

ضرورت استفاده از یک سیستم کنترل نسخه:

در طول روند تولید یک برنامه، چه به صورت تیمی و یا حتی انفرادی، بارها برای برنامه نویسان این نیاز پیش می‌آید که به نسخه‌های قدیمی‌تر فایل‌های خود دسترسی داشته باشند تا بتوانند آنچه را که در قبل نوشته‌اند مورد بازبینی قرار دهند. شاید کسانی که با سیستم‌های مدیریت نسخه آشنایی ندارند، این کار را با استفاده از copy و paste کردن فایل‌ها در پوشه‌های جداگانه انجام دهند؛ اما روند توسعه یک برنامه در محیط عملی، امکان استفاده از چنین روشی را به ما نمی‌دهد. زیرا مدیریت این فایل‌ها علی‌الخصوص در پروژه‌های تیمی، بعد از مدتی بسیار دشوار خواهد شد. بنابراین نیاز به سیستمی احساس می‌شود که بتواند این کار را به صورت خودکار انجام دهد.

وظیفه اصلی یک سیستم مدیریت کد، ایجاد یک رویه خودکار جهت دنبال کردن تغییرات فایل‌های ما است به طوری که بگویید هر فایل در چه زمانی، توسط چه کسی، به چه دلیل، چه تغییری کرده است.

آشنایی با Git:

Git توسط سازنده سیستم عامل لینوکس یعنی آقای Linus Torvalds و برای مدیریت کدهای آن ساخته شد که بعدها توسط Linux-BitKeeper ارتقا یافت. BitKeeper یک سیستم مدیریت کد توزیع شده است که البته رایگان نیست. تیم BitKeeper در ابتدا پروژه لینوکس را به صورت رایگان پشتیبانی می‌کرد اما در سال 2005 این حمایت را قطع کرد. در این هنگام تیم توسعه لینوکس تصمیم گرفت که خود یک سیستم مدیریت کد توزیع شده ایجاد کند. آن‌ها این سیستم را با Perl و C نوشتند و آن را برای اجرا شدن بر روی انواع سیستم عامل‌ها نظیر لینوکس ویندوز و حتی مک آماده کردند اهداف اصلی Git عبارتند از:

- 1) سرعت بالا
- 2) سادگی
- 3) قدرت پشتیبانی بالا از Merge/Branching
- 4) یک سیستم کاملاً توزیع شده
- 5) قابلیت توسعه برای پروژه‌های بزرگ

تفاوت سیستم‌های متمرکز و توزیع شده:

سیستم‌های کنترل نسخه را می‌توان بر اساس خصوصیات مختلف در دسته‌های متفاوتی قرار داد اما از نظر معماری سیستم، به دو دسته‌ی زیر تقسیم می‌شوند:

- ۱) VCS (Version Control System) - سیستم‌های مدیریت نسخه متمرکز
 - ۲) DVCS (Distributed Version Control System) - سیستم‌های مدیریت نسخه توزیع شده
- در ادامه مقاله تفاوت این دو روش را بیان خواهیم نمود و به بررسی مزایا و معایب آن‌ها خواهیم پرداخت

تعریف Repository:

مخزن یا همان Repository محلی است که یک سیستم مدیریت نسخه از آن برای نگهداری تغییرات فایل‌ها استفاده می‌کند. در سیستم‌های VCS این مخزن به صورت متمرکز یا اصطلاحاً Centralized Repository می‌باشد. به این معنا که یک Repository بر روی یک ماشین، خواه سیستم خود برنامه نویسی (در پروژه‌های انفرادی) و خواه یک سرور قرار دارد (در پروژه‌های تیمی) و برنامه

نویسان تغییرات فایل‌های خود را به سمت این سرور می‌فرستند و این سرور وظیفه نگهداری تمامی نسخه‌ها و اطلاعات مربوطه از برنامه نویسان مختلف را به عهده دارد. اشکال این روش در این است که برنامه نویس تنها به نسخه جاری که بر روی سیستم خود است دسترسی دارد و اگر بنا به دلیلی بخواهد از نسخه‌های پیشین استفاده کند باید آن را از سرور بخواهد که این کار مشکل دیگری ایجاد می‌کند و آن این است که ممکن است برنامه نویس همیشه در موقعیتی نباشد که بتواند به سرور دسترسی داشته باشد. به همین دلیل این روش وابستگی زیادی برای برنامه نویس ایجاد می‌کند اما پیاده سازی این روش آسان‌تر از مدل توزیع شده است.

در مدل توزیع شده علاوه بر یک مخزن که بر روی یک سرور قرار داد و تمامی نسخه‌ها در آن جا نگهداری می‌شود، هر برنامه نویس یک نسخه محلی مخزن را نیز در اختیار دارد. به این ترتیب وابستگی برنامه نویس به سرور کاهش می‌یابد؛ همچنین می‌توان با ایجاد SubRepositoryها یک ساختار درختی ایجاد نمود که هر کدام از این زیر سیستم‌ها در نهایت اطلاعات را در سرور اصلی قرار می‌دهند. علاوه بر این به دلیل ساختار توزیع شده، امکان بک آپ گیری در این روش مطمئن‌تر است. زیرا تنها وابسته به یک سرور نیست و می‌تواند بر روی سیستم‌های مختلف توزیع شده باشد. البته از اشکالات این روش پیچیدگی پیاده سازی بیشتر آن نسبت به سیستم‌های متمرکز است.

اما سوال این جا است که ما حقیقتا چه چیزی را باید ذخیره کنیم ؟

پاسخ به این سوال بسیار ساده است: هر آنچه برای ما مهم است که این شامل فایل‌های کد، فایل‌های پیکربندی، خروجی‌های نظیر dll و غیره است. البته در این بین استثنائاتی نظیر فایل‌های EXE و یا پکیج‌های نصب شده وجود دارد که در بسیاری از موارد نیازی به پیگیری نسخه‌های آن‌ها نیست اما تمامی این‌ها وابسته به نظر برنامه نویس است.

در ادامه مقالات ما به تعاریف مورد نیاز در سیستم‌های مدیریت کد، ساختار Git و چگونگی نصب و استفاده آن خواهیم پرداخت.

نظرات خوانندگان

نویسنده: محسن
تاریخ: ۱۳۹۱/۰۵/۱۱ ۳:۱۵

سلام

من هم به صورت تجربی باهاش کار کردم برای همین با بعضی از اصطلاحات مشکل دارم و معنی شون رو نفهمیدم مثل push یا pull و ... بسیار خوبه که مرجعی به زبان فارسی این مفاهیم رو توضیح بده .

نویسنده: احمد احمدی
تاریخ: ۱۳۹۱/۰۶/۲۰ ۲۱:۴۴

سلام

تشکر از مقالات مفیدتون - بنده تا بحال از سیستم مدیریت کد استفاده نکردم . به نظر شما برای شروع ، بهتر هست که از چه سیستمی شروع کنم ؟ تعریف SVN و Git رو شنیدم ، اما نیاز به راهنمایی دقیق تری دارم . با تشکر

نویسنده: AliReza
تاریخ: ۱۳۹۱/۰۸/۲۳ ۹:۵۱

سلام

مقاله خیلی خوبی بود من قبلا SVN را کمی کار کردم ولی خوب نتونستم از آن استفاده کنم

ولی چطور Git را تهیه کنیم

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۸/۲۳ ۹:۵۸

[برچسب Git](#) را در سایت دنبال کنید. در قسمت سوم آن به نحوه تهیه و نصب اشاره شده.

نویسنده: مهرداد
تاریخ: ۱۳۹۲/۰۲/۱۸ ۱۲:۳۳

عالی بود

نویسنده: علی پناهی
تاریخ: ۱۳۹۲/۱۰/۲۰ ۲۲:۴۸

کسی در مورد نصب svn یا git بر روی کامپیوتر خونه و اتصال کاربران از طریق اینترنت می تونه راهنمایی بکنه؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۱۰/۲۰ ۲۳:۴

- نصب آن را در [قسمت 4](#) این سری پیگیری کنید. کار با یک سرور ریموت را در [قسمت 9](#) آن مطالعه کنید. البته در این بین تمام قسمت ها را باید مطالعه کنید تا [نظم منطقی آن](#) برقرار شود.

- اینترنت خانه شما اگر IP ثابت دارد، به همین ترتیب از بیرون قابل استفاده خواهد بود (البته [پورت 9418](#) را باید در فایروال سیستم باز کنید). اگر ندارد یک [VPS](#) ارزان بخرید و Git را روی آن نصب کنید یا با ISP خودتان برای گرفتن IP ثابت مذاکره کنید (می فروشند). یا اینکه مثلا سایتی مانند [BitBucket](#) ، مخزن Git خصوصی رایگان تا 5 نفر عضو گروه را نیز به شما ارائه می دهد.

در ادامه آموزش Git، به بررسی مفاهیم مورد استفاده در این سیستم مدیریت کد می‌پردازیم. البته ذکر این نکته ضروری است که ممکن است برخی از تعاریف زیر، برای افرادی که تا کنون با اینگونه سیستم‌ها کار نکرده‌اند، مبهم باشد. اما مشکلی نیست؛ زیرا در دروس بعدی کار با Git، به صورت عملی، این مفاهیم به شکل دقیق‌تر و کاربردی‌تر بیان می‌شوند. هدف در اینجا تنها ایجاد یک تصویر کلی از نحوه کار سیستم‌های مدیریت کد توزیع شده است.

تعاریف زیر هر چند برای Git نوشته شده‌اند، اما می‌توانند در بقیه DVCS‌ها نیز کاربرد داشته باشند.

:Commit

بعد از آن که برنامه نویسان از صحت کدهای خود مطمئن شدند، برای ثبت وضعیت فعلی باید آن‌ها را commit کنند. با این کار یک نسخه جدید از فایل‌ها ایجاد می‌شود. به این ترتیب امکان بازگشت به نقطه فعلی در آینده به وجود خواهد آمد.

:Pushing

بعد از انجام عملیات Commit، معمولاً برنامه نویسان می‌خواهند کدهای نوشته شده را با دیگران به اشتراک بگذارند. این کار به وسیله عملیات Pushing صورت می‌گیرد. بنابراین pushing عبارت است از عملی که با استفاده از آن داده‌ها از یک Repository به Repository دیگر جهت به اشتراک گذاری انتقال می‌یابد. معمولاً به این مخزن Upstream Repository می‌گویند. Upstream Repository یک مخزن عمومی برای تمامی برنامه نویسانی است که تغییرات فایل‌های خود را در آنجا push می‌کنند.

:Pulling

عملیات Pushing تنها نیمی از آن چیزی است که برنامه نویسان برای حفظ به روز بودن کدهای خود به آن احتیاج دارند. در بسیاری از موارد آن‌ها نیاز دارند تا تغییرات فایل‌ها و آخرین به روز رسانی‌ها را نیز دریافت کنند. این کار در دو مرحله متفاوت انجام می‌شود:

(1) بازیابی داده‌ها از مخزن عمومی (fetch)

(2) الحاق داده‌های دریافت شده با داده‌های فعلی

معمولاً در بسیاری از سیستم‌های مدیریت کد، چون به هر دوی این عملیات توأمان نیاز است، با یک دستور هر دو کار انجام می‌شود. به مجموع عملیات فوق Pulling گویند.

Branchها (شاخه‌ها):

Branch و یا همان شاخه، به ما این امکان را می‌دهد که بتوانیم برای قسمت‌های مختلف یک پروژه که روند تولید آن‌ها با هم ارتباط مستقیمی ندارند، سوابق فایلی متفاوتی را ایجاد کنیم.

به عنوان مثال تصور کنید که در یک پروژه سه تیم متفاوت وجود دارد

(1) تیم توسعه برنامه

2) تیم تست و اشکال یابی

3) واحد گرافیکی

در این حالت منطقی است به جای آن که سوابق فایل‌ها برای همه یکسان باشد، هر تیم، شاخه مخصوص به خود را داشته باشد، تا تنها تغییرات فایل‌های مربوطه را پیگیری کند و در نهایت بعد از آن که از صحت کار خود مطمئن شد، آن را در یک شاخه اصلی برای استفاده دیگر تیم‌ها قرار دهد.

در Git شاخه اصلی master نام دارد و فایل‌ها به صورت پیش فرض در این شاخه قرار داده می‌شوند. استاندارد کار بر آن است که در شاخه master تنها فایل‌های نهائی قرار گیرند.

Merging:

به عملیات ادغام دو یا چند شاخه با یکدیگر Merging گفته می‌شود. در بعضی موارد، در روند توسعه یک برنامه نیاز است که شاخه‌هایی جهت مدیریت بهتر کد ایجاد شود. اما بعد از توسعه این قسمت‌ها، می‌توان شاخه‌های ایجاد شده را با هم ادغام نمود تا تغییرات فایل‌ها در یک شاخه قرار گیرند. مثلاً در یک تیم توسعه فرض کنید دو گروه وجود دارند که کدهای مربوط به دسترسی داده را می‌نویسند و هر دو را در یک شاخه فایل‌های خود، نگهداری می‌کنند. گروه اول بر روی کلاس‌های انتزاعی و گروه دوم بر روی کلاس‌های عملی کار می‌کنند. به منظور اینکه گروه دوم به اشتباه کلاس‌های انتزاعی را که هنوز کامل نیستند پیاده سازی نکند، دو شاخه از شاخه اصلی ایجاد می‌شود و هر گروه در شاخه‌ای مجزا قرار می‌گیرد. گروه اول تنها کلاس‌های انتزاعی را در شاخه مشترک قرار می‌دهد که کار آنها تمام شده باشد و گروه دوم تنها همان کلاس‌ها را پیاده سازی و در شاخه مشترک می‌گذارد. بعد از آنکه کار این دو بخش پایان گرفت می‌توان هر سه شاخه را در یک شاخه مثلاً بخش کدهای دسترسی داده قرار داد.

البته عملیات Merging می‌تواند باعث ایجاد مشکلی به نام Conflict شود که خوشبختانه Git روش‌هایی را برای مدیریت این مشکل دارد که در مقالات بعد به آن اشاره خواهد شد.

Locking:

با استفاده از این کار می‌توان مانع تغییر یک فایل توسط برنامه نویسان دیگر شد. معولا Locking به 2 صورت است

1) Strict Locking

2) Optimistic Locking

در روش اول بعد از آن که فایلی قفل شد همان کسی که فایل را قفل کرده تنها امکان تغییر آن را خواهد داشت؛ که البته این روش مناسب سیستم‌های توزیع شده نیست.

در روش دوم فرض بر این است که تغییراتی را که هر کس بر روی فایل می‌دهد، به گونه‌ای باشد که هنگام ادغام این تغییرات، اختلالی ایجاد نشود. یعنی وظیفه بر عهده مصرف کننده فایل است که آگاهی داشته باشد چگونه فایل را تغییر دهد. هنگامی که فایلی به این روش قفل می‌شود، اگر در حین تغییر فایل توسط ما، شخص دیگری فایل را تغییر داده باشد و آن را pull کرده باشد ما در زمان push فایل با خطا مواجه می‌شویم. سیستم از ما می‌خواهد که ابتدا تغییرات فایل را pull کنیم و سپس فایل را push نمائیم. در هنگام pull اگر برنامه نویسی قوانین تغییرات فایل را رعایت نکرده باشد، ممکن است اعمال تغییرات با خطا همراه گردد.

تعاریف فوق بخشی از مفاهیم اولیه مورد نیاز Git بود. اما ما در ادامه به بررسی objectهای Git و همچنین نحوه ذخیره سازی و مدیریت فایل‌ها در این سیستم مدیریت کد خواهیم پرداخت.

نظرات خوانندگان

نویسنده: پژمان
تاریخ: ۱۸:۱۸ ۱۳۹۱/۰۵/۱۲

سلام؛ خسته نباشید. با تشکر.

من قبلا با svn کار کردم. به نظر می‌رسه که در git این commit به مخزن local است نه مخزن اصلی یا upstream در اینجا. درسته؟

نویسنده: حسام امامی
تاریخ: ۱۹:۵۹ ۱۳۹۱/۰۵/۱۲

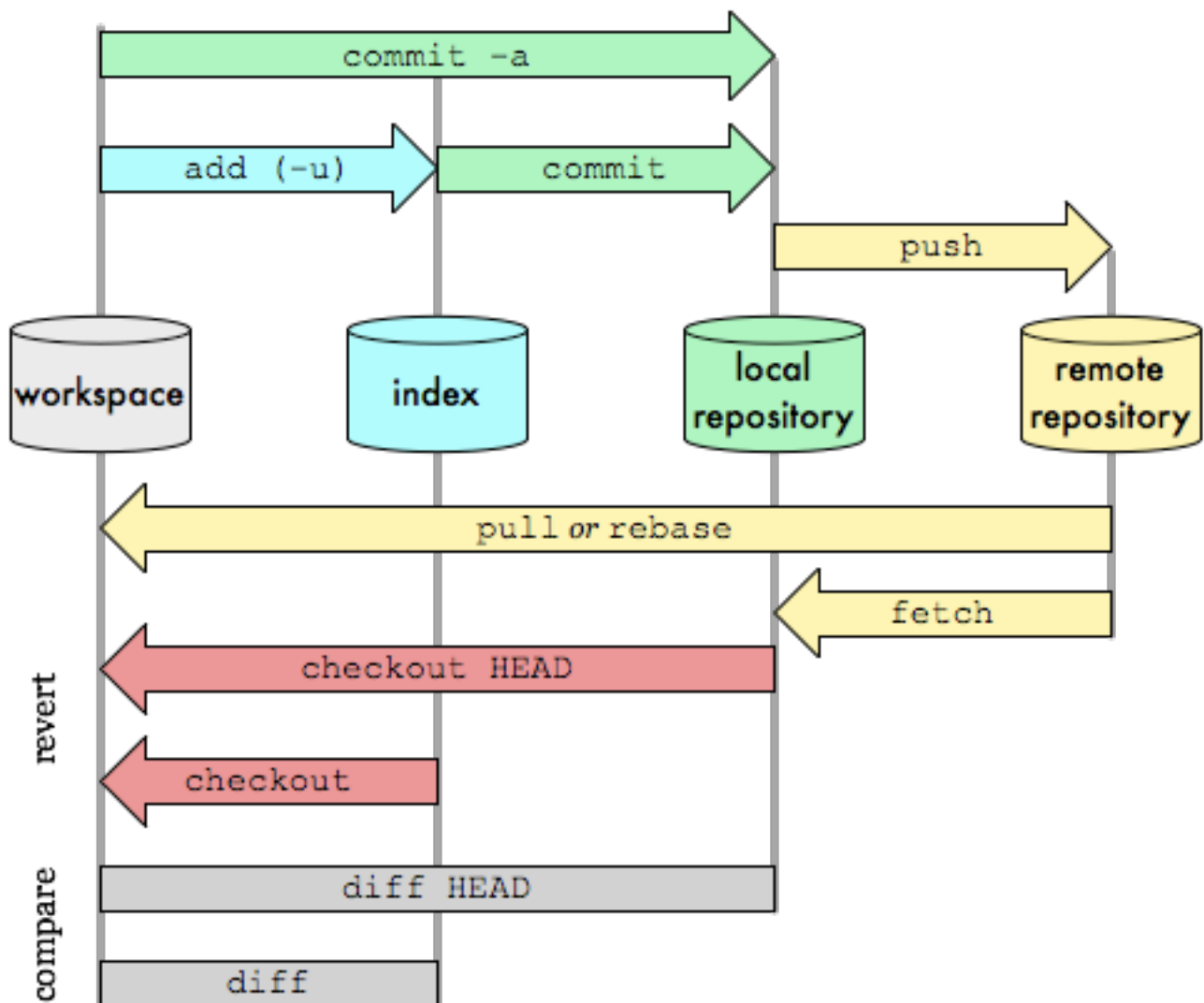
با سلام بله شما ابتدا باید در مخزن محلی commit را انجام دهید بعد در صورتی بخواهید، می‌توانید اطلاعات را در مخزن push، upstream کنید

نویسنده: وحید نصیری
تاریخ: ۹:۲۱ ۱۳۹۱/۰۶/۱۹

گردش کاری توضیح داده شده [به صورت تصویری](#) :

Git Data Transport Commands

<http://osteele.com>



شاید از دید بسیاری از برنامه نویسان بررسی نحوه عملکرد Git چندان اهمیتی نداشته باشد، زیرا آن‌ها سیستمی کارا برای مدیریت کدهای خود لازم دارند و نیازی نمی‌بینند که به جزئیات رفتار Git توجه کنند؛ به همین دلیل در بسیاری از منابع آموزشی این مفاهیم به این شکل گردآوری نشده است. اما من ترجیح دادم برای مدیریت و استفاده بهتر از Git حتی الامکان مطالب کاربردی را از پشت صحنه عملکرد Git در این بخش قرار دهم.

Working Tree (Directory): پوشه‌ی روتی است که فایل‌های پروژه در آن نگهداری می‌شود. این پوشه باید حاوی پوشه‌ای به نام git باشد که محتویات این پوشه، در اصل Repository ما را تشکیل می‌دهند.

اشیا در Git:

برای درک بهتر عملکرد سیستم مدیریت کد Git بهتر است نگاهی به اجزای تشکیل دهنده آن داشته باشیم. به طور کلی Git دارای 4 نوع object است، که هر کدام وظیفه خاصی را به عهده دارند:

1) Tree: شیئی Tree دقیقاً مانند دایرکتوری‌ها در یک سیستم مدیریت فایل است. در واقع treeها ساختاری درختی را ایجاد می‌کنند تا وضعیت فایل‌ها و پوشه‌ها را در Repository حفظ نمایند. هر tree توسط یک کد منحصر به فرد SHA-1 نام گذاری می‌شود.

2) BLOB(Binary large object): اگر با سیستم‌های مدیریت داده نظیر SQL Server کار کرده باشید قطعاً با BLOB آشنایی دارید. BLOBها در واقع چیزی نیستند جز یک مجموعه از بایت‌ها که می‌توانند حاوی هر چیزی باشند (نظیر عکس، فایل متنی، فایل‌های اجرایی و...) در Git فایل‌ها به صورت BLOB و به شکل کامل ذخیره می‌شوند. همچنین مقدار هش شده محتویات فایل‌ها با استفاده از SHA-1 در خود فایل ذخیره می‌شود. به این ترتیب در صورت تغییر در فایل، مقدار هش جدید با مقدار موجود در فایل فرق کرده و Git متوجه می‌شود که فایل دچار تغییر شده است. نکته قابل توجه این است که بر خلاف بسیاری از سیستم‌های مدیریت کد، در هر بار تغییر فایل، Git تنها تغییرات را ذخیره نمی‌کند بلکه از کل محتوا یک snapshot می‌گیرد. شاید به نظر بسیاری تهیه این snapshotهای فراوان باعث زیاد شدن حجم Repository شود، اما Git هوشمندانه تنها فایل‌هایی را مجدداً ذخیره می‌نماید که مقدار آن‌ها تغییر کرده است. در غیر این صورت یک نشانگر به فایل موجود در snapshot جدید ایجاد می‌کند.

3) Commit: این شیئی، یک snapshot از وضعیت فعلی Working Tree است. در واقع با هر بار دستور commit این object ایجاد شده و حداقل حاوی اطلاعات زیر است:

مقدار هش درختی که به آن اشاره می‌کند

نام ثبت کننده دستور commit

توضیحی درباره علت ایجاد commit

خود commit نیز توسط یک کد منحصر به فرد SHA-1 شناخته می‌شود.

4) Tag: چون کار کردن با کدهای هش commit ممکن مشکل باشد، می‌توان از تگ‌ها به عنوان نامی برای commit استفاده نمود. خود تگ می‌تواند حاوی توضیحاتی باشد.

آشنایی با Stage(Index):

هر فایل قبل از آنکه بخواهد در Repository توسط دستور commit ذخیره شود باید ابتدا به Stage آورده شود. در این حالت Git تغییرات فایل را دنبال کرده و سپس می‌توان توسط دستور commit فایل را در Repository وارد کرد. بنابراین ذخیره یک فایل در Git دارای سه مرحله است:

Modified : یعنی فایل تغییر کرده اما به stage اضافه نشده است

Staged: فایل تغییر کرده به stage اضافه شده است.

Committed: فایل در Repository ذخیره شده است.

:head

اشاره‌گری است که به آخرین شئی commit اشاره می‌کند. هر Repository می‌تواند یک head برای هر شاخه‌ی مختلف داشته باشد؛ اما در هر لحظه تنها یک head به عنوان head جاری شناخته می‌شود که معمولا آن را با حروف بزرگ یعنی HEAD مشخص می‌کنند.

تا این مرحله شما تقریبا تمامی آنچه که برای شروع کار با Git را لازم دارید آموخته‌اید. البته همان‌طور که در ابتدا اشاره کردم این مباحث دارای جزئیات بسیاری است اما تا این اندازه برای کار با Git کفایت می‌کند. در صورتیکه به نکات خاصی احتیاج پیدا کنیم، در طول بیان دستورات Git به آن‌ها اشاره خواهد شد. در قسمت بعد نحوه‌ی نصب و پیکربