etmək üçün çox çalışır. Hətta Git alt modullarını da istifadə edə bilərsiniz (baxmayaraq ki, onlar Perforce istifadəçiləri üçün qəribə görünəcəklər) və branch-ları birləşdirə bilərsiniz (bu Perforce tərəfində inteqrasiya kimi qeyd olunacaq).

Serverinizin administratorunu Git Fusion qurmağa inandıra bilmirsinizsə, bu alətləri birlikdə istifadə etməyin bir yolu var.

Git-p4

Git-p4, Git və Perforce arasında iki tərəfli bir körpüdür. Tamamilə Git deponuzun içərisində işləyir, beləliklə Perforce serverinə hər hansı bir girişə ehtiyacınız olmayacaq (əlbəttə ki, istifadəçi m əlumatları xaricində). Git-p4, Git Fusion qədər çevik deyil və ya bir həll yolu tamamlamamaqla yanaşı, istədiyiniz işlərin əksəriyyətini server mühitinə müdaxilə olmadan etməyinizə imkan verir.

Git-p4 ilə işləmək üçün `PATH`-nızın bir yerində `p4` alətinə ehtiyacınız olacaq. Bu yazıdan etibarənhttp://www.perforce.com/downloads/Perforce/20-User[] saytında sərbəst şəkildə əldə edilə bilər.

Ayarlamaq

Məsələn, Perforce serverini yuxarıda göstərildiyi kimi Git Fusion OVA-dan çalışdıracağıq, lakin Git Fusion serverini bypass edərək birbaşa Perforce versiya nəzarətinə keçəcəyik.

`p4` əmr sətri müştərisini (git-p4-dən asılı olan) istifadə etmək üçün bir neçə mühit dəyişənini təyin etməlisiniz:

```
$ export P4PORT=10.0.1.254:1666
$ export P4USER=john
```

Başlayırıq

Git-də olduğu kimi, ilk əmr klonlaşdırmaqdır:

```
$ git p4 clone //depot/www/live www-shallow
Importing from //depot/www/live into www-shallow
Initialized empty Git repository in /private/tmp/www-shallow/.git/
Doing initial import of //depot/www/live/ from revision #head into
refs/remotes/p4/master
```

Bu, Git baxımından “shallow” bir klon yaradır; yalnız ən son Perforce revizyonu Git’e idxal olunur; unutmayın, Perforce hər bir istifadəçiyə hər düzəliş vermək üçün dizayn edilməyib. Bu, Git’i Perforce müştərisi kimi istifadə etmək üçün kifayətdir, lakin digər məqsədlər üçün bu kifayət deyil.

Tamamlandıqdan sonra tam işləyən bir Git depomuz var:

```
$ cd myproject
$ git log --oneline --all --graph --decorate
* 70eaf78 (HEAD, p4/master, p4/HEAD, master) Initial import of //depot/www/live/ from
the state at revision #head
```

Perforce server üçün necə bir “p4” remote-nun olduğunu, ancaq hər şeyin standart bir klon kimi göründüyünə diqqət yetirin. Əslində, bu bir az yanıltıcıdır; həqiqətən orada bir remote yoxdur.

```
$ git remote -v
```

Bu depoda heç bir remote yoxdur. Git-p4, serverin vəziyyətini təmsil etmək üçün bəzi referanslar yaratdı və bunlar `git log` üçün remote-dan gələn referanslara bənzəyir, lakin onlar Git tərəfindən idarə olunmur və onlara push etmək olmur.

Workflow

Yaxşı, biraz iş görək. Güman edək ki, çox vacib bir xüsusiyyət üzərində bir az irəliləyiş əldə etdiniz və bunu komandanızın qalan hissəsinə göstərməyə hazırsınız.

```
$ git log --oneline --all --graph --decorate
* 018467c (HEAD, master) Change page title
* c0fb617 Update link
* 70eaf78 (p4/master, p4/HEAD) Initial import of //depot/www/live/ from the state at
revision #head
```

Perforce serverinə təqdim etməyə hazır olduğumuz iki yeni commit götürdük. Bu gün başqa birinin işlədiyini yoxlayaq:

```
$ git p4 sync
git p4 sync
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12142 (100%)
$ git log --oneline --all --graph --decorate
* 75cd059 (p4/master, p4/HEAD) Update copyright
| * 018467c (HEAD, master) Change page title
| * c0fb617 Update link
|/
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Göründüyü kimi onlar idi və `master` və `p4/master` ayrıldı. Perforce-in branching sistemi Git kimi *heç bir şey* deyil, bu səbəbdən birləşmə commit-lərini təqdim etməyin heç bir mənası yoxdur. Git-p4, commit-lərinizi yenidən düzəltməyinizi tövsiyə edir və hətta bunu etmək üçün qısa yolla gəlir:

```
$ git p4 rebase
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
No changes to import!
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
Applying: Update link
Applying: Change page title
 index.html | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Çıxışdan çox şey bilə bilərsiniz, ancaq `git p4 rebase` `git p4 sync` üçün bir qısayoldur, ardından `git rebase p4/master` izləyin. Xüsusilə birdən çox branch-la işləyəndə bundan biraz ağıllıdır, amma yaxşı bir yanaşma var.

İndi tariximiz yenidən xəttidir və dəyişikliklərimizi Perforce-a qaytarmaq üçün hazırıq. `git p4 submit` əmri, `p4/master` və `master` arasındakı hər Git commit-i üçün yeni bir Perforce revizyonu yaratmağa çalışacaq. Çalıştırmamız bizi sevdiyimiz redaktora salır və faylın məzmunu belə bir şeyə bənzəyir:

```
# A Perforce Change Specification.
#
#  Change:      The change number. 'new' on a new changelist.
#  Date:        The date this specification was last modified.
#  Client:      The client on which the changelist was created.  Read-only.
#  User:        The user who created the changelist.
#  Status:      Either 'pending' or 'submitted'. Read-only.
#  Type:        Either 'public' or 'restricted'. Default is 'public'.
#  Description: Comments about the changelist.  Required.
#  Jobs:        What opened jobs are to be closed by this changelist.
#               You may delete jobs from this list.  (New changelists only.)
#  Files:       What opened files from the default changelist are to be added
#               to this changelist.  You may delete files from this list.
#               (New changelists only.)

Change:  new

Client:  john_bens-mbp_8487

User: john

Status:  new

Description:
   Update link

Files:
   //depot/www/live/index.html   # edit


######## git author ben@straub.cc does not match your p4 account.
######## Use option --preserve-user to modify authorship.
######## Variable git-p4.skipUserNameCheck hides this message.
######## everything below this line is just the diff #######
--- //depot/www/live/index.html  2014-08-31 18:26:05.000000000 0000
+++ /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/index.html   2014-
08-31 18:26:05.000000000 0000
@@ -60,7 +60,7 @@
 </td>
 <td valign=top>
 Source and documentation for
-<a href="http://www.perforce.com/jam/jam.html">
+<a href="jam.html">
 Jam/MR</a>,
 a software build tool.
 </td>
```

Bu, əsasən git-p4-in daxil etdiyi şeylər istisna olmaqla, `p4 submit` düyməsini işə salmaqla gördüyünüz eyni məzmundur. Git-p4, bir commit və ya dəyişiklik üçün bir ad verməli olduqda, Git

və Perforce parametrlərinizi fərdi olaraq hörmət etməyə çalışır, lakin bəzi hallarda onu ləğv etmək istəyirsiniz. Məsələn, idxal etdiyiniz Git commit-ini Perforce istifadəçi hesabına sahib olmayan bir iştirakçı yazmışsa, nəticədə dəyişikliklərin yazdıqları kimi görünməsini istəyə bilərsiniz (siz də yox).

Git-p4, bu Perforce dəyişikliyinin məzmunu olaraq Git commit-indən mesajı faydalı bir şəkildə idxal etdi, buna görə etməli olduğumuz şey iki dəfə saxlamaq və çıxmaqdır (hər bir commit üçün bir dəfə). Nəticədə shell çıxışı belə görünəcəkdir:

```
$ git p4 submit
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-
mbp_8487/john_bens-mbp_8487/depot/www/live/
Synchronizing p4 checkout...
... - file(s) up-to-date.
Applying dbac45b Update link
//depot/www/live/index.html#4 - opened for edit
Change 12143 created with 1 open file(s).
Submitting change 12143.
Locking 1 files ...
edit //depot/www/live/index.html#5
Change 12143 submitted.
Applying 905ec6a Change page title
//depot/www/live/index.html#5 - opened for edit
Change 12144 created with 1 open file(s).
Submitting change 12144.
Locking 1 files ...
edit //depot/www/live/index.html#6
Change 12144 submitted.
All commits applied!
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12144 (100%)
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
$ git log --oneline --all --graph --decorate
* 775a46f (HEAD, p4/master, p4/HEAD, master) Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Nəticə sanki sadəcə baş verənlərə ən yaxın bənzətmə olan bir `git push` işlətmişik.

Qeyd edək ki, bu müddət ərzində hər Git commit-i Perforce dəyişikliyinə çevrilir; onları tək bir d əyişiklik qrupuna yığmaq istəyirsinizsə, bunu `git p4 submit`-i işə salmadan əvvəl interaktiv bir b ərpa ilə edə bilərsiniz. Dəyişiklik olaraq təqdim olunan bütün commit-lərin SHA-1 hash-larının d əyişdiyini də unutmayın; çünki git-p4 çevirdiyi hər bir commit-in sonuna bir sətir əlavə edir:

```
$ git log -1
commit 775a46f630d8b46535fc9983cf3ebe6b9aa53145
Author: John Doe <john@example.com>
Date:   Sun Aug 31 10:31:44 2014 -0800

    Change page title

    [git-p4: depot-paths = "//depot/www/live/": change = 12144]
```

Birləşdirmə commit-ini təqdim etməyə çalışsanız nə olur? Gəlin bir cəhd edək. Budur düşdüyümüz vəziyyət:

```
$ git log --oneline --all --graph --decorate
* 3be6fd8 (HEAD, master) Correct email address
*   1dcbf21 Merge remote-tracking branch 'p4/master'
|\
| * c4689fc (p4/master, p4/HEAD) Grammar fix
* | cbacd0a Table borders: yes please
* | b4959b6 Trademark
|/
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Git və Perforce tarixi 775a46f-dən sonra ayrılır. Git tərəfinin iki commit var, daha sonra Perforce başlığı ilə birləşdirmə, sonra başqa bir commit vardır. Bunları Perforce tərəfindəki tək bir d əyişiklik üzərində təqdim etməyə çalışacağıq. İndi təqdim etməyə çalışsaq nə olacağını görək:

```
$ git p4 submit -n
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-
mbp_8487/john_bens-mbp_8487/depot/www/live/
Would synchronize p4 checkout in /Users/ben/john_bens-mbp_8487/john_bens-
mbp_8487/depot/www/live/
Would apply
  b4959b6 Trademark
  cbacd0a Table borders: yes please
  3be6fd8 Correct email address
```

`-n` flag-ı `--dry-run`-nın qısaldılmasıdır və submit əmri real olaraq işləsəydi nə olacağını bildirməyə çalışır. Bu vəziyyətdə, Perforce serverində hələ mövcud olmayan birləşdirilməmiş üç commit-ə uyğun üç Perforce dəyişiklikləri yaratdığımıza bənzəyir. Bu, tam olaraq istədiyimiz kimi səslənir, gəlin nəticəyə baxaq:

```
$ git p4 submit
[···]
$ git log --oneline --all --graph --decorate
* dadbd89 (HEAD, p4/master, p4/HEAD, master) Correct email address
* 1b79a80 Table borders: yes please
* 0097235 Trademark
* c4689fc Grammar fix
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Göndərmədən əvvəl yenidən müəyyənləşdirdiyimiz kimi keçmişimiz xətti oldu (tam olaraq nə baş verdi). Bu, keçmişinizin Perforce ilə bir araya sığmayacağından qorxmadan Git tərəfində branch-ları yaratmaq, işləmək, atmaq və birləşdirməkdə sərbəst ola biləcəyiniz deməkdir.. Yenidən düzə ldə bilsəniz, bir Perforce serverinə kömək edə bilərsiniz.

Branching

Perforce layihənizin birdən çox branch-ı varsa, hələ də şansınız var; git-p4 bunu Git kimi hiss etdirəcək şəkildə idarə edə bilər. Deyək ki, Perforce deposunuz belə qurulub:

```
//depot
   └─── project
        ├─── main
        └─── dev
```

Git-p4 bilər və deyək ki, belə bir görünüş spesifikasiyasına sahib bir `dev` branch-ı var: Bu vəziyyəti tamamilə aşkar edin və düzgün bir şey edin:

```
//depot/project/main/... //depot/project/dev/...
```

Git-p4 bu vəziyyəti avtomatik olaraq aşkar edə bilər və düzgün işi edə bilər:

```
$ git p4 clone --detect-branches //depot/project@all
Importing from //depot/project@all into project
Initialized empty Git repository in /private/tmp/project/.git/
Importing revision 20 (50%)
    Importing new branch project/dev

    Resuming with change 20
Importing revision 22 (100%)
Updated branches: main dev
$ cd project; git log --oneline --all --graph --decorate
* eae77ae (HEAD, p4/master, p4/HEAD, master) main
| * 10d55fb (p4/project/dev) dev
| * a43cfae Populate //depot/project/main/... //depot/project/dev/....
|/
* 2b83451 Project init
```

Depo yolundakı “@all” spesifikatoruna diqqət yetirin; bu git-p4-ə yalnız bu alt ağac üçün son d əyişiklikləri deyil, bu yollara toxunan bütün dəyişiklikləri klonlamasını söyləyir. Bu, Git-in bir klon konsepsiyasına daha yaxındır, ancaq uzun bir tarixə sahib bir layihə üzərində işləyirsinizsə, biraz vaxt ala bilər.

`--detect-branches` flag-ı git-p4-ə Perforce-in branch xüsusiyyətlərini istifadə edərək branch-ları Git reflərinə uyğunlaşdırmağı məsləhət görür. Bu uyğunlaşmalar Perforce serverində yoxdursa (Perforce’u istifadə etmək üçün tamamilə etibarlı bir yoldur), git-p4-ə branch uyğunlaşmalarının nə olduğunu söyləyə bilərsiniz və eyni nəticəni əldə edə bilərsiniz:

```
$ git init project
Initialized empty Git repository in /tmp/project/.git/
$ cd project
$ git config git-p4.branchList main:dev
$ git clone --detect-branches //depot/project@all .
```

`git-p4.branchList` konfiqurasiya dəyişənini `main:dev` olaraq təyin etmək git-p4-ə “main” and “dev”-in hər iki branch olduğunu, ikincisinin isə birincinin övladı olduğunu bildirir.

İndi `git checkout -b dev p4/project/dev` və bəzi işlər görsək, git-p4 `git p4 submit` etdiyimiz zaman doğru branch-ı hədəf alacaq qədər ağıllıdır. Təəssüf ki, git-p4 shallow klonları və çoxsaylı branch-ları qarışdıra bilməzr; nəhəng bir layihəniz varsa və birdən çox branch üzərində işləmək ist əyirsinizsə, təqdim etmək istədiyiniz hər branch üçün bir dəfə `git p4 clone` işlətməlisiniz.

Branch-lar yaratmaq və ya birləşdirmək üçün bir Perforce müştərisindən istifadə etməlisiniz. Git-p4 yalnız mövcud branch-lara sinxronizasiya edə və təqdim edə bilər və eyni anda yalnız bir xətti dəyişiklik edə bilər. Git-də iki branch-ı birləşdirirsinizsə və yeni dəyişiklikləri göndərməyə çalışsanız, qeyd olunacaq bir çox fayl dəyişikliyidir; branch-ların inteqrasiyada iştirak etdiyi metadata itiriləcəkdir.

Git və Perforce Nəticəsi

Git-p4 Perforce server ilə bir Git iş axını istifadə etməyi mümkün edir və bu işdə olduqca yaxşıdır. Bununla birlikdə, Perforce-un mənbədən məsul olduğunu və yalnız Git'i yerli işləmək üçün istifadə etdiyinizi unutmamalısınız. Yalnız Git commit-lərini bölüşmək üçün həqiqətən diqqətli olun; digər insanların istifadə etdiyi bir remote-nuz varsa, əvvəllər Perforce serverinə təqdim olunmamış commit-lər verməyin.

Perforce və Git-in mənbə nəzarəti üçün müştəri kimi istifadəsini sərbəst şəkildə qarışdırmaq ist əyirsinizsə və server administratorunu onu quraşdırmağa inandıra bilsəniz, Git Fusion, Git-i Perforce server üçün birinci dərəcəli versiya-nəzarət müştərisinə çevirir.

Git və TFS

Git, Windows developer-ləri ilə populyarlaşır və Windows-da kod yazırsınızsa, Microsoft Team Foundation Server (TFS) istifadə etmək üçün böyük bir şans var. TFS, qüsur və iş elementlərinin izlənməsi, Scrum və digərləri üçün proses dəstəyi, kodun nəzərdən keçirilməsi və versiya nəzar ətini əhatə edən bir əməkdaşlıq paketidir. Qarşıda bir az qarışıqlıq var: **TFS** həm Git'i, həm də **TFVC** (Team Foundation Version Control) adını verdikləri öz xüsusi VNCS-lərini istifadə edərək n əzarət mənbəyi kodunu dəstəkləyən serverdir. Git dəstəyi TFS üçün bir qədər yeni bir xüsusiyy ətdir (2013 versiyası ilə göndərmə), buna görə əvvəldən gələn bütün vasitələr, əsasən TFVC ilə çalışsalar da versiya nəzarət hissəsini "TFS" adlandırırlar.

Özünüzü TFVC istifadə edən bir komandada görürsəniz, ancaq versiya nəzarət müştərisi olaraq Git'i istifadə etsəniz, sənin üçün bir layihə var.

Hansı Tool

Əslində iki var: git-tf və git-tfs.

Git-tfs (https://github.com/git-tfs/git-tfs saytında tapın) bir .NET layihəsidir və (bu yazı kimi) yalnız Windows-da işləyir. Git depoları ilə işləmək üçün yüksək performanslı və Git deposunun bağırsaqları ilə çox rahatlıq təmin edən Git-in kitabxana yönümlü tətbiqi olan libgit2 üçün .NET bağlamalarından istifadə edir. Libgit2, Git-in tam bir tətbiqi deyil, beləliklə fərqi ödəmək üçün git-tfs həqiqətən bəzi əməliyyatlar üçün əmr sətiri Git müştəri çağıracaq, buna görə də Git depoları ilə əlaqəli süni məhdudiyyətlər yoxdur. TFVC xüsusiyyətlərini dəstəkləməsi çox yetkindir, çünki serverlərlə əməliyyatlar üçün Visual Studio assemblilərindən stifadə edir. Bu o assemblilərə girişə ehtiyacınız olacaq deməkdir, yəni Visual Studio-nun son versiyasını (versiya 2012-dən bəri Express daxil olmaqla 2010-cu ildən bəri hər hansı bir nəşr) və ya Visual Studio SDK-nı quraşdırmanız lazımdır.

Git-tf, End-of-Life (EOL), heç bir yeniləmə almayacaq. Artıq Microsoft tərəfindən dəstəklənmir.

Git-tf (hansı ki evi https://archive.codeplex.com/?p=gittf ünvanındadı) bir Java layihəsidir və bu ş əkildə Java işləmə müddəti olan istənilən kompüterdə işləyir. JGit (Git'in bir JVM tətbiqi) vasitəsi ilə Git depoları ilə əlaqə qurur, yəni Git funksiyaları baxımından heç bir məhdudiyyətə sahib deyildir. Bununla birlikdə, TFVC üçün dəstəyi git-tfs ilə müqayisədə məhduddur - məsələn branch-ları dəst əkləmir.

Beləliklə, hər bir vasitənin müsbət və mənfi cəhətləri var və bir-birinə üstünlük verən çoxlu və ziyyət var. Hər ikisinin əsas istifadəsini bu kitabda nəzərdən keçirəcəyik.

Bu təlimatları izləmək üçün TFVC əsaslı bir depoya daxil olmağınız lazımdır. Bunlar vəhşi təbiətdə Git və ya Subversion depoları qədər çox deyil, buna görə özünüzdən birini yaratmanız lazım ola bilər. Codeplex (https://archive.codeplex.com/)) və ya Visual Studio Online (https://visualstudio.microsoft.com) hər ikisi bunun üçün yaxşı seçimdir.

Başlanğıc: `git-tf`

Etdiyiniz ilk şey hər hansı bir Git layihəsində olduğu kimi klondur. `git-tf` ilə görünən budur:

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main project_git
```

Birinci arqument bir TFVC kolleksiyasının URL'si, ikincisi `$/project/branch` şəklində, üçüncüsü isə yaradılacaq yerli Git deposuna gedən yoldur (sonuncusu istəyə bağlıdır). Git-tf hər dəfə yalnız bir branch ilə işləyə bilər; fərqli bir TFVC branch-ında chekinlər etmək istəyirsinizsə, bu branch-dan yeni bir klon etməlisiniz.

Bu, tamamilə işləyən bir Git deposu yaradır:

```
$ cd project_git
$ git log --all --oneline --decorate
512e75a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Checkin message
```

Buna yalnız son dəyişikliklər endirildiyi anlamına bir *shallow* klonu deyilir. TFVC, hər bir müşt ərinin tarixin tam bir nüsxəsini əldə etməsi üçün nəzərdə tutulmamışdır, beləliklə git-tf varsayılan olaraq yalnız son versiyasını alır, bu da daha sürətlidir.

Bir az vaxtınız varsa, ehtimal ki, bütün layihə tarixçəsini `--deep` seçimindən istifadə edərək klonlamağa dəyər:

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main \
  project_git --deep
Username: domain\user
Password:
Connecting to TFS...
Cloning $/myproject into /tmp/project_git: 100%, done.
Cloned 4 changesets. Cloned last changeset 35190 as d44b17a
$ cd project_git
$ git log --all --oneline --decorate
d44b17a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Goodbye
126aa7b (tag: TFS_C35189)
8f77431 (tag: TFS_C35178) FIRST
0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
        Team Project Creation Wizard
```

`TFS_C35189` kimi adlarla etiketlərə diqqət yetirin; bu, hansı Gitin TFVC dəyişiklikləri ilə əlaqəli olduğunu bilməyinizə kömək edən bir xüsusiyyətdir. Bu onu təmsil etmək üçün gözəl bir yoldur, çünki sadə bir günlük əmri ilə hansınızın TFVC-də mövcud olan bir snapshot ilə əlaqəli olduğunu görə bilərsiniz. Bunlar lazım deyil (və əslində onları `git config git-tf.tag false` ilə söndürə bil ərsiniz) - git-tf, `.git/git-tf` faylında həqiqi commit-changeset uyğunlaşmalarını saxlayır.

Başlanğıc: `git-tfs`

Git-tfs klonlama bir az fərqli davranır. Baxaq:

```
PS> git tfs clone --with-branches \
    https://username.visualstudio.com/DefaultCollection \
    $/project/Trunk project_git
Initialized empty Git repository in C:/Users/ben/project_git/.git/
C15 = b75da1aba1ffb359d00e85c52acb261e4586b0c9
C16 = c403405f4989d73a2c3c119e79021cb2104ce44a
Tfs branches found:
- $/tfvc-test/featureA
The name of the local branch will be : featureA
C17 = d202b53f67bde32171d5078968c644e562f1c439
C18 = 44cd729d8df868a8be20438fdeeefb961958b674
```

`--with-branches` flag-na diqqət yetirin. Git-tfs, TFVC branch-larını Git branch-larına uyğunlaşdırma qabiliyyətinə malikdir və bu bayraq hər TFVC branch-ı üçün yerli bir Git branch-ının qurulmasını bildirir. TFS-də branched zamanı və ya birləşdiyiniz təqdirdə bu çox tövsiyə olunur, lakin TFS 2010-dan əvvəlki bir serverlə işləməyəcək - git-tfs onları adi qovluqlardan ayıra bilməz, çünki bu buraxılışdan əvvəl “branch-lar” sadəcə qovluq idi.

Nəticədə çıxan Git deposuna nəzər salaq:

```
PS> git log --oneline --graph --decorate --all
* 44cd729 (tfs/featureA, featureA) Goodbye
* d202b53 Branched from $/tfvc-test/Trunk
* c403405 (HEAD, tfs/default, master) Hello
* b75da1a New project
PS> git log -1
commit c403405f4989d73a2c3c119e79021cb2104ce44a
Author: Ben Straub <ben@straub.cc>
Date:   Fri Aug 1 03:41:59 2014 +0000

    Hello

    git-tfs-id:
[https://username.visualstudio.com/DefaultCollection]$/myproject/Trunk;C16
```

Klonun başlanğıc nöqtəsini (TFVC-də `Trunk`) və bir child branch-ını (TFVC-də `featureA`) təmsil edən iki local branch, `master` və `featureA` var. Ayrıca, `tfs` "remote"-da bir neçə referensa sahib olduğunu görə bilərsiniz: TFVC branch-larını təmsil edən `default` və `featureA`. Git-tfs klonladığınız branch-ı `tfs/default`-a xəritələşdirir və digərləri öz adlarını alır.

Diqqəti cəlb edən başqa bir şey, mesajlardakı `git-tfs-id:` sətirləridir. Etiket əvəzinə git-tfs, TFVC d əyişikliklərini Git commit-ləri ilə əlaqələndirmək üçün bu markerlərdən istifadə edir. Bu, Git commit-lərinizin TFVC-yə göndərilmədən əvvəl və sonra fərqli bir SHA-1 qarışığına sahib olacağı mənasını verir.

Git-tf[s] Workflow

Hansı alətdən istifadə etməyinizdən asılı olmayaraq, problemlərlə qarşılaşmamaq üçün bir neçə Git konfiqurasiya dəyərini təyin etməlisiniz.

```
$ git config set --local core.ignorecase=true
$ git config set --local core.autocrlf=false
```

İstəyəcəyiniz açıq bir şey layihə üzərində işləməkdir. TFVC və TFS, iş axınınıza mürəkkəblik əlavə edə biləcək bir neçə xüsusiyyətə malikdir:

1. TFVC-də təmsil olunmayan xüsusiyyət branch-ları bir az mürəkkəblik əlavə edir. Bu, TFVC və Gitin branch-ları təmsil etdiyi **çox** fərqli yollarla əlaqəlidir.
2. Qeyd edək ki, TFVC, istifadəçilərə serverdəki faylları "checkout" və kilidləməyə imkan verir ki, heç kim onu düzəldə bilməz. Şübhəsiz ki, bu, onları yerli deponuzda düzəltməyinizə mane olmayacaq, ancaq dəyişikliklərinizi TFVC serverinə göndərmə vaxtı gələndə mane ola bil ərsiniz.
3. TFS, girişə icazə verilməzdən əvvəl bir TFS build-test dövrünün uğurla başa çatması lazım olan "gated" qeydlər konsepsiyasına malikdir. Bu, TFVC-də burada "shelve" funksiyasından istifadə edir, hansı ki burada ətraflı danışmırıq. Bunu git-tf ilə manual olaraq saxta edə bilərsiniz və git-tfs gate-aware olan `checkintool` əmrini təmin edir.

Qısaca, burada əhatə edəcəyimiz bu məsələlərin əksəriyyətini kənara qoyan və ya çəkindirən xoşbəxt yoldur.

Workflow: `git-tf`

Deyək ki, bir az iş görmüsünüz, bir neçə Git `master` üzərində commit götürdünüz və TFVC serverində irəliləyişinizi bölüşməyə hazırsınız. Budur Git depomuz:

```
$ git log --oneline --graph --decorate --all
* 4178a82 (HEAD, master) update code
* 9df2ae3 update readme
* d44b17a (tag: TFS_C35190, origin_tfs/tfs) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

`4178a82` commit-indəki ani görüntüyü götürmək və TFVC serverinə ötürmək istəyirik. Əvvəlcə hər şey: gəlin görək komanda yoldaşlarımızdan biri son əlaqə qurandan bəri bir şey edib etməyib:

```
$ git tf fetch
Username: domain\user
Password:
Connecting to TFS...
Fetching $/myproject at latest changeset: 100%, done.
Downloaded changeset 35320 as commit 8ef06a8. Updated FETCH_HEAD.
$ git log --oneline --graph --decorate --all
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
| * 4178a82 (HEAD, master) update code
| * 9df2ae3 update readme
|/
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

Deyəsən başqası da işləyir və indi fərqli tarixə sahibik. Gitin parladığı yer budur, amma necə davam edəcəyimizə dair iki seçimimiz var:

1. Birləşdirmə commit-i etmək, Git istifadəçisi olaraq təbiidir (nəticə olaraq `git pull`-da elə bunu edir) və git-tf bunu sizin üçün sadə bir `git tf pull` ilə edə bilər. Ancaq unutmayın ki, TFVC bu şəkildə düşünmür və birləşməni push etsəniz tarixiniz hər iki tərəfdə fərqli görünməyə başlayacaq, bu da qarışıq ola bilər. Bununla birlikdə, bütün dəyişikliklərinizi bir dəyişiklik kimi təqdim etməyi planlaşdırırsınızsa, bu, yəqin ki, ən asan seçimdir.
2. Rebasing, tariximizi xətti edir, yəni hər bir Git commit-imizi bir TFVC dəyişikliyinə çevirmə seçimimiz var. Ən çox seçim açıq qaldığı üçün bunu bu şəkildə etməyinizi məsləhət görürük; git-tf hətta `git tf pull --rebase` ilə sizin üçün asanlaşdırır.

Seçim sizindir. Bu nümunə üçün biz geri qayıdacağıq:

```
$ git rebase FETCH_HEAD
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

İndi TFVC serverinə giriş yazmağa hazırıq. Git-tf, sonuncudan başlayaraq bütün dəyişiklikləri əks etdirən tək bir dəyişiklik etmə seçimini (default olaraq `--shallow`) və hər bir Git commit-i üçün yeni bir dəyişiklik yaratma seçimini verir (`--deep`). Bu nümunə üçün yalnız bir dəyişiklik düzəldəcəyik:

```
$ git tf checkin -m 'Updating readme and code'
Username: domain\user
Password:
Connecting to TFS...
Checking in to $/myproject: 100%, done.
Checked commit 5a0e25e in as changeset 35348
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, tag: TFS_C35348, origin_tfs/tfs, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

TFVC-nin `5a0e25e` commit-i ilə eyni snapshot-nu saxladığını göstərən yeni bir `TFS_C35348` etiketi var. Qeyd etmək vacibdir ki, hər bir Gitin TFVC-də dəqiq bir həmkarına sahib olması lazım deyil; məsə lən, `6eb3eb5` commit-i serverin heç bir yerində yoxdur.

Əsas iş axını budur. Unutmamaq istədiyiniz bir neçə başqa məqam var:

- Branching yoxdur. Git-tf hər dəfə yalnız bir TFVC branch-ından Git depoları yarada bilər.
- TFVC və ya Git istifadə edərək əməkdaşlıq edin, lakin hər ikisini istifadə etməyin. Eyni TFVC deposunun fərqli git-tf klonlarında, baş ağrısına səbəb olmayan fərqli yekunlaşdırıcı SHA-1 hash-ları ola bilər.
- Komandanızın iş axını Git-də iş birliyini və TFVC ilə vaxtaşırı sinxronizasiyanı əhatə edirsə,

yalnız Git depolarından biri ilə TFVC-yə qoşulun.

Workflow: `git-tfs`

Git-tfs istifadə edərək eyni ssenaridən keçək. Git depomuzdakı `master` branch-na etdiyimiz yeni commit-lər:

```
PS> git log --oneline --graph --all --decorate
* c3bd3ae (HEAD, master) update code
* d85e5a2 update readme
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 (tfs/default) Hello
* b75da1a New project
```

İndi gəlin görək başqaları işimizi görüb ya yox:

```
PS> git tfs fetch
C19 = aea74a0313de0a391940c999e51c5c15c381d91d
PS> git log --all --oneline --graph --decorate
* aea74a0 (tfs/default) update documentation
| * c3bd3ae (HEAD, master) update code
| * d85e5a2 update readme
|/
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Bəli, iş yoldaşımızın yeni `aea74a0` əmri kimi görünən yeni bir TFVC dəyişiklik dəsti əlavə etdiyi və `tfs/default` uzaq branch-ın hərəkət etdiyi ortaya çıxdı.

Git-tf-də olduğu kimi, bu fərqli tarixi necə həll etmək üçün iki əsas seçimimiz var:

1. Xətti bir tarixi qorumaq üçün yenidən bərpa edin.
2. Həqiqətən baş verənləri qorumaq üçün birləşdirin.

Bu vəziyyətdə, hər bir Git-in bir TFVC dəyişikliyinə çevrildiyi bir “dərin” yoxlama edəcəyik, buna görə geri qayıdaq.

```
PS> git rebase tfs/default
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
PS> git log --all --oneline --graph --decorate
* 10a75ac (HEAD, master) update code
* 5cec4ab update readme
* aea74a0 (tfs/default) update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

İndi TFVC serverinə kodumuzu yoxlayaraq töhfəmizi tamamlamağa hazırıq. HEAD-dən tapılan ilk `tfs` uzaq branch-ına gedən yolda hər bir Git əməli üçün bir TFVC dəyişikliyi yaratmaq üçün burada `rcheckin` əmrindən istifadə edəcəyik (`checkin` əmri yalnız bir dəyişik yaradır; Git kimi commit-lər götürür).

```
PS> git tfs rcheckin
Working with tfs remote: default
Fetching changes from TFS to minimize possibility of late conflict...
Starting checkin of 5cec4ab4 'update readme'
 add README.md
C20 = 71a5ddce274c19f8fdc322b4f165d93d89121017
Done with 5cec4ab4b213c354341f66c80cd650ab98dcf1ed, rebasing tail onto new TFS-
commit...
Rebase done successfully.
Starting checkin of b1bf0f99 'update code'
 edit .git\tfs\default\workspace\ConsoleApplication1/ConsoleApplication1/Program.cs
C21 = ff04e7c35dfbe6a8f94e782bf5e0031cee8d103b
Done with b1bf0f9977b2d48bad611ed4a03d3738df05ea5d, rebasing tail onto new TFS-
commit...
Rebase done successfully.
No more to rcheckin.
PS> git log --all --oneline --graph --decorate
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

TFVC serverinə hər uğurlu qeydiyyatdan keçdikdən sonra git-tfs-in qalan işləri əvvəlki işlərinə necə qaytardığına diqqət yetirin. Bunun səbəbi SHA-1 hash-ları dəyişdirən commit mesajlarının altına `git-tfs-id` sahəsini əlavə etməsidir.

Bu tam olaraq dizayn edilmiş şəkildədir və narahat olacağınız bir şey yoxdur, ancaq bunun baş verdiyini, xüsusən də Git commit-lərini başqaları ilə bölüşdüyünüzü bilməlisiniz.

TFS, versiya nəzarət sistemi ilə inteqrasiya olunan bir çox xüsusiyyətə malikdir, məsələn iş itemları, təyin olunmuş reviewer-lərin, gated checkins və s. Bu xüsusiyyətlərlə yalnız bir əmr sətiri alətindən istifadə etmək çətin ola bilər, amma xoşbəxtlikdən git-tfs qrafik qeyd alətini çox asanlıqla işə salmağa imkan verir:

```
PS> git tfs checkintool
PS> git tfs ct
```

Belə görünəcəkdir:

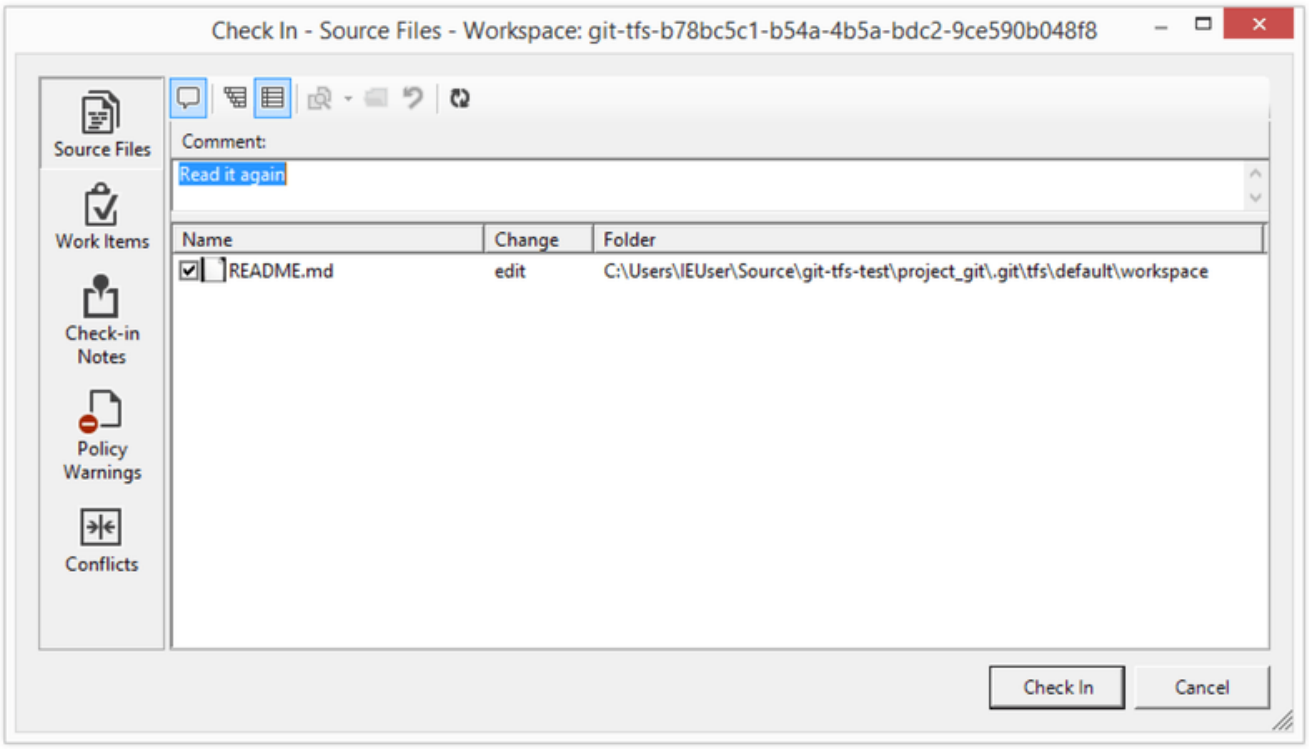


Figure 148. The git-tfs checkin tool

Bu, Visual Studio içərisindən başlayan eyni dialoq olduğundan TFS istifadəçilərinə tanış görünür.

Git-tfs ayrıca Git deposundan TFVC branch-larını idarə etməyə imkan verir. Nümunə olaraq birini yaradaq:

```
PS> git tfs branch $/tfvc-test/featureBee
The name of the local branch will be : featureBee
C26 = 1d54865c397608c004a2cadce7296f5edc22a7e5
PS> git log --oneline --graph --decorate --all
* 1d54865 (tfs/featureBee) Creation branch $/myproject/featureBee
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

TFVC-də bir branch yaratmaq, bu branch-ın mövcud olduğu bir dəyişiklik dəsti əlavə etmək dem əkdir və bu, Git commit-i kimi proqnozlaşdırılır. Git-tfs `tfs/featureBee` uzaq branch-ını **yaratdığını** da unutmayın, lakin `HEAD` hələ də `master`-ə işarə edir. Newly-minted branch-da işləmək istəyirsinizs ə, yeni commit-lərinizi, bəlkə də həmin commit-dən mövzu commit-i yaratmaqla `1d54865` commit-i üzərində qurmaq istəyirsiniz.

Git və TFS Qısa Məzmunu

Git-tf və Git-tfs TFVC serverlə əlaqə yaratmaq üçün əla vasitədir. Git gücünü yerli olaraq istifadə etməyə, mərkəzi TFVC serverinə davamlı gediş-gəlişdən çəkinməyə və bütün komandanızı Git-ə köçməyə məcbur etmədən bir developer olaraq həyatınızı asanlaşdırmağa imkan verirlər. Windows-da işləyirsinizsə (çox güman ki, komandanız TFS istifadə edir), xüsusiyyət dəsti daha tamamlandığından git-tfs istifadə etmək istərdiniz, ancaq başqa bir platformada işləsəniz, daha m əhdud olan git-tf istifadə edəcəksiniz. Bu fəsildəki alətlərin əksəriyyətində olduğu kimi, bu versiya idarəetmə sistemlərindən birini kanonik olaraq seçməlisiniz və digərini tabe bir şəkildə istifadə etməlisiniz - ya Git ya da TFVC əməkdaşlıq mərkəzi olmalıdır, lakin hər ikisi deyil.

Git'ə Miqrasiya

Başqa bir VNS'də mövcud bir kod bazanız varsa, ancaq Git istifadə etməyə qərar verdiyiniz tə qdirdə layihənizi bu və ya digər şəkildə köçürməlisiniz. Bu bölmə ümumi sistemlər üçün bəzi idxalçıları nəzərdən keçirir və sonra öz xüsusi idxalatçısının necə inkişaf etdiriləcəyini göstərir. Keçid edən istifadəçilərin əksəriyyətini təşkil etdiyindən və onlar üçün yüksək keyfiyyətli alətlərin gəlməsi asan olduğundan, peşəkarlıqla istifadə olunan bir neçə daha böyük SCM sistemindən m əlumatları necə idxal edəcəyinizi öyrənəcəksiniz.

Subversion

`git svn` istifadəsi ilə bağlı əvvəlki bölümü oxumusunuzsa, bu təlimatları asanlıqla `git svn clone` deposuna getmək üçün istifadə edə bilərsiniz; sonra, Subversion serverindən istifadəni dayandırın, yeni bir Git serverinə keçin və istifadə etməyə başlayın. Tarixçəni istəsəniz, məlumatı Subversion serverindən çıxara bildiyiniz müddətdə (bir müddət ala bilər) bacara bilərsiniz.

Bununla birlikdə, idxal mükəmməl deyil; və çox uzun çəkəcəyi üçün bunu da edə bilərsiz. İlk

problem müəllif məlumatlarıdır. Subversion-da, hər bir iş görən şəxsin sistemdə commit m əlumatında qeyd olunan bir istifadəçisi var. Əvvəlki hissədəki nümunələr, `blame` çıxışı və `git svn log` kimi bəzi yerlərdə `schacon` göstərir. Bunu daha yaxşı Git müəllif məlumatlarına uyğunlaşdırmaq istəyirsinizsə, Subversion istifadəçilərindən Git müəlliflərinə qədər bir xəritəyə ehtiyacınız var. Bu istifadəçi şəklində belə bir formatda olan `users.txt` adlı bir fayl yaradın:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

SVN-nin istifadə etdiyi müəllif adlarının siyahısını əldə etmək üçün bunu edə bilərsiniz:

```
$ svn log --xml --quiet | grep author | sort -u | \
  perl -pe 's/.*>(.*?)<.*/$1 = /'
```

Gündəlik çıxışı XML formatında yaradır, sonra yalnız müəllif məlumatları olan sətirləri saxlayır, dublikatları atır, XML etiketlərini silir. Aydındır ki, bu yalnız `grep`, `sort` və `perl` quraşdırılmış bir maşın üzərində işləyir. Sonra, hər bir girişin yanına ekvivalent Git istifadəçi məlumatlarını əlavə edə bilmək üçün bu çıxışı `users.txt` faylınıza yönləndirin.

Bunu bir Windows maşınında sınayırsınızsa, problemlə qarşılaşacağınız nöqtə budur. Microsoft,https://docs.microsoft.com/en-us/azure/devops/repos/git/perform-migration-from-svn-to-git[] saytında bəzi yaxşı məsləhətlər və nümunələr verir.

Müəllif məlumatlarını daha dəqiq bir şəkildə göstərməsinə kömək etmək üçün bu faylı `git svn`-ə t əqdim edə bilərsiniz. Ayrıca,`git svn`-ə Subversion-un normal olaraq idxal etdiyi meta məlumatları daxil etməməsini, `--no-metadata`-nı `clone` ya da `init` əmrinə keçirməklə də deyə bilərsiniz. Metadata, Gitin idxal zamanı meydana gətirəcəyi hər bir mesajın içərisində bir `git-svn-id` var. Bu, Git log-unuzu şişirə bilər və bir az anlaşılmaz edə bilər.

Git deposundakı commit-ləri orijinal SVN deposuna qaytarmaq istədiyiniz zaman metadatanı saxlamalısınız. Sinxronizasiyanı commit log-da istəmirsinizsə, `--no-metadata` parametrini əlavə etməyə çəkinməyin.

Bu, `import` əmrinizi belə göstərir:

```
$ git svn clone http://my-project.googlecode.com/svn/ \
      --authors-file=users.txt --no-metadata --prefix "" -s my_project
$ cd my_project
```

İndi `my_project` qovluğunuzda daha yaxşı bir Subversion idxal etməlisiniz. Buna bənzər commit-lər yerinə:

```
commit 37efa680e8473b615de980fa935944215428a35a
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk

    git-svn-id: https://my-project.googlecode.com/svn/trunk@94 4c93b258-373f-11de-
    be05-5f7a86268029
```

they look like this:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk
```

Müəllif sahəsi həm daha yaxşı görünür, həm də `git-svn-id` artıq yoxdur. Ayrıca bir az post-import sonrası təmizləmə etməlisiniz. Birincisi, `git svn` qurduğu qəribə referansları təmizləməlisiniz. Ə vvəlcə yazıları qəribə uzaq branch-lardan daha çox həqiqi etiket olduqları üçün daşıyacaqsınız, sonra qalan branch-ları local olduqları üçün köçürəcəksiniz.

Etiketləri uyğun Git etiketi olmaq üçün hərəkət etdirin:

```
$ for t in $(git for-each-ref --format='%(refname:short)' refs/remotes/tags); do git
tag ${t/tags\//} $t && git branch -D -r $t; done
```

Bu, `refs/remotes/tags/` ilə başlayan uzaq branch-lar olan referansları alır və onları həqiqi (yüngül) etiketlər edir. Ardından, referansların qalan hissəsini local branch-lar halına gətirmək üçün `refs/remotes` altına köçürün:

```
$ for b in $(git for-each-ref --format='%(refname:short)' refs/remotes); do git branch
$b refs/remotes/$b && git branch -D -r $b; done
```

Elə ola bilər ki, `@xxx` (harada ki xxx ədəddir) şəkilçisi olan bəzi əlavə branch-lar görərsiniz, Subversionda isə yalnız bir branch görürsünüz. Bu, əslində Git üçün heç bir sintaktik həmkarı olmayan bir şey olan “peg-revisions” adlı Subversion xüsusiyyətidir. Beləliklə, `git svn` branch-ın düzəldilməsini həll etmək üçün svn-də yazdığınız şəkildə svn versiya nömrəsini branch-ın adına əlavə edir. Artıq peg-revisions düzəlişləri ilə maraqlanmırsınızsa, sadəcə onları silin:

```
$ for p in $(git for-each-ref --format='%(refname:short)' | grep @); do git branch -D
$p; done
```

İndi bütün köhnə branch-lar əsl Git branch-larıdır və bütün köhnə etiketlər əsl Git etiketləridir. T

əmizləmək üçün son bir şey var. Təəssüf ki, `git svn`, Subversionun standart branch-ına uyğunlaşan `trunk` adlı əlavə bir branch yaradır, lakin `trunk` `master` ilə eyni yerə işarə edir. `master` daha idiomatik olaraq Git olduğundan əlavə branch-ı necə silmək olar:

```
$ git branch -d trunk
```

Ediləcək son şey yeni Git serverinizi bir remote olaraq əlavə edib ona push etməkdir. Serverinizi məsafədən əlavə etmək üçün bir nümunə:

```
$ git remote add origin git@my-git-server:myrepository.git
```

Bütün branch-larınızın və etiketlərinizin yuxarı qalxmasını istədiyiniz üçün indi bunu edə bil ərsiniz:

```
$ git push origin --all
$ git push origin --tags
```

Bütün branch-larınız və etiketləriniz təmiz bir idxalda yeni Git serverinizdə olmalıdır.

Mercurial

Mercurial və Git versiyalarını təmsil etmək üçün kifayət qədər oxşar modellərə sahib olduğundan və Git bir az daha çevik olduğundan, bir deponu Mercurial-dan Git-ə çevirmək kifayət qədər sad ədir, bunun üçün "hg-fast-export" adlı bir vasitə istifadə etmək lazımdır:

```
$ git clone https://github.com/frej/fast-export.git
```

Dönüşümün ilk addımı çevirmək istədiyiniz Mercurial deposunun tam bir klonunu əldə etməkdir:

```
$ hg clone <remote repo URL> /tmp/hg-repo
```

Növbəti addım bir müəllif mapping faylı yaratmaqdır. Mercurial, dəyişikliklər üçün müəllif sahə sinə qoyacaqlarına görə Gitdən biraz daha bağışlayır, buna görə də ev təmizləmək üçün yaxşı vaxtdır. Bunu yaratmaq, `bash` shell-ində bir sətir əmridir:

```
$ cd /tmp/hg-repo
$ hg log | grep user: | sort | uniq | sed 's/user: *//' > ../authors
```

Bu, layihənizin tarixinin nə qədər olacağına görə bir neçə saniyə çəkəcək və bundan sonra `/tmp/authors` faylı belə görünəcək:

```
bob
bob@localhost
bob <bob@company.com>
bob jones <bob <AT> company <DOT> com>
Bob Jones <bob@company.com>
Joe Smith <joe@company.com>
```

Bu nümunədə, eyni şəxs (Bob) biri fərqli görünən və biri Git commiti üçün tamamilə etibarsız olan dörd fərqli ad altında dəyişikliklər yaratdı. Hg-fast-export hər sətri bir qaydaya çevirərək bunu düzəltməyimizə imkan verir: `"<input>"="<output>"`, `<input>` ilə `<output>` arasında mapping. `<input>` və `<output>` string-lərinin içərisində, python `string_escape` kodlaması ilə başa düşülən bütün qaçış ardıcıllığı dəstəklənir. Müəllif mapping faylında uyğun bir `<input>` yoxdursa, o müəllif dəyişdirilm ədən Git-ə göndəriləcək. Bütün istifadəçi adları yaxşı görünürsə, bu fayla heç ehtiyacımız olmayacaq. Bu nümunədə faylımızın belə görünməsini istəyirik:

```
"bob"="Bob Jones <bob@company.com>"
"bob@localhost"="Bob Jones <bob@company.com>"
"bob <bob@company.com>"="Bob Jones <bob@company.com>"
"bob jones <bob <AT> company <DOT> com>"="Bob Jones <bob@company.com>"
```

Eyni növ mapping faylı, Mercurial adının Git tərəfindən icazə verilmədiyi zaman branch və etiketl ərin adını dəyişdirmək üçün istifadə edilə bilər.

Növbəti addım yeni Git depomuzu yaratmaq və ixrac skriptini çalışdırmaqdır:

```
$ git init /tmp/converted
$ cd /tmp/converted
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
```

`-r` flag-ı hg-fast-export-ə çevirmək istədiyimiz Mercurial deposunu harada tapacağımızı və `-A` flag-ı isə author-mapping faylını harada tapacağımızı bildirir (branch və etiket mapping faylları müvafiq olaraq `-B` və `-T` flag-ları ilə təyin olunur). Skript Mercurial dəyişikliklərini təhlil edir və Git'in "fast-import" xüsusiyyəti üçün bir skriptə çevirir (bir az sonra ətraflı şəkildə müzakirə edəcəyik).

Bu bir az çəkir (baxmayaraq ki, şəbəkə üzərində olandan *daha çox* sürətli olur) və nəticə olduqca açıqdır:

```
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
Loaded 4 authors
master: Exporting full revision 1/22208 with 13/0/0 added/changed/removed files
master: Exporting simple delta revision 2/22208 with 1/1/0 added/changed/removed files
master: Exporting simple delta revision 3/22208 with 0/1/0 added/changed/removed files
[…]
master: Exporting simple delta revision 22206/22208 with 0/4/0 added/changed/removed
files
master: Exporting simple delta revision 22207/22208 with 0/2/0 added/changed/removed
files
master: Exporting thorough delta revision 22208/22208 with 3/213/0
added/changed/removed files
Exporting tag [0.4c] at [hg r9] [git :10]
Exporting tag [0.4d] at [hg r16] [git :17]
[…]
Exporting tag [3.1-rc] at [hg r21926] [git :21927]
Exporting tag [3.1] at [hg r21973] [git :21974]
Issued 22315 commands
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:     120000
Total objects:       115032 (    208171 duplicates                  )
      blobs  :        40504 (    205320 duplicates      26117 deltas of      39602
attempts)
      trees  :        52320 (      2851 duplicates      47467 deltas of      47599
attempts)
      commits:        22208 (         0 duplicates          0 deltas of          0
attempts)
      tags   :            0 (         0 duplicates          0 deltas of          0
attempts)
Total branches:         109 (         2 loads     )
      marks:        1048576 (     22208 unique    )
      atoms:           1952
Memory total:          7860 KiB
       pools:          2235 KiB
     objects:          5625 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit      = 8589934592
pack_report: pack_used_ctr            =      90430
pack_report: pack_mmap_calls          =      46771
pack_report: pack_open_windows        =          1 /          1
pack_report: pack_mapped              =  340852700 /  340852700
---------------------------------------------------------------------

$ git shortlog -sn
   369  Bob Jones
   365  Joe Smith
```

Bunun üçün demək olar ki, hamısı var. Mercurial etiketlərin hamısı Git etiketlərinə, Mercurial branch-ları və bookmark-ları Git branch-larına çevrilmişdir. İndi deponu yeni server tərəfindəki evinə aparmağa hazırsınız:

```
$ git remote add origin git@my-git-server:myrepository.git
$ git push origin --all
```

Bazaar

Bazaar, Git kimi bir DVCS vasitədir və nəticədə bir Bazaar deposunu Git-ə çevirmək olduqca sad ədir. Bunu həyata keçirmək üçün `bzr-fastimport` pluginini idxal etməlisiniz.

bzr-fastimport pluginn alınması

Fastimport pluginin quraşdırılması proseduru UNIX kimi əməliyyat sistemlərində və Windows-da fərqlidir. Birinci halda, ən sadə, lazım olan bütün asılılıqları quracaq olan `bzr-fastimport` paketinin quraşdırılmasıdır.

Məsələn, Debian və törəməsi ilə aşağıdakıları edərdiniz:

```
$ sudo apt-get install bzr-fastimport
```

RHEL ilə aşağıdakıları edərdiniz:

```
$ sudo yum install bzr-fastimport
```

Fedora ilə, 22 buraxılışından bəri yeni paket meneceri dnf:

```
$ sudo dnf install bzr-fastimport
```

Paket mövcud deyilsə, onu bir plugin kimi qura bilərsiniz:

```
$ mkdir --parents ~/.bazaar/plugins     # creates the necessary folders for the
plugins
$ cd ~/.bazaar/plugins
$ bzr branch lp:bzr-fastimport fastimport   # imports the fastimport plugin
$ cd fastimport
$ sudo python setup.py install --record=files.txt   # installs the plugin
```

Bu pluginin işləməsi üçün `fastimport` Python moduluna ehtiyacınız olacaq. Mövcud olub olmadığını yoxlayıb aşağıdakı əmrlərlə quraşdıra bilərsiniz:

```
$ python -c "import fastimport"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named fastimport
$ pip install fastimport
```

Mövcud deyilsə, https://pypi.python.org/pypi/fastimport/ ünvanından yükləyə bilərsiniz.

İkinci halda (Windows-da) `bzr-fastimport` avtomatik olaraq bağımsız versiya və standart quraşdırma ilə quraşdırılır (bütün checkbox-ların işarələnməsinə icazə verin). Yəni bu vəziyyətdə sizin heç bir əlaqəniz yoxdur.

Bu nöqtədə, Bazaar deposun idxal etmək yolu tək bir branch-ınız olduğuna və ya bir neçə branch-ı olan bir depo ilə işlədiyinizə görə fərqlənir.

Tək bir branch ilə layihə

İndi Bazaar deponuz olan qovluqda `cd` və Git deposunu işə salın:

```
$ cd /path/to/the/bzr/repository
$ git init
```

İndi Bazaar deponuzu sadəcə ixrac edə və aşağıdakı əmrdən istifadə edərək Git deposuna çevirə bilərsiniz:

```
$ bzr fast-export --plain . | git fast-import
```

Layihənin ölçüsündən asılı olaraq, Git deponuz bir neçə saniyədən bir neçə dəqiqəyə qədər sürətlə qurulur.

Əsas branch və işləyən bir branch olan bir layihə işi

Branch-ları olan bir Bazaar deposunu da idxal edə bilərsiniz. İki branch-nızın olduğunu düşünək: biri əsas branch-ı təmsil edir (myProject.trunk), digəri işləyən branch-dır (myProject.work).

```
$ ls
myProject.trunk myProject.work
```

Git deposunu və `cd`-ni yaradın:

```
$ git init git-repo
$ cd git-repo
```

`master` branch-ını git-ə pull edin:

```
$ bzr fast-export --export-marks=../marks.bzr ../myProject.trunk | \
git fast-import --export-marks=../marks.git
```

İşləyən branch-ı Git-ə pull edin:

```
$ bzr fast-export --marks=../marks.bzr --git-branch=work ../myProject.work | \
git fast-import --import-marks=../marks.git --export-marks=../marks.git
```

İndi `git branch` sizə `work` branch-ını olduğu kimi `master` branch-ını da göstərir. Tam olduğundan əmin olmaq üçün qeydləri yoxlayın və `marks.bzr` və `marks.git` fayllarından qurtulun.

Səhnələşdirmə sahəsinin sinxronlaşdırılması

Sahib olduğunuz branch sayınız və istifadə etdiyiniz idxal metodundan asılı olmayaraq səhnəl əşdirmə sahəniz `HEAD` ilə sinxronizasiya edilmir və bir neçə branch-ın idxalı ilə iş qovluğunuz da sinxronizasiya edilmir. Bu vəziyyət asanlıqla aşağıdakı əmrlə həll olunur:

```
$ git reset --hard HEAD
```

.bzrignore ilə ignoring edilmiş faylları ignore etmək

İndi ignore etmək üçün fayllara nəzər salaq. Ediləcək ilk şey, `.bzrignore` adını `.gitignore` olaraq d əyişdirməkdir. `.bzrignore` faylı "!!" ilə və ya "RE:" başlayan bir və ya bir neçə sətir varsa, onu d əyişdirməlisiniz və Bazaarın ignore etdiyi faylları ignore etmək üçün bəlkə də bir neçə `.gitignore` faylı yaratmalısınız.

Nəhayət, migrasiya üçün bu dəyişikliyi ehtiva edən bir commit yaratmalısınız:

```
$ git mv .bzrignore .gitignore
$ # modify .gitignore if needed
$ git commit -am 'Migration from Bazaar to Git'
```

Deponuzu serverə göndərmək

Budur! İndi deponu yeni ev serverinə push edə bilərsiniz:

```
$ git remote add origin git@my-git-server:mygitrepository.git
$ git push origin --all
$ git push origin --tags
```

Git deponuz istifadəyə hazırdır.

Perforce

İdxal etməyə baxacağınız növbəti sistem Perforce'dir. Yuxarıda müzakirə etdiyimiz kimi, Git və Perforce-un bir-birlərinə danışmalarına icazə verməyin iki yolu var: git-p4 və Perforce Git Fusion.

Perforce Git Fusion

Git Fusion bu prosesi kifayət qədər ağrısız edir. Bir konfiqurasiya sənədindən istifadə edərək layihə parametrlərinizi, istifadəçi xəritələrinizi və branch-larınızı konfiqurasiya etməyiniz kifayətdir (Git Fusion-da müzakirə olunduğu kimi) və deponu klonlayın. Git Fusion, yerli bir Git deposuna bənzər bir şey buraxır, daha sonra istəsəniz local bir Git hostuna push etməyə hazırdır. İstəsəniz, Perforce'ı Git ev sahibi kimi istifadə edə bilərsiniz.

Git-p4

Git-p4 idxal vasitəsi kimi də çıxış edə bilər. Nümunə olaraq Jam layihəsini Perforce Public Depot-dan idxal edəcəyik. Müştərinizi qurmaq üçün, Perforce deposuna işarə etmək üçün P4PORT mühit dəyişkənini ixrac etməlisiniz:

```
$ export P4PORT=public.perforce.com:1666
```

Ardından izləmək üçün əlaqə qurmaq üçün bir Perforce deposuna ehtiyacınız var. Nümunələrimiz üçün public.perforce.com saytındakı ictimai depodan istifadə ed əcəyik, ancaq daxil olduğunuz hər hansı bir depodan istifadə edə bilərsiniz.

Depo və layihə yolunu və layihəni idxal etmək istədiyiniz yolu təmin edərək Perforce serverindən Jam layihəsini idxal etmək üçün `git p4 clone` əmrini işə salın:

```
$ git-p4 clone //guest/perforce_software/jam@all p4import
Importing from //guest/perforce_software/jam@all into p4import
Initialized empty Git repository in /private/tmp/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 9957 (100%)
```

Bu xüsusi layihənin yalnız bir branch-ı var, ancaq branch görünüşləri ilə (və ya yalnız bir qovluq yığını) konfiqurasiya edilmiş branch-larınız varsa, bütün layihə branch-larını da idxal etmək üçün `--detect-branches` flagını `git p4 clone` üçün istifadə edə bilərsiniz. Bu barədə bir az daha ətraflı m əlumat üçün Branching bölməsinə baxın.

Bu nöqtədə demək olar ki, bitirdiniz. `p4import` qovluğuna gedib `git log`-ı işlətsəniz, idxal etdiyiniz işi görə bilərsiniz:

```
$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800

    Correction to line 355; change </UL> to </OL>.

    [git-p4: depot-paths = "//public/jam/src/": change = 8068]

commit aa21359a0a135dda85c50a7f7cf249e4f7b8fd98
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).

    [git-p4: depot-paths = "//public/jam/src/": change = 7304]
```

Hər git mesajında `git-p4` bir identifikator buraxdığını görə bilərsiniz. Perforce dəyişiklik nömrə sinə daha sonra müraciət etməyiniz lazım olduğu təqdirdə bu identifikatoru orada saxlamaq yaxşıdır. Bununla birlikdə, identifikatoru silmək istəsəniz, indi yeni depoda işə başlamazdan əvvəl bunu etməliyik. Tanımlayıcı sətirləri kütləvi şəkildə silmək üçün `git filter-branch` istifadə edə bilərsiniz:

```
$ git filter-branch --msg-filter 'sed -e "/^\[git-p4:/d"'
Rewrite e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
Ref 'refs/heads/master' was rewritten
```

`git log`-ı işə salırsınızsa, commit-lər üçün bütün SHA-1 hesablama cəmlərinin dəyişdiyini görə bil ərsiniz, lakin `git-p4` string-ləri artıq commit mesajlarında yoxdur:

```
$ git log -2
commit b17341801ed838d97f7800a54a6f9b95750839b7
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800

    Correction to line 355; change </UL> to </OL>.

commit 3e68c2e26cd89cb983eb52c024ecdfba1d6b3fff
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).
```

İdxalınız yeni Git serverinizə keçməyə hazırdır.

TFS

Komandanız mənbə nəzarətini TFVC-dən Git-ə çevirirsə, əldə edə biləcəyiniz ən yüksək əslinə uyğun dönüşümünü istəyərsiniz. Bu o deməkdir ki, interop bölməsi üçün həm git-tfs, həm də git-tf-ni əhatə etdiyimiz halda, yalnız bu hissə üçün git-tfs-i əhatə edəcəyik, çünki git-tfs branch-ları dəst əkləyir və bu git-tf istifadə etmək olduqca çətindir.

Bu, birtərəfli dönüşümdür. Nəticədə Git deposu orijinal TFVC layihəsi ilə əlaqə qura bilməyəcək.

Ediləcək ilk şey istifadəçi adlarının xəritəsidir. TFVC, dəyişikliklər üçün müəllif sahəsinə daxil olanlarla kifayət qədər sərbəstdir, lakin Git insan tərəfindən oxunaqlı bir ad və e-poçt ünvanı ist əyir. Bu məlumatı `tf` əmr sətrini müştərisindən ala bilərsiniz:

```
PS> tf history $/myproject -recursive > AUTHORS_TMP
```

Bu, layihənin tarixindəki bütün dəyişiklikləri tutur və *İstifadəçi* sütununun (ikincisi) məlumatlarını çıxarmaq üçün işləyəcəyimiz AUTHORS_TMP sənədinə qoyuruq. Faylı açın və sütunun hansı işarə də başlayıb bitdiyini və yerini dəyişdirin, aşağıdakı əmr sətrində `cut` əmrini `11-20` parametrlərini tapılanlarla əvəz edin:

```
PS> cat AUTHORS_TMP | cut -b 11-20 | tail -n+3 | sort | uniq > AUTHORS
```

`cut` əmri hər sətirdən yalnız 11 ilə 20 arasındakı simvolları saxlayır. `tail` əmri sahə başlıqları və ASCII-art vurğulayan ilk iki sətiri atlayır. Bütün bunların nəticəsi, təkrarlamaları aradan qaldırmaq üçün `sort` və `uniq` şəkillərinə göndərilir və `AUTHORS` adlı bir faylda saxlanılır. Növbəti addım manualdır; git-tfs-in bu fayldan səmərəli istifadə etməsi üçün hər sətir bu formatda olmalıdır:

```
DOMAIN\username = User Name <email@address.com>
```

Sol tərəfdəki hissə TFVC-dən “User” sahəsidir və bərabər işarəsinin sağ tərəfindəki hissə Git commit-ləri üçün istifadə ediləcək istifadəçi adıdır.

Bu faylı əldə etdikdən sonra bir sonrakı iş, maraqlandığınız TFVC layihəsinin tam bir klonunu yaratmaqdır:

```
PS> git tfs clone --with-branches --authors=AUTHORS
https://username.visualstudio.com/DefaultCollection $/project/Trunk project_git
```

Bundan sonra, commit mesajların altından `git-tfs-id` bölmələrini təmizləmək lazımdır. Aşağıdakı əmr bunu edəcəkdir:

```
PS> git filter-branch -f --msg-filter 'sed "s/^git-tfs-id:.*$//g"' '--' --all
```

Bu, “git-tfs-id:” ilə başlayan hər hansı bir sətri boşluqla əvəz etmək üçün Git-bash mühitindən `sed` əmrini istifadə edir və Git bundan sonra ignore qalacaq.

Hamısı bitdikdən sonra yeni bir remote əlavə etməyə, bütün branch-larınızı yuxarıya qaldırmağa və komandanızın Git-dən işə başlamasına hazırsınız.

Xüsusi İdxalçı

Sisteminiz yuxarıda göstərilənlərdən biri deyilsə, onlayn bir idxalçı axtarmalısınız - keyfiyyətli idxalçılar CVS, Clear Case, Visual Source Safe hətta arxivlər siyahısı daxil olmaqla bir çox digər sistemlər üçün mövcuddur.

Bu vasitələrdən heç biri sizin üçün işləmirsə, daha qaranlıq bir alətiniz varsa və ya başqa bir xüsusi idxal prosesinə ehtiyacınız varsa, `git fast-import` istifadə etməlisiniz. Bu əmr, müəyyən Git məlumatlarını yazmaq üçün stdin-dən sadə təlimatları oxuyur. Git obyektlərini bu şəkildə yaratmaq, xam Git əmrlərini işə salmaqdan və ya xam obyektləri yazmağa çalışmaqdan daha asandır (daha çox məlumat üçün Git’in Daxili İşləri bax). Bu şəkildə, idxal etdiyiniz sistemdən lazımi məlumatları oxuyan və stdout üçün birbaşa təlimatları yazan bir idxal skriptini yaza bil ərsiniz. Daha sonra bu proqramı işə sala və nəticəsini `git fast-import` vasitəsilə ötürə bilərsiniz.

Sürətlə nümayiş etdirmək üçün sadə bir idxalçı yazacaqsınız. Tutaq ki, `current`-də işləyirsiniz, zaman zaman qovluğu bir time-stamped `back_YYYY_MM_DD` ehtiyat qovluğuna kopyalayaraq proyektinizi backup edirsiniz və bunu Git-ə idxal etmək istəyirsiniz. Qovluq quruluşunuz belə görünür:

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03
current
```

Bir Git qovluğunu idxal etmək üçün Gitin məlumatlarını necə saxladığını nəzərdən keçirməlisiniz. Xatırladığınız kimi, Git əsas etibarilə məzmunun bir snapshot-una işarə edən obyektlərin əlaqəli siyahısıdır. Etməli olduğunuz şey məzmunun snpshot-larının nə olduğunu, məlumatları onlara yönəldən şeyləri və daxil olma qaydalarını `fast-import` etməyinizdir. Strategiyanız snapshot-lardan bir-bir keçmək və hər bir qovluğu əvvəlki ilə əlaqələndirərək hər bir kataloqun məzmunu ilə commit-lər yaratmaq olacaq.

Git-Enforced Siyasət Nümunəsi-də etdiyimiz kimi, bunu Ruby-də yazacağıq, çünki ümumiyyətlə işl əmək və oxunması asan olur. Bu nümunəni tanış olduğunuz hər hansı bir şeydə olduqca asanlıqla yaza bilərsiniz - yalnız müvafiq məlumatları `stdout`-a yazdırmaq lazımdır. Və Windows üzərində işləyirsinizsə, bu, sətirlərinizin sonunda vaqon qayıtmalarını daxil etməməyə xüsusi diqqət yetirməli olduğunuz mənasını verir - `git fast-import` yalnız Windows istifadə etdiyi daşıyıcı xətləri (CRLF) deyil, yalnız xətt lentləri (LF) istəməyə gəldikdə çox dəqiqdir.

Başlamaq üçün hədəf qovluğunu dəyişəcəksiniz və hər biri commit olaraq idxal etmək istədiyiniz bir snapshot-u olan hər subdirectory-ni müəyyənləşdirəcəksiniz. Hər subdirectory-ni keçib ixrac

etmək üçün lazım olan əmrləri çap edəcəksiniz. Əsas dövrəniz belə görünür:

```
last_mark = nil

# loop through the directories
Dir.chdir(ARGV[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end
```

Hər bir qovluğun içərisində əvvəlki snapshot-un manifestini və işarəsini götürən və bunun manifestini və işarəsini qaytaran `print_export` işlədirsiniz; bu şəkildə onları düzgün bir şəkildə bağlaya bilərsiniz. "Mark" commit-ə verdiyiniz bir identifikator üçün `fast-import` terminidir; commit-lər yaradarkən, hər birinə digər commit-lərdən əlaqələndirmək üçün istifadə edə biləc əyiniz bir işarə verirsiniz. Beləliklə, `print_export` metodunda ediləcək ilk şey qovluq adından bir işarənin yaradılmasıdır:

```
mark = convert_dir_to_mark(dir)
```

Bunu bir dizin qovluğu yaratmaqla və indeks dəyərini işarə olaraq istifadə etməklə edəcəksiniz, çünki bir işarə tam olmalıdır. Metodunuz belə görünür:

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

Commit-inizin tam bir təmsilçiliyinə sahib olduğunuz üçün, commit meta məlumatları üçün bir tarixə ehtiyacınız var. Tarix qovluq adına ifadə olunduğundan, onu təhlil edəcəksiniz. `Print_export` falınızdakı növbəti sətir:

```
date = convert_dir_to_date(dir)
```

where `convert_dir_to_date` is defined as:

```
def convert_dir_to_date(dir)
  if dir == 'current'
    return Time.now().to_i
  else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
  end
end
```

Bu, hər bir qovluğun tarixi üçün bir tam dəyər qaytarır. Hər bir commit üçün lazım olan son meta məlumat parçası, qlobal dəyişəndə sabit kodladığınız ötürmə məlumatlarıdır:

```
$author = 'John Doe <john@example.com>'
```

İndi idxalçı üçün commit məlumatlarını çap etməyə başlamağa hazırsınız. İlkin məlumatda commit obyektini və hansı branch-da olduğunuzu, sonra yaratdığınız işarəni, məlumat verən m əlumatı və commit mesajını, sonra varsa əvvəlki commit-i təyin etdiyiniz bildirilir. Kod belə görünür:

```
# print the import information
puts 'commit refs/heads/master'
puts 'mark :' + mark
puts "committer #{$author} #{date} -0700"
export_data('imported from ' + dir)
puts 'from :' + last_mark if last_mark
```

Saat dilimini (-0700) kodlaşdırın, çünki bunu etmək asandır. Başqa bir sistemdən idxal edirsinizsə, zaman zonasını ofset kimi göstərməlisiniz. Commit mesajı xüsusi bir formatda ifadə olunmalıdır:

```
data (size)\n(contents)
```

Format məlumat sözündən, oxunacaq məlumatın ölçüsündən, yeni bir sətirdən və nəhayət m əlumatdan ibarətdir. Faylın məzmununu daha sonra müəyyənləşdirmək üçün eyni formatı istifadə etməlisiniz, çünki köməkçi bir metod, `export_data` yaradırsınız:

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

Qalan yalnız hər bir snapshot üçün fayl məzmununun müəyyənləşdirilməsidir. Bu çox asandır, çünki hər biriniz bir kataloqdadır - qovluqdakı hər bir sənədin məzmunu və ardından `deleteall` əmrini çap edə bilərsiniz. Daha sonra Git hər bir snapshot-u uyğun şəkildə qeyd edəcək:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Qeyd: Bir çox sistem öz reviziyalarını bir commit-dən digərinə dəyişiklik kimi qəbul etdiyini düşündüyündən, sürətli idxal da hər bir faylda hansı faylların əlavə olunduğunu, silindiyini və ya dəyişdirildiyini və yeni məzmununun nə olduğunu müəyyənləşdirmək üçün əmrlər götürə bilər. Snapshot-lar arasındakı fərqləri hesablaya bilər və yalnız bu məlumatları verə bilərsiniz, ancaq bunu etmək daha mürəkkəbdir - Git-ə bütün məlumatları verə və onu başa düşməsinə icazə verə bilərsiniz. Bu, məlumatlarınıza daha uyğun gəlirsə, məlumatlarınızı bu şəkildə necə təqdim edəc əyiniz barədə ətraflı məlumat üçün `fast-import` man page-ini yoxlayın.

Yeni fayl məzmununun siyahısı və ya dəyişdirilmiş bir sənədin yeni məzmunu ilə göstərilməsi formatı belədir:

```
M 644 inline path/to/file
data (size)
(file contents)
```

Budur, 644 rejimi (executable fyllarınız varsa, bunun əvəzinə 755-i aşkarlamalı və təyin etm əlisiniz) və sətir içərisindəki məzmunu bu sətirdən dərhal sonra sadalayacağınızı söyləyir. `inline_data` metodunuz belə görünür:

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

Daha əvvəl müəyyənləşdirdiyiniz `export_data` metodundan təkrar istifadə edirsiniz, çünki bu, commit mesajı məlumatlarınızı təyin etməyinizlə eynidir.

Etməli olduğunuz son şey cari işarəni növbəti təkrarlamaya ötürmək üçün qaytarmaqdır:

```
return mark
```

Windows-da işləyirsinizsə, əlavə bir addım əlavə etdiyinizə əmin olmalısınız. Daha əvvəl də qeyd edildiyi kimi, Windows yeni sətir simvolları üçün CRLF istifadə edir, `git fast-import` isə yalnız LF gözləyir. Bu problemi həll etmək və `git fast-import`-ı xoşbəxt etmək üçün ruby-ə CRLF əvəzinə LF istifadə etməsini söylə məlisiniz:

```
$stdout.binmode
```

Bu qədər. Budur skript bütövlükdə:

```
#!/usr/bin/env ruby

$stdout.binmode
$author = "John Doe <john@example.com>"

$marks = []
def convert_dir_to_mark(dir)
    if !$marks.include?(dir)
        $marks << dir
    end
    ($marks.index(dir)+1).to_s
end

def convert_dir_to_date(dir)
    if dir == 'current'
        return Time.now().to_i
    else
        dir = dir.gsub('back_', '')
        (year, month, day) = dir.split('_')
        return Time.local(year, month, day).to_i
    end
end

def export_data(string)
    print "data #{string.size}\n#{string}"
end

def inline_data(file, code='M', mode='644')
    content = File.read(file)
    puts "#{code} #{mode} inline #{file}"
    export_data(content)
end

def print_export(dir, last_mark)
    date = convert_dir_to_date(dir)
    mark = convert_dir_to_mark(dir)

    puts 'commit refs/heads/master'
```

```
    puts "mark :#{mark}"
    puts "committer #{$author} #{date} -0700"
    export_data("imported from #{dir}")
    puts "from :#{last_mark}" if last_mark

    puts 'deleteall'
    Dir.glob("**/*").each do |file|
        next if !File.file?(file)
        inline_data(file)
    end
    mark
end

# Loop through the directories
last_mark = nil
Dir.chdir(ARGV[0]) do
    Dir.glob("*").each do |dir|
        next if File.file?(dir)

        # move into the target directory
        Dir.chdir(dir) do
            last_mark = print_export(dir, last_mark)
        end
    end
end
```

Bu skripti işlədirsinizsə, buna bənzər bir məzmun əldə edəcəksiniz:

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer John Doe <john@example.com> 1388649600 -0700
data 29
imported from back_2014_01_02deleteall
M 644 inline README.md
data 28
# Hello

This is my readme.
commit refs/heads/master
mark :2
committer John Doe <john@example.com> 1388822400 -0700
data 29
imported from back_2014_01_04from :1
deleteall
M 644 inline main.rb
data 34
#!/bin/env ruby

puts "Hey there"
M 644 inline README.md
(...)
```

İmporter-i çalıştırmak üçün bu çıxışı idxal etmək istədiyiniz Git qovluğunda olarkən `git fast-import` vasitəsilə ötürün. Yeni bir qovluq yaradıb sonra bir başlanğıc nöqtəsi üçün içərisindəki `git init` -i işə sala və sonra skriptinizi işə sala bilərsiniz:

```
$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:       5000
Total objects:           13 (         6 duplicates                  )
      blobs  :            5 (         4 duplicates          3 deltas of          5
attempts)
      trees  :            4 (         1 duplicates          0 deltas of          4
attempts)
      commits:            4 (         1 duplicates          0 deltas of          0
attempts)
      tags   :            0 (         0 duplicates          0 deltas of          0
attempts)
Total branches:           1 (         1 loads     )
      marks:           1024 (         5 unique    )
      atoms:              2
Memory total:          2344 KiB
       pools:          2110 KiB
     objects:           234 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit      = 8589934592
pack_report: pack_used_ctr            =         10
pack_report: pack_mmap_calls          =          5
pack_report: pack_open_windows        =          2 /          2
pack_report: pack_mapped              =       1457 /       1457
---------------------------------------------------------------------
```

Gördüyünüz kimi uğurla başa çatdıqda, həyata keçirdikləri barədə bir çox statistika verir. Bu v əziyyətdə, 1 branch-a 4 commit üçün cəmi 13 obyekt idxal etdiniz. İndi yeni tarixçənizi görmək üçün `git log`-u işə sala bilərsiniz:

```
$ git log -2
commit 3caa046d4aac682a55867132ccdfbe0d3fdee498
Author: John Doe <john@example.com>
Date:   Tue Jul 29 19:39:04 2014 -0700

    imported from current

commit 4afc2b945d0d3c8cd00556fbe2e8224569dc9def
Author: John Doe <john@example.com>
Date:   Mon Feb 3 01:00:00 2014 -0700

    imported from back_2014_02_03
```

Budur - gözəl, təmiz bir Git deposu. Heç bir şeyin yoxlanılmadığını qeyd etmək vacibdir - əvvəlcə iş qovluğunda heç bir faylınız yoxdur. Bunları əldə etmək üçün branchnızı `master`-i n olduğu yerə sıfırlamalısınız:

```
$ ls
$ git reset --hard master
HEAD is now at 3caa046 imported from current
$ ls
README.md main.rb
```

`fast-import` aləti ilə daha çox şey edə bilərsiniz - fərqli rejimləri, ikili məlumatları, birdən çox branch-ı və birləşmə, etiketlər, irəliləyiş göstəriciləri və s. Daha mürəkkəb ssenarilərin bir sıra nümunələri Git qaynaq kodunun `contrib/fast-import` qovluğunda mövcuddur.

Qısa Məzmun

Git'i digər versiya nəzarət sistemləri üçün bir müştəri olaraq istifadə etmək və ya demək olar ki, mövcud bir deposu məlumat itirmədən Git'ə idxal etmək üçün rahat hiss etməlisiniz. Növbəti fə sildə, Git'in xam daxili hissələrini əhatə edəcəyik, buna görə ehtiyac duyduğunuz hər bir baytı hazırlaya bilərsiniz.

Git’in Daxili İşləri

Bu fəslə daha əvvəlki bir fəsildən keçib gəlmiş ola bilərsiz və ya bütün kitabı bu vaxta qədər ardıcıl oxuduqdan sonra buraya gəlmiş ola bilərsiz - hər iki halda da Git’in daxili işlərini və tətbiqini nəz ərdən keçirəcəyik. Bu məlumatları başa düşməyin Git’in nə qədər faydalı və güclü olduğunu qiym ətləndirmək üçün nə qədər vacib olduğunu gördük, lakin başqaları bunun yeni başlayanlar üçün qarışıq və lazımsız dərəcədə mürəkkəb ola biləcəyini iddia edirlər. Beləliklə, bu müzakirəni kitabdakı son fəsildə etdik ki, öyrənmə müddətində erkən və ya sonra oxuya biləsiniz. Qərar verm əyi sizin ixtiyarınıza veririk.

İndi isə buradasınız, başlayaq. Birincisi, hələ aydın deyilsə, Git, üstündə bir VNS istifadəçi interfeysi ilə məzmunla əlaqəli bir fayl sistemidir. Bunun nə demək olduğunu bir azdan öyrənəcəksiniz.

Git’in ilk günlərində (əksərən əvvəl 1.5) istifadəçi interfeysi cilalanmış bir VNS əvəzinə bu fayl sistemini vurğuladığı üçün daha mürəkkəb idi. Son bir neçə ildə, istifadəçi interfeysi, orada olan h ər hansı bir sistem qədər təmiz və istifadəsi asan olana qədər təmizləndi; Bununla birlikdə, stereotip mürəkkəb və öyrənilməsi çətin olan ilk Git UI ilə əlaqədardır.

Məzmunun ünvanlandığı fayl sistemi təbəqəsi qəribə dərəcədə cool-dur, buna görə əvvəlcə bu f əsildə əhatə edəcəyik; sonra nəqliyyat mexanizmləri və nəticədə həll edə biləcəyiniz deponun saxlanılması tapşırıqlarını öyrənəcəksiniz.

Plumbing və Porcelain

Bu kitab, ilk növbədə Git’i 30-a yaxın `checkout`, `branch`, `remote` və s. kimi alt komanda ilə necə istifadə edəcəyimizi əhatə edir. Ancaq Git əvvəlcə tam user-friendly bir VNS əvəzinə versiya nəzarət sistemi üçün bir vasitə dəsti olduğundan, aşağı səviyyəli iş görən və UNIX tərzində zəncirlənmək və ya skriptlərdən çağırılmaq üçün hazırlanmış bir sıra alt komanda var. Bu əmrlərə ümumiyyətlə Git-in “plumbing” əmrləri deyilir, daha çox user-friendly əmrlərə isə “porcelain” əmrləri deyilir.

İndiyə qədər fərq etdiyiniz kimi bu kitabın ilk doqquz fəsli demək olar ki, yalnız porcelain əmrləri ilə əlaqədardır. Ancaq bu fəsildə əsasən aşağı səviyyəli plumbing əmrləri ilə məşğul olacaqsınız, çünki bunlar sizə Gitin daxili işlərinə giriş imkanı verir və Git-in necə və niyə etdiyini göstərməyə kömək edir. Bu əmrlərin əksəriyyəti əmr sətrində manual olaraq istifadə olunmaq üçün deyil, daha çox yeni alətlər və xüsusi skriptlər üçün bloklar kimi istifadə olunur.

Yeni və ya mövcud bir qovluqda `git init` işlətdiyinizdə Git Gitin saxladığı və idarə etdiyi hər şeyin yerləşdiyi `.git` qovluğunu yaradır. Deponuzun back up-nı çıxarmaq və ya klonlamaq istəyirsinizsə, bu tək qovluğu başqa yerə kopyalamaq sizə lazım olan hər şeyi verir. Bu fəslin hamısı, əsasən bu qovluqda görə biləcəyiniz şeylərdən bəhs edir. Yeni başlatılmış `.git` qovluğunun adətən belə görünür:

```
$ ls -F1
config
description
HEAD
hooks/
info/
objects/
refs/
```

Git versiyanızdan asılı olaraq orada bir neçə əlavə məzmun görə bilərsiniz, ancaq bu təzə bir `git init` deposudur - varsayılan olaraq gördüyünüz budur. `Description` faylı yalnız GitWeb proqramı t ərəfindən istifadə olunur, buna görə narahat olmayın. `Config` faylı proyektə xüsusi konfiqurasiya seçimlərinizi ehtiva edir və `info` qovluğu ``.gitignore` faylında izləmək istəmədiyiniz ignored pattern-lər üçün qlobal bir xaric sənədini (excludes saxlayır. `hooks` qovluğu, Git Hook'ları bölm əsində ətraflı müzakirə olunan müştəri və ya server tərəfindəki hook skriptlərinizi ehtiva edir.

Bu dörd vacib giriş buraxır: `HEAD` və (hələ yaradılmayacaq) `index` faylları və `object` və `refs` qovluqları. Bunlar Gitin əsas hissələridir. `Object` qovluğu verilənlər bazanızdakı bütün məzmunu, `refs` qovluğu göstəriciləri həmin məlumatdakı obyektləri (branch-lar, etiketlər, remote-lar və daha çox) saxlayır, `HEAD` faylı hazırda yoxladığınız branch-ı göstərir və `index` faylı Git-in səhnələşdirmə sahəsi məlumatlarınızı saxladığı yerdir. İndi Git'in necə işlədiyini görmək üçün bu hissələrin hər birinə ətraflı baxacaqsınız.

Git Obyektləri

Git, məzmuna yönəldilə bilən bir sistemdir. Əla. Bəs bunun mənası nədir? Bu, Git-in mərkəzində sadə bir əsas dəyər məlumat deposu olduğu deməkdir. Bunun mənası budur ki, Git deposuna hər cür məzmun daxil edə bilərsiniz, bunun üçün Git bu məzmunu almaq üçün daha sonra istifadə edə biləcəyiniz unikal bir key-i geri qaytaracaqdır.

Nümayiş olaraq, bəzi məlumatları götürən, `.git/objects` qovluğunda (*object database*) saxlayan və indi buna istinad edən unikal key-i verən `git hash-object` plumbing əmrinə baxaq.

Əvvəlcə yeni bir Git deposunu işə salırsınız və `objects` qovluğunda (əvvəlcədən) heç bir şey olmadığını yoxlayırsınız:

```
$ git init test
Initialized empty Git repository in /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
```

Git, `objects` qovluğunu işə saldı və içərisində `pack` və `info` alt qovluqlarını yaratdı, lakin orada normal fayllar yoxdur. İndi yeni bir məlumat obyekti yaratmaq və onu yeni Git verilənlər

bazasında manual şəkildə saxlamaq üçün `git hash-object`-dən istifadə edək:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

Ən sadə şəkildə, `git hash-object` verdiyiniz məzmunu alacaq və *would*-nin Git verilənlər bazasında saxlamaq üçün istifadə etdiyi unikal key-i qaytaracaqdı. Daha sonra `-w` seçimi əmrə key-i sadəcə qaytarmağı deyil, həmin obyekti verilənlər bazasına yazmağı əmr edir. Nəhayət, `--stdin` seçimi, `git hash-object`-ə məzmunun stdin-dən işlənməsinin alınmasını deyir; Əks təqdirdə, əmr, məzmunun daxil olduğu əmrin sonunda bir fayl adı arqumentinin istifadə olunacağını gözləyir.

Yuxarıda göstərilən əmrdən çıxan nəticə 40 simvollu bir cəmi hash-dir. Bu SHA-1 hash-ı - saxladığınız məzmunun cəmi və üstəlik, bir azdan öyrənəcəyiniz bir başlıqdır. İndi Git-in m əlumatlarınızı necə saxladığını görə bilərsiniz:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Əgər siz yenidən `objects` qovluğunu araşdırırsınızsa, indi o yeni məzmun üçün özündə bir fayl daxil etdiyini görə bilərsiniz. Git əvvəlcə məzmunu onun SHA-1 nəzarət cəmi və onun başlığı ilə adlandırılan bir məzmuna görə tək bir fayl kimi saxlayır. Alt qovluq SHA-1-in ilk 2 simvolu ilə adlanır və fayl adı qalan 38 simvoldur.

Verilənlər bazanızda bir məzmun olduqdan sonra, bu məzmunu `git cat-file` əmri ilə yoxlaya bil ərsiniz. Bu əmr, Git obyektlərini yoxlamaq üçün bir növ İsveçrə ordusu bıçağıdır. `-P`, `cat-file` keçid əmrinə əvvəlcə məzmunun növünü müəyyənləşdirməyi, sonra isə uyğun şəkildə göstərməyi əmr edir:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

İndi Git-ə məzmun əlavə edib yenidən geri pull edə bilərsiniz. Bunu fayllardakı məzmunu ilə də edə bilərsiniz. Məsələn, bir faylda bəzi sadə versiya nəzarəti edə bilərsiniz. Əvvəlcə yeni bir fayl yaradın və məzmununu verilənlər bazanıza yazın:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Sonra bir neçə yeni məzmunu fayla yazın və yenidən qeyd edin:

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

İndi obyekt verilənlər bazanızda bu yeni faylın hər iki versiyası da var (orada saxladığınız ilk m əzmun kimi):

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Bu nöqtədə, həmin `test.txt` faylının yerli nüsxəsini silə bilərsiniz, sonra ilk verilənlər bazasını saxladığınız obyekt verilənlər bazasından almaq üçün Git-dən istifadə edin:

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

Və ya ikinci versiya:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

Ancaq faylınızın hər bir versiyası üçün SHA-1 key-ni xatırlamaq praktik deyil; üstəlik, sisteminizdə fayl adını deyil, yalnız məzmunu saxlayırsınız. Bu obyekt növünə *blob* deyilir.

Git-dən SHA-1 key-ni nəzərə alaraq Git-dəki`git cat-file –t` ilə hər hansı bir obyektin növünü sizə izah etməsini istəyə bilərsiniz:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

Ağac Obyektləri

Növbəti araşdıracağımız Git obyekt növü, fayl adının saxlanılması problemini həll edən və eyni zamanda bir qrup faylları birlikdə saxlamağınıza imkan verən *tree*-dir. Git, məzmunu UNIX fayl sisteminə bənzər, lakin biraz sadələşdirilmiş şəkildə saxlayır.

Bütün məzmun ağac və blob obyektləri, UNIX qovluq girişlərinə uyğun olan ağaclar və inodes və ya fayl məzmununa az və ya çox uyğun gələn blob-larla saxlanılır. Tək bir ağac obyekti, hər biri əlaq əli rejimi, növü və fayl adı ilə bir blob və ya alt ağacın SHA-1 hash olan bir və ya daha çox giriş daxil edir. Məsələn, bir layihədəki ən son ağac belə görünə bilər:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

`Master^{tree}` sintaksisiniz, `master` branch-nızdakı sonuncu işarə ilə göstərilən ağac obyektini mü əyyənləşdirir. Diqqət yetirin ki, `lib` alt kataloqu blok deyil, o başqa bir ağaca işarə edir:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

Hansı shell-dən istifadə etdiyinizə görə, `master^{tree}` sintaksisini istifadə edərkən səhvlərlə qarşılaşa bilərsiniz.

Windows-dakı CMD-də `^` işarəsi qaçmaq üçün istifadə olunur, buna görə bunun qarşısını almaq üçün onu ikiqat artırmalısınız: `git cat-file -p master^^{tree}". PowerShell istifadə edərkən, parametrin səhv təhlil edilməməsi üçün {} simvoldan istifadə olunan parametrlərə istinad edilməlidir: `git cat-file -p 'master^{tree}'`. ZSH istifadə edirsinizsə, `^` işarəsi globbing üçün istifadə olunur, buna görə bütün ifadəni dırnaq işarələrinə əlavə etməlisiniz: `git cat-file -p 'master^{tree}"`.

Konseptual olaraq, Gitin saxladığı məlumatlar belə bir şeyə bənzəyir:

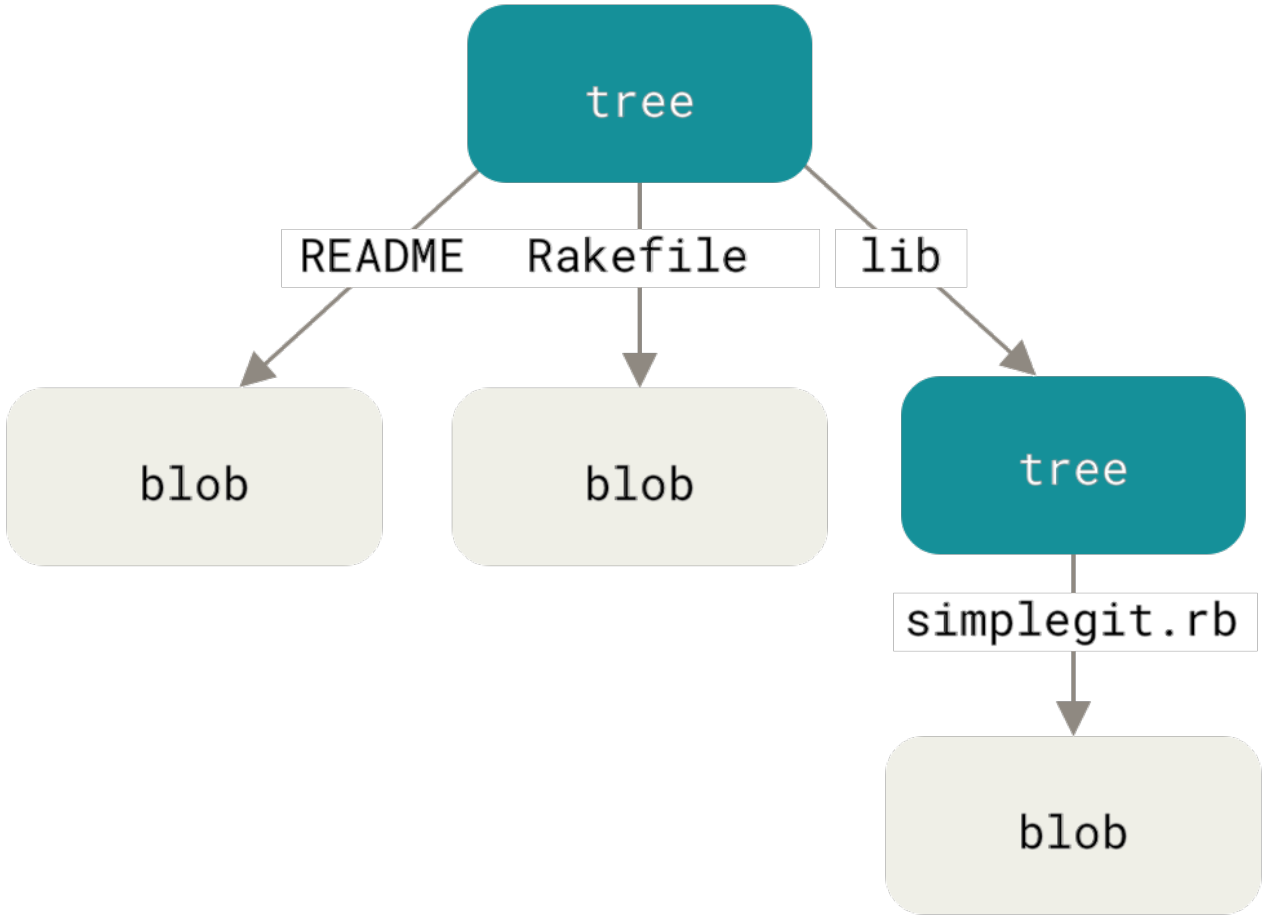


Figure 149. Git data modelinin sadə versiyası

Öz ağacınızı olduqca asanlıqla yarada blərsiniz. Git normal olaraq quruluş sahənizin və ya indeksinizin vəziyyətini götürərək ondan bir sıra ağac obyektlərini yazaraq bir ağac yaradır Bel əliklə, bir ağac obyekti yaratmaq üçün əvvəlcə bəzi sənədləri quraraq bir indeks qurmalısınız. Single entry – ilə sadə indeks qurmaq üçün `test.txt` faylınızın ilk versiyası — `git update-index` plumbing əmrindən istifadə edə bilərsiniz. Bu əmrdən süni şəkildə yeni bir səhnə sahəsinə `test.txt` faylının əvvəlki versiyasını əlavə etmək üçün istifadə edirsiniz. Fayl hazırlama sahənizdə hələ olmadığı üçün `--add` seçimini keçməlisiniz (hələ də staging sahəniz qurulmayıbsa) və əlavə etdiyiniz fayl olmadığı üçün `--cacheinfo` qovluğunuzda ancaq verilənlər bazanızdadır. Sonra rejimi, SHA-1 və fayl adını təyin edirsiniz:

```
$ git update-index --add --cacheinfo 100644 \
  83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

Bu vəziyyətdə, `100644` rejimini təyin edirsiniz, yəni normal bir fayldır. Digər seçimlər `100755`-dir, y əni icra edilə bilən bir fayldır; və simvolik bir əlaqəni ifadə edən `120000`seçimidir. Rejim normal UNIX rejimlərindən götürülmüşdür, lakin daha az çevikdir - bu üç rejim Git-dəki fayllar (bloblar) üçün etibarlı olanlardır (baxmayaraq ki, digər rejimlər kitabçalar və alt modullar üçün istifadə olunur).

İndi quruluş sahəsini bir ağac obyektinə yazmaq üçün `git write-tree` istifadə edə bilərsiniz. Heç bir `-w` seçiminə ehtiyac yoxdur - bu əmri çağırmaq avtomatik olaraq indiki vəziyyətdə bir ağac obyektini indeks vəziyyətindən yaradır:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

Daha əvvəl gördüyünüz eyni `git cat-file` əmrini istifadə edərək ağac obyekti olduğunu da doğrulaya bilərsiniz:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

İndi `test.txt`-in ikinci versiyası və yeni bir fayl ilə yeni bir ağac yaradacaqsınız:

```
$ echo 'new file' > new.txt
$ git update-index --add --cacheinfo 100644 \
  1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
$ git update-index --add new.txt
```

İndi səhnələşdirmə sahənizdə `test.txt`-in yeni versiyası ilə yanaşı yeni `new.txt` faylı var. O ağacı yazın (staging sahəsinin və ya indeksin bir ağac obyektinə yazılmasına istinadən) və necə göründüyünə baxın:

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Diqqət yetirin ki, bu ağacın həm fayl girişləri, həm də `test.txt` SHA-1 əvvəldən qalan “version 2” SHA-1 (`1f7a7a`) versiyası var. Yalnız əyləncə üçün ilk ağacı alt qovluq olaraq bu ağaca əlavə edəc əksiniz. Quruluş sahənizə ağacları `git read-tree`-ni işə salaraq oxuya bilərsiniz. Bu vəziyyətdə, mövcud əmri `--prefix` seçimindən istifadə edərək subtree olaraq quruluş sahənizə oxuya bil ərsiniz:

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579      bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Yeni yazdığınız ağacdan bir iş qovluğu yaratmışsanız, iş qovluğunun üst səviyyəsindəki iki faylı və `test.txt` sənədinin ilk versiyasını ehtiva edən `bak` adlı bir alt qovluğu əldə edəcəksiniz. Git-in bu

strukturlar üçün ehtiva etdiyi məlumatları belə hesab edə bilərsiniz:

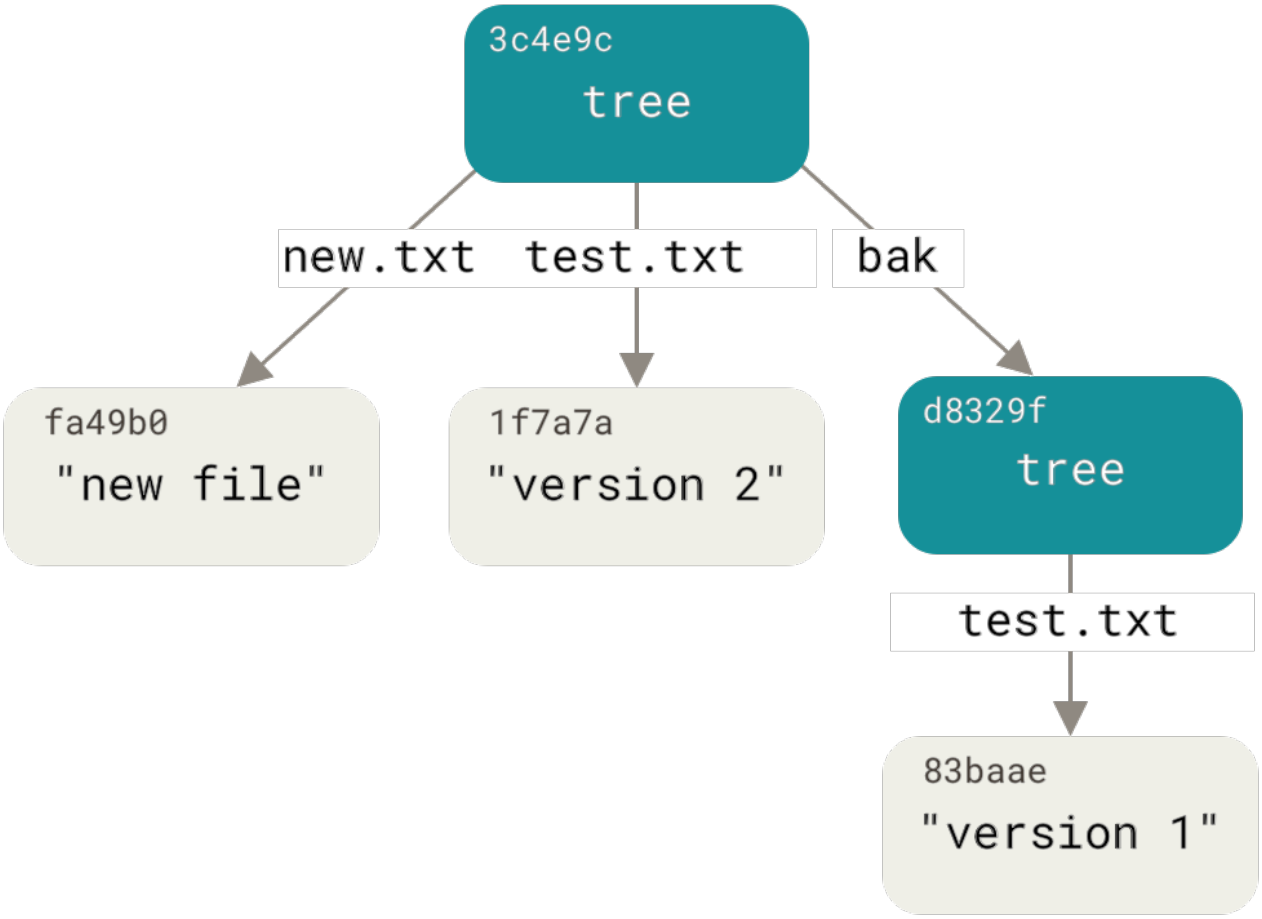


Figure 150. The content structure of your current Git data

Commit Obyektləri

Yuxarıda sadalananların hamısını etmisinizsə, indi izləmək istədiyiniz layihənizin fərqli anlarını əks etdirən üç ağacınız var, lakin əvvəlki problem yerində qalır: anlıq snapshot-ları xatırlamaq üçün hər üç SHA-1 dəyərini xatırlamalısınız. Snapshot-ları kimin saxladığı, nə vaxt və ya nə üçün saxlandığı barədə heç bir məlumatınız isə yoxdur. Commit obyektinin sizin üçün saxladığı əsas m əlumatlar budur.

Bir öhdəlik obyekti yaratmaq üçün, `commit-tree` işə salırsınız və bir ağac SHA-1 təyin edirsiniz və ə gər varsa, əvvəlcədən obyektlər yaradırsınız. Yazdığınız ilk ağacdan başlayın:

```
$ echo 'First commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Fərqli yaradılış müddəti və müəllif məlumatları səbəbindən fərqli bir hash dəyəri əldə edəcəksiniz. Üstəlik, prinsip etibarilə hər hansı bir commit obyekti m əlumatların verildiyi dəqiq şəkildə çoxaldıla bilsə də, bu kitabın inşaatının tarixi təfərrüatları, çap edilən commit-lərin verilmiş commit-lərə uyğun olmaya biləc əyini göstərir. Bu bölmədə daha çox əmsal və etiket hash-lərini öz checksum-larınızla əvəz edin.

Artıq yeni commit-inizə `git cat-file`-dan baxa bilərsiniz:

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

First commit
```

Commit obyekti üçün format sadədir: həmin nöqtədə layihənin snapshot-u üçün ən yüksək səviyy əli ağacı təyin edir; valideyn varsa, commit-i götürür (yuxarıda göstərilən commit obyektinin valideynləri yoxdur); author/committer məlumatı (istifadəçi adınız və istifadəçi poçtunuzun konfiqurasiya parametrlərindən və zaman damğasından istifadə edən); boş bir sətir və sonra commit mesajı.

Sonra, hər biri özündən əvvəl gələn commit-ə istinad edən digər iki commit obyektini yazacaqsınız:

```
$ echo 'Second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'Third commit'  | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Each of the three commit objects points to one of the three snapshot trees you created. Oddly enough, you have a real Git history now that you can view with the `git log` command, if you run it on the last commit SHA-1: Hər üç commit obyekti yaratdığınız üç snapshot ağacından birini göst ərir. Qəribədir ki, son olaraq SHA-1-də işləsəniz, `git log` əmri ilə baxa biləcəyiniz gerçək bir Git tarixiniz var:

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

  Third commit

 bak/test.txt | 1 +
 1 file changed, 1 insertion(+)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:14:29 2009 -0700

  Second commit

 new.txt  | 1 +
 test.txt | 2 +-
 2 files changed, 2 insertions(+), 1 deletion(-)

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700

    First commit

 test.txt | 1 +
 1 file changed, 1 insertion(+)
```

Əla. Front end əmrlərindən heç birini istifadə etmədən Git tarixçəsi yaratmaq üçün aşağı səviyyə əməliyyatları tamamlamısınız.

Siz `git add` və `git commit` əmrlərini çalışdırdığınızda Git bunu edir - dəyişən fayllar üçün blob-ları saxlayır, indeksləri yeniləyir, ağacları və üst səviyyə istinad obyektlərini və onlardan dərhal əvvəl gələn vəzifələri yazır. Bu üç əsas Git obyekti — the blob, the tree və the commit — əvvəlcə `.git/objects` qovluğunda ayrı fayllar kimi saxlanılır. İndi bütün obyektlər, içində saxladıqları şərh edilmiş şəkildə nümunələr qovluğundadır:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Bütün daxili göstəriciləri təqib etsəniz, belə bir obyekt qrafiki əldə edirsiniz: . Git qovluğunuzdakı bütün əlçatan obyektlər image::images/data-model-3.png[All the reachable objects in your Git directory]

Object Deposu

Daha əvvəl Git obyekt verilənlər bazasına sadiq olduğunuz hər bir obyektlə birlikdə saxlanılan bir başlığın olduğunu qeyd etmişdik. Git-in obyektlərini necə saxladığını görmək üçün bir dəqiqə ayıraq. Blob obyektinin necə saxlandığını - bu halda “what is up, doc?”-- sətrini Ruby skript dilində interaktiv şəkildə görəcəksiniz.

İnteraktiv Ruby rejimini `irb` əmri ilə işə sala bilərsiniz:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git əvvəlcə obyekt tipini təyin etməklə başlayan bir başlıq qurur — bu halda bu bir blob sayılır. Başlığın bu birinci hissəsinə Git, boşluğun ardından məzmunun bayt ölçüsünü və son sıfır baytını əlavə edir:

```
>> header = "blob #{content.bytesize}\0"
=> "blob 16\u0000"
```

Git, başlığı və orijinal məzmunu birləşdirir və sonra bu yeni məzmunun SHA-1 checksum cəmini hesablayır. Ruby-də bir string-in SHA-1 dəyərini SHA1 toplama kitabxanasını `require` əmri ilə daxil edib daha sonra sətirlə `Digest::SHA1.hexdigest()` işə salaraq hesablaya bilərsiniz:

```
>> store = header + content
=> "blob 16\u0000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Gəlin onu Git `hash-object` output-u ilə müqayisə edək. Burada input-a yeni bir xətt əlavə edilməməsi üçün `echo -n` istifadə edirik.

```
$ echo -n "what is up, doc?" | git hash-object --stdin
bd9dbf5aae1a3862dd1526723246b20206e5fc37
```

Git, zlib kitabxanası ilə Ruby-də edə biləcəyiniz yeni məzmunu zlib ilə sıxışdırır. Əvvəlcə kitabxananı tələb etməlisiniz və sonra məzmunda `Zlib::Deflate.deflate()` işə salmalısınız:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\x9CK\xCA\xC9OR04c(\xCFH,Q\xC8,V(-\xD0QH\xC9O\xB6\a\x00_\x1C\a\x9D"
```

Nəhayət, zlib-deflated məzmununuzu diskdəki bir obyektə yazacaqsınız. Yazmaq istədiyiniz obyektin path-ini təyin edəcəksiniz (SHA-1 dəyərinin ilk iki simvolu alt direktoriya adı və son 38 simvol həmin qovluqdakı fayl adıdır). Ruby-də, mövcud olmadıqda alt qovluğu yaratmaq üçün `FileUtils.mkdir_p()` funksiyasından istifadə edə bilərsiniz. Daha sonra, faylı `File.open()` ilə açın və əvvəlki zlib-compressed məzmunu nəticəsi ilə ortaya çıxan fayl sapı üzərinə bir `write()` çağırışı ilə yazın:

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

Gəlin `git cat-file` istifadə edərək obyektin məzmununu yoxlayaq:

```
---
$ git cat-file -p bd9dbf5aae1a3862dd1526723246b20206e5fc37
what is up, doc?
---
```

Və budur – siz tamamlanmış bir Git blob obyekti yaratdınız.

Bütün Git obyektləri eyni şəkildə, yalnız fərqli növlər ilə saxlanılır – o, simvol bloku əvəzinə, commit başlığı və ya ağacla başlayacaq. Blob məzmunu təxminən hər şey ola bilsə də, commit və ağac məzmunu çox xüsusi formatlandırılmışdır.

Git Referansları

Deponuzun tarixini qeydiyyat prosesindən, məsələn `1a410e` kimi görmək mümkündürsə, bu tarixç əyə baxmaq üçün `git log 1a410e` kimi bir şey işlədə bilərsiniz, amma yenə də `1a410e`-nin bu tarix üçün başlanğıc nöqtəsi olaraq istifadə etmək istədiyiniz commit prosesi olduğunu unutmamalısınız. Bunun əvəzinə, həmin SHA-1 dəyərini sadə bir ad altında saxlaya biləcəyiniz bir faylınız olsaydı daha asan olardı, belə ki, xam SHA-1 dəyərindən çox bu sadə addan istifadə edə bilərsiniz.

Git-də bu sadə adlara `references'' və ya ``refs'' deyilir; həmin SHA-1 dəyərlərini ehtiva edən sənədləri `.git/refs` qovluğunda tapa bilərsiniz. Mövcud layihədə bu qovluqda heç bir fayl yoxdur, ancaq sadə bir quruluş içərisindədir:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
```

Ən son commit-nizin harada olduğunu xatırlamağa kömək edəcək yeni bir istinad yaratmaq üçün texniki olaraq bu qədər sadə bir şey edə bilərsiniz:

```
$ echo 1a410efbd13591db07496601ebc7a059dd55cfe9 > .git/refs/heads/master
```

İndi Git əmrlərinizdəki SHA-1 dəyəri əvəzinə yeni yaratdığınız head referansından istifadə edə bil ərsiniz:

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Referans fayllarını birbaşa redaktə etməyiniz tövsiyə edilmir; Bunun əvəzinə, Git, bir referansı yeniləmək istəyirsinizsə, bunu etmək üçün daha təhlükəsiz `git update-ref` əmrini istifadə edə bil ərsiniz:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

Əsasən Gitdəki bir branch budur: sadə bir göstərici və ya bir iş xəttinin rəhbərinə referans. İkinci

commit-də bir branch yaratmaq üçün bunu edə bilərsiniz:

```
$ git update-ref refs/heads/test cac0ca
```

Branch-ınızda yalnız aşağıda sadalananların işi olacaq:

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

İndi Git verilənlər bazanız konseptual olaraq belə görünür:

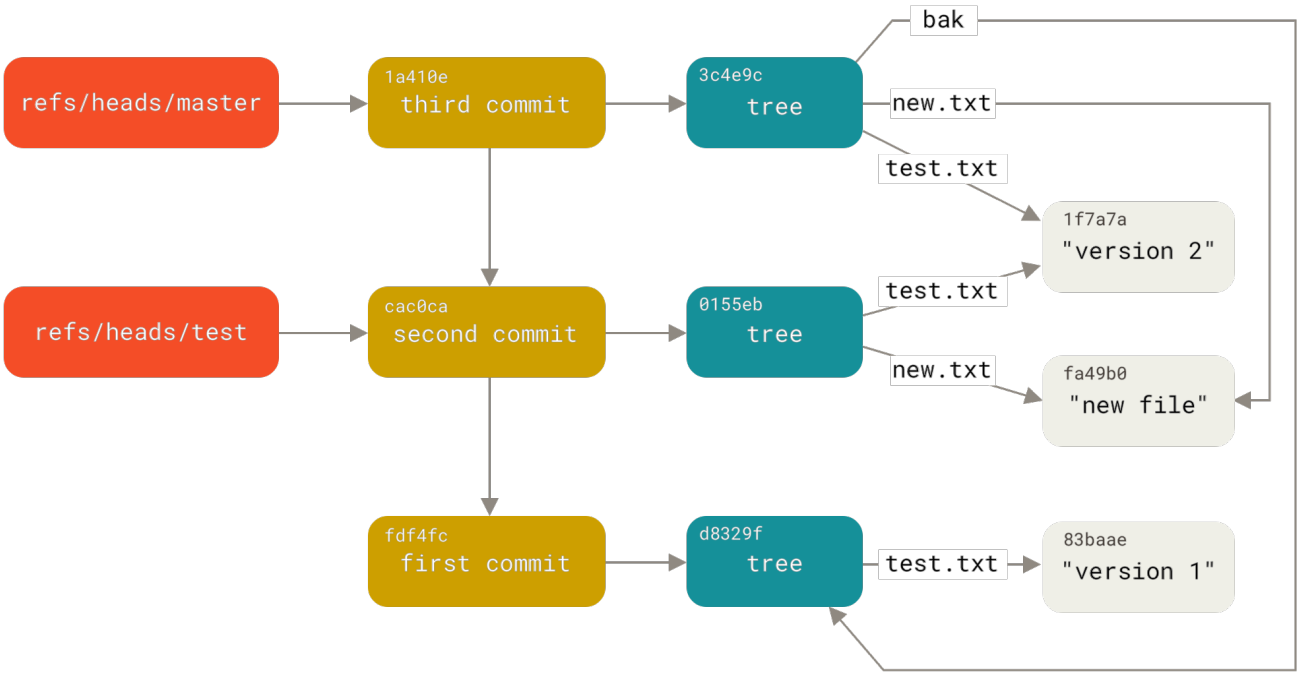


Figure 151. Git directory objects with branch head references included

`git branch <branch>` kimi əmrləri işə saldığınız zaman Git əsasən yaratmaq istədiyiniz hər hansı bir faylın son commit-i olan SHA-1-i əlavə etmək üçün `update-ref` əmrini işə salır.

HEAD

İndi sual, `git branch <branch>` işlətdiyiniz zaman Git sonuncu commit-dən SHA-1-i necə bilir? Cavab HEAD faylıdır.

Ümumiyyətlə HEAD faylı, hazırda olduğunuz branch-a simvolik bir referansdır. Simvolik istinad dedikdə, normal bir istinaddan fərqli olaraq başqa bir istinad üçün bir göstərici ehtiva etdiyini nə zərdə tuturuq.

Lakin bəzi nadir hallarda HEAD faylında git obyektinin SHA-1 dəyəri ola bilər. Deponuzu "detached HEAD" vəziyyətinə qoyan bir etiket, commit və ya uzaq bir branch-ı çıxardıqda bu baş verir.

Fayla baxsanız, normal olaraq belə bir şey görəcəksiniz:

```
$ cat .git/HEAD
ref: refs/heads/master
```

`git checkout test` işlətsəniz, Git faylını belə görünmək üçün yeniləyir:

```
$ cat .git/HEAD
ref: refs/heads/test
```

`git commit` əmrini işlətdiyinizdə, commit obyektinin əsas hissəsinin HEAD-dəki referansının işarə etdiyi SHA-1 dəyəri olduğunu ifadə edərək, commit obyektini yaradır.

Bu faylı manual olaraq da düzəldə bilərsiniz, lakin bunun üçün daha etibarlı bir əmr mövcuddur: `git symbolic-ref`. HEAD-in dəyərini bu əmr vasitəsilə oxuya bilərsiniz:

```
$ git symbolic-ref HEAD
refs/heads/master
```

Eyni əmrdən istifadə edərək HEAD dəyərini də təyin edə bilərsiniz:

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

Referans üslubundan kənarda simvolik bir istinad təyin edə bilməzsiniz:

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

Tags

Git'in üç əsas obyekt növünü (*blobs*, *trees* və *commits*) müzakirə etməyi bitirdik, indi dördüncüsünə baxaq. *Tag* obyekti bir commit obyektinə çox oxşayır - etiket, tarix, mesaj və göstərici ehtiva edir. Əsas fərq odur ki, bir etiket obyekti ümumiyyətlə bir ağaca deyil, bir commit-ə işarə edir. Bu branch referansına bənzəyir, amma heç vaxt tərpənmir - həmişə eyni commit-i göstərir, lakin ona daha dost bir ad verir.

Git'in Əsasları-də müzakirə edildiyi kimi iki növ etiket var: izahatlı və yüngül. Belə bir şey işlədərək yüngül bir etiket edə bilərsiniz:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

Bütün bunlar yüngül bir etiketdir - heç vaxt hərəkət etməyən bir referans. Şərhli etiket daha mür əkkəbdir. İzahatlı bir etiket yaratsanız, Git bir etiket obyekti yaradır və sonra birbaşa commit-ə

deyil, ona işarə etmək üçün bir referans yazır. Bunu izahatlı bir etiket yaratmaqla görə bilərsiniz (`-a` seçimindən istifadə edərək):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'Test tag'
```

Yaratdığı obyekt SHA-1 dəyəri budur:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

İndi SHA-1 dəyərində `git cat-file -p` işlədin:

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700

Test tag
```

Nişan girişinin etiketlədiyiniz SHA-1 dəyərini göstərdiyinə diqqət yetirin. Bir commit-i göstərməyə ehtiyac olmadığına da diqqət yetirin; istənilən Git obyektini etiketləyə bilərsiniz. Məsələn, Git qaynaq kodunda qoruyucu GPG ümumi açarını bir blob obyekti olaraq əlavə etdi və sonra etiketl ədi. Bunu Git deposunun bir klonunda işlədərək ümumi açara baxa bilərsiniz:

```
$ git cat-file blob junio-gpg-pub
```

Linux kernel deposunda ayrıca bir işarə etməyən bir etiket obyekti var - ilk etiket mənbə kodunun idxalının başlanğıc ağacına nöqtələr yaratdı.

Remote-lar

Görəcəyiniz üçüncü reefrans növü remote bir referansdır. Bir remote əlavə edib ona push etsəniz, Git bu məsafəyə son göndərdiyiniz dəyəri hər bir branch üçün `refs/remotes` qovluğunda saxlayır. Məsələn, `origin` adlı bir remote əlavə edə və `master` branch-ınızı ona push edə bilərsiniz:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master
Counting objects: 11, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
Total 7 (delta 2), reused 4 (delta 1)
To git@github.com:schacon/simplegit-progit.git
  a11bef0..ca82a6d  master -> master
```

Daha sonra `refs/remotes/origin/master` faylını yoxlayaraq server ilə son əlaqə qurduğunuz mənbə remote-undakı `master` branch-ının nə olduğunu görə bilərsiniz:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Uzaqdan referanslar branch-lardan (`refs/heads` referansları) əsasən read-only sayılmaları ilə fərql ənir. Birinə `git checkout` edə bilərsiniz, ancaq Git HEAD-i birinə yönəltməyəcək, buna görə onu heç bir zaman `commit` əmri ilə yeniləməyəcəksiniz. Git onları bu branch-ların həmin serverlərdə olduğu son bilinən vəziyyətə qədər bookmark-lar kimi idarə edir.

Packfile'lar

Əvvəlki hissədəki nümunədəki bütün təlimatları izlədiyiniz təqdirdə, 11 obyektlə - dörd blob, üç ağac, üç commit və bir etiketi olan bir test Git deposuna sahib olmalısınız:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git bu faylların məzmununu zlib ilə sıxır və siz çox şey saxlamırsınız, buna görə bütün bu fayllar toplu olaraq yalnız 925 bayt tutur. İndi Git-in maraqlı bir xüsusiyyətini nümayiş etdirmək üçün depoya daha əhəmiyyətli bir məzmun əlavə edəcəksiniz. Nümayiş etmək üçün Grit kitabxanasından `repo.rb` faylını əlavə edəcəyik — söhbət 22K mənbə kodu faylıdan gedir:

```
$ curl https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb >
repo.rb
$ git checkout master
$ git add repo.rb
$ git commit -m 'Create repo.rb'
[master 484a592] Create repo.rb
 3 files changed, 709 insertions(+), 2 deletions(-)
 delete mode 100644 bak/test.txt
 create mode 100644 repo.rb
 rewrite test.txt (100%)
```

Yaranan ağaca baxsanız, yeni `repo.rb` blob obyektiniz üçün hesablanmış SHA-1 dəyərini görə bil

ərsiniz:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

`git cat-file` istifadə edərək obyektin hansı böyüklükdə olduğunu görə bilərsiniz:

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
22044
```

Bu zaman, faylda bir balaca dəyişiklik edin və nə olduğuna baxın:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'Modify repo.rb a bit'
[master 2431da6] Modify repo.rb a bit
 1 file changed, 1 insertion(+)
```

Son commit-in yaratdığı ağacı yoxlayın və bu zaman çox maraqlı bir şey görəcəksiniz:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob b042a60ef7dff760008df33cee372b945b6e884e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Bu blob indi fərqli bir blob-dur, yəni 400 sətirlik bir faylın sonuna yalnız bir sətir əlavə etməyinizə baxmayaraq, Git bu yeni məzmunu tamamilə yeni bir obyekt kimi saxlayır:

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
22054
```

Diskinizdə təxminən iki eyni 22K obyekt var (hər biri təxminən 7K-a sıxılmışdır). Git onlardan birini tam olaraq saxlaya bilər, amma ikinci obyekti yalnız onunla birincisi arasındakı delta kimi saxlaya bilsəydi yaxşı olmazdı?

Belə görünür ki edə bilər. Git-in obyektləri diskdə saxladığı ilkin formata “loose” bir obyekt formatı deyilir. Bununla birlikdə, bəzən Git, yerdən qənaət etmək və daha səmərəli olmaq üçün bu obyektlərdən bir neçəsini bir “packfile” adlanan tək bir binary fayla yığır. Ətrafınızda çox boş obyekt varsa, `git gc` əmrini manual olaraq idarə etsəniz və ya remote serverə bassanız, Git bunu edəcəkdir. Nə baş verdiyini görmək üçün `git gc` əmrini axtararaq Git-dən obyektləri yığmasını manual olaraq istəyə bilərsiniz:

```
$ git gc
Counting objects: 18, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (18/18), done.
Total 18 (delta 3), reused 0 (delta 0)
```

`objects` qovluğuna baxsanız, bir çox obyektinizin getdiyini və bəzi faylların əmələ gəldiyini görəc əksiniz:

```
$ find .git/objects -type f
.git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack
```

Qalan obyektlər heç bir commit ilə göstərilməyən blob-lardır – nümunə olaraq `what is up, doc?` misalı və əvvəllər yaratdığınız “test content” nümunəsi bloblarını göstərə bilərik. Onları heç vaxt heç bir commit-ə əlavə etmədiyiniz üçün asılmış sayılır və yeni paketinizə yığılmır.

Digər sənədlər yeni packfile-niz və indeksinizdir. Packfile, sisteminizdən silinmiş bütün obyektl ərin məzmununu ehtiva edən tək bir fayldır. İndeks, müəyyən bir obyekti tez bir zamanda axtara bilmək üçün həmin packfile içərisindəki offsets-i ehtiva edən bir fayldır. Gözəl olan budur ki, 'gc` əmrini çalıştırmadan əvvəl diskdəki obyektlər toplu olaraq 15K ölçülü olmasına baxmayaraq, yeni packfile yalnız 7K-dır. Disk istifadənizi obyektlərinizi pack edərəck yarıya endirdiniz.

Bəs Git bunu necə edir? Git obyektləri paketlədikdə, eyni adlı və ölçülü faylları axtarır və faylın bir versiyasından digərinə yalnız deltaları saxlayır. Siz packfile-in içərisinə baxa və Git-in yerə qənaət etmək üçün nə etdiyini görə bilərsiniz. `Git verify-pack` plumbing əmri, nəyin paketləndiyini görm əyə imkan verir:

```
$ git verify-pack -v .git/objects/pack/pack-
978e03944f5c581011e6998cd0e9e30000905586.idx
2431da676938450a4d72e260db3bf7b0f587bbc1 commit 223 155 12
69bcdaff5328278ab1c0812ce0e07fa7d26a96d7 commit 214 152 167
80d02664cb23ed55b226516648c7ad5d0a3deb90 commit 214 145 319
43168a18b7613d1281e5560855a83eb8fde3d687 commit 213 146 464
092917823486a802e94d727c820a9024e14a1fc2 commit 214 146 610
702470739ce72005e2edff522fde85d52a65df9b commit 165 118 756
d368d0ac0678cbe6cce505be58126d3526706e54 tag    130 122 874
fe879577cb8cffcdf25441725141e310dd7d239b tree   136 136 996
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree   36 46 1132
deef2e1b793907545e50a2ea2ddb5ba6c58c4506 tree   136 136 1178
d982c7cb2c2a972ee391a85da481fc1f9127a01d tree   6 17 1314 1 \
  deef2e1b793907545e50a2ea2ddb5ba6c58c4506
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree   8 19 1331 1 \
  deef2e1b793907545e50a2ea2ddb5ba6c58c4506
0155eb4229851634a0f03eb265b69f5a2d56f341 tree   71 76 1350
83baae61804e65cc73a7201a7252750c76066a30 blob   10 19 1426
fa49b077972391ad58037050f2a75f74e3671e92 blob   9 18 1445
b042a60ef7dff760008df33cee372b945b6e884e blob   22054 5799 1463
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob   9 20 7262 1 \
  b042a60ef7dff760008df33cee372b945b6e884e
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob   10 19 7282
non delta: 15 objects
chain length = 1: 3 objects
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack: ok
```

Burada, `repo.rb` faylınızın ilk versiyası olduğunu xatırladığınız `033b4` blob, faylın ikinci versiyası olan `b042a` blob-una istinad edir. Output-dakı üçüncü sütun paketdəki obyektin ölçüsüdür, buna görə `b042a`ın 22K faylını götürdüyünü, ancaq `033b4`ün yalnız 9 bayt yer aldığını görə bil ərsiniz.

Maraqlısı budur ki, faylın ikinci versiyası toxunulmaz saxlanılır, orijinal versiyası isə delta kimi saxlanılır — bu, faylın ən son versiyasına daha sürətli daxil olma ehtimalının olmasıdır.

Bununla əlaqədar həqiqətən gözəl şeylərdən biri, hər an yenidən paketlənə bilməsidir. Git arada verilənlər bazanızı avtomatik olaraq yenidən paketlənəcək və hər zaman daha çox yer saxlamağa çalışacaq, ancaq istədiyiniz zaman əlinizlə `git gc` düyməsini işə salmaqla manual olaraq yenidən paketləyə bilərsiniz.

Refspec

Bu kitab boyunca uzaq branch-lardan yerli referanslara qədər sadə xəritələrdən istifadə etdik, lakin daha mürəkkəb ola bilər. Tutaq ki, son cüt bölmələri ilə birlikdə izlədiniz və kiçik bir yerli Git deposu yaratdınız və indi ona bir *remote* əlavə etmək istəyirsiniz:

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

Yuxarıdakı əmri işə salmaq, depolarınızın `.git/config` faylına bir bölmə əlavə edir, pultun adını (`origin`), remote deponun URL-ini və gətirmək üçün istifadə ediləcək *refspec* qeyd edir:

```
[remote "origin"]
    url = https://github.com/schacon/simplegit-progit
    fetch = +refs/heads/*:refs/remotes/origin/*
```

Refspecin formatı, əvvəlcə istəyə bağlı bir `+` ardından `<src>:<dst>`, burada `<src>` remote tərəfdəki referanslar üçün nümunədir və `<dst>` olduğu yerdir və bu referanslar yerli olaraq izlənilәcəkdir. `+` Git-ə sürətli irəliləməsə də referansı yeniləməsini söyləyir.

Avtomatik olaraq bir `git remote add origin` əmri ilə yazılan varsayılan vəziyyətdə, Git serverdəki `refs/heads/` altındakı bütün referansları götürür və yerli olaraq `refs/remotes/origin/` yazır. Belə liklə, serverdə bir `master` branch-ı varsa, aşağıdakılardan hər hansı biri ilə həmin branch-ın jurnalına daxil ola bilərsiniz:

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

Hamısı ekvivalentdir, çünki Git hər birini `refs/remotes/origin/master`-ə genişləndirir.

Bunun əvəzinə Git'in remote serverdəki digər branch-ları deyil, yalnız `master` branch-ını hər dəfə pull down istəyirsinizsə, gətirmə xəttini yalnız bu branch-a istinad etmək üçün dəyişə bilərsiniz:

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

Bu, həmin remote üçün `git fetch` üçün yalnız standart refspecdir. Yalnız birdəfəlik gətirmək ist əyirsinizsə, komanda xəttində də xüsusi refspeci göstərə bilərsiniz. Uzaqdakı `master` branch-nı local olaraq `origin/mymaster`-a çəkmək üçün işlədə bilirsiniz:

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Ayrıca birdən çox spesifikasiya təyin edə bilərsiniz. Əmr sətrində belə bir neçə branch-ı pull down bilərsiniz:

```
$ git fetch origin master:refs/remotes/origin/mymaster \
     topic:refs/remotes/origin/topic
From git@github.com:schacon/simplegit
 ! [rejected]        master     -> origin/mymaster  (non fast forward)
 * [new branch]      topic      -> origin/topic
```

Bu vəziyyətdə, sürətli master istinad kimi göstərilmədiyi üçün `master` branch pull etməsi rədd edildi. Refspecin qarşısındakı `+` işarəsini göstərərək bunu ləğv edə bilərsiniz.

Konfiqurasiya faylınızda əldə etmək üçün birdən çox refspecs də göstərə bilərsiniz. Həmişə `origin` remote-undan `master` və `experiment` branch-larını almaq istəyirsinizsə, iki sətir əlavə edin:

```
[remote "origin"]
    url = https://github.com/schacon/simplegit-progit
    fetch = +refs/heads/master:refs/remotes/origin/master
    fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

Git 2.6.0 olduğundan, birdən çox branch-a uyğun olmaq üçün pattern-də qismən qlobuslardan istifadə edə bilərsiniz, belə ki, bu işləyir:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

Daha yaxşısı, daha çox quruluşla eyni şeyi etmək üçün ad boşluqlarından (və ya qovluqlardan) istifadə edə bilərsiniz. Bir sıra branch-ları push edən bir QA komandanız varsa və `master` branch-nı və QA komandasının hər hansı bir branch-ını əldə etmək istəyirsinizsə, başqa bir şey yoxdursa, belə bir konfiqurasiya bölməsindən istifadə edə bilərsiniz:

```
[remote "origin"]
    url = https://github.com/schacon/simplegit-progit
    fetch = +refs/heads/master:refs/remotes/origin/master
    fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

Bir QA komandasını branch-ları push edən, developerləri branch-ları push edən və inteqrasiya komandalarını uzaqdan branch-lara push edən və iş birliyi olan kompleks bir iş axını prosesiniz varsa, onları bu şəkildə asanlıqla adlandıra bilərsiniz.

Pushing Refspecs

Namespace referansları bu şəkildə ala biləməyiniz çox yaxşıdır, amma QA komandası ilk növbədə branch-larını `qa/` namespace-ə necə çevirir? Bunu push etmək üçün refspecs istifadə edərək həyata keçirirsiniz.

QA komandası, `master` branch-nı remote server üzerindəki `qa/master` push etmək istəyirsə, bunları işlədə bilər:

```
$ git push origin master:refs/heads/qa/master
```

Git'in hər dəfə `git push origin` işə saldıqda bunu avtomatik olaraq etməsini istəsələr, konfiqurasiya sənədlərinə bir `push` dəyəri əlavə edə bilərlər:

```
[remote "origin"]
    url = https://github.com/schacon/simplegit-progit
    fetch = +refs/heads/*:refs/remotes/origin/*
    push = refs/heads/master:refs/heads/qa/master
```

Yenə də bu, local bir `master` branch-ını default olaraq remote `qa/master` branch-na push etməyə səb əb olacaqdır.

Refspec-i bir depodan götürüb başqa birinə push etmək üçün istifadə edə bilm əzsiniz. Bunu edə biləcəyiniz bir nümunə üçün GitHub Public Depolarınızı Yenil əyin-ə müraciət edin.

Reference-ları Silmək

Remote serverdən referansları silmək üçün refspec-dən belə bir şey istifadə edərək istifadə edə bil ərsiniz:

```
$ git push origin :topic
```

Refspec `<src>:<dst>` olduğu üçün `<src>` hissəsini tərk edərək, bu, əsasən `topic` branch-nı uzaqdan heç bir şey etməməyi tələb edir və onu silir.

Və ya daha yeni sintaksisdən istifadə edə bilərsiniz (Git v1.7.0-dan bəri mövcuddur):

```
$ git push origin --delete topic
```

Transfer Protokolları

Git iki depo arasında məlumatları iki əsas şəkildə ötürə bilər: “dumb' protokolu və ``smart” protokolu. Bu bölmə bu iki əsas protokolun necə işlədiyini tez bir zamanda əhatə edəcəkdir.

The Dumb Protokolu

Yalnız bir oxunuşda HTTP üzərindən təqdim ediləcək bir depo qurursanız, dumb protokolundan istifadə edilməsi ehtimalı böyükdür. Bu protokol “dumb” adlanır, çünki nəqliyyat prosesi zamanı server tərəfində Git-ə məxsus bir kod tələb olunmur; gətirmə prosesi, müştərinin serverdəki Git deposunun tərtibatını qəbul edə biləcəyi bir sıra HTTP `GET` istəkləridir.

Bu günlərdə dumb protokolu kifayət qədər nadir hallarda istifadə olunur. Təhlük əsizliyi təmin etmək və ya özəlləşdirmək çətindir, buna görə də Git host-larının əksəriyyəti (həm cloud əsaslı, həm də on-premises) istifadə etməkdən imtina edə cəkdir. Ümumiyyətlə bir azdan izah etdiyimiz daha ağıllı protokoldan istifadə etməyiniz tövsiyə olunur.

Simplegit kitabxanası üçün `http-fetch` prosesini izləyək:

```
$ git clone http://server/simplegit-progit.git
```

Bu əmrin ilk işi `info/refs` faylını pull etməkdir. Bu fayl `update-server-info` əmri ilə yazılmışdır, bu səbəbdən HTTP nəqlinin düzgün işləməsi üçün bunu `post-receive` hook olaraq təmin etməlisiniz:

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949     refs/heads/master
```

Artıq remote istinadların və SHA-1-lərin siyahısı var. Daha sonra, HEAD istinadının nə olduğunu axtarırsınız, belə ki bitirdikdən sonra nəyi yoxlayacağınızı biləcəksiniz:

```
=> GET HEAD
ref: refs/heads/master
```

Prosesi tamamladıqdan sonra `master` branch-nı yoxlamalısınız. Bu anda gəzinti prosesinə başlamağa hazırsınız. Başlanğıc nöqtəniz `info/refs` faylında gördüyünüz `ca82a6` commit obyekti olduğundan, bunu əldə etməyə başlayırsınız:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

Bir obyekt geri alırsınız – hanı ki, bu obyekt serverdə boş formatda olur və onu statik bir HTTP GET istəyi üzərinə götürmüşdünüz. Siz onu zlib-uncompress edə bilər, başlığı soyub, məzmuna baxa bilərsiniz:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

Change version number
```

Sonra, sizin alacağınız daha iki obyektiniz var - yeni aldığımız commit-in göstərdiyi məzmun ağacından olan `cfda3b` və valideyn olan `085bb3`:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

O sizə bir sonrakı obyekt commit-ini verir. Ağac obyektini götürün:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Oy! – belə görünür ki bu ağac obyekti serverdə boş formatda deyil, buna görə 404 cavabını geri alırsınız. Bunun bir neçə səbəbi ola bilər - obyekt alternativ bir depoda ola bilər və ya bu depodakı packfile-da da ola bilər. Git əvvəlcə siyahıda göstərilən alternativləri yoxlayır:

```
=> GET objects/info/http-alternates
(empty file)
```

Əgər o, alternativ URL-lərin siyahısı ilə geri qayıdırsa, Git boş faylları və packfile-ları yoxlayır - bu, bir-birinə çəngəllənən layihələrin diskdəki obyektləri paylaşması üçün gözəl bir mexanizmdir. Bununla birlikdə, bu vəziyyətdə alternativlər göstərilmədiyindən, obyektiniz packfile-da olmalıdır. Bu serverdə hansı packfile-ların mövcud olduğunu görmək üçün bunların siyahısını ehtiva edən (`update-server-info` tərəfindən yaradılan) `objects/info/packs` faylını əldə etməlisiniz.

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

Serverdə yalnız bir packfile var, buna görə obyektiniz açıq-aydın şəkildə oradadır, ancaq əmin olmaq üçün indeks faylını yoxlayacaqsınız. Serverdə birdən çox packfile-nız varsa, bu da sizə lazım olan obyekti hansı packfile-da görə biləcəyiniz üçün faydalıdır:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

Artıq packfile indeksinə sahib olduğunuzdan, obyektinizin içində olub olmadığını görə bilərsiniz - çünki indeks packfile-nızdakı obyektlərin SHA-1-lərini və həmin obyektlərin əvəzlərini siyahıya alır. Hədəfiniz oradadır, davam edin və bütün packfile-ı əldə edin:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

Indi sizin ağac obyektiniz mövcuddur və siz commit-lərinizi yeritməyə davam edə bilərsiniz. Onların hamısı həm də hal-hazırda yüklədiyiniz packfile-da olduğundan serverə başqa request-lər göndərməyinizə ehtiyac yoxdur. Git əvvəlində yüklədiyiniz HEAD arayışı ilə işarə edilən `master` branch-ın işləyən bir nüsxəsini yoxlayır.

The Smart Protokolu

Dumb protokolu sadə, lakin bir az səmərəsizdir və müştəridən serverə məlumat yazmaqla işləyə bilmir. Smart protokolu daha çox məlumat ötürmə metodudur, lakin Git haqqında ağıllı bir proses tələb edir – o, yerli məlumatları oxuya bilir, müştərinin nəyə ehtiyac duyduğunu tapır və bunun üçün xüsusi bir paket yarada bilir. Məlumatların ötürülməsi üçün iki proses dəsti mövcuddur:

uploading data üçün bir cüt və downloading data üçün bir cüt.

Uploading Data

Verilənləri remote prosesə yükləmək üçün Git, `send-pack` və `receive-pack` proseslərindən istifadə edir. `send-pack` prosesi müştəri üzərində işləyir və remote-dakı `receive-pack` prosesinə qoşulur.

SSH

Məsələn, proyektinizdə `git push origin master` işlətdiyinizi düşünək və `origin` SSH protokolundan istifadə edən bir URL olaraq təyin olunduğunu deyək. Git, SSH üzərindən serverinizə bir əlaqə quran `send-pack` prosesini işə salır. O, remote serverdə belə bir şeyə bənzəyən bir SSH çağırışı vasitəsilə bir əmr işlətməyə çalışır:

```
$ ssh -x git@server "git-receive-pack 'simplegit-progit.git'"
00a5ca82a6dff817ec66f4437202690a93763949 refs/heads/master□report-status \
    delete-refs side-band-64k quiet ofs-delta \
    agent=git/2:2.1.1+github-607-gfba4028 delete-refs
0000
```

`git-receive-pack` əmri, hazırda sahib olduğu hər bir istinad üçün dərhal bir sətirlə cavab verir – y əni, bu halda yalnız `master` branch-ı və onun SHA-1-i. Birinci sətirdə serverin imkanlarının siyahısı (burada, `report-status`, `delete-refs` və digərləri, o cümlədən müştəri identifikatoru) var.

Məlumatlar chunk-lara ötürülür. Hər bir chunk, onun nə qədər olduğunu göstərən 4 simvollu bir hex dəyəri ilə başlayır (uzunluğun özünün 4 baytı da daxil olmaqla). Chunk-lar adətən tək bir m əlumat sətri və arxadakı bir xətti qidalandırır. İlk chunk-nız 165 hexadecimal üçün 00a5 ilə başlayır, yəni chunk 165 bayt uzunluğundadır. Növbəti chunk 0000-dir, yəni server istinadlar siyahısı ilə hazırlanır.

Artıq serverin vəziyyətini bildiyindən, `send-pack` prosesi, serverin etmədiyi commit-i müəyyənl əşdirir. Bu push-un yeniləyəcəyi hər istinad üçün, `send-pack` prosesi bu məlumatı `receive-pack` prosesinə izah edir. Məsələn, `master` branch-ı yeniləyirsinizsə və bir `experiment` branch-na əlavə edirsinizsə, `send-pack` cavabı belə görünə bilər:

```
0076ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83b3ff6
\
    refs/heads/master report-status
006c0000000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d
\
    refs/heads/experiment
0000
```

Git, xəttin uzunluğu, köhnə SHA-1, yeni SHA-1 və yenilənən istinadla yenilədiyiniz hər bir istinad üçün bir sətir göndərir. Birinci sətir də müştərinin imkanlarına malikdir. Bütün *0'ların SHA-1 dəy əri əvvəllər heç bir şey olmadığı demək deyil - çünki təcrübə istinadını siz əlavə edirsiniz. Əgər bir arayışı silsəniz, əksini görəcəksiniz: bütün '0*-lar sağ tərəfdədir.

Sonra müştəri serverdə hələ mövcud olmayan bütün obyektlərin paketini göndərir. Nəhayət, server müvəffəqiyyət (və ya uğursuzluq) göstəricisi ilə cavab verir:

```
000eunpack ok
```

HTTP(S)

Bu proses əsasən HTTP-də də eynidir, baxmayaraq ki, qarşılıqlı əlaqə biraz fərqlidir. Əlaqə bu ist əklə başladı:

```
=> GET http://server/simplegit-progit.git/info/refs?service=git-receive-pack
001f# service=git-receive-pack
00ab6c5f0e45abd7832bf23074a333f739977c9e8188 refs/heads/master□report-status \
    delete-refs side-band-64k quiet ofs-delta \
    agent=git/2:2.1.1~vmg-bitmaps-bugaloo-608-g116744e
0000
```

İlk müştəri-server mübadiləsi başa çatdı. Müştəri daha sonra başqa bir sorğu göndərir, bu dəfə `send-pack`-nin verdiyi məlumatlarla `POST` göndərilir.

```
=> POST http://server/simplegit-progit.git/git-receive-pack
```

`POST` sorğusuna, `send-pack` output-u və packfile faydalı yük kimi daxildir. Server daha sonra HTTP cavabı ilə müvəffəq və ya uğursuz olduğunu göstərir.

HTTP protokolunun bu məlumatları yığılmış köçürmə kodlaşdırmasının içərisinə əlavə edə biləc əyini unutmayın.

Downloading Data

Verilənləri yüklədiyiniz zaman, `fetch-pack` və `upload-pack` prosesləri iştirak edir. Müştəri hansı m əlumatların ötürüləcəyini müzakirə etmək üçün remote-dakı bir `upload-pack` prosesinə qoşulan bir `fetch-pack` prosesinə başlayır.

SSH

SSH üzərindən gətirmə edirsinizsə, `fetch-pack` belə bir şey işlədir:

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

`fetch-pack` qoşulduqdan sonra, `upload-pack` geriyə belə bir şey göndərir:

```
00dfca82a6dff817ec66f44342007202690a93763949 HEAD□multi_ack thin-pack \
    side-band side-band-64k ofs-delta shallow no-progress include-tag \
    multi_ack_detailed symref=HEAD:refs/heads/master \
    agent=git/2:2.1.1+github-607-gfba4028
003fe2409a098dc3e53539a9028a94b6224db9d6a6b6 refs/heads/master
0000
```

Bu, `receive-pack` cavabları ilə çox oxşardır, lakin imkanları fərqlidir. Bundan əlavə, HEAD-in işarə etdiyini geri göndərir (`symref=HEAD:refs/heads/master`), buna görə müştəri bunun bir klon olub olmadığını yoxlayacağını bilir.

Bu nöqtədə, `fetch-pack` prosesi hansı obyektlərə sahib olduğuna baxır və ehtiyac duyduğu obyektl ərə "want" və sonra SHA-1 göndərərək cavab verir. Artıq sahib olduğu bütün obyektləri "have" və sonra SHA-1 ilə göndərir. Bu siyahının sonunda, ehtiyac duyduğu məlumatların paket sənədini göndərməyə başlamaq üçün `upload-pack` prosesini başlamaq üçün "done" yazır:

```
003cwant ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0009done
0000
```

HTTP(S)

Fetch əməliyyatı üçün əl sıxma iki HTTP request-i alır. Birincisi, dumb protokolda eyni nöqtədə istifadə olunan bir `GET`-dir:

```
=> GET $GIT_URL/info/refs?service=git-upload-pack
001e# service=git-upload-pack
00e7ca82a6dff817ec66f44342007202690a93763949 HEAD□multi_ack thin-pack \
    side-band side-band-64k ofs-delta shallow no-progress include-tag \
    multi_ack_detailed no-done symref=HEAD:refs/heads/master \
    agent=git/2:2.1.1+github-607-gfba4028
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

Bu, SSH bağlantısı üzərindən `git-upload-pack` çağırmağa çox oxşayır, lakin ikinci mübadilə ayrı bir istək olaraq həyata keçirilir:

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7
0032have 441b40d833fdfa93eb2908e52742248faf0ee993
0000
```

Yenə də, bu yuxarıdakı ilə eyni formatdadır. Bu sorğunun cavabı müvəffəq və ya uğursuz olduğunu göstərir və paket sənədini də əhatə edir.

Protokolun Nəticəsi

Bu bölmə köçürmə protokollarının çox əsas bir icmalını ehtiva edir. Protokolda `multi_ack` və ya `side-band` imkanları kimi bir çox digər xüsusiyyətlər var, lakin bunları əhatə etmək bu kitabın ə hatə dairəsindən xaricdədir. Sizə müştəri və server arasında ümumi geri və irəli bir fikir verməyə çalışdıq; bundan daha çox biliyə ehtiyacınız varsa, ehtimal ki, Git mənbə koduna nəzər yetirmək istərdiniz.

Maintenance və Məlumatların Bərpası

Bəzən bəzi təmizlik işləri görməli ola bilərsiniz - deponu daha yığcam etmək, idxal olunan deponu təmizləmək və ya itirilmiş işi bərpa etmək. Bu bölmə bu ssenarilərdən bəzilərini əhatə edəcəkdir.

Maintenance

Bəzən, Git avtomatik olaraq “auto gc” adlı bir əmr işlədir. Çox vaxt bu əmr heç bir şey etmir. Bununla birlikdə, bir çox boş obyekt (packfile-da olmayan obyektlər) və ya çox çox paket varsa, Git tam hüquqlu bir `git gc` əmrini işə salır. “gc” zibil yığmaq deməkdir və bu əmr bir sıra şeyləri yerinə yetirir: bütün boş obyektləri toplayır və onları paketlərə yerləşdirir, paketləri bir böyük paketə birləşdirir hər hansı bir commit-dən əlçatmaz olan və bir neçə aylıq olan obyektləri silir.

Avtomatik gc-ni aşağıdakı kimi manual olaraq işə sala bilərsiniz:

```
$ git gc --auto
```

Yenə də bu tam olaraq heç bir şey etmir. Git-in həqiqi bir gc əmrini işə salması üçün təxminən 7,000 boş obyekt və ya 50-dən çox paketiniz olmalıdır. Bu məhdudiyyətləri sırasıyla `gc.auto` və` gc.autopacklimit` konfiqurasiya parametrləri ilə dəyişdirə bilərsiniz.

`gc`-nin edəcəyi başqa bir şey, istinadlarınızı bir fayla yığmaqdır. Tutaq ki, deponuzda aşağıdakı branch-lar və etiketlər var:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

Əgər siz `git gc` işlədirsinizsə, artıq bu fayllar `refs` qovluğunda mövcud olmayacaq. Git bunları effektivlik naminə belə görünən `.git/packed-refs` adlı bir fayla köçürəcəkdir:

```
$ cat .git/packed-refs
# pack-refs with: peeled fully-peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

Bir istinadı yeniləsəniz, Git bu faylı redaktə etmir, əksinə `refs/heads` üçün yeni bir fayl yazır. Mü əyyən bir istinad üçün uyğun SHA-1 əldə etmək üçün Git, `refs` qovluğunda həmin istinadı yoxlayır və sonra `packed-refs` faylını geri yük kimi yoxlayır. Beləliklə, `refs` qovluğunda bir müraciət tapa bilmirsinizsə, ehtimal ki,`packed-refs` faylınızdadır.

Faylın `^` ilə başlayan son sətirinə diqqət yetirin. Bu o deməkdir ki, birbaşa yuxarıdakı etiket izahatlı bir etiketdir və həmin sətir izlənilmiş etiketin göstərdiyi commit-dir.

Data Recovery

Git səyahətinizin bir nöqtəsində təsadüfən bir commit-i itirə bilərsiniz. Ümumiyyətlə, bu, üzərində işləyən bir branch-ı zorla sildiyiniz üçün baş verir və nəticədə branch-ı istədiyinizi ortaya qoyur; ya da bir branch-ı yenidən sıfırlayırsınız, beləliklə bir şey istədiyiniz commit-dən imtina edirsiniz. Bunun baş verdiyini düşünsək, commit-lərinizi necə geri ala bilərsiniz?

Test anbarınızdakı `master` branch-ı köhnə bir commit-ə yenidən bərpa edən və sonra itirilmiş commit-ləri bərpa edən bir nümunə göstərək. Əvvəlcə bu deponuzun harada olduğunu nəzərdən keçirək:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b Modify repo a bit
484a59275031909e19aadb7c92262719cfcdf19a Create repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

İndi, `master` branch-ınızı ortadakı commit-ə köçürün:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef Third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Ən yaxşı iki səmərəli işi itirmisiniz - bu commit-lərin əldə edilə biləcəyi bir branch-nız yoxdur. Ən son SHA-1 əmrini tapmalı və sonra ona işarə edən bir budaq əlavə etməlisiniz. Hiylə budur: son SHA-1 commit-ni tapmaq - bu yadınızda qalan kimi deyil, düzdür?

Çox vaxt, ən sürətli yol `git reflog` adlı bir vasitə istifadə etməkdir. İşləyərkən, Git hər dəfə d əyişdirdiyiniz zaman HEAD-in nə olduğunu səssizcə qeyd edir. Branch-ları hər dəfə işlətdiyiniz və ya dəyişdirdiyiniz zaman reflog yenilənir. Reflog ayrıca, Git Referansları-də qeyd etdiyimiz kimi, yalnız ref fayllarınıza SHA-1 dəyərini yazmaq əvəzinə istifadə etmək üçün başqa bir səbəb olan `git update-ref` əmri ilə də yenilənir. İstədiyiniz zaman harada olduğunuzu `git reflog` çalıştırarak görə bilərsiniz:

```
$ git reflog
1a410ef HEAD@{0}: reset: moving to 1a410ef
ab1afef HEAD@{1}: commit: Modify repo.rb a bit
484a592 HEAD@{2}: commit: Create repo.rb
```

Burada yoxladığımız iki işi görə bilərik, lakin burada çox məlumat yoxdur. Eyni məlumatları daha faydalı bir şəkildə görmək üçün, `git log -g` işlədə bilərik ki, bu da reflog-unuz üçün normal bir günlük output-u təmin edəcəkdir.

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:22:37 2009 -0700

    Third commit

commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

       Modify repo.rb a bit
```

Göründüyü kimi alt commit itirdiyiniz işdir, buna görə yeni commit yaratmaqla onu bərpa edə bil ərsiniz. Məsələn, bu əməli (ab1afef) bərpa etmək üçün `recover-branch` başlada bilərsiniz:

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b Modify repo.rb a bit
484a59275031909e19aadb7c92262719cfcdf19a Create repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Əla - indi `recovery-branch` adlı bir branch-nız var, əvvəllər `master` branch-nızın olduğu yerlərdə ilk iki iş yenidən əlçatandır. Bundan sonra, itkinizin bir səbəbdən reflog-da olmadığını düşünək – bunu `recovery-branch`-nı çıxarıb təkrar qeydini silməklə simulyasiya edə bilərsiniz. İndi ilk iki

commit-ə heç bir şey çatmır:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Reflog məlumatları `.git/logs/` qovluğunda saxlanıldığından, sizdə heç bir reflog yoxdur. Bu anda bu işi necə bərpa edə bilərsiniz? Bir üsul, verilənlər bazanızın bütövlüyünü yoxlayan `git fsck` yardım proqramından istifadə etməkdir. Bunu `--full` seçimi ilə işə salırsınızsa, başqa bir obyekt t ərəfindən göstərilməyən bütün obyektləri göstərir:

```
$ git fsck --full
Checking object directories: 100% (256/256), done.
Checking objects: 100% (18/18), done.
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

Bu vəziyyətdə, “dangling commit” sətrindən sonra itkin commit-nizi görə bilərsiniz. Eyni SHA-1-ə işarə edən bir qol əlavə edərək onu eyni şəkildə bərpa edə bilərsiniz.

Obyektlərin Silinməsi

Git haqqında çox yaxşı şey var, amma bu problemə səbəb ola biləcək bir xüsusiyyətdir, yəni, bir `git clone`-un hər faylın hər versiyası da daxil olmaqla layihənin bütün tarixini yükləməsidir. Bu ə gər mənbə kodu olarsa yaxşıdır, çünki Git bu məlumatları səmərəli şəkildə sıxmaq üçün yüksək d ərəcədə optimize edilmişdir. Bununla birlikdə, layihənizin tarixinin hər hansı bir nöqtəsində kimsə tək bir nəhəng fayl əlavə etsə, bütün klonlar, sonrakı commit-lərdə layihədən çıxarılan olsa da, bütün zaman üçün bu klonu yükləmək məcburiyyətində qalacaq. Tarixdən əlçatan olduğundan h əmişə mövcud olacaq.

Subversion və ya Perforce depolarını Git-ə çevirdiyiniz zaman bu böyük bir problem ola bilər. Bu sistemlərdəki bütün tarixi yükləmədiyiniz üçün bu əlavə bir neçə nəticəyə səbəb olur. Başqa bir sistemdən idxal etmisinizsə və ya başqa bir şəkildə deponuzun lazım olduğundan daha böyük olduğunu görmünüzsə, burada böyük obyektləri necə tapa və silə biləcəyinizi görə bilərsiniz.

Xəbərdarlıq: bu texnika sizin commit tarixçəniz üçün dağıdıcıdır. O, böyük bir fayl arayışını silmək üçün dəyişdirməli olduğunuz ilk ağacdan bəri hər bir commit obyektini yenidən yazır. Bunu idxaldan dərhal sonra, kimsə işin commit üzərində qurulmasına başlamazdan əvvəl etsəniz, yaxşıdır - əks halda, bütün dəstəkçilərinizə işlərini yeni commit-lərinizə qaytarmaları lazım olduğunu bildirməlisiniz.

Nümayiş üçün test deponuza böyük bir fayl əlavə edəcək, növbəti commit-də götürüb tapacaqsınız və depodan qalıcı olaraq çıxaracaqsınız. Əvvəlcə tarixçənizə böyük bir obyekt əlavə edin:

```
$ curl https://www.kernel.org/pub/software/scm/git/git-2.1.0.tar.gz > git.tgz
$ git add git.tgz
$ git commit -m 'Add git tarball'
[master 7b30847] Add git tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tgz
```

Vay - layihənizə böyük bir tarball əlavə etmək istəmirdiniz. Ən yaxşısı bundan yaxa qurtarmaq üçün:

```
$ git rm git.tgz
rm 'git.tgz'
$ git commit -m 'Oops - remove large tarball'
[master dadf725] Oops - remove large tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

İndi, məlumat bazanızı `gc` edin və nə qədər yer istifadə etdiyinizi görün:

```
$ git gc
Counting objects: 17, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

Nə qədər yer istifadə etdiyinizi görmək üçün `count-objects` əmrini işə sala bilərsiniz:

```
$ git count-objects -v
count: 7
size: 32
in-pack: 17
packs: 1
size-pack: 4868
prune-packable: 0
garbage: 0
size-garbage: 0
```

`Size-pack` girişi paketlərinizin kilobayt ölçüsündədir, buna görə demək olar ki, 5MB istifadə edirsiniz. Son commit-dən əvvəl, 2K-a yaxın istifadə edirdiniz - açıq şəkildə, əvvəlki commit-dən faylın silinməsi tarixçənizdən silinmədi. Hər kəs bu deponu klonladıqda, yalnız bu kiçik layihəni əldə etmək üçün bütün 5 MB-ı klonlamalı olacaq, çünki siz təsadüfən böyük bir fayl əlavə etmisiniz. Gəlin bundan qurtulaq.

Əvvəlcə onu tapmaq lazımdır. Bu vəziyyətdə, bunun hansı fayl olduğunu artıq bilirsiniz. Ancaq güman etmədiniz ki; hansı fayl və ya sənədlərin bu qədər yer tutduğunu necə müəyyənləşdir

ərdiniz? `git gc`-i işlədirsinizsə, bütün obyektlər packfile-dadır; böyük obyektləri `git verify-pack` adlı başqa bir plumbing əmrini işə salmaqla və sənəddəki fayl ölçüsü olan üçüncü sahəyə görə ayırmaqla təyin edə bilərsiniz. Yalnız son bir neçə ən böyük sənədlə maraqlandığınız üçün onu `tail` əmri ilə ötürə bilərsiniz:

```
$ git verify-pack -v .git/objects/pack/pack-29…69.idx \
  | sort -k 3 -n \
  | tail -3
dadf7258d699da2c8d89b09ef6670edb7d5f91b4 commit 229 159 12
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob   22044 5792 4977696
82c99a3e86bb1267b236a4b6eff7868d97489af1 blob   4975916 4976258 1438
```

Aşağıdakı böyük obyekt: 5 MB. Hansı fayl olduğunu tapmaq üçün qısa müddətdə Xüsusi bir Commit-Mesaj Formatının Tətbiq Edilməsi-da istifadə etdiyiniz `rev-list` əmrini istifadə edəc əksiniz. `--objects`-i `rev-list`-ə keçirsəniz, bütün SHA-1'ləri və bunlarla əlaqəli fayl yolları ilə blob SHA-1'ləri siyahıya alır. Blob adınızı tapmaq üçün bundan istifadə edə bilərsiniz:

```
$ git rev-list --objects --all | grep 82c99a3
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

İndi keçmişdəki bütün ağaclardan bu faylı silməlisiniz. Bu faylı hansı commit-in dəyişdirdiyini asanlıqla görə bilərsiniz:

```
$ git log --oneline --branches -- git.tgz
dadf725 Oops - remove large tarball
7b30847 Add git tarball
```

Bu faylı Git tarixinizdən tamamilə silmək üçün `7b30847`-dən aşağıda olan bütün commit-ləri yenid ən yazmalısınız. Bunu etmək üçün Tarixi Yenidən Yazmaq-də istifadə etdiyiniz `filter-branch`-dan istifadə edirsiniz:

```
$ git filter-branch --index-filter \
  'git rm --ignore-unmatch --cached git.tgz' -- 7b30847^..
Rewrite 7b30847d080183a1ab7d18fb202473b3096e9f34 (1/2)rm 'git.tgz'
Rewrite dadf7258d699da2c8d89b09ef6670edb7d5f91b4 (2/2)
Ref 'refs/heads/master' was rewritten
```

`--index-filter` seçimi, Tarixi Yenidən Yazmaq-də istifadə olunan `--tree-filter` seçiminə bənzəyir, yalnız diskdə qeyd edilmiş sənədləri dəyişdirən bir əmr ötürmək əvəzinə, hər dəfə quruluş sah ənizi və ya indeksinizi dəyişdirirsiniz.

Müəyyən bir faylı `rm file` kimi bir şeylə silmək əvəzinə, `git rm --cached` ilə silməlisiniz - diskdən yox, indeksdən çıxarmalısınız. Bunu bu şəkildə etməyinizin səbəbi sürətdir - çünki Git filterinizi işə salmadan əvvəl diskdəki hər bir düzəlişə baxmaq məcburiyyətində qalmadığı üçün proses çox daha sürətli ola bilər. Həmin tapşırığı `--tree-filter` ilə də yerinə yetirə bilərsiniz. `git rm` seçimi

üçün `--ignore-unmatch` seçimi, silmək istədiyiniz şablonun olmadığı təqdirdə səhv etməməsini tələb edir. Nəhayət, `filter-branch`-dan tarixinizi yalnız `7b30847` commit-indən yenidən yazmasını xahiş edirsiniz, çünki bu problemin burada başladığını bilirsiniz. Əks təqdirdə, başlanğıcdan başlayacaq və lazımsız olaraq daha uzun çəkəcəkdir.

Tarixinizdə artıq həmin fayla istinad yoxdur. Bununla birlikdə reflogunuz və Git-in `.git/refs/original` altında `filter-branch` etdiyiniz zaman əlavə etdiyi yeni bir refs dəsti hələ də mövcuddur, buna görə onları silməli və sonra verilənlər bazasını yenidən paketləməlisiniz. Yenid ən qablaşdırmadan əvvəl bu köhnə commit-lərə işarə edən bir şeydən qurtulmalısınız:

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 15, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (15/15), done.
Total 15 (delta 1), reused 12 (delta 0)
```

Gəlin nə qədər yer saxladığınıza baxaq:

```
$ git count-objects -v
count: 11
size: 4904
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

Paketlənmiş depo ölçüsü 8K-a qədərdir, bu da 5MB-dən daha yaxşıdır. Ölçü dəyərindən görə bil ərsiniz ki, böyük obyekt hələ də boş obyektlərinizdədir, buna görə də getməyib; lakin vacib olan odur ki, o, push və ya sonrakı bir klon üzərinə köçürülməyəcəkdir. Həqiqətən istəsəniz, `--expire` seçimi ilə `git prune` düyməsini basaraq obyekti tamamilə silə bilərsiniz:

```
$ git prune --expire now
$ git count-objects -v
count: 0
size: 0
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

Mühit Dəyişənləri

Git həmişə bir `bash` shell-inin içərisində işləyir və necə davranacağını təyin etmək üçün bir sıra shell mühiti dəyişənlərindən istifadə edir. Bəzən bunların nə olduğunu və Git'in istədiyiniz kimi davranması üçün necə istifadə edilə biləcəyini bilmək çox faydalıdır. Bu, Git'in diqqət yetirdiyi bütün mühit dəyişənlərinin tam siyahısı deyil, ancaq biz ən faydalılarını əhatə edəcəyik.

Qlobal Davranış

Kompüter proqramı kimi Git'in bəzi ümumi davranışları ətraf mühit dəyişənlərindən asılıdır.

`GIT_EXEC_PATH`, Git'in alt proqramlarını harada axtardığını təyin edir (`git-commit`, `git-diff` və digərl əri kimi). Hazırkı ayarı `git --exec-path`-i işə salaraq yoxlaya bilərsiniz.

`HOME` ümumiyyətlə fərdiləşdirilə bilməz (Çünki, çox şey ondan asılıdır), ancaq Git-in qlobal konfiqurasiya faylını axtardığı yerdir. Qlobal konfiqurasiya ilə tamamlanmış, həqiqətən portativ bir Git quraşdırma istəyirsinizsə, portativ Git-in shell profilindəki `HOME` seçimini ləğv edə bilərsiniz.

`PREFIX` bənzərdir, lakin sistem miqyasında konfiqurasiya üçündür. Git bu faylı `$PREFIX/etc/gitconfig`-də axtarır.

`GIT_CONFIG_NOSYSTEM`, ayarlandığı təqdirdə, sistem səviyyəsində konfiqurasiya faylının istifadəsini dayandırır. Sistem konfiqurasiyanız əmrlərinizə müdaxilə edirsə, dəyişdirmək və ya silmək üçün girişiniz yoxdursa, onda bu faydalı olacaqdır.

`GIT_PAGER` əmr sətrində çox səhifəlik çıxışı göstərmək üçün istifadə olunan proqramı idarə edir. Bu ayarlanmayıbsa, `PAGER` geri dönüş kimi istifadə ediləcək.

`GIT_EDITOR`, istifadəçinin bəzi mətnləri düzəltməsi lazım olduqda (məsələn, commit mesajı) Git-in işə salacağı redaktordur. Ayarlanmadıqda isə, `EDİTOR` istifadə ediləcək.

Depo Yerləri

Git, cari depo ilə necə əlaqəli olduğunu müəyyən etmək üçün bir neçə mühit dəyişənlərindən istifadə edir.

`GIT_DIR`, `.git` qovluğunun yeridir. Bu göstərilməyibsə, Git hər bir addımda `.git` qovluğu axtararaq, `~` və ya `/`-a çatana qədər qovluq ağacını gəzir.

`GIT_CEILING_DIRECTORIES`, `.git` qovluğu axtarma davranışına nəzarət edir. Yüklənməsi ləng olan qovluqlara (məsələn, bir lent sürücüsündə və ya yavaş bir şəbəkə bağlantısı daxilində) daxil olsanız, xüsusilə shell istəyinizi qurarkən Git çağırıldığı təqdirdə, Git-dən başqa bir müddətdən ə vvəl sınamağı dayandırmaq istəyə bilərsiniz.

`GIT_WORK_TREE` boş olmayan bir depo üçün iş qovluğunun kökünün yerləşməsidir. Əgər `--git-dir` və ya `GIT_DIR` göstərilsə də, `--work-tree`, `GIT_WORK_TREE` və ya `core.worktree` qeyd edilmirsə, cari iş qovluğu iş ağacınızın ən üst səviyyəsi sayılır.

`GIT_INDEX_FILE` indeks faylına gedən path-dır (yalnız boş olayan depolarda).

`GIT_OBJECT_DIRECTORY` ümumiyyətlə `.git/objects`-də yerləşən qovluğun yerini təyin etmək üçün istifadə edilə bilər.

`GIT_ALTERNATE_OBJECT_DIRECTORIES` Git-ə `GIT_OBJECT_DIRECTORY`-də olmadıqda obyektlərin harada yoxlanılacağını bildirən iki nöqtə ilə ayrılmış bir siyahıdır (`/dir/one:/dir/two:... ` kimi formatlaşdırılmışdır). Tamamilə eyni məzmuna sahib olan böyük faylları olan bir çox layihəniz olursa, bunların çox nüsxəsini saxlamamaq üçün istifadə edilə bilər.

Pathspec-lər

Bir `pathspec`, wildcards istifadəsi də daxil olmaqla Git-dəkilərə path-ları necə təyin etdiyinizə aiddir. Bunlar həm `.gitignore` faylında, həm də əmr sətrində (` git add * .c `) istifadə olunur.

`GIT_GLOB_PATHSPECS` və `GIT_NOGLOB_PATHSPECS` wildcards-ın pathspecs-dəki standart davranışını idarə edir. Əgər *GIT_GLOB_PATHSPECS* 1 olaraq ayarlanırsa, wildcard simvolları wildcards rolunu oynayır (standart olaraq); `GIT_NOGLOB_PATHSPECS` 1-ə ayarlanmışsa, joker simvollar yalnız özləri ilə uyğunlaşır, yəni `*.c` adları `.c` ilə bitən hər hansı bir faylla deyil, yalnız *named* ``* .c" faylına uyğun gəlir. Bunu ayrı hallarda `:(glob)*.c`-də olduğu kimi `:(glob)` və ya `:(hərfi)` ilə başlayan path işar əsini ləğv edə bilərsiniz. `GIT_LITERAL_PATHSPECS` yuxarıdakı davranışların hər ikisini deaktiv edir; heç bir wildcard simvolu işləməyəcək və yalnış prefikslər də söndürülür.

`GIT_ICASE_PATHSPECS` bütün path xüsusiyyətlərini işlərə həssas olmayan bir şəkildə işləməyə quraşdırır.

Committing

Git commit obyektinin son yaradılması, ümumiyyətlə bu mühit dəyişənlərini əsas məlumat mənb əyi kimi istifadə edən və yalnız mövcud olmadıqda konfiqurasiya dəyərlərinə qayıdan `git-commit-tree` tərəfindən edilir.

`GIT_AUTHOR_NAME` ‘`author '' sahəsindəki insan tərəfindən oxuna bilən addır.

`GIT_AUTHOR_EMAIL` “author” sahəsi üçün e-poçtdur.

`GIT_AUTHOR_DATE` “author” sahəsi üçün istifadə olunan zaman damğasıdır.

`GIT_COMMITTER_NAME` “committer” sahəsi üçün insan adı təyin edir.

`GIT_COMMITTER_EMAIL` “committer” sahəsi üçün e-poçtdur.

`GIT_COMMITTER_DATE` “committer” sahəsi üçün zaman damğasıdır.

`EMAIL` `user.email` konfiqurasiya dəyərinin təyin edilməməsi halında geri göndərilən e-poçt adresidir. *this* ayarlanmadıqda, Git sistem istifadəçisi və host adlarına qayıdır.

Networking

Git, HTTP üzərindən şəbəkə əməliyyatları aparmaq üçün `curl` kitabxanasından istifadə edir, buna görə də `GIT_CURL_VERBOSE` Git-ə həmin kitabxana tərəfindən yaradılan bütün mesajları buraxmasını söyləyir. Bu, əmr sətrində `curl –v` etməyə oxşayır.

GIT_SSL_NO_VERIFY Git-ə SSL sertifikatlarının təstiqlənməsini söyləyir. HTTPS üzərindən Git depolarına xidmət göstərmək üçün öz imzanızla təsdiqlənmiş bir sertifikat istifadə edirsinizsə və ya bir Git server quraşdırmasının ortasındasnızsa, lakin hələ tam bir sertifikat yükləməmisinizsə, bu bəzən lazım ola bilər.

Bir HTTP əməliyyatının məlumat dərəcəsi saniyədə GIT_HTTP_LOW_SPEED_TIME saniyədən uzun müddət ərzində GIT_HTTP_LOW_SPEED_LIMIT baytdan aşağı olarsa, Git bu əməliyyatı ləğv edəcəkdir. Bu dəyərlər http.lowSpeedLimit və` http.lowSpeedTime` konfiqurasiya dəyərlərini ləğv edir.

GIT_HTTP_USER_AGENT 12, HTTP üzərindən əlaqə qurarkən Git tərəfindən istifadə edilən istifadəçi agent sətirini təyin edir. Standartlıq git / 2.0.0 kimi bir dəyərdir.

Diffing and Birləşdirmə

GIT_DIFF_OPTS bir az səhvdir. Yalnız etibarlı dəyərlər, git diff əmrində göstərilən kontekst sətirl ərinin sayını idarə edən -u <n> və ya '--unified = <n>'-dir.

GIT_EXTERNAL_DIFF diff.external konfiqurasiya dəyəri üçün yalnış olaraq istifadə olunur. Ayarlandığı təqdirdə, Git, git diff çağırıldıqda bu proqramı çağıracaqdır.

GIT_DIFF_PATH_COUNTER və GIT_DIFF_PATH_TOTAL GIT_EXTERNAL_DIFF və ya diff.external tərəfindən t əyin olunan proqramın içərisindən faydalıdır. Birincisi, bir seriyadakı hansı faylın fərqləndiyini göstərir (1-dən başlayaraq), ikincisi isə topludakı faylların ümumi sayını təşkil edir.

GIT_MERGE_VERBOSITY rekursiv birləşmə strategiyası üçün output-a nəzarət edir. İcazəli dəyərlər aşağıdakılardır:

- 0, ehtimal ki, tək bir səhv mesajı xaricində heç bir şey çıxarmaz.
- 1 yalnız konfliktləri göstərir.
- 2 də fayl dəyişikliklərini göstərir.
- 3 fayl dəyişmədiyi üçün skip olunduğunu göstərir.
- 4 işləndikcə bütün path-ları göstərir.
- 5 və yuxarısı ətraflı debugging məlumatlarını göstərir.

Standart dəyər 2-dir. ==== Debugging

really Git-in nə etdiyini bilmək istəyirsiniz? Git-in içərisində kifayət qədər əksiksiz bir iz var və yalnız bunları işə salmaq lazımdır. Bu dəyişənlərin mümkün qiymətləri aşağıdakılardır:

*``true", 1" və ya `2" - iz kateqoriyası stderr-ə yazılır. * "/ İlə başlayan mütləq path - trace output-u həmin fayla yazılacaqdır.

GIT_TRACE xüsusi bir kateqoriyaya yerləşməyən ümumi izləri idarə edir. Buraya taxma adların genişləndirilməsi və digər alt proqramlara nümayəndəlik daxildir.

```
$ GIT_TRACE=true git lga
20:12:49.877982 git.c:554               trace: exec: 'git-lga'
20:12:49.878369 run-command.c:341       trace: run_command: 'git-lga'
20:12:49.879529 git.c:282               trace: alias expansion: lga => 'log' '--graph'
'--pretty=oneline' '--abbrev-commit' '--decorate' '--all'
20:12:49.879885 git.c:349               trace: built-in: git 'log' '--graph' '--
pretty=oneline' '--abbrev-commit' '--decorate' '--all'
20:12:49.899217 run-command.c:341       trace: run_command: 'less'
20:12:49.899675 run-command.c:192       trace: exec: 'less'
```

`GIT_TRACE_PACK_ACCESS` packfila-a girişin izlənilməsinə nəzarət edir. Birinci sahə əldə edilən packfile-dir, ikincisi həmin fayl içindəki offsetdir:

```
$ GIT_TRACE_PACK_ACCESS=true git status
20:10:12.081397 sha1_file.c:2088        .git/objects/pack/pack-c3fa...291e.pack 12
20:10:12.081886 sha1_file.c:2088        .git/objects/pack/pack-c3fa...291e.pack 34662
20:10:12.082115 sha1_file.c:2088        .git/objects/pack/pack-c3fa...291e.pack 35175
# […]
20:10:12.087398 sha1_file.c:2088        .git/objects/pack/pack-e80e...e3d2.pack
56914983
20:10:12.087419 sha1_file.c:2088        .git/objects/pack/pack-e80e...e3d2.pack
14303666
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

`GIT_TRACE_PACKET` şəbəkə əməliyyatları üçün paket səviyyəsində izləmə imkanı verir.

```
$ GIT_TRACE_PACKET=true git ls-remote origin
20:15:14.867043 pkt-line.c:46           packet:          git< # service=git-upload-
pack
20:15:14.867071 pkt-line.c:46           packet:          git< 0000
20:15:14.867079 pkt-line.c:46           packet:          git<
97b8860c071898d9e162678ea1035a8ced2f8b1f HEAD\0multi_ack thin-pack side-band side-
band-64k ofs-delta shallow no-progress include-tag multi_ack_detailed no-done
symref=HEAD:refs/heads/master agent=git/2.0.4
20:15:14.867088 pkt-line.c:46           packet:          git<
0f20ae29889d61f2e93ae00fd34f1cdb53285702 refs/heads/ab/add-interactive-show-diff-func-
name
20:15:14.867094 pkt-line.c:46           packet:          git<
36dc827bc9d17f80ed4f326de21247a5d1341fbc refs/heads/ah/doc-gitk-config
# […]
```

`GIT_TRACE_PERFORMANCE` performans məlumatlarının qeyd edilməsinə nəzarət edir. Output, hər bir xüsusi `git` çağırışının nə qədər çəkdiyini göstərir.

```
$ GIT_TRACE_PERFORMANCE=true git gc
20:18:19.499676 trace.c:414             performance: 0.374835000 s: git command: 'git'
'pack-refs' '--all' '--prune'
20:18:19.845585 trace.c:414             performance: 0.343020000 s: git command: 'git'
'reflog' 'expire' '--all'
Counting objects: 170994, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (43413/43413), done.
Writing objects: 100% (170994/170994), done.
Total 170994 (delta 126176), reused 170524 (delta 125706)
20:18:23.567927 trace.c:414             performance: 3.715349000 s: git command: 'git'
'pack-objects' '--keep-true-parents' '--honor-pack-keep' '--non-empty' '--all' '--
reflog' '--unpack-unreachable=2.weeks.ago' '--local' '--delta-base-offset'
'.git/objects/pack/.tmp-49190-pack'
20:18:23.584728 trace.c:414             performance: 0.000910000 s: git command: 'git'
'prune-packed'
20:18:23.605218 trace.c:414             performance: 0.017972000 s: git command: 'git'
'update-server-info'
20:18:23.606342 trace.c:414             performance: 3.756312000 s: git command: 'git'
'repack' '-d' '-l' '-A' '--unpack-unreachable=2.weeks.ago'
Checking connectivity: 170994, done.
20:18:25.225424 trace.c:414             performance: 1.616423000 s: git command: 'git'
'prune' '--expire' '2.weeks.ago'
20:18:25.232403 trace.c:414             performance: 0.001051000 s: git command: 'git'
'rerere' 'gc'
20:18:25.233159 trace.c:414             performance: 6.112217000 s: git command: 'git'
'gc'
```

`GIT_TRACE_SETUP` Git-in ünsiyyət qurduğu mühit və mühit haqqında nəyi kəşf etdiyi barədə məlumat göstərir.

```
$ GIT_TRACE_SETUP=true git status
20:19:47.086765 trace.c:315             setup: git_dir: .git
20:19:47.087184 trace.c:316             setup: worktree: /Users/ben/src/git
20:19:47.087191 trace.c:317             setup: cwd: /Users/ben/src/git
20:19:47.087194 trace.c:318             setup: prefix: (null)
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Miscellaneous

`GIT_SSH`, göstərildiyi təqdirdə, Git bir SSH host-na qoşulmağa çalışarkən `ssh` yerinə çağırılan bir proqramdır. `$ GIT_SSH [istifadəçi adı @] host [-p <port>] <command>` kimi çağırılır. Qeyd edək ki, bu, `ssh`-in necə çağırılacağını fərdiləşdirməyin ən asan yolu deyil; əlavə komanda sətri parametrl ərini dəstəkləməyəcək, buna görə bir wrapper ssenarisi yazmalı və işarə etmək üçün `GIT_SSH` ayarlamalısınız. Bunun üçün sadəcə `~/.ssh/config` faylını istifadə etmək daha asandır.

`GIT_ASKPASS` `core.askpass` konfiqurasiya dəyəri üçün keçiddir. Bu, Git-in istifadəçidən bir əmr sətri arqumenti kimi bir mətn istəməsini gözləyə bilən creditentals tələb etməsi lazım olan və `stdout`-da cavabı qaytarmalı olduğu zaman çağırılan proqramdır (bu alt sistem haqqında daha çox məlumat üçün buraya baxın: Etibarlı Yaddaş).

`GIT_NAMESPACE` ad boşluğuna daxil olan ref-lərə girişə nəzarət edir və `--namespace` bayrağına bərab ərdir. Bu əsasən server tərəfində faydalıdır, burada tək bir depoda birdən çox hook saxlaya bil ərsiniz, yalnız ref-ləri ayrı saxlamalısınız. `GIT_FLUSH` Git-i stdout-a tədricən yazarkən tamponlanmamış I/O istifadə etməyə məcbur etmək üçün istifadə edilə bilər. 1 dəyəri Git-in daha tez-tez yuyulmasına, 0 dəyəri bütün çıxışın tamponlanmasına səbəb olur. Standart dəyər (bu dəyiş ən təyin edilməyibsə), fəaliyyətə və çıxış rejiminə görə uyğun bir tamponlama sxemi seçməkdir.

`GIT_REFLOG_ACTION` reflog-a yazılmış təsviri mətni təyin etməyə imkan verir. Misal olaraq:

```
$ GIT_REFLOG_ACTION="my action" git commit --allow-empty -m 'My message'
[master 9e3d55a] My message
$ git reflog -1
9e3d55a HEAD@{0}: my action: My message
```

Qısa Məzmun

Bu nöqtədə Git'in arxa planda nə etdiyini və bir dərəcədə necə tətbiq olunduğunu olduqca yaxşı başa düşməlisiniz. Bu fəsildə bir sıra plumbing əmrləri - kitabın qalan hissəsində öyrəndiyiniz porcelain əmrlərdən daha aşağı səviyyəli və daha sadə əmrlər verilib. Git'in daha aşağı səviyyədə necə işlədiyini başa düşmək, nə üçün etdiyini başa düşməyi asanlaşdırmalı və eyni zamanda öz iş axınınızı sizin üçün uyğunlaşdırmaq üçün öz alətlərinizi və köməkçi skriptlərinizi yazmalısınız.

Məzmununa yönəldilə bilən bir fayl sistemi olaraq Git, sadəcə bir VNS-dən çox asanlıqla istifadə edə biləcəyiniz çox güclü bir vasitədir. Bu texnologiyanı öz sərbəst tətbiq etmənizi təmin etmək və Git'i daha inkişaf etmiş üsullarla istifadə edərək daha rahat hiss etmək üçün Git daxili biliklərinizi yeni istifadə edə biləcəyinizə ümid edirəm.

Appendix A: Digər Mühitlərdə Git

Kitabın hamısını oxumusunuzsa, əmr sətrində Git'in necə istifadə ediləcəyi barədə çox şey öyr əndiniz. Lokal fayllarla işləyə, deponuzu bir şəbəkə vasitəsilə başqalarına bağlaya və başqaları ilə səmərəli işləyə bilərsiniz. Ancaq hekayə bununla bitmir; Git ümumiyyətlə daha böyük bir ekosistemin bir hissəsi kimi istifadə olunur və terminal həmişə onunla işləmək üçün ən yaxşı yol deyil. İndi Git'in faydalı ola biləcəyi bəzi digər mühitlərə və digər tətbiqetmələrin (sizin də daxil olmaqla) Git ilə necə işlədiyinə nəzər salacağıq.

Qrafik interfeyslər

Gitin yerli mühiti terminaldadır. Əvvəlcə yeni xüsusiyyətlər ortaya çıxır və yalnız komanda xə ttində Gitin tam gücü tamamilə sizin ixtiyarınızdadır. Ancaq düz mətn bütün tapşırıqlar üçün ən yaxşı seçim deyil; bəzən vizual bir nümayəndəlik sizə lazım olan şeydir və bəzi istifadəçilər point-and-click interfeysi ilə daha rahatdırlar.

Fərqli interfeyslərin fərqli iş axınlarına uyğunlaşdırıldığını qeyd etmək vacibdir. Bəzi müştərilər müəllifin effektiv hesab etdiyi xüsusi bir iş üsulunu dəstəkləmək üçün yalnız diqqətlə hazırlanmış Git funksionallığının alt hissəsini ifşa edirlər. Bu işığa baxıldıqda, bu vasitələrin heç birinə digərl ərindən "daha yaxşı" olduğu deyilə bilməz, sadəcə məqsədləri üçün daha uyğun olurlar. Bu qrafik müştərilərin əmr sətri müştərisinin edə bilməyəcəyi bir şey olmadığına da diqqət yetirin; əmr sətri, depolarınızla işləyərkən ən çox gücə və nəzarətə sahib olacağınız yerdir.

`gitk` və `git-gui`

Git'i quraşdırdığınızda, vizual alətləri, `gitk` və `git-gui`-də əldə edirsiniz. `gitk` qrafik tarixçəsi görüntüləyicisidir. Bunu `git log` və `git grep` üzərində güclü bir GUI shell-i kimi düşünün. Keçmişdə baş verən bir şey tapmağa çalışarkən istifadə etdiyiniz və ya layihənizin tarixini görselleştiren bu vasitədir.

Gitk-i əmr sətrindən çağırmaq asandır. Yalnız bir Git deposuna `cd` yazın:

```
$ gitk [git log options]
```

Gitk, bir çox komanda xətti seçimini qəbul edir, bunların əksəriyyəti əsas `git log` əməliyyatına ötürülür. Bunlardan ən çox faydalı olan flag-larından biri `--all`-dur, hansı ki gitk-in yalnız HEAD deyil, *hər* ref-dən əldə edilə bilən işləri göstərməsini söyləyir.. Gitk interfeysi belə görünür:

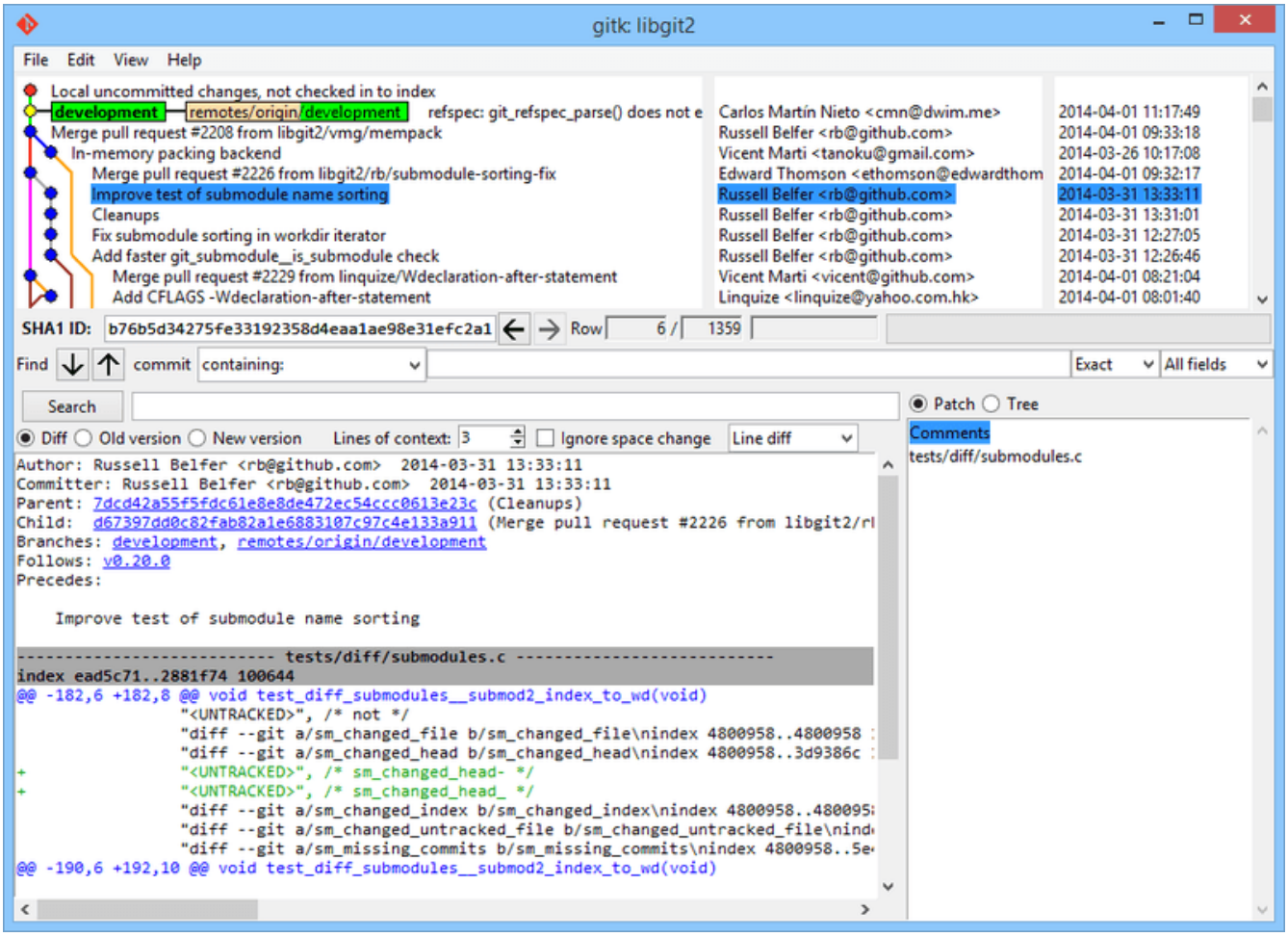


Figure 152. `gitk` tarixçəsi

Yuxarıda bir az `git log --graph` çıxışına bənzəyən bir şey var; hər nöqtə bir commit-i, sətirlər valideyn münasibətlərini və reflər rəngli qutular şəklində göstərilir.

Sarı nöqtə HEAD-i, qırmızı nöqtə isə commit-ə çevrilməli olan dəyişiklikləri təmsil edir. Aşağıda seçilmiş commit-in görünüşü var; sol tərəfdəki şərhlər və patch, sağdakı summary görünüşü. Arada tarix axtarmaq üçün istifadə edilən nəzarət toplusu var. Digər tərəfdən, `git-gui`, ilk növbədə commit-lərin hazırlanması üçün bir vasitədir. Əmr sətrindən çağırmaq da asandır:

```
$ git gui
```

Və bu belə görünür:

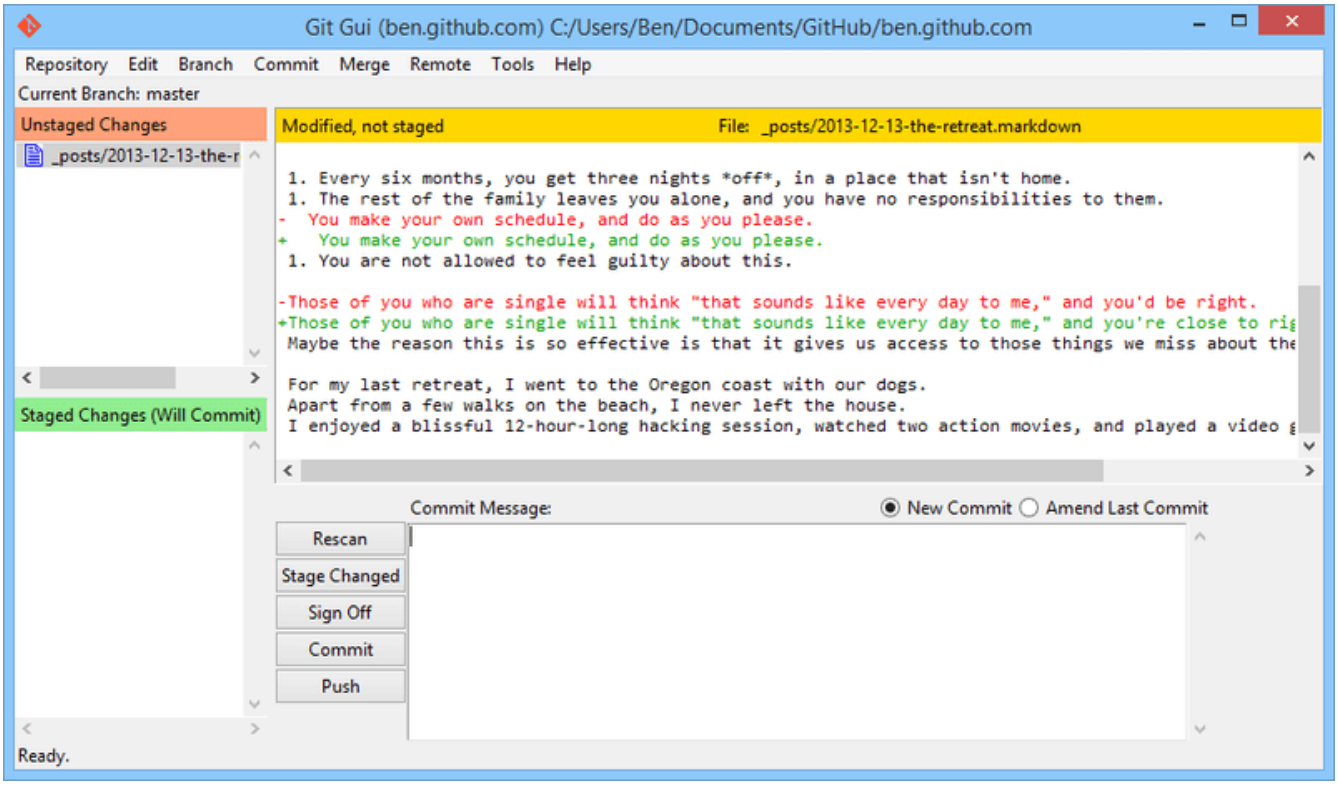


Figure 153. `git-gui` commit tool

Solda indeks var; unstaged dəyişikliklər üstdə, staged dəyişikliklər altındadır. İconlara click edərək iki vəziyyət arasında bütün faylları hərəkət etdirə və ya adını click edərək görüntüləmək üçün bir sənəd seçə bilərsiniz.

Sağ üstdə hazırda seçilmiş fayl üçün dəyişiklikləri göstərən diff görünüşü var. Bu ərazidə sağ click edərək fərdi ovları (və ya fərdi xətləri) səhnələşdirə bilərsiniz.

Sağ altda mesaj və action sahəsi var. Mesajınızı mətn qutusuna yazın və `git commit`-ə bənzər bir şey etmək üçün “Commit” düyməsini basın. Ayrıca, “Staged Changes” sahəsini son commit-in məzmunu ilə yeniləyəcək olan “Amend” radio düyməsini seçərək son commit-ə düzəliş edə bilərsiniz. Sonra sadəcə bəzi dəyişiklikləri stage və ya unstage, commit mesajını dəyişdirə və köhnə commit-i yenisi ilə əvəz etmək üçün yenidən “Commit” düyməsinə basa bilərsiniz.

`gitk` və `git-gui` task-oriented vasitələrin nümunələridir. Hər biri müəyyən bir məqsəd üçün hazırlanmışdır (müvafiq olaraq tarixə baxmaq və commit-lər yaratmaq) və bu tapşırıq üçün lazım olmayan xüsusiyyətləri buraxmaq.

MacOS və Windows üçün GitHub

GitHub, iki iş axını yönümlü Git müştəri yaratdı: biri Windows, digəri macOS üçün. Bu müştərilər iş axını yönümlü alətlərin yaxşı bir nümunəsidir - Git-in işləkliyini ortaya qoymaqdansa, əksinə, birlikdə işləyən, geniş yayılmış istifadə olunan xüsusiyyətlərə diqqət yetirirlər. Bunlar belə görünür:

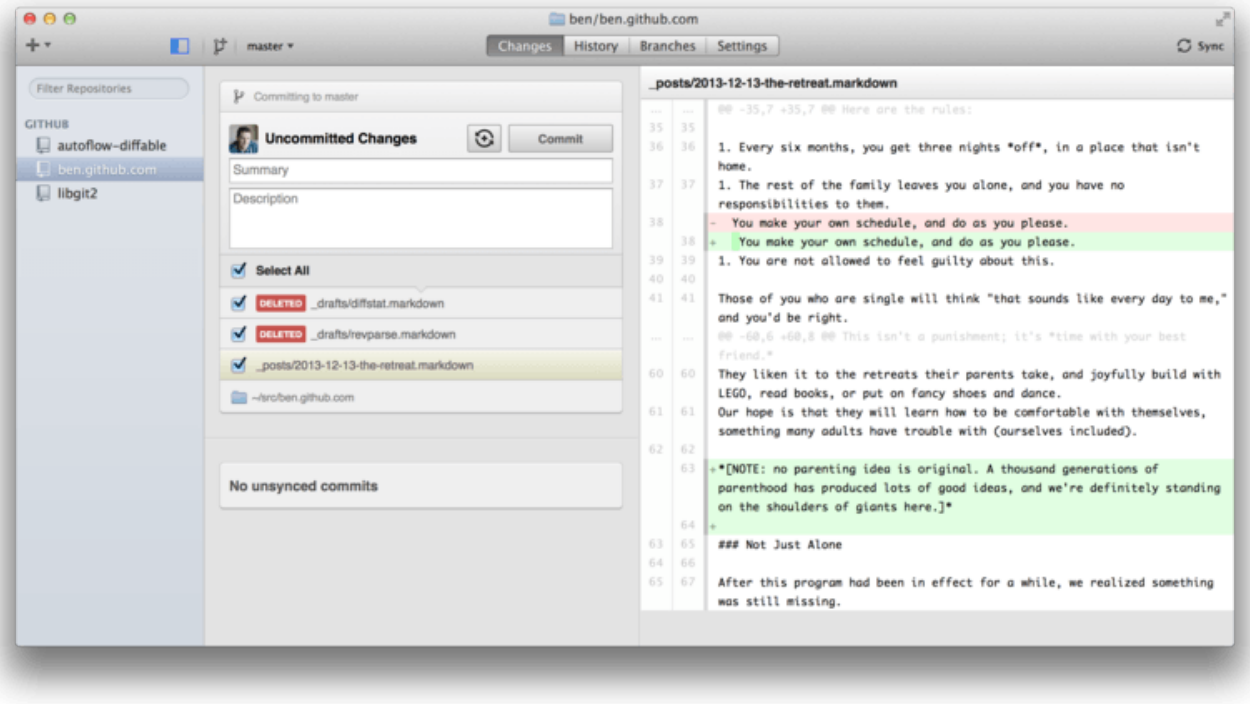


Figure 154. GitHub macOS üçün

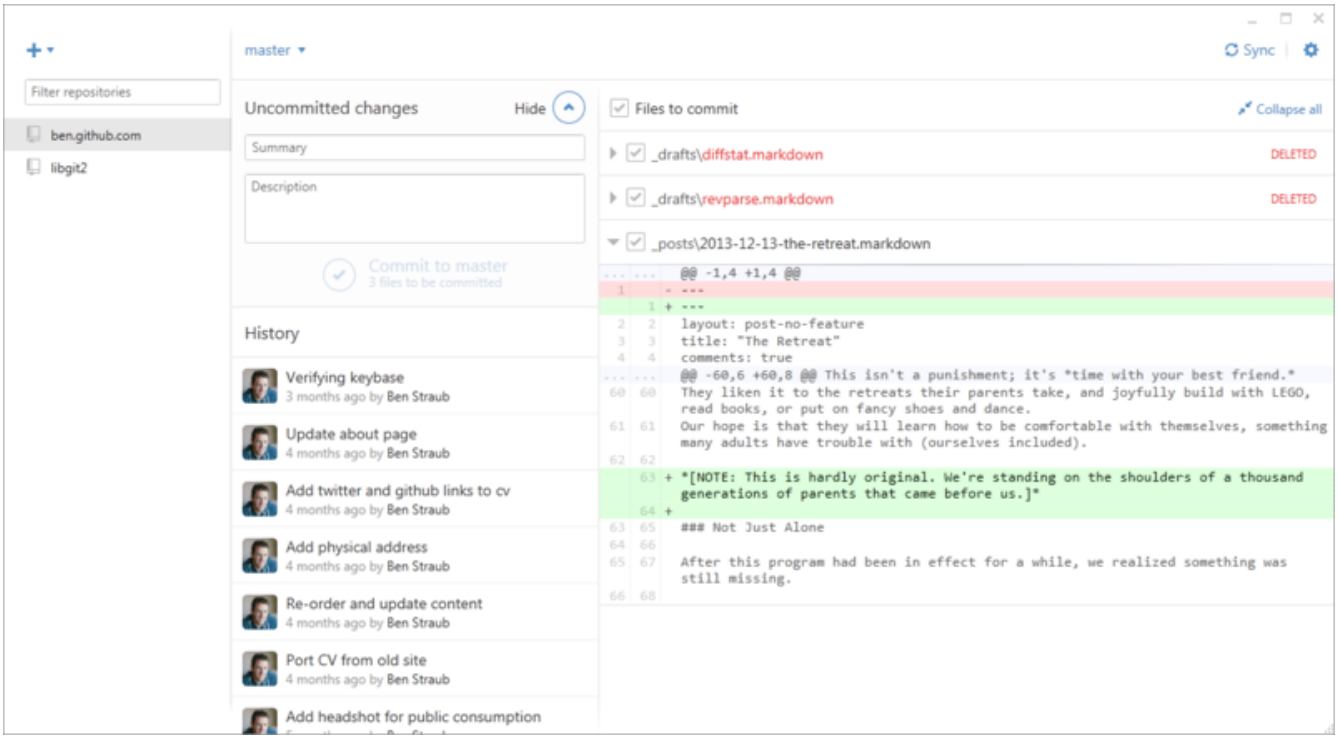


Figure 155. GitHub Windows üçün

Çox oxşar görünmək və işləmək üçün hazırlanmışdır, buna görə bu hissədə onları bir məhsul kimi nəzərdən keçirəcəyik. Bu alətlər barədə ətraflı bir baxış keçirməyəcəyik (öz sənədləri var), ancaq "dəyişikliklər" görünüşünə (vaxtınızın çox hissəsini sərf edəcəyiniz) sürətli bir tura çıxmağa dəyər.

- Solda müştərinin izlədiyi depoların siyahısı; bu sahənin yuxarı hissəsindəki “+” iconunu click edərək (yerli olaraq klonlama və ya əlavə etməklə) bir depo əlavə edə bilərsiniz.
- Mərkəzdə bir commit mesajı daxil etməyə imkan verən və hansı faylların daxil edilməli olduğunu seçən bir commit-input sahəsi var. Windows-da, commit tarixçəsi birbaşa bunun

altında göstərilir; macOS-da ayrı bir bölmədə var.

- Sağda iş qovluğunda nəyin dəyişdirildiyini və ya seçilmiş commit-ə hansı dəyişikliklərin daxil edildiyini göstərən fərqli bir görünüş var.
- Diqqət yetirən son şey, şəbəkə üzərindəki qarşılıqlı əlaqənin əsas üsulu olan yuxarı sağdakı "Sync" düyməsidir.

Bu alətlərdən istifadə etmək üçün bir GitHub hesabına ehtiyacınız yoxdur. GitHub xidmətini və tövsiyə olunan iş axınını vurğulamaq üçün nəzərdə tutulsa da, hər hansı bir depo ilə məmnuniyyətlə işləyəcək və istənilən Git host ilə şəbəkə əməliyyatları aparacaqlar.

Quraşdırma

Windows üçün GitHub, https://windows.github.com, MacOS üçün GitHub isə https://mac.github.com saytından yüklənə bilər. Tətbiqlər ilk dəfə işə salındıqda adınızı və e-poçt adresinizi konfiqurasiya etmək kimi ilk dəfə Git quraşdırılmasını həyata keçirir və hər ikisi credential caches və CRLF davranışı kimi bir çox ümumi konfiqurasiya variantları üçün ağlabatan standart parametrləri quraşdırırlar.

Hər ikisi də "evergreen"-dır - yeniləmələr yüklənir və tətbiqlər açıq vəziyyətdə arxa planda quraşdırılır. Bu, faydalı şəkildə Git-in paketli bir versiyasını ehtiva edir, yəni manual yeniləməkdən narahat olmayacağınız mənasını verir. Windows-da müştəri PowerShell-i Posh-git ilə işə salmaq üçün bir qısayol daxildir və bu bölmədə daha sonra danışacağıq.

Növbəti addım alətə işləmək üçün bəzi depolar verməkdir. Müştəri sizə GitHub'da girə biləcəyiniz depoların siyahısını göstərir və onları bir addımda klonlaya bilər. Yerli bir deposunuz varsa, yalnız qovluğunu Finder və ya Windows Explorer-dən GitHub müştəri pəncərəsinə sürükləyin və soldakı depolar siyahısına daxil ediləcək.

Recommended Workflow

Quraşdırıldıqdan və konfiqurasiya edildikdən sonra, bir çox ümumi Git tapşırıqları üçün GitHub müştərisini istifadə edə bilərsiniz. Bu alət üçün nəzərdə tutulan iş axını bəzən "GitHub Flow" adlanır. Bunu GitHub Axını-də daha ətraflı şəkildə izah edirik, lakin ümumi fikir budur ki, (a) bir branch-a sadiq qalacaqsınız və (b) bir uzaqdan idarəedici cihazla sinxronizasiya olunacaqsınız.

Branch management iki vasitənin ayrıldığı sahələrdən biridir. MacOS-da, yeni bir branch yaratmaq üçün pəncərənin yuxarı hissəsində bir düymə var:

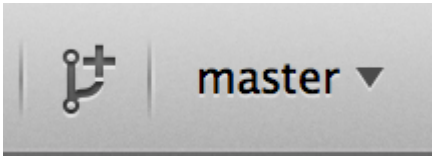


Figure 156. "Create Branch" button macOS-da

Windows-da, bu branch dəyişən widget-a yeni branch-ın adını yazaraq edilir:

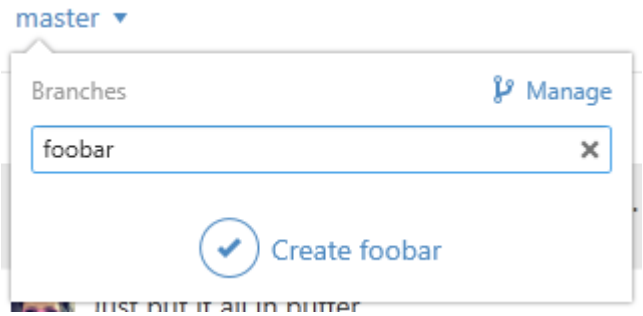


Figure 157. Windows-da bir branch yaratmaq

Branch-ınız yaradıldıqdan sonra yeni commit-lər götürmək kifayət qədər sadədir. İş qovluğunuzda bəzi dəyişikliklər edin və GitHub müştəri pəncərəsinə keçdiyiniz zaman hansı sənədlərin d əyişdirildiyini sizə göstərəcəkdir. Commit mesajı daxil edin, daxil etmək istədiyiniz sənədləri seçin və “Commit” düyməsini basın (ctrl-enter və ya -enter).

Şəbəkə üzərindəki digər depolar ilə qarşılıqlı əlaqənizin əsas yolu “Sync” xüsusiyyətidir. Git daxili olaraq pushing, fetching, merging, and rebasing üçün ayrı əməliyyatlara sahibdir, lakin GitHub müştəriləri bunların hamısını çoxpilləli bir xüsusiyyətə çevirir. Sinxronizasiya düyməsini basdıqda nə baş verir:

1. `git pull --rebase`. Birləşmə konflikti səbəbindən bu uğursuz olarsa, `git pull --no-rebase` v əziyyətinə qayıdın.
2. `git push`.

Bu üslubda işləyərkən şəbəkə əmrlərinin ən çox yayılmış ardıcıllığıdır, buna görə onları bir əmrdə sıxmaq çox vaxta qənaət edir.

Qısa Məzmun

Bu alətlər, dizayn etdikləri iş axını üçün çox uyğundur. Developers və non-developers bir neçə d əqiqə ərzində bir layihədə əməkdaşlıq edə bilərlər və bu cür iş axını üçün ən yaxşı təcrübələrin çoxu alətlərə bişirilir. Developers and non-developers alike can be collaborating on a project within minutes, and many of the best practices for this kind of workflow are baked into the tools. Bununla birlikdə, iş axınınız fərqlidirsə və ya şəbəkə əməliyyatlarının necə və nə vaxt ediləcəyinə daha çox nəzarət etmək istəyirsinizsə, başqa bir müştəri və ya əmr satırından istifadə etməyinizi məsləhət görürük.

Başqa GUI-lər

Bir sıra başqa qrafik Git müştəriləri var və onlar Gitin edə biləcəyi hər şeyi ortaya qoymağa çalışan tətbiqetmələrə qədər xüsusi, tək məqsədli vasitələrdən istifadə edirlər. Rəsmi Git veb saytında https://git-scm.com/downloads/guis saytında ən populyar müştərilərin seçilmiş siyahısı var. Daha əhatəli bir siyahı Git wiki saytında, https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools#Graphical_Interfaces saytında mövcuddur.

Visual Studio’da Git

Visual Studio 2013 Update 1-dən başlayaraq, Visual Studio istifadəçiləri birbaşa IDE-də

quraşdırılmış bir Git Müştərisinə sahibdirlər. Visual Studio bir müddətdir mənbə nəzarəti inteqrasiya xüsusiyyətlərinə malikdir, lakin mərkəzləşdirilmiş, fayl kilidləmə sistemlərinə yön əldilmişdir və Git bu iş axını üçün yaxşı uyğun gəlmədi. Visual Studio 2013-un Git dəstəyi bu köhnə xüsusiyyətdən uzaqlaşdı və nəticə Studio ilə Git arasında daha yaxşı bir uyum oldu.

Xüsusiyyəti tapmaq üçün Git tərəfindən idarə olunan bir layihəni açın (və ya sadəcə mövcud bir layihəyə daxil olun) və menyudan View > Team Explorer seçin. Bir az belə görünən "Connect" görünüşünü görəcəksiniz:

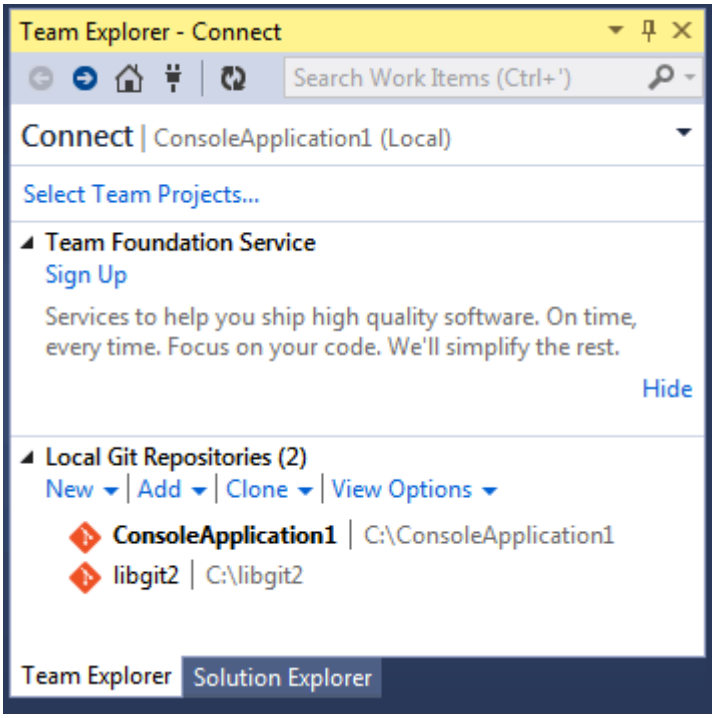


Figure 158. Team Explorer-dən Git deposuna qoşulma

Visual Studio, açdığınız və Git nəzarətində olan bütün layihələri xatırlayır və altındakı siyahıda mövcuddur. Orada istədiyinizi görmürsənsə, "Add" link-ni vurun və iş qovluğunun path-nı yazın. Local Git depolarında birinə double clicking sizi Visual Studio'da bir Git deposu üçün "Ev" görünüşü kimi görünən Ev görünüşünə aparır.

Bu, Git action-ları həyata keçirmək üçün bir mərkəzdir; kod yazarkən, yəqin ki, vaxtınızın çox hiss əsini "Changes" görünüşündə keçirəcəksiniz, ancaq komanda yoldaşlarınızın etdikləri dəyişiklikləri geri çəkmə vaxtı gəldikdə, "Unsynced Commits" və "Branches"-dan istifadə edəcəksiniz.

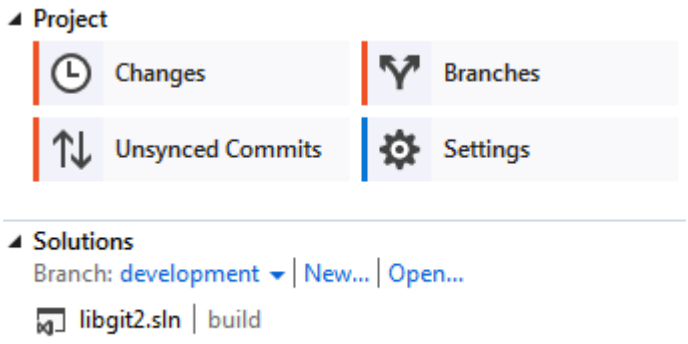


Figure 159. Visual Studio'da bir Git deposu üçün "Ev" görünüşü

Visual Studio artıq Git üçün güclü bir iş odaklı bir interfeysə sahibdir. Xətti tarix görünüşü, diff

görüntüləyici, uzaqdan əmrlər və bir çox digər imkanlar daxildir.

Visual Studio daxilində Git istifadəsi haqqında daha çox məlumat üçün: https://docs.microsoft.com/en-us/azure/devops/repos/git/command-prompt?view=azure-devops.

Visual Studio Code'da Git

Visual Studio Kodunda quraşdırılmış git dəstəyi var. Git versiyası 2.0.0 (və ya daha yeni) quraşdırılmış olmalıdır.

Xüsusiyyətlər:

- Kanalda düzəliş etdiyiniz sənədin fərqinə baxın.
- Git Status Çubuğu (sol altda) cari branch-;, çirkli göstəriciləri, gələn və gedən öhdəlikləri göst ərir.
- Ən çox görülən git əməliyyatlarını redaktordan edə bilərsiniz:
  - Bir deponu başladın.
  - Deponu klonlayın.
  - Branch-lar və etiketlər yaradın.
  - Mərhələ və dəyişikliklər commit edin.
  - Remote branch ilə push/pull/sync edin.
  - Birləşdirmə konfliktlərini həll edin.
  - Fərqlərə baxın.
- Bir extension ilə, GitHub Pull Requests-i də idarə edə bilərsiniz: https://marketplace.visualstudio.com/items?itemName=GitHub.vscode-pull-request-github.

Rəsmi sənədləri burada tapa bilərsiniz: https://code.visualstudio.com/Docs/editor/versioncontrol.

Eclipse'də Git

Eclipse, Git əməliyyatlarına kifayət qədər tam bir interfeys təmin edən Egit adlı bir plugin ilə gönd ərilir. Git Perspektivinə keçidlə əldə edilir ((Window > Open Perspective > Other..., və "Git"-i seç).

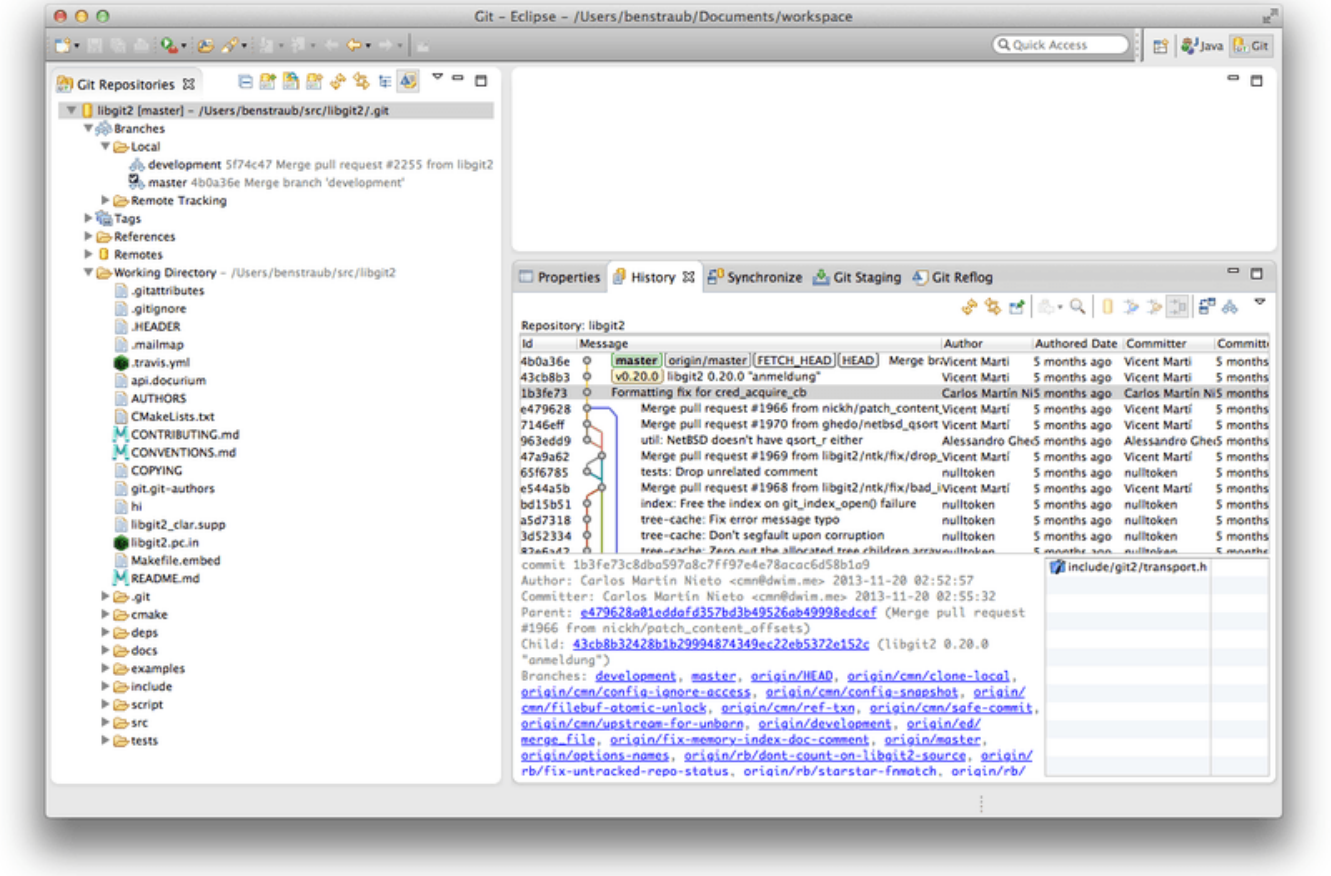


Figure 160. Eclipse's EGit environment

EGit, Help > Help Contents daxil olaraq və məzmun siyahısından "EGit Documentation" nodunu seçərək tapa biləcəyiniz çoxlu möhtəşəm sənədlərlə gəlir.

Sublime Text'də Git

3.2 versiyasından etibarən Sublime Text redaktorda git inteqrasiyasına malikdir.

Xüsusiyyətləri:

- Xüsusiyyətlər bunlardır: Kənar çubuğu badge/icon olan fayl və qovluqların git vəziyyətini göst ərəcəkdir.
- .gitignore dosyanızdakı fayl və qovluqlar kənar paneldə solğun olacaq.
- Vəziyyət çubuğunda, cari git branch-ını və nə qədər dəyişiklik etdiyinizi görə bilərsiniz.
- Bir fayldakı bütün dəyişikliklər artıq kanaldakı işarələr vasitəsilə görünür.
- Sublime Merge git müştəri funksionallığının bir hissəsini Sublime Text-dən istifadə edə bil ərsiniz. Bunun üçün Sublime Merge quraşdırılmalıdır. Baxın: https://www.sublimemerge.com/.

Sublime Text üçün rəsmi sənədlərə burada tapa bilərsiniz: https://www.sublimetext.com/docs/3/git_integration.html.

Bash'da Git

Bash istifadəçisisinizsə, Git ilə təcrübənizi daha dost etmək üçün shell-nizin bəzi xüsusiyyətlərinə

toxuna bilərsiniz. Git, həqiqətən, bir neçə shell üçün pluginlərlə təchiz olunur, lakin varsayılan olaraq açıq deyil.

Əvvəlcə Git qaynaq kodundan `contrib/completion/git-completion.bash` faylının bir kopyasını çıxarmalısınız. Ana qovluğunuz kimi lazımlı bir yerə kopyalayın və bunu `.bashrc`- yə əlavə edin:

```
. ~/git-completion.bash
```

Tamamlandıqdan sonra qovluğunuzu Git deposuna dəyişin və yazın:

```
$ git chec<tab>
```

...və Bash avtomatik olaraq `git checkout` gedəcəkdir. Bu, Git'in bütün alt əmrləri, komanda xətti parametrləri və uyğun olduğu yerlərdə məsafələr və ref adları ilə işləyir.

Mövcud qovluğun Git deposu haqqında məlumat göstərmək üçün təklifinizi fərdiləşdirmək də faydalıdır. Bu, istədiyiniz qədər sadə və ya mürəkkəb ola bilər, lakin ümumiyyətlə cari branch və iş qovluğunun vəziyyəti kimi insanların çoxunun istədiyi bir neçə əsas məlumat var. Bunları t əklifinizə əlavə etmək üçün Gitin qaynaq deposundan contrib/completion/git-prompt.sh` faylını ev qovluğuna kopyalayın, `.bashrc`-inizə belə bir şey əlavə edin:

```
. ~/git-prompt.sh
export GIT_PS1_SHOWDIRTYSTATE=1
export PS1='\w$(__git_ps1 " (%s)")\$ '
```

`\w` cari iş qovluğunu yazdırmaq deməkdir, `\$` təklifin `$`hissəsini yazdırır və `__git_ps1 " (%s)"` ilə `git-prompt.sh` tərəfindən verilən funksiyanı formatlaşdırma arqumenti ilə çağırır. İndi Git tərə findən idarə olunan bir layihənin içərisində olduğunuz zaman bash təklifiniz belə görünəcək:

Figure 161. Customized `bash` prompt

Bu skriptlərin hər ikisi faydalı sənədlərlə gəlir; daha çox məlumat üçün `git-complete.bash` və `git-prompt.sh` məzmununa nəzər yetirin.

Zsh'də Git

Zsh ayrıca Git üçün tab-completion kitabxanası ilə təchiz olunur. Onu istifadə etmək üçün sadəcə `.zshrc`-inizdə `autoload -Uz compinit && compinit`-i işlədin. Zsh interfeysi Bash-dan biraz daha güclüdür:

```
$ git che<tab>
check-attr        -- display gitattributes information
check-ref-format  -- ensure that a reference name is well formed
checkout          -- checkout branch or paths to working tree
checkout-index    -- copy files from index to working directory
cherry            -- find commits not merged upstream
cherry-pick       -- apply changes introduced by some existing commits
```

Birmənalı olmayan tab-completions yalnız siyahıda deyil; faydalı izahatlara malikdirlər və nişanı dəfələrlə vuraraq siyahıda qrafik olaraq gəzə bilərsiniz. Bu, Git əmrləri, arqumentləri və depo iç ərisindəki şeylərin adları (ref və uzaq məsafələr kimi), həmçinin fayl adları və Zsh-in nəyi tamamlayacağını bildiyi bütün şeylərlə işləyir.

Zsh, `vcs_info` adlanan versiya nəzarət sistemlərindən məlumat almaq üçün bir çərçivəyə sahibdir. Branch adını sağ tərəfdəki bildirişə daxil etmək üçün bu sətirləri `~/.zshrc` faylınıza əlavə edin:

```
autoload -Uz vcs_info
precmd_vcs_info() { vcs_info }
precmd_functions+=( precmd_vcs_info )
setopt prompt_subst
RPROMPT=\$vcs_info_msg_0_
# PROMPT=\$vcs_info_msg_0_'%# '
zstyle ':vcs_info:git:*' formats '%b'
```

Bu, qabığınız Git deposunun içərisində olduqda, terminal pəncərəsinin sağ tərəfindəki mövcud branch-ın göstərilməsi ilə nəticələnir. Əlbətdə sol tərəf də dəstəklənir; yalnız PROMPT üçün tapşırıqdan imtina edin. Biraz buna bənzəyir:

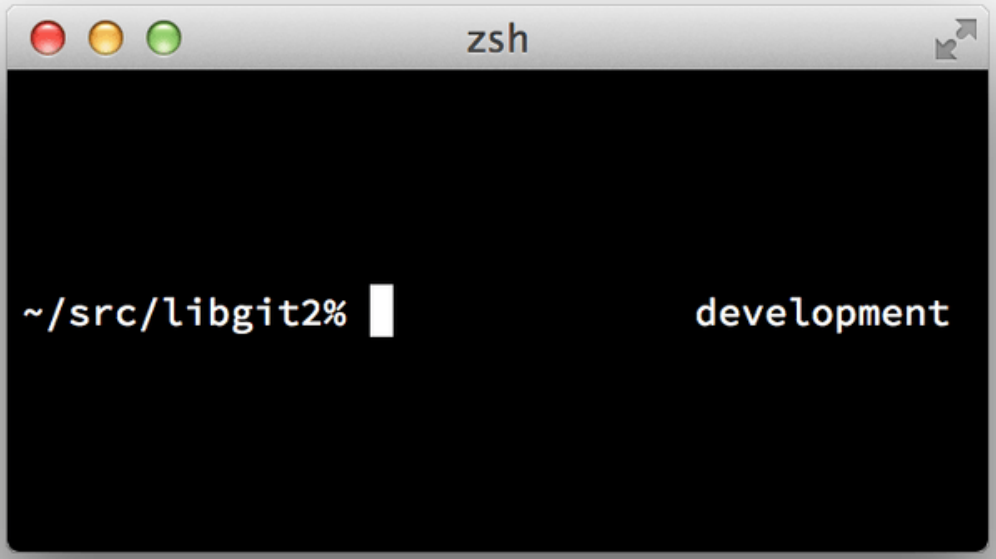


Figure 162. Özəlləştirilmiş `zsh` prompt

`vcs_info` haqqında daha çox məlumat üçün fayllarını `zshcontrib(1)` dərslik səhifəsində və ya http://zsh.sourceforge.net/Doc/Release/User-Contributions.html#Version-Control-Information.

`vcs_info` əvəzinə Git ilə göndərilən, `git-prompt.sh` adlanan istəmə özəlləştirmə skriptinə üstünlük verə bilərsiniz; Ətraflı məlumat üçün https://github.com/git/git/blob/master/contrib/completion/git-prompt.sh səhifəsinə baxın. `git-prompt.sh` həm Bash, həm də Zsh ilə uyğundur.

Zsh kifayət qədər güclüdür ki, onu daha da yaxşılaşdırmağa həsr olunmuş bütün çərçivələr var. Bunlardan biri "oh-my-zsh" adlanır və https://github.com/robbyrussell/oh-my-zsh saytında tapa bil ərsiniz.

oh-my-zsh-in plagin sistemi güclü git tab-completion ilə gəlir və əksəriyyəti versiya nəzarəti m əlumatlarını əks etdirən müxtəlif sürətli "mövzulara" malikdir. An example of an oh-my-zsh theme bu sistemlə nəyin edilə biləcəyinin yalnız bir nümunəsidir.

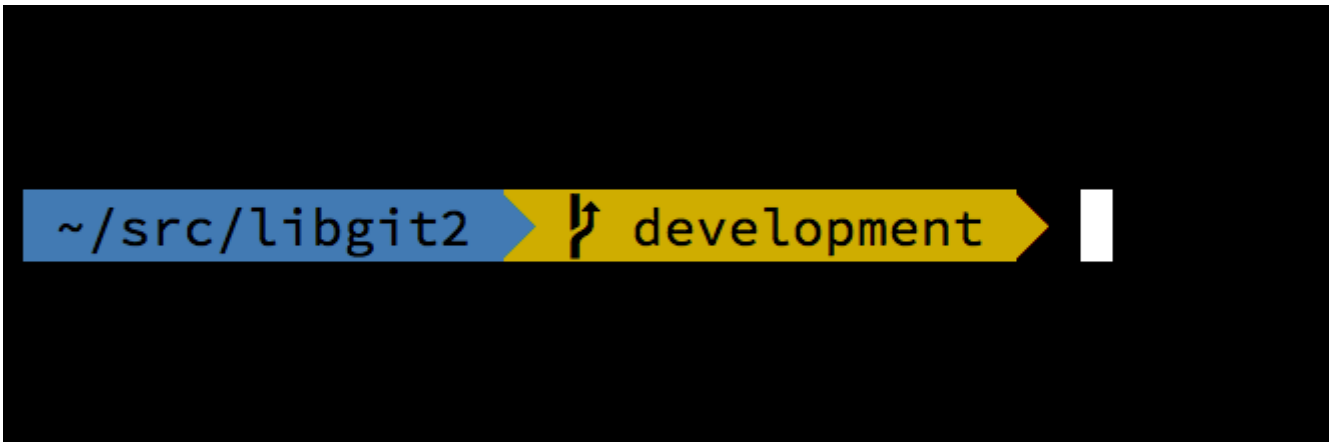


Figure 163. An example of an oh-my-zsh theme

PowerShell'də Git

Windows'dakı köhnə komanda xətti terminalı (`cmd.exe`)həqiqətən xüsusi bir Git təcrübəsi edə bilmir, ancaq PowerShell istifadə edirsinizsə, şansınız var. Linux və ya macOS-da PowerShell Core işlədirsinizsə, bu da işləyir. Posh-git adlı paket ((https://github.com/dahlbyk/posh-git) güclü tab-completion imkanları ilə yanaşı, depo statusunuzun üstündə qalmanıza kömək edəcək inkişaf etmiş bir təklif də verir. Belə görünür:

Figure 164. PowerShell with Posh-git

Installation

Ön şərtlər (yalnız Windows)

PowerShell skriptlərini maşınınızda işə salmadan əvvəl yerli `ExecutionPolicy`-i `RemoteSigned` olaraq ayarlamalısınız (əsasən `Undefined` və `Restricted` istisna olmaqla). `RemoteSigned` əvəzinə `AllSigned` seçsəniz, eyni zamanda yerli skriptlərin (özünüzün) icrası üçün rəqəmsal imzalanmalıdır. `RemoteSigned` ilə yalnız `ZoneIdentifier Internet` olaraq təyin edilmiş (vebdən yüklənmiş) skriptlərin imzalanması lazımdır, digərlərinin yox. İdarəçisinizsə və onu həmin maşındakı bütün istifadəçilər üçün təyin etmək istəyirsinizsə, `-Scope LocalMachine` istifadə edin. Normal bir istifadəçisinizsə, inzibati hüquqa sahib deyilsinizsə, onu yalnız sizin üçün təyin etmək üçün `-Scope CurrentUser` istifadə edə bilərsiniz.

PowerShell Scopes haqqında daha çox məlumat: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_scopes.

PowerShell ExecutionPolicy haqqında daha çox məlumat: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.security/set-executionpolicy.

Bütün istifadəçilər üçün `ExecutionPolicy` dəyərini `RemoteSigned` olaraq təyin etmək üçün növbəti əmrdən istifadə edin:

```
> Set-ExecutionPolicy -Scope LocalMachine -ExecutionPolicy RemoteSigned -Force
```

PowerShell Gallery

PackageManagement quraşdırılmış ən azı PowerShell 5 və ya PowerShell 4 varsa, sizin üçün posh-git yükləmək üçün paket menecerindən istifadə edə bilərsiniz.

PowerShell Gallery haqqında daha çox məlumat: https://docs.microsoft.com/en-us/powershell/scripting/gallery/overview.

```
> Install-Module posh-git -Scope CurrentUser -Force
> Install-Module posh-git -Scope CurrentUser -AllowPrerelease -Force # Newer beta
version with PowerShell Core support
```

Bütün istifadəçilər üçün posh-git yükləmək istəyirsinizsə, bunun əvəzinə `-Scope AllUsers` istifadə edin və yüksək PowerShell konsolundan əmri icra edin. İkinci əmr `Modul 'PowerShellGet' Install-Module tərəfindən yüklənmədi` kimi bir səhvlə uğursuz olarsa, başqa bir əmr işlətməyiniz lazımdır:

```
> Install-Module PowerShellGet -Force -SkipPublisherCheck
```

Sonra geri qayıdıb yenidən cəhd edə bilərsiniz. Bu, Windows PowerShell ilə göndərilən modulların fərqli bir nəşr sertifikatı ilə imzalanması səbəbindən baş verir.

PowerShell Prompt-u Yeniləmək

Git məlumatını təklifinizə daxil etmək üçün posh-git modulunun idxal edilməsi lazımdır. PowerShell hər dəfə başlayanda posh-git idxal etmək üçün, idxal bəyanatını `$profile` skriptinizə əlavə edəcək olan `Add-PoshGitToProfile` əmrini yerinə yetirin. Bu skript hər dəfə yeni bir PowerShell konsolu açdığınız zaman icra olunur. Unutmayın ki, birdən çox `$profile` skript var. Biri konsol üçün, digəri də istifadə üçün.

```
> Import-Module posh-git
> Add-PoshGitToProfile -AllHosts
```

Mənbədən

https://github.com/dahlbyk/posh-git/releases saytından bir posh-git release-ni yükləyin və uncompress edin. Ardından `posh-git.psd1` faylının tam yolunu istifadə edərək modulu import edin:

```
> Import-Module <path-to-uncompress-folder>\src\posh-git.psd1
> Add-PoshGitToProfile -AllHosts
```

Bu, `profile.ps1` faylınıza müvafiq sətir əlavə edəcək və PowerShell-i növbəti dəfə açdığınız zaman posh-git aktiv olacaq.

Prompt-da göstərilən Git statusu xülasəsi məlumatlarının təsviri üçün baxın: https://github.com/dahlbyk/posh-git/blob/master/README.md#git-status-summary-information Posh-git prompt-nuzu necə düzəltmək barədə daha ətraflı məlumat üçün baxın: https://github.com/dahlbyk/posh-git/blob/

master/README.md#customization-variables.

Qısa Məzmun

Git’in gücünü gündəlik işinizdə istifadə etdiyiniz alətlərin içərisindən necə istifadə edəcəyinizi və öz proqramlarınızdan Git depolarına necə daxil olmağı öyrəndiniz.

Appendix B: Proqramlara Git Daxil Etmək

Tətbiqiniz developerlər üçündürsə, mənbə nəzarəti ilə inteqrasiyadan faydalanma ehtimalı yaxşıdır. Sənəd redaktorları kimi developer olmayan tətbiqlər belə, versiya nəzarət xüsusiyyətl ərindən faydalana bilər və Git modeli bir çox fərqli ssenari üçün çox yaxşı işləyir.

Git'i proqramınızla birləşdirmək lazımdırsa, əslində iki seçiminiz var: bir shell düzəldin və `git` komanda xətti proqramını çağırın və ya tətbiqinizə bir Git kitabxanasını daxil edin. Burada komanda xətti inteqrasiyasını və ən populyar bir neçə Git kitabxanasını əhatə edəcəyik.

Əmr-sətri Git

Seçimlərdən biri bir shell prosesi hazırlamaq və işi yerinə yetirmək üçün Git əmr sətri alətindən istifadə etməkdir. Bunun kanonik olmağının faydası var və Gitin bütün xüsusiyyətləri dəstəklənir. Bu da olduqca asan olur, çünki əksər iş vaxtı mühitləri əmr sətri arqumentləri ilə bir prosesi çağırmaq üçün nisbətən sadə bir quruluşa sahibdir. Ancaq bu yanaşmanın bəzi mənfi cəhətləri var.

Biri, bütün çıxışların düz mətndə olmasıdır. Bu, səmərəsiz və səhvlərə meylli ola biləcək tərəqqi və nəticə məlumatlarını oxumaq üçün Git-in vaxtaşırı dəyişən çıxış formatını təhlil etməlisiniz.

Başqa biri səhv bərpa olunmamasıdır. Bir depo bir şəkildə pozulubsa və ya istifadəçinin səhv bir konfiqurasiya dəyəri varsa, Git sadəcə bir çox əməliyyatı yerinə yetirməkdən imtina edəcəkdir.

Başqa bir proses idarə edir. Git, istənməyən mürəkkəblik əlavə edə bilən ayrı bir proses üzərində bir shell mühitini saxlamağınızı tələb edir. Bu proseslərin bir çoxunu əlaqələndirməyə çalışmaq (xüsusən bir neçə prosesdən eyni depo əldə etmək mümkün olduqda) olduqca çətin ola bilər.

Libgit2

Sərəncamınızdakı başqa bir seçim də Libgit2 istifadə etməkdir. Libgit2, digər proqramların daxilində istifadə üçün gözəl bir API-yə sahib olmağa yönəlmiş Git'in asılılıqsız bir tətbiqidir. https://libgit2.orgsaytında tapa bilərsiniz.

Əvvəlcə C API-nin nə olduğuna nəzər salaq. Və budur, qasırğa turu:

```
// Open a repository
git_repository *repo;
int error = git_repository_open(&repo, "/path/to/repository");

// Dereference HEAD to a commit
git_object *head_commit;
error = git_revparse_single(&head_commit, repo, "HEAD^{commit}");
git_commit *commit = (git_commit*)head_commit;

// Print some of the commit's properties
printf("%s", git_commit_message(commit));
const git_signature *author = git_commit_author(commit);
printf("%s <%s>\n", author->name, author->email);
const git_oid *tree_id = git_commit_tree_id(commit);

// Cleanup
git_commit_free(commit);
git_repository_free(repo);
```

İlk iki sətir Git deposunu açır. `git_repository` növü, yaddaşdakı cache-a sahib bir deponun idar əedicisini təmsil edir. Deponun iş qovluğuna və ya `.git` qovluğuna gedən yolu dəqiq bildiyiniz zaman bu ən sadə metoddur. Axtarış üçün seçimləri ehtiva edən `git_repository_open_ext`-də var, `git_clone` və uzaq bir deponun yerli klonunu hazırlayan dostlar, və tamamilə yeni bir depo yaratmaq üçün `git_repository_init` istifadə olunur.

Kodun ikinci hissəsi, HEAD-in sonunda göstərdiyi commit-i əldə etmək üçün rev-parse sintaksisindən istifadə edir (bu barədə daha çox məlumat üçün Branch Referansları bax). Geri qaytarılmış tip, bir depo üçün Git obyekt bazasında mövcud olanı təmsil edən `git_object` göst əricisidir. `git_object` əslində bir neçə müxtəlif növ obyekt üçün “parent” növüdür; “child” növl ərinin hər biri üçün yaddaş düzəni, `git_object` ilə eynidir, buna görə də doğru birinə ata bilərsiniz. Bu vəziyyətdə, `git_object_type(commit) GIT_OBJ_COMMIT` qaytarır,buna görə `git_commit` göstəricisinə ötürülməsi təhlükəsizdir.

Növbəti hissə, commit-in xüsusiyyətlərinə necə çatacağını göstərir. Buradakı son sətirdə `git_oid` növü istifadə olunur; bu Libgit2’nin bir SHA-1 hash üçün təqdimatıdır.

Bu nümunədən bir-iki pattern ortaya çıxmağa başladı:

- Bir göstərici elan etsəniz və bir referansı Libgit2 çağırışına göndərsəniz, bu çağırış, ehtimal ki, tam bir error kodu qaytaracaqdır. A `0 'dəyəri müvəffəqiyyəti göstərir; daha az bir şey səhvdir.
- Libgit2 sizin üçün bir göstəricini doldurursa, onu sərbəst buraxmağa cavabdehsiniz.
- Libgit2 bir çağırışdan `const` göstəricisini qaytarırsa, onu sərbəst buraxmanız lazım deyil, ancaq aid olduğu obyekt sərbəst buraxıldıqda etibarsız olacaqdır.
- C yazmaq biraz ağrılıdır.

Bu son, Libgit2 istifadə edərkən C yazacağınızın çox ehtimal olunmadığını göstərir. Xoşbəxtlikdən, xüsusi dilinizdən və mühitinizdən Git depoları ilə işləməyi asanlaşdıran bir sıra dilə bağlı bağlamalar mövcuddur. Güclü adlanan və https://github.com/libgit2/rugged ünvanında olan Libgit2

üçün Ruby bağlayıcılarından istifadə edərək yazılmış yuxarıdakı nümunəyə nəzər salaq.

```
repo = Rugged::Repository.new('path/to/repository')
commit = repo.head.target
puts commit.message
puts "#{commit.author[:name]} <#{commit.author[:email]}>"
tree = commit.tree
```

Gördüyünüz kimi, kod daha az qarışıqdır. Birincisi, Rugged istisnalardan istifadə edir; səhv şərtl ərinə siqnal vermək üçün `ConfigError` və ya `ObjectError` kimi şeyləri qaldıra bilər. İkincisi, resursların açıq şəkildə sərbəst buraxılması yoxdur, çünki Ruby garbage-collected-dir. Bir az daha mürəkkəb bir nümunəyə nəzər salaq: sıfırdan commit hazırlamaq

```
blob_id = repo.write("Blob contents", :blob) ①

index = repo.index
index.read_tree(repo.head.target.tree)
index.add(:path => 'newfile.txt', :oid => blob_id) ②

sig = {
    :email => "bob@example.com",
    :name => "Bob User",
    :time => Time.now,
}

commit_id = Rugged::Commit.create(repo,
    :tree => index.write_tree(repo), ③
    :author => sig,
    :committer => sig, ④
    :message => "Add newfile.txt", ⑤
    :parents => repo.empty? ? [] : [ repo.head.target ].compact, ⑥
    :update_ref => 'HEAD', ⑦
)
commit = repo.lookup(commit_id) ⑧
```

① Yeni bir faylın məzmunu olan yeni bir blob yaradın.

② İndeksi başlıqlı commit ağac ilə doldurun və yeni faylı `newfile.txt` yoluna əlavə edin.

③ Bu, ODB-də yeni bir ağac yaradır və onu yeni commit üçün istifadə edir.

④ Həm müəllif, həm də müəllif sahələri üçün eyni imzanı istifadə edirik.

⑤ Commit mesajı.

⑥ Commit yaradarkən, yeni commit-in valideynlərini göstərməlisiniz. Bu, tək valideyn üçün HEAD tip-indən istifadə edir.

⑦ Rugged (və Libgit2) bir öhdəlik götürərkən istəyə görə bir referansı yeniləyə bilər.

⑧ Qayıdış dəyəri yeni bir əmr obyekti olan SHA-1 hash-ıdır, bundan sonra `Commit` obyektini əldə etmək üçün istifadə edə bilərsiniz.

Ruby kodu gözəl və təmizdir, lakin Libgit2 heavy lifting etdiyindən bu kod da çox sürətli işləyəc əkdir. Rubyist deyilsinizsə, Other Bindings bəzi digər bağlantılara toxunuruq.

Ətraflı funksionallıq

Libgit2, Git əsas xaricində olmayan bir neçə xüsusiyyətə malikdir. Bir nümunə taxıla biləndir: Libgit2, bir neçə növ əməliyyat üçün xüsusi “arxa planlar” təmin etməyə imkan verir, beləliklə şeyləri Git-dən fərqli bir şəkildə saxlaya bilərsiniz. Libgit2, digər şeylər arasında, konfiqurasiya, ref storage və obyekt verilənlər bazası üçün xüsusi geri imkanları verir.

Gəlin bunun necə işlədiyini nəzərdən keçirək. Aşağıdakı kod, Libgit2 komandası tərəfindən təmin edilmiş geri nümunələr toplusundan götürüldü (https://github.com/libgit2/libgit2-backends). Obyekt databazası üçün xüsusi bir geribildirim necə qurulur:

```
git_odb *odb;
int error = git_odb_new(&odb); ①

git_odb_backend *my_backend;
error = git_odb_backend_mine(&my_backend, /*…*/); ②

error = git_odb_add_backend(odb, my_backend, 1); ③

git_repository *repo;
error = git_repository_open(&repo, "some-path");
error = git_repository_set_odb(repo, odb); ④
```

Səhvlərin tutulduğunu, ancaq handle edilmədiyini unutmayın. Ümid edirik ki, kodunuz bizimkind ən daha yaxşıdır.__

① Həqiqi işi görənlər “backends” üçün konteyner rolunu oynayacaq boş bir obyekt verilənlər bazasını (ODB) “frontend”-i işə salın.

② Xüsusi bir ODB backendi başlatın.

③ Backend-i frontend-ə əlavə edin.

④ Bir depo açın və obyektləri axtarmaq üçün ODB-dən istifadə edəcəyik.

Bəs bu `git_odb_backend_mine` nədir? Yaxşı, bu öz ODB həyata keçirməyiniz üçün konstruktordur və `git_odb_backend` quruluşunu düzgün doldurduqca orada istədiyinizi edə bilərsiniz. Budur nəyə b ənzədiyinə baxaq:

```
typedef struct {
    git_odb_backend parent;

    // Some other stuff
    void *custom_context;
} my_backend_struct;

int git_odb_backend_mine(git_odb_backend **backend_out, /*…*/)
{
    my_backend_struct *backend;

    backend = calloc(1, sizeof (my_backend_struct));

    backend->custom_context = …;

    backend->parent.read = &my_backend__read;
    backend->parent.read_prefix = &my_backend__read_prefix;
    backend->parent.read_header = &my_backend__read_header;
    // …

    *backend_out = (git_odb_backend *) backend;

    return GIT_SUCCESS;
}
```

Buradakı ən incə bir məhdudiyyət, budur ki `my_backend_struct`-ın ilk üzvü `git_odb_backend` quruluşu olmalıdır; bu yaddaş sxeminin Libgit2 kodunun olmasını gözlədiyini təmin edir. Qalan hissəsi özbaşınadır; bu quruluş lazım olduğunuz qədər böyük və ya kiçik ola bilər.

Başlatma funksiyası quruluş üçün bir az yaddaş ayırır, xüsusi kontekst qurur və sonra dəstəklədiyi `parent` strukturunun üzvlərini doldurur. Tam çağırış imzaları dəsti üçün Libgit2 mənbəyindəki `include/git2/sys/odb_backend.h` faylına nəzər yetirin; xüsusi istifadə vəziyyətiniz bunlardan hansını dəstəkləmək istədiyinizi müəyyənləşdirməyə kömək edəcəkdir.

Other Bindings

Libgit2 bir çox dildə bağlanır. Bu yazıdan etibarən daha dolğun bağlama paketlərindən bir neçəsini istifadə edərək kiçik bir nümunə göstərdik; kitabxanalar C++, Go, Node.js, Erlang, və JVM, daxil olmaqla bir çox başqa dillər üçün mövcuddur. Rəsmi bağlamalar kolleksiyasını https://github.com/libgit2 saytındakı depolara baxmaqla tapa bilərsiniz. Yazacağımız kod HEAD tərəfindən göstərilən commit-in öhdəsindən gələn mesajı qaytaracaqdır (`git log -1` kimi).

LibGit2Sharp

Bir .NET və ya Mono tətbiqi yazırsınızsa, LibGit2Sharp (https://github.com/libgit2/libgit2sharp) axtardığınız şeydir. Bağlamalar C# -də yazılıb və xam Libgit2 zənglərini doğma hiss olunan CLR API-lərlə bağlamağa çox diqqət yetirilib. Nümunə proqramımız belə görünür:

```
new Repository(@"C:\path\to\repo").Head.Tip.Message;
```

Masaüstü Windows tətbiqetmələrində, işə başlamağınıza kömək edəcək bir NuGet paketi də var.

objective-git

Tətbiqiniz bir Apple platformasında işləyirsə, ehtimal ki, tətbiq dili olaraq Objective-C istifadə edirsiniz. Obyektiv-Git (https://github.com/libgit2/objective-git) bu mühit üçün Libgit2 bağlamalarının adıdır. Nümunə proqram belə görünür:

```
GTRepository *repo =
    [[GTRepository alloc] initWithURL:[NSURL fileURLWithPath: @"/path/to/repo"]
error:NULL];
NSString *msg = [[[repo headReferenceWithError:NULL] resolvedTarget] message];
```

Objective-git Swift ilə tam qarşılıqlı əlaqəlidir, buna görə də Objective-C-ni geridə qoymusunuzsa qorxmayın.

pygit2

Pythondakı Libgit2 üçün bağlamalara Pygit2 deyilir və https://www.pygit2.org-də tapa bilərsiniz. Nümunə proqramımız:

```
pygit2.Repository("/path/to/repo") # open repository
    .head                          # get the current branch
    .peel(pygit2.Commit)           # walk down to the commit
    .message                       # read the message
```

Əlavə Oxu

Əlbətdə ki, Libgit2-nin imkanlarına tam bir baxış bu kitabın əhatəsi xaricindədir. Libgit2'nin özü haqqında daha çox məlumat istəyirsinizsə, https://libgit2.github.com/libgit2 ünvanında API sənədl əri və https://libgit2.github.com/docs adresində bir sıra təlimatlar var.

Digər bağlamalar üçün birləşdirilmiş README və testləri yoxlayın; tez-tez orada daha da oxumaq üçün kiçik təlimatlar və göstəricilər var.

JGit

Bir Java proqramı daxilində Git istifadə etmək istəyirsinizsə, JGit adlı tam özəllikli Git kitabxanası var. JGit, Git'in yerli olaraq Java dilində yazılmış və Java cəmiyyətində geniş istifadə edilən nisbət ən tam bir tətbiqdir. JGit layihəsi Eclipse çətirinin altındadır və evinə https://www.eclipse.org/jgit/ ünvanından baxmaq olar.

Quraşdırılması

Layihənizi JGit ilə bağlamaq və ona qarşı kod yazmağa başlamağın bir sıra yolu var. Yəqin ki, ən asan olanı Maven istifadə etməkdir - inteqrasiya pom.xml faylınızdakı `<dependencies>` etiketinə aşağıdakı snippet əlavə etməklə həyata keçirilir:

```
<dependency>
    <groupId>org.eclipse.jgit</groupId>
    <artifactId>org.eclipse.jgit</artifactId>
    <version>3.5.0.201409260305-r</version>
</dependency>
```

`Versiya` çox güman ki, bunu oxuduğunuz zaman inkişaf etmiş olacaq; Yenilənmiş depo məlumatları üçünhttps://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit[] bölməsinə baxın. Bu addım atıldıqdan sonra, Maven avtomatik olaraq ehtiyacınız olan JGit kitabxanalarını əldə edəcək və istifadə edəcəkdir.

İkili bağlantıları özünüz idarə etmək istəsəniz, əvvəlcədən qurulmuş JGit binarları https://www.eclipse.org/jgit/download saytından əldə edilə bilər. Bu kimi bir əmr işlədərək onları layihənizdə qura bilərsiniz:

```
javac -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App.java
java -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App
```

Plumbing

JGit, API-nin iki əsas səviyyəsinə malikdir: plumbing and porcelain. Bunların terminologiyası Git'in özündən irəli gəlir və JGit təxminən eyni bölgələrə bölünür: porcelain API'ləri ümumi istifadəçi s əviyyəli hərəkətlər üçün əlverişli bir yerdir (normal istifadəçinin Git əmr sətrini istifadə edəcəyi şeylər), plumbing API'ləri isə birbaşa aşağı səviyyəli depo obyektləri ilə qarşılıqlı əlaqə yaratmaq üçündür.

Əksər JGit sessiyalarının başlanğıc nöqtəsi `Repository` sinifidir və etmək istədiyiniz ilk şey bunun nümunəsini yaratmaqdır. Bir fayl sistemi əsaslı depo üçün (bəli, JGit digər saxlama modellərinə imkan verir), bu, `FileRepositoryBuilder` istifadə edərək həyata keçirilir:

```
// Create a new repository
Repository newlyCreatedRepo = FileRepositoryBuilder.create(
    new File("/tmp/new_repo/.git"));
newlyCreatedRepo.create();

// Open an existing repository
Repository existingRepo = new FileRepositoryBuilder()
    .setGitDir(new File("my_repo/.git"))
    .build();
```

Proqramın, Git deposunu tapmaq üçün lazım olan hər şeyi təmin etmək üçün proqramınızın harada olduğunu dəqiq bilməməsindən asılı olmayaraq, səlis bir API var. Ətraf mühit dəyişənl ərindən (`.readEnvironment()`)istifadə edə bilər, iş qovluğundakı bir yerdən başlayaraq (`.setWorkTree(…).findGitDir()`) axtarır və ya yuxarıdakı kimi bilinən `.git` qovluğunu aça bilər.

Bir `Repository` nümunəsinə sahib olduqda, onunla hər şeyi edə bilərsiniz. Bu da sürətli bir nümunə:

```
// Get a reference
Ref master = repo.getRef("master");

// Get the object the reference points to
ObjectId masterTip = master.getObjectId();

// Rev-parse
ObjectId obj = repo.resolve("HEAD^{tree}");

// Load raw object contents
ObjectLoader loader = repo.open(masterTip);
loader.copyTo(System.out);

// Create a branch
RefUpdate createBranch1 = repo.updateRef("refs/heads/branch1");
createBranch1.setNewObjectId(masterTip);
createBranch1.update();

// Delete a branch
RefUpdate deleteBranch1 = repo.updateRef("refs/heads/branch1");
deleteBranch1.setForceUpdate(true);
deleteBranch1.delete();

// Config
Config cfg = repo.getConfig();
String name = cfg.getString("user", null, "name");
```

Burada çox şey var, buna görə hər bir hissədən keçək. Birinci sətir `master` referansına bir işarə alır. JGit avtomatik olaraq `refs/heads/master`-də yaşayan *actual* `master` ref-ni tutur və arayış haqqında məlumat almağa imkan verən bir obyekti qaytarır.

Ad (`.getName()`) və ya birbaşa istinadın hədəf obyekti (`.getObjectId()`) və ya simvolik bir ref (`.getTarget()`) ilə göstərilən istinad əldə edə bilərsiniz. Ref obyektləri etiket reflərini və obyektlərini təmsil etmək üçün də istifadə olunur, beləliklə etiketin “peeled” olub olmadığını soruşa bilərsiniz, yəni etiket obyektlərinin bir (potensial uzun) sətrinin son hədəfinə işarə edir.

İkinci sətir, ObjectId nümunəsi olaraq qaytarılmış olan `master` referansının hədəfini alır. ObjectId, Git’in obyekt bazasında mövcud ola biləcək və ya olmaya bilən bir obyektin SHA-1 hash-ını təmsil edir.

Üçüncü sətir oxşardır, lakin JGit-in yenidən təhlil sintaksisini necə işlədiyini göstərir (bu barədə daha çox məlumat üçün Branch Referansları); Git’in başa düşdüyü hər hansı bir obyekt göst

əricisini keçə bilərsiniz və JGit həmin obyekt üçün ya etibarlı bir ObjectId ya da `null` qaytaracaq.

Növbəti iki sətirdə bir obyektin kontektinin necə yüklənəcəyi göstərilir. Bu nümunədə obyektin m əzmunu birbaşa stdout-a axın etmək üçün `ObjectLoader.copyTo()`-ı çağırırıq, lakin ObjectLoader-də bir obyektin növünü və ölçüsünü oxumaq, həmçinin bayt massivi kimi qaytarmaq üçün metodlar var.

Böyük obyektlər üçün (harada `.isLarge()` `true`qayıdır), hamısını bir anda yaddaşına çəkmədən oxuya bilən InputStream-ə bənzər bir obyekt əldə etmək üçün `.openStream()` adlandıra bilərsiniz.

Sonrakı bir neçə sətir yeni bir branch yaratmaq üçün nə lazım olduğunu göstərir. Bir RefUpdate nümunəsi yaradırıq, bəzi parametrləri konfiqurasiya edirik və dəyişikliyi tetiklemek üçün `.update()` çağırırıq. Bunu birbaşa izləmək eyni branch-ı silmək üçün koddur. Qeyd edək ki, bunun üçün `.setForceUpdate(true)` tələb olunur; əks təqdirdə `.delete()` çağırışı `REJECTED` qaytaracaq və heç bir şey olmayacaq.

Son nümunə, Git konfiqurasiya sənədlərindən `user.name` dəyərinin necə alınacağını göstərir. Bu Config misalı əvvəllər yerli konfiqurasiya üçün açdığımız depodan istifadə edir, lakin avtomatik olaraq qlobal və sistem konfiqurasiya fayllarını aşkarlayacaq və onlardan dəyərləri də oxuyacaqdır.

Bu, tam plumbing API'sinin yalnız kiçik bir nümunəsidir; daha çox metod və sinif mövcuddur. Burada JGit-in istisnaların tətbiqi ilə edilən səhvləri necə həll etməsi göstərilmir. JGit API-ləri bəzən standart Java istisnalarını (məsələn, `IOException`) atırlar, lakin JGit-ə məxsus istisna növləri də var (məsələn, `NoRemoteRepositoryException`, `CorruptObjectException`və `NoMergeBaseException`)

Porcelain

Plumbing API-ləri kifayət qədər tamamlanmışdır, lakin indeksə bir fayl əlavə etmək və ya yeni bir commit götürmək kimi ümumi hədəflərə çatmaq üçün onları bir-birinə bağlamaq çətin ola bilər. JGit bu işdə kömək etmək üçün daha yüksək səviyyəli API dəsti təmin edir və bu API-lərə giriş nöqtəsi `Git` sinifidir:

```
Repository repo;
// construct repo...
Git git = new Git(repo);
```

Git sinfi, olduqca mürəkkəb bir davranış qurmaq üçün istifadə edilə bilən yüksək səviyyəli *builder* stil metodlarından ibarətdir. Bir nümunəyə baxaq - `git ls-remote` kimi bir şey edək:

```
CredentialsProvider cp = new UsernamePasswordCredentialsProvider("username",
"p4ssw0rd");
Collection<Ref> remoteRefs = git.lsRemote()
    .setCredentialsProvider(cp)
    .setRemote("origin")
    .setTags(true)
    .setHeads(false)
    .call();
for (Ref ref : remoteRefs) {
    System.out.println(ref.getName() + " -> " + ref.getObjectId().name());
}
```

Bu, Git sinfi ilə ümumi bir nümunədir; metodlar zəncirvari metod çağırışlarını parametrləri təyin etməyə imkan verən bir əmr obyektini qaytarır, bunlar `.call()`. Bu vəziyyətdə `origin` remote-dan etiketlər istəyirik, head-ləri yox. Doğrulama üçün bir `CredentialsProvider` obyektinin istifadəsinə də diqqət yetirin.

Git sinfi vasitəsilə bir çox digər əmrlər mövcuddur, lakin `add`, `blame`, `commit`, `clean`, `push`, `rebase`, `revert`, və `reset`-lə məhdudlaşmır.

Əlavə Oxu

Bu JGit-in tam imkanlarından yalnız kiçik bir nümunədir. Əgər maraqlanırsınızsa və daha çox öyr ənmək istəyirsinizsə, məlumat və ilham axtaracaq yeriniz budur:

- https://www.eclipse.org/jgit/documentation-dan rəsmi JGit API sənədləri ilə tanış ola bilərsiniz. Bunlar standart Javadoc'dur, buna görə sevdiyiniz JVM IDE bunları yerli olaraq da qura biləc əkdir.
- https://github.com/centic9/jgit-cookbook ünvanındakı JGit Cookbook-da JGit ilə xüsusi tapşırıqların yerinə yetirilməsinə dair bir çox nümunə var.

go-git

Git'i Golang'da yazılmış bir xidmətə inteqrasiya etmək istədiyiniz təqdirdə təmiz bir Go kitabxanası tətbiqi də mövcuddur. Bu tətbiqin heç bir yerli asılılığı yoxdur və bu səbəbdən manual yaddaş idarəetməsi səhvlərinə meylli deyil. CPU, Yaddaş profilləri, race detektoru və s. kimi standart Golang performans analizi alətləri üçün də transparent-dır.

go-git genişlənməyə, uyğunluğa yönəldilib və https://github.com/go-git/go-git/blob/master/COMPATIBILITY.md sənədləşdirilmiş santexnika API-lərinin çoxunu dəstəkləyir.

Go API-lərindən istifadənin əsas nümunəsi:

```
import "github.com/go-git/go-git/v5"

r, err := git.PlainClone("/tmp/foo", false, &git.CloneOptions{
    URL:      "https://github.com/go-git/go-git",
    Progress: os.Stdout,
})
```

Bir `Depo` nümunəsinə sahib olduqdan sonra məlumat əldə edə və üzərində mutasiyalar həyata keçirə bilərsiniz:

```
// retrieves the branch pointed by HEAD
ref, err := r.Head()

// get the commit object, pointed by ref
commit, err := r.CommitObject(ref.Hash())

// retrieves the commit history
history, err := commit.History()

// iterates over the commits and print each
for _, c := range history {
    fmt.Println(c)
}
```

Ətraflı Funksionallıq

go-git-in az nəzərə çarpan inkişaf etmiş xüsusiyyətləri vardır, bunlardan biri də Libgit2 arxa planlarına bənzər bir taxılan saxlama sistemidir. Varsayılan tətbiq çox sürətli olan yaddaş içi depolamadır.

```
r, err := git.Clone(memory.NewStorage(), nil, &git.CloneOptions{
    URL: "https://github.com/go-git/go-git",
})
```

Taxıla bilən yaddaş bir çox maraqlı seçim təqdim edir. Məsələn, https://github.com/go-git/go-git/tree/master/_examples/storage istinadlar, obyektlər və konfiqurasiyanı Aerokosmik bazasında saxlamağa imkan verir.

Digər bir xüsusiyyət çevik bir fayl sistemi soyutlamasıdır. https://pkg.go.dev/github.com/go-git/go-billy/v5?tab=doc#Filesystem istifadə edərək, bütün sənədləri fərqli şəkildə, hamısını qablaşdırmaqla saxlamaq asandır. Yəni diskdə bir arxiv və ya hamısını yaddaşda saxlamaq olar.

Başqa bir inkişaf etmiş istifadəyə, https://github.com/go-git/go-git/blob/master/_examples/custom_http/main.go da tapılan kimi yaxşı tənzimlənən HTTP müştərisi daxildir.

```
customClient := &http.Client{
    Transport: &http.Transport{ // accept any certificate (might be useful for
testing)
        TLSClientConfig: &tls.Config{InsecureSkipVerify: true},
    },
    Timeout: 15 * time.Second,  // 15 second timeout
        CheckRedirect: func(req *http.Request, via []*http.Request) error {
        return http.ErrUseLastResponse // don't follow redirect
    },
}

// Override http(s) default protocol to use our custom client
client.InstallProtocol("https", githttp.NewClient(customClient))

// Clone repository using the new client if the protocol is https://
r, err := git.Clone(memory.NewStorage(), nil, &git.CloneOptions{URL: url})
```

Əlavə Oxu

Go-git qabiliyyətlərinin tam müalicəsi bu kitabın əhatəsi xaricindədir. Go-git haqqında daha çox məlumat istəyirsinizsə, https://pkg.go.dev/github.com/go-git/go-git/v5ünvanında API sənədləri və https://github.com/go-git/go-git/tree/master/_examples-də bir sıra istifadə nümunələri var.

Dulwich

Həm də təmiz bir Python Git tətbiqi var - Dulwich. Layihə https://www.dulwich.io/ altında aparılır Bu birbaşa çıxmağa çağırmayan, əksinə saf Python istifadə edən depoların (həm local, həm də remote) yerləşdirilməsi üçün bir interfeys təmin etmək məqsədi daşıyır. Performansı əhəmiyyətli dərəcədə yaxşılaşdıran istəyə bağlı bir C extensions-na malikdir.

Dulwich git dizaynını izləyir və iki əsas API səviyyəsini ayırır: plumbing and porcelain.

Aşağıdakı API-nin sonuncu commit-i yerinə yetirmək mesajına daxil olmaq üçün istifadə edilmə sinə dair bir nümunə:

```
from dulwich.repo import Repo
r = Repo('.')
r.head()
# '57fbe010446356833a6ad1600059d80b1e731e15'

c = r[r.head()]
c
# <Commit 015fc1267258458901a94d228e39f0a378370466>

c.message
# 'Add note about encoding.\n'
```

Yüksək səviyyəli porcelain API istifadə edərək commit jurnalını çap etmək üçün aşağıdakılardan istifadə etmək olar:

```
from dulwich import porcelain
porcelain.log('.', max_entries=1)

#commit: 57fbe010446356833a6ad1600059d80b1e731e15
#Author: Jelmer Vernooij <jelmer@jelmer.uk>
#Date:   Sat Apr 29 2017 23:57:34 +0000
```

Əlavə Oxu

- Rəsmi sənədlər https://www.dulwich.io/apidocs/dulwich.html-da mövcuddur.
- https://www.dulwich.io/docs/tutorial-dəki rəsmi təlimatda Dulwich ilə xüsusi tapşırıqların yerinə yetirilməsinə dair bir çox nümunə var.

Appendix C: Git Əmrləri

Kitab boyunca onlarla Git əmrini təqdim etdik və hekayəyə yavaş-yavaş daha çox əmr əlavə edərək bir hekayə daxilində tətbiq etmək üçün çox çalışdıq. Lakin əmrlərin istifadəsi nümunələri bu kitabda bir qədər səpələnmiş formadadır.

Bu əlavədə kitab boyunca müraciət etdiyimiz bütün Git əmrlərini istifadə etdikləri şeylərə görə qruplaşdıraraq nəzərdən keçirəcəyik. Hər bir əmrin ümumiyyətlə nələr etdiyindən danışacağıq və sonra kitabın harasında istifadə etdiyimizi göstərəcəyik.

Quraşdırma və Konfiqurasiya

Çox istifadə olunan iki əmr var, Gitin ilk çağırışlarından adi gündəlik danışıqlara və istinadlara There are two commands that are used quite a lot, from the first invocations of Git to common every day tweaking and referencing, the `config` and `help` commands.

git config

Git-in yüzlərlə şey etmək üçün standart bir yolu var. Bunların çoxu üçün Git-ə default olaraq fərqli bir şəkildə etməsini söyləyə və ya seçimlərinizi təyin edə bilərsiniz. Bu, Git-ə adınızın nə olduğunu izah etməkdən, xüsusi terminal rəng seçimlərinə və ya hansı redaktordan istifadə etdiyinizə qədər hər şeyi əhatə edir. Bu əmrin oxuduğu və yazacağı bir neçə sənəd var, beləcə dəyərləri global olaraq və ya müəyyən depolara təyin edə bilərsiniz.

`git config` əmri demək olar ki, kitabın hər fəslində istifadə edilmişdir.

İlk Dəfə Git Quraşdırması-də Git istifadə etməyə başlamadan əvvəl adımız, e-poçt adresimiz və redaktor seçimimizi göstərmək üçün istifadə etdik.

Git Alias'lar-də hər dəfə onları yazmaq məcburiyyətində qalmamağınız üçün uzun seçim ardıcıllığına qədər genişlənən shorthand əmrləri yaratmaq üçün ondan necə istifadə edə biləc əyinizi göstərdik.

Rebasing -də `git pull` əmrini işə saldığınızda `--rebase`-i varsayılan etmək üçün istifadə etdik.

Etibarlı Yaddaş-də HTTP parollarınız üçün standart bir mağaza qurmaq üçün istifadə etdik.

Keyword Expansion-də Git-ə daxil olan və çıxan məzmun üzərində smudge və təmiz filtrlərin necə qurulacağını göstərdik.

Son olaraq, əsasən Git Konfiqurasiyası-nin tamamı əmrə həsr edilmişdir.

git config core.editor əmrləri

Sizin Redaktorunuz-un içindəki konfiqurasiya təlimatlarını müşayiət edən bir çox redaktor aşağıdakı kimi qurula bilər:

Table 4. Exhaustive list of `core.editor` configuration commands

Editor	Configuration command
Atom	`git config --global core.editor "atom --wait"`
BBEdit (Mac, with command line tools)	`git config --global core.editor "bbedit -w"`
Emacs	git config --global core.editor emacs
Gedit (Linux)	`git config --global core.editor "gedit --wait --new-window"`
Gvim (Windows 64-bit)	`git config --global core.editor "'C:/Program Files/Vim/vim72/gvim.exe' --nofork '%*'"` (Also see note below)
Kate (Linux)	`git config --global core.editor "kate"`
nano	`git config --global core.editor "nano -w"`
Notepad (Windows 64-bit)	`git config core.editor notepad`
Notepad++ (Windows 64-bit)	git config --global core.editor "*C:/Program Files/Notepad* `/notepad`*.exe* -multiInst -notabbar -nosession -noPlugin" (Also see note below)
Scratch (Linux)	`git config --global core.editor "scratch-text-editor"`
Sublime Text (macOS)	`git config --global core.editor "/Applications/Sublime\ Text.app/Contents/SharedSupport/bin/subl --new-window --wait"`
Sublime Text (Windows 64-bit)	`git config --global core.editor "'C:/Program Files/Sublime Text 3/sublime_text.exe' -w"` (Also see note below)
Textmate	`git config --global core.editor "mate -w"`
Textpad (Windows 64-bit)	`git config --global core.editor "'C:/Program Files/TextPad 5/TextPad.exe' -m` (Also see note below)
Vim	`git config --global core.editor "vim"`
VS Code	`git config --global core.editor "code --wait"`
WordPad	`git config --global core.editor '"C:\Program Files\Windows NT\Accessories\wordpad.exe"'`
Xi	`git config --global core.editor "xi --wait"`

Windows 64 bit sistemində 32 bitlik bir redaktorunuz varsa, proqram yuxarıdakı cədvəldəki kimi `C:\Program Files (x86)\` rather than `C:\Program Files\` quraşdırılacaqdır.

git help

`git help` əmri hər hansı bir əmr haqqında Git ilə göndərilən bütün sənədləri göstərmək üçün istifadə olunur. Bu əlavədəki daha90i8 v populyar olanların əksəriyyəti haqqında ümumi bir m əlumat verərkən, hər bir əmr üçün mümkün olan bütün seçimlərin və flag-ların tam siyahısı üçün hər zaman `git help <command>` əmrini işə sala bilərsiniz.

Kömək Almaq-də `git help` əmrini təqdim etdik və sizə Server qurmaq-də `git shell` haqqında daha çox məlumat tapmaq üçün necə istifadə edəcəyinizi göstərdik.

Layihələrin Alınması və Yaradılması

Git deposu əldə etməyin iki yolu var. Biri onu şəbəkədəki və ya başqa bir yerdəki mövcud depodan kopyalamaq, digəri isə mövcud bir qovluqda yenisini yaratmaqdır.

git init

Bir qovluğu götürüb yeni bir Git deposuna çevirmək üçün `git init` əmrini işə salaraq versiyanı idarə etməyə başlaya bilərsiniz.

Bunu əvvəlcə Git Deposunun Əldə Edilməsi-də təqdim edirik və burada işləməyə başlamaq üçün yeni bir depo yaratdığımızı göstəririk.

Uzaq Branch'lar içərisində varsayılan branch-ı "master"-dən necə dəyişdirə biləcəyiniz barədə qısa danışırıq.

Bu əmri Serverə Boş Depo Daxil Edilməsi içindəki bir server üçün boş bir deponu yaratmaq üçün istifadə edirik.

Nəhayət, Plumbing və Porcelain-də əslində səhnə arxasında gördüyü işlərin bəzi detallarına nəzər salırıq.

git clone

`git clone` əmri əslində digər bir neçə əmrin ətrafındakı bir şeydir. Yeni bir qovluq yaradır, içə risinə girir və boş bir Git deposu halına gətirmək üçün `git init` işlədir, ötürdüyünüz URL-ə bir remote olaraq (`git remote add`) əlavə edir (varsayılan olaraq `origin` adlanır), o remote depodan bir `get fetch` və `git checkout` ilə sonra işləmə qovluğunuzdakı son commit-i yoxlayır.

`git clone` əmri kitab boyu onlarla yerdə istifadə olunur, ancaq sadəcə bir neçə maraqlı yeri sadalayacağıq.

Bu əsasən bir neçə nümunədən keçdiyimiz Mövcud Deponu Klonlaşdırmaq-də təqdim olunur və izah olunur.

Serverdə Git Əldə Etmək-də işləyən qovluğu olmayan Git deposunun bir kopyasını yaratmaq üçün `--bare` seçimindən istifadə edirik.

Bundling-də paketlənmiş Git deposunu açmaq üçün istifadə edirik.

Nəhayət, Bir Layihəni Submodullarla Klonlaşdırmaq-də submodullarla deponun klonlaşdırılmasını bir az daha asanlaşdırmaq üçün `--recurse-submodules` seçimini öyrənirik.

Kitab vasitəsilə başqa bir çox yerdə istifadə olunsa da, bunlar bir qədər bənzərsiz və ya bir az f ərqli şəkildə istifadə olunduğu yerlərdir.

Sadə Snapshotting

Məzmunun qurulması və tarixinizə committingetməsi üçün əsas iş axını üçün yalnız bir neçə əsas əmr var.

git add

`git add` əmri işləmə qovluğundan növbəti commit üçün səhnələşdirmə sahəsinə (və ya "index") m əzmun əlavə edir. `git commit` əmri işlədildikdə varsayılan olaraq yalnız bu quruluş sahəsinə baxır, buna görə də `git add` növbəti commit snapshot-nun tam olaraq necə görünməsini istədiyinizi hazırlamaq üçün istifadə olunur.

Bu əmr Git-də inanılmaz dərəcədə vacib bir əmrdir və bu kitabda onlarla dəfə qeyd edilir və ya istifadə olunur. Tapıla bilən bəzi unikal istifadəni tez bir zamanda əhatə edəcəyik.

Əvvəlcə `git add`-ı Yeni Faylların Izlənməsi-də ətraflı şəkildə təqdim edirik və izah edirik.

Əsas Birləşmə Konfiliktləri-də birləşmə konfliktlərini həll etmək üçün bundan necə istifadə edəc əyimizi qeyd edirik.

Interaktiv Səhnələşdirmə içərisində dəyişdirilmiş bir sənədin yalnız müəyyən hissələrini interaktiv şəkildə səhnələşdirmək üçün istifadə edirik.

Nəhayət, onu Ağac Obyektləri-də aşağı səviyyədə təqlid edirik, beləliklə pərdə arxasında nə işlə m əşğul olduğuna dair bir fikir əldə edə bilərsiniz.

git status

`git status` əmri iş qovluğunuzdakı və quruluş sahənizdəki fərqli vəziyyətləri sizə göstərəcəkdir. Hansı fayllar dəyişdirilib və səhnələşdirilməyib və hansının səhnələşdirildiyi, lakin hələ commit edilmədiyini göstərəcəkdir. Normal formada, bu mərhələlər arasında faylları necə köçürəcəyinə dair bəzi əsas göstərişləri göstərəcəkdir.

Əvvəlcə `status`-u Fayllarınızın Vəziyyətinin Yoxlanılması-da həm əsas, həm də sadələşdirilmiş formada əhatə edirik. Kitab boyunca istifadə etdiyimiz müddətdə, `git status` əmri ilə edə biləc əyiniz hər şey orada əksini tapmışdır.

git diff

Hər hansı iki ağac arasındakı fərqləri görmək istədiyiniz zaman `git diff` əmri istifadə olunur. Bu, iş mühitinizlə səhnələşdirmə sahəniz (öz-özünə `git diff`), quruluş sahənizlə son işiniz (`git diff --staged`) arasındakı fərq və ya iki iş (`git diff master branchB`) arasındakı fərq ola bilər.

Əvvəlcə Mərhələli və Mərhələsiz Dəyişikliklərə Baxış bölməsində `git diff`-in əsas istifadəsinə baxırıq, burada hansı dəyişikliklərin səhnələşdirildiyini və hələ səhnələşdirilmədiyini necə göstər əcəyimizi göstəririk.

Bunu Commit Guidelines içərisində `--check` seçimi etmədən əvvəl mümkün boşluq problemlərini axtarmaq üçün istifadə edirik.

Nəyin Təqdim Olunduğunu Müəyyənləşdirmək içərisindəki `git diff A...B` sintaksisiyle branch-lar arasındakı fərqləri daha təsirli şəkildə necə yoxlayacağımızı görürük.

Boşluq fərqlərini `-b` ilə filtrdən keçirmək və konfliktli faylların müxtəlif mərhələlərini İnkişaf etmiş Birləşmə-də `--theirs`, `--ours` və `--base` ilə necə müqayisə etmək üçün istifadə edirik.

Nəhayət, submodul dəyişikliklərini Submodullarla başlayaq içindəki `--submodule` ilə effektiv şəkildə müqayisə etmək üçün istifadə edirik.

git difftool

`git difftool` əmri, sadəcə quraşdırılmış `git diff` əmrindən başqa bir şey istifadə etmək istəsəniz, iki ağac arasındakı fərqi göstərmək üçün xarici bir vasitə işə salır.

Bunu yalnız Mərhələli və Mərhələsiz Dəyişikliklərə Baxış bölümündə qısaca qeyd edirik.

git commit

`git commit` əmri, `git add` ilə hazırlanmış bütün fayl məzmunlarını götürür və verilənlər bazasında yeni qalıcı bir snapshot qeyd edir və sonra branch göstəricisini cari branch üzərinə aparır.

Əvvəlcə, Dəyişikliklərinizin Commit'lənməsi bölməsində committing-in əsaslarını əhatə edirik. Orada gündəlik iş axınlarında `git add` addımını atlamaq üçün `-a` flag-ının necə istifadə ediləcəyini və redaktoru işlətmək əvəzinə komanda xəttində bir commit mesajı ötürmək üçün `-m` flag-ının necə istifadə ediləcəyini nümayiş etdiririk.

Ləğv Edilən İşlər (Geri qaytarılan) bölməsində ən son commit-i təkrarlamaq üçün `--amend` seçimindən istifadə edirik.

Nutshell'də Branch'lar-də `git commit`-in nə etməsi və niyə belə etməsi barədə daha ətraflı məlumat veririk.

Commit-ləri İmzalamaq-də `-S` flag-ı ilə kriptoqrafik olaraq necə imza atacağımıza baxdıq.

Nəhayət, `git commit` əmrinin arxa planda nə etdiyinə və Commit-ləri İmzalamaq-də necə tətbiq olunduğuna baxırıq.

git reset

`git reset` əmri, əvvəlki şeyləri geri qaytarmaq üçün istifadə olunur. `HEAD` göstəricisi ətrafında hərə kət edir və istəyə bağlı olaraq `index` və ya quruluş sahəsini dəyişdirir və istəsəniz `--hard` istifadə etsəniz iş qovluğunu da dəyişə bilər. Bu son seçim bu əmrin səhv istifadə edildiyi təqdirdə işinizi itirməsini mümkün edir, buna görə istifadə etməzdən əvvəl başa düşdüyünüzdən əmin olun.

Əvvəlcə `git reset`-in ən sadə istifadəsini Mərhələli Bir Faylın Mərhələlərə Ayrılmaması bölməsində effektiv şəkildə əhatə edirik, burada işlədiyimiz bir faylı səhnələşdirmək üçün `git add` istifadə edirik.

Daha sonra tamamilə bu əmri izah etməyə həsr olunmuş Reset Demystified-də bir az ətraflı izah edirik.

Birləşməni Ləğv etmək-də birləşməni ləğv etmək üçün `git reset --hard` istifadə edirik, burada `git reset` üçün bir az sarğı olan `git merge --abort` istifadə olunur.

git rm

`git rm` əmri, Git üçün quruluş sahəsindən və iş qovluğundan sənədləri silmək üçün istifadə olunur. Növbəti commit üçün bir faylın silinməsini mərhələləndirdiyinə görə `git add`-ə bənzəyir.

`git rm` əmrini Faylların Silinməsi bölməsində, faylları təkrarən silmək və yalnız səhnələşdirmə sahəsindən çıxarmaqla yanaşı, işləmə qovluğunda `--cached` ilə tərk etmək daxil olmaqla ətraflı şəkildə əhatə edirik.

Kitabdakı `git rm`-nin digər fərqli istifadəsi Obyektlərin Silinməsi-dədir, burada `git filter-branch` işləyərkən `--ignore-unmatch`-ı qısaca istifadə edib izah edirik, bu silməyə çalışdığımız sənədin olmadığı zaman səhv etmir. Bu scripting məqsədləri üçün faydalı ola bilər.

git mv

`git mv` əmri bir faylı köçürmək üçün yeni bir əmrdir və sonra yeni faylda `git add` və köhnə faylda `git rm` əmrini verir.

Bu əmri yalnız Daşınan Fayllar-də qısaca qeyd edirik.

git clean

`git clean` əmri istənməyən faylları iş qovluğunuzdan silmək üçün istifadə olunur. Bura müvəqqəti build artifacts-ın çıxarılması və ya konflikt fayllarının birləşdirilməsi daxil ola bilər.

İş Qovluğunuzun Təmizlənməsi içindəki təmiz əmrdən istifadə edə biləcəyiniz bir çox seçim və ssenarini əhatə edirik.

Branching və Birləşmə

Git-də branching və birləşmə funksiyalarının çoxunu həyata keçirən bir neçə əmr var.

git branch

`git branch` əmri əslində bir branch idarəetmə vasitəsidir. Sahədəki branch-ların siyahısını verə bil ər, yeni bir branch yarada, branch-ları silə və branch-ların adını dəyişə bilər.

Git'də Branch-nin çox hissəsi `branch` əmrinə həsr olunmuşdur və bütün fəsildə istifadə olunur. Ə vvəlcə onu Təzə Branch Yaratmaq-də təqdim edirik və digər xüsusiyyətlərinin əksəriyyətindən (siyahı və silmə)Branch İdarəedilməsi-də keçirik.

İzləmə Branch-ları-də tracking branch qurmaq üçün `git branch -u` seçimindən istifadə edirik.

Nəhayət, Git Referansları daxilində arxa planda gördüklərindən bir neçəsini keçirik.

git checkout

`git checkout` əmri branch-ları dəyişdirmək və məzmunu iş qovluğunuza göndərmək üçün istifadə olunur.

İlk olaraq əmrlə Switching Branches-də `git branch` əmri ilə qarşılaşırıq.

İzləmə Branch-ları içərisində `--track` flag-ı ilə branch-ları izləməyə başlamaq üçün bundan necə istifadə edəcəyimizi görürük.

Bunu Konfliktləri Yoxlamaq içindəki `--conflict=diff3` ilə fayl konfliktlərini yenidən tətbiq etmək üçün istifadə edirik.

Reset Demystified içərisindəki `git reset` ilə əlaqəsini daha ətraflı təfərrüatlarına nəzər salırıq.

Nəhayət, HEAD-də bəzi tətbiq detallarına nəzər salırıq.

git merge

`git merge` vasitəsi, bir və ya daha çox branch-ı yoxladığınız branch-a birləşdirmək üçün istifadə olunur. Daha sonra mövcud branch-ı birləşmə nəticəsinə aparacaqdır.

`git merge` əmri ilk dəfə Sadə Branching-də tətbiq edilmişdir. Kitabın müxtəlif yerlərində istifadə olunmasına baxmayaraq, `merge` əmrinin çox az dəyişikliyi var - ümumiyyətlə birləşdirmək ist ədiyiniz tək branch-ın adı ilə `git merge <branch>` bəs edir.

Squashed birləşdirmənin necə ediləcəyini (Git işi birləşdirdiyi, ancaq birləşdiyiniz branch-ın tarixini qeyd etmədən yeni bir commit kimi göründüyünü) Forked Public Layihəsi sonunda necə izah etdik.

Birləşdirmə prosesi və əmri, o cümlədən `-Xignore-space-change` əmri və bir problem birləşməsini l əğv etmək üçün `--abort` flag-ı daxil olmaqla çox şeyi İnkişaf etmiş Birləşmə-də araşdırdıq.

Layihənizin Commit-ləri İmzalamaq-də GPG imzalamasından istifadə edirsə, birləşmədən əvvəl imzaların necə təsdiqlənəcəyini öyrəndik.

Nəhayət, Subtree Subtree Birləşdirməsi-də birləşməsini öyrəndik.

git mergetool

`git mergetool` əmri, Git-də birləşmə ilə bağlı problemləriniz olması halında xarici birləşmə köm əkçisini işə salır.

Bunu Əsas Birləşmə Konfiliktləri-də tez qeyd edirik və Xaricdən Birləşdirmə və Diff Vasitələri öz xarici birləşmə alətinizi necə tətbiq edəcəyiniz barədə ətraflı məlumat veririk.

git log

`git log` əmri, son commit snapshot-dan bir layihənin əldə edilə bilən qeyd edilmiş tarixçəsini göst ərmək üçün istifadə olunur. Varsayılan olaraq yalnız hazırda olduğunuz branch-ın tarixçəsini göst ərəcək, ancaq içindən keçmək üçün fərqli və ya hətta çoxlu head-lar və ya branch-lar verilə bilər. Həm də commit səviyyəsində iki və ya daha çox branch arasındakı fərqləri göstərmək üçün tez-tez istifadə olunur.

Bu əmr kitabın təxminən hər fəslində bir layihənin tarixini göstərmək üçün istifadə olunur

Commit Tarixçəsinə Baxış bölümündə əmri təqdim edirik və bir qədər dərindən əhatə edirik. Orada hər bir öcommit-də nələrin təqdim edildiyi barədə bir fikir əldə etmək üçün `-p` və `--stat` seçimlərinə və bəzi sadə tarix və müəllif filtrləmə seçimləri tarixə daha qısaca baxmaq üçün `--pretty` və `--oneline` seçimlərinə baxırıq.

Təzə Branch Yaratmaq-də branch göstəricilərimizin harada yerləşdiyini asanlıqla vizuallaşdırmaq üçün onu `--decorate` seçimi ilə istifadə edirik və fərqli tarixlərin necə göründüyünü görmək üçün `--graph` seçimindən də istifadə edirik.

Private Kiçik Komanda və Commit Aralıqları bölmələrində, commit-lərin başqa bir branch-a nəzər ən bir branch-a unikal olduğunu görmək üçün `git log` əmrini istifadə etmək üçün `branchA...branchB` sintaksisini əhatə edirik. Commit Aralıqları-də bu vəziyyəti olduqca geniş izah edirik.

Birləşdirmə Log-u və Üçqat Nöqtə-də nə olduğunu görmək üçün `branchA...branchB` formatı və `--left-right` sintaksisindən istifadə edirik ki, bu və ya digər branch-da, lakin hər ikisində deyil, nə baş verdiyini görək. Birləşdirmə Log-u bölməsində birləşdirmə commit-ində olan konfliktlərə baxmaq üçün `--cc` seçiminin yanında birləşdirmə konflikti debugging-nə kömək etmək üçün `--merge` seçimindən necə istifadə edəcəyimizə də baxırıq.

RefLog Qısa Adları bölməsində keçid yerinə bu vasitə ilə Git reflog-una baxmaq üçün `-g` seçimind ən istifadə edirik.

Axtarış bölməsində bir funksiyanın tarixini görmək kimi kodda tarixən baş verən bir şey üçün kifayət qədər mürəkkəb axtarışlar aparmaq üçün `-S` və `-L` seçimlərindən istifadə etməyə baxırıq.

Commit-ləri İmzalamaq bölməsində `git log` çıxışında hər bir commit-ə doğrulama sətri əlavə etm ək üçün `--show-signature` - etibarlı bir şəkildə imzalanmış olub olmadığına əsaslanaraq necə istifadə ediləcəyini görürük.

git stash

`git stash` əmri branch-da bitməmiş iş görmədən iş qovluğunuzu təmizləmək üçün müvəqqəti olmayan işləri keçici olaraq saxlamaq üçün istifadə olunur.

Stashing və Təmizləmə-də bu əhatə olunub.

git tag

Kod tarixinin müəyyən bir nöqtəsinə qalıcı bir bookmark vermək üçün `git tag` əmri istifadə olunur. Ümumiyyətlə bu, relizlər kimi şeylər üçün istifadə olunur.

Bu əmr Etiketləmə-də ətraflı şəkildə təqdim olunur və Buraxılışlarınızı Etiketləmək tətbiq edilir.

Ayrıca, `-s` flag-ı ilə bir GPG imzalı etiketin yaradılmasını və İşinizin İmzalanması içərisində `-v` flag-ı ilə necə təsdiqlənəcəyini də əhatə edirik.

Layihələrin Paylaşılması və Yenilənməsi

Git-də şəbəkəyə daxil olan çox sayda əmr yoxdur, əmrlərin hamısı yerli verilənlər bazasında işl

əyir. İşlərinizi bölüşməyə və ya başqa bir yerdən dəyişikliklər etməyə hazır olduğunuzda, remote depolarla əlaqəli bir neçə əmr var.

git fetch

`git fetch` əmri remote bir depo ilə əlaqə qurur və mövcud depoda olmayan bütün məlumatları toplayır və yerli verilənlər bazasında saxlayır.

Əvvəlcə bu əmrə Uzaqdan Fetching və Pulling bölümündə baxırıq və Uzaq Branch'lar-də istifadəsi nümunələrini görməyə davam edirik.

Bunu həmçinin Layihəyə Töhfə vermək misallarından bir neçəsində də istifadə edirik.

Bunu Pull Request Referləri içindəki standart boşluğun xaricində olan tək bir spesifik istinad almaq üçün istifadə edirik və Bundling içindəki bir paketdən necə götürüləcəyini görürük.

`git fetch`-in Refspec-də varsayılandan bir az fərqli bir şey etməsini təmin etmək üçün yüksək dərə cədə xüsusi refspecs quraşdırdıq.

git pull

`git pull` əmri, əsasən Git-in təyin etdiyiniz məsafədən alacağı və sonra dərhal olduğunuz branch-a birləşdirməyə çalışacağı `git fetch` və `git merge` əmrlərinin birləşməsidir.

Bunu Uzaqdan Fetching və Pulling-də tez bir zamanda təqdim edirik və Uzaqdan Yoxlama-də işə saldığınız zaman nəyin necə birləşəcəyini göstərəcəyik.

Rebase etdiyiniz zaman yenidən yazın-də rebasing çətinliklərinə kömək etmək üçün necə istifadə edəcəyimizi də görürük.

Uzaq Branch'ları Yoxlamaq-də birdəfəlik dəyişikliklər etmək üçün URL ilə necə istifadə edəcəyimizi göstəririk.

Nəhayət, çəkdiyiniz commit-lərin Commit-ləri İmzalamaq daxilində imzalanmış GPG olduğunu t əsdiqləmək üçün ona `--verify-signatures` seçimini istifadə edə biləcəyinizi çox tez qeyd edirik.

git push

`git push` əmri, başqa bir depo ilə əlaqə qurmaq, yerli verilənlər bazanızın remote olmayan birinin nə olduğunu hesablamaq və sonra fərqi digər depoya köçürmək üçün istifadə olunur. Digər depolara yazılı giriş tələb edir və buna görə normal bir şəkildə təsdiqlənir.

Əvvəlcə Uzaqdan Pushing etmək-də `git push` əmrinə baxırıq. Burada bir branch-ı remote bir depoya push etməyin əsaslarını əhatə edirik. Pushing (İtələmə)-də müəyyən branch-ları pushing etməyə bir az daha dərindən gedirik və İzləmə Branch-ları-də izləmə branch-larını avtomatik olaraq push etmək üçün necə quracağımızı görürük. Uzaq Branch-ların Silinməsi -də serverdəki bir branch-ı git push ` ilə silmək üçün `--delete` flag-ından istifadə edirik.

Layihəyə Töhfə vermək boyunca, branch-larda işi birdən çox məsafədən bölüşmək üçün `git push` istifadə etmək üçün bir neçə nümunəyə baxırıq.

Etiketləri Paylaşmaq bölməsində `--tags` seçimi ilə etdiyiniz etiketləri bölüşmək üçün necə istifadə edəcəyimizi görürük.

Submodul Dəyişikliklərini Yayımlamaq-də, submodullardan istifadə edərkən həqiqətən faydalı ola biləcək superproject-dən əvvəl bütün submodullarımızın işinin dərc olunduğunu yoxlamaq üçün `--recurse-submodules` seçimini istifadə edirik.

Başqa Müştəri Hook'ları -də bir push etmə başa çatmadan çalışdıra biləcəyimiz bir ssenari olan `pre-push` hook-undan bəhs etməliyik.

Nəhayət, Pushing Refspecs-də normal istifadə olunan ümumi qısayollar əvəzinə tam refspec ilə push etməyə baxırıq. Bu, bölüşmək istədiyiniz işi dəqiqləşdirməyə kömək edə bilər.

git remote

`git remote` əmri, remote depolarınızı qeyd etmək üçün bir idarəetmə vasitəsidir. Uzun URL'ləri "origin" kimi qısa tutacaq kimi saxlamağa imkan verir, beləcə, onları daima yazmağa ehtiyac qalmaz. Bunlardan bir neçəsinə sahib ola bilərsiniz və bunları əlavə etmək, dəyişdirmək və silmək üçün `git remote` əmri istifadə olunur.

Bu əmr, Uzaqdan İşləmək-də siyahıya daxil olmaq, əlavə edilməsi, silinməsi və adlandırılması da daxil olmaqla ətraflı şəkildə verilmişdir.

Kitabdakı təqribən hər sonrakı fəsildə də istifadə olunur, lakin həmişə standart `git remote add <name> <url>` formatındadır.

git archive

`git archive` əmri, layihənin müəyyən bir snapshot-nun arxiv sənədini yaratmaq üçün istifadə olunur.

Buraxılış Hazırlamaq bölüşmək üçün bir layihənin tarball-unu yaratmaq üçün `git archive` istifadə edirik.

git submodule

`git submodule` əmri normal depolar içərisində xarici depoları idarə etmək üçün istifadə olunur. Bu kitabxanalar və ya digər paylaşılan mənbələr üçün ola bilər. `submodule` əmrində bu qaynaqları idarə etmək üçün bir neçə alt əmr var (`add`, `update`, `sync` və s.).

Bu əmr yalnız Alt Modullar-də qeyd olunur və tamamilə əhatə olunur.

Yoxlama və Müqayisə

git show

`git show` əmri bir Git obyektini sadə və insan tərəfindən oxunaqlı bir şəkildə göstərə bilər. Normalda bunu bir etiket və ya commit barədə məlumat göstərmək üçün istifadə edərdiniz.

Əvvəlcə Əlavə Etiketlər-də izahatlı etiket məlumatlarını göstərmək üçün istifadə edirik.

Daha sonra bunu müxtəlif revizyon seçimlərimizi həll etdiyimiz commit-ləri göstərmək üçün Reviziya Seçimi bölməsində bir az istifadə edirik.

`git show` ilə etdiyimiz daha maraqlı şeylərdən biri Manual Faylı Yenidən Birləşdirmə birləşmə kofilikti zamanı müxtəlif mərhələlərin spesifik fayl məzmunlarını çıxarmaqd;r.

git shortlog

`git shortlog` əmri,`git log` nəticəsini ümumiləşdirmək üçün istifadə olunur. `git log` əmrinin yerinə yetirəcəyi eyni variantlardan çoxunu alacaqdır, lakin bütün commit-ləri sadalamaq əvəzinə mü əllif tərəfindən qruplaşdırılan commit-lərin xülasəsi təqdim ediləcək.

Qısa Yol-da gözəl bir dəyişiklik tarixi yaratmaq üçün necə istifadə edəcəyimizi göstərdik.

git describe

`git describe` əmri, commit-i həll edən və bir qədər insan tərəfindən oxunaqlı və dəyişməyəcək bir string istehsal edən hər şeyi almaq üçün istifadə olunur.

Bu commit SHA-1 qədər birmənalı, lakin daha başa düşülən bir commit-in təsvirini almaq üçün bir yoldur.

Bir Build Nömrəsi Yaratmaq və Buraxılış Hazırlamaq-də `git describe`-dən istifadə edərək buraxılış sənədimizə ad vermək üçün bir sətir əldə edirik.

Debugging

Git, kodunuzdakı bir problemi həll etməyə kömək etmək üçün istifadə olunan bir neçə əmrə malikdir. Bu, bir şeyin harada tanıdıldığını, kimin təqdim etdiyini anlamağa qədər uzanır.

git bisect

`git bisect` aləti, avtomatik ikili axtarış edərək bir səhv və ya problemi səbəb olan ilk commit-in nə olduğunu müəyyən etmək üçün istifadə edilən inanılmaz dərəcədə faydalı bir debugging vasit əsidir.

Tamamilə İkili Axtarış-də ilə əhatə olunmuşdur və yalnız bu hissədə qeyd edilmişdir.

git blame

`git blame` əmri, sənədin hər sətirində bir dəyişiklik təqdim edən sonuncu və hansı commit-in mü əllifi olduğu hər hansı bir sənədin sətirlərini şərh edir. Bu, kodunuzun müəyyən bir bölməsi haqqında daha çox məlumat istəyən şəxsi tapmaq üçün faydalıdır.

Fayl Annotasiyası-də əhatə olunmuşdur və yalnız həmin hissədə qeyd edilmişdir.

git grep

`git grep` əmri layihənizin köhnə versiyaları daxil olmaqla qaynaq kodunuzdakı hər hansı bir

fayldakı hər hansı bir string-i və ya regular expression-i tapmaqda sizə kömək edə bilər.

Bu Git Grep-da əhatə olunmuşdur və yalnız həmin hissədə qeyd edilmişdir.

Patching

Gitdəki bir neçə əmr, tətbiq etdikləri dəyişikliklər baxımından commit-lərin düşüncə konsepsiyası ətrafında mərkəzləşmişdir, sanki commit seriyası bir sıra patch-lardan ibarətdir.

Bu əmrlər branch-larınızı bu şəkildə idarə etməyə kömək edir.

git cherry-pick

`git cherry-pick` əmri, bir Git commit-ində tətbiq olunan dəyişikliyi götürmək və hazırda olduğunuz branch-da yeni bir commit olaraq yenidən tətbiq etməyə çalışmaq üçün istifadə olunur. Bu, bütün dəyişiklikləri alan branch-da birləşmək əvəzinə branch-dan bir-bir və ya iki commit götürmək üçün faydalı ola bilər.

Cherry picking Rebasing və Cherry-Picking İş Axınları bölməsində təsvir edilir.

git rebase

`git rebase` əmri əsasən avtomatlaşdırılmış bir `cherry-pick`-dir. Bir sıra vəzifələri müəyyənləşdirir və sonra cherry-picks ilə onları başqa yerdə eyni qaydada bir-bir seçir.

Rebasing Rebasing bölməsində təfərrüatlı şəkildə açıqlanmışdır.

Bunu `--onto` flag-nı da istifadə edərək tarixinizi Dəyişdirmək-da iki ayrı depoya ayırma nümunəsi zamanı praktikada istifadə edirik.

Rerere-də rebasing zamanı birləşmə konfliktinə rast gəlirik.

Bunu Birdən Çox Commit Mesajının Dəyişdirilməsi-da `-i` seçimi ilə interaktiv bir skript yazma rejimində də istifadə edirik.

git revert

`git revert` əmri əslində tərs bir `git cherry-pick`-dir. Hədəf etdiyiniz commit-dəki dəyişikliyin tam əksini tətbiq edən yeni bir commit yaradır, mahiyyətcə ləğv edir və ya geri qaytarır.

Birləşdirmə commit-ini ləğv etmək üçün bunu Commit-ləri Tərs Çevirmək-də istifadə edirik.

E-poçt

Git özü də daxil olmaqla bir çox Git layihəsi tamamilə poçt siyahıları üzərində aparılır. Git, asanlıqla e-poçtla göndərə biləcəyiniz patch-lar yaratmaqdan bir e-poçt qutusundan bu patch-arı t ətbiq etməyə qədər bu prosesi asanlaşdırmağa kömək edən bir sıra alətlərə malikdir.

git apply

`git apply` əmri, `git diff` və ya hətta GNU diff əmri ilə yaradılan patch-ı tətbiq edir. Bu, `patch` əmrinin bir neçə kiçik fərqlə edə biləcəyinə bənzəyir.

Bunu istifadə edə biləcəyinizi və bunu edə biləcəyiniz şərtləri Elektron Poçtdan Patch'ların Tətbiq Olunması-də əhatə etmişik.

git am

`git am` əmri, e-poçt gələnlər qutusundan, xüsusən mbox formatlı olan patch-ları tətbiq etmək üçün istifadə olunur. Bu e-poçt üzərindən patch-lar almaq və onları asanlıqla layihənizə tətbiq etmək üçün faydalıdır.

Patch'ı `am` ilə Tətbiq Etmək-də `git am` ətrafında istifadə və iş axınını `--resolved`, `-i` və `-3` seçimlə rindən istifadə etməklə əhatə etdik.

Ayrıca, `git am` ətrafında iş axınında kömək etmək üçün istifadə edə biləcəyiniz bir çox hooks var və hamısı E-poçt İş Axını Hook'ları-da əhatə olunmuşdur.

Bundan əlavə, E-poçt Bildirişləri-də patch formatlı GitHub Pull Request dəyişikliklərini tətbiq etm ək üçün istifadə edirik.

git format-patch

`git format-patch` əmri, düzgün şəkildə formatlanmış poçt siyahısına göndərmək üçün istifadə edə biləcəyiniz mbox formatında bir sıra patch-lar yaratmaq üçün istifadə olunur.

E-poçt Üzərindən Public Layihə-da `git format-patch` alətindən istifadə edərək bir layihəyə töhfə vermə nümunəsindən keçirik.

git imap-send

`git imap-send` əmri `git format-patch` ilə yaradılan bir poçt qutusunu bir IMAP draft qovluğuna yükləyir.

E-poçt Üzərindən Public Layihə-də `git imap-send` vasitəsi ilə patch-lar göndərərək bir layihəyə töhfə vermə nümunəsini nəzərdən keçiririk.

git send-email

`git send-email` əmri e-poçt üzərindən `git format-patch` ilə yaradılan patch-ları göndərmək üçün istifadə olunur.

E-poçt Üzərindən Public Layihə-də `git send-email` vasitəsi ilə patch-lar göndərərək bir layihəyə töhfə vermək nümunəsini əhatə edirik.

git request-pull

`git request-pull` əmri sadəcə birinə e-poçt göndərmək üçün nümunə mesaj gövdəsi yaratmaq

üçün istifadə olunur. Bir ümumi serverdə bir branch-ınız varsa və kiməsə bu dəyişiklikləri e-poçt üzərindən göndərmədən necə birləşdirəcəyini bildirmək istəyirsinizsə, bu əmri işə sala və nəticəni dəyişiklikləri pull etmək istədiyiniz şəxsə göndərə bilərsiniz.

Forked Public Layihəsi içərisində bir pull mesajı yaratmaq üçün `git request-pull` istifadə qaydalarını nümayiş etdiririk.

Xarici Sistemlər

Git, digər versiya nəzarət sistemləri ilə inteqrasiya etmək üçün bir neçə əmrlə gəlir.

git svn

`git svn` əmri, bir müştəri olaraq Subversion versiyası idarəetmə sistemi ilə əlaqə qurmaq üçün istifadə olunur. Bu, Git-i bir Subversion serverindən ödəmə və commit götürmək üçün istifadə edə biləcəyiniz deməkdir.

Bu əmr Git və Subversion-də daha ətraflı əhatə olunur.

git fast-import

Digər versiya idarəetmə sistemləri və ya hər hansı bir formatdan idxal etmək üçün digər formatı Git-in asanlıqla yaza biləcəyi bir şeyə sürətlə uyğunlaşdırılması üçün `git fast-import` istifadə edə bilərsiniz.

Bu əmr Xüsusi İdxalçı-də daha ətraflı əhatə olunur.

İdarəetmə

Bir Git deposunu idarə edirsinizsə və ya bir şeyi böyük bir şəkildə düzəltməlisinizsə, Git sizə kömək etmək üçün bir sıra inzibati əmrlər təqdim edir.

git gc

`git gc` əmri deponuzda “garbage collection” işlədir, verilənlər bazanızdakı lazımsız sənədləri silər və qalan faylları daha səmərəli formata yığar.

Bu əmr normal olaraq sizin üçün arxa planda işləyir, istəsəniz manual olaraq idarə edə bilərsiniz. Bunun bəzi nümunələrini Maintenance-də nəzərdən keçiririk.

git fsck

`git fsck` əmri daxili verilənlər bazasında problem və uyğunsuzluqlar olub olmadığını yoxlamaq üçün istifadə olunur.

Asılı obyektləri axtarmaq üçün bunu yalnız Data Recovery-də bir dəfə istifadə edirik.

git reflog

`git reflog` əmri, tarixlərin yenidən yazılması nəticəsində itirmiş ola biləcəyiniz commit-ləri tapmaq üçün işləyərkən bütün branch-larınızın rəhbərlərinin olduğu bir gündəliyə keçir.

Bu əmri əsasən RefLog Qısa Adları bölməsində əhatə edirik, burada normal istifadəni və `git log` çıxışı ilə eyni məlumatlara baxmaq üçün `git log -g` istifadə qaydalarını göstəririk.

Ayrıca Data Recovery-də belə bir itirilmiş branch-ı bərpa etmək üçün praktik bir nümunəsini inc əliyəcəyik.

git filter-branch

`git filter-branch` əmri, bir faylın hər yerdə silinməsi və ya layihənin çıxarılması üçün bütün deponun bir tək bir alt qovluğa filtrləmək kimi müəyyən yüklənmə qaydalarına uyğun olaraq yenidən yazmaq üçün istifadə olunur.

Hər Commit-dən Bir Sənədin Silinməsi bölməsində əmri izah edirik və `--commit-filter`, `--subdirectory-filter` və `--tree-filter` kimi bir neçə fərqli variantı araşdırırıq.

Git-p4 və TFS içərisində xaricdən gətirilən xarici depoları düzəltmək üçün istifadə edirik.

Plumbing Əmrləri

Kitabda qarşılaşdığımız bir çox aşağı səviyyəli plumbing əmrləri də var idi.

İlk qarşılaşdığımız, Pull Request Referləri içərisində serverdəki xam istinadlara baxmaq üçün `ls-remote` istifadə etdirik .

Səhnələşdirmə sahənizin necə göründüyünə baxmaq üçün Manual Faylı Yenidən Birləşdirmə, Rerere və Index içərisində `ls-files` istifadə edirik.

Ayrıca Branch Referansları-də `rev-parse`-ni qeyd edərək hər bir sətri götürüb SHA-1 obyektinə çeviririk.

Bununla birlikdə, əhatə etdiyimiz aşağı səviyyəli plumbing əmrlərinin çoxu Git'in Daxili İşləri-da az-çox diqqət mərkəzindədir. Kitabın qalan hissəsində bunların istifadəsinin qarşısını almağa çalışdıq.

Index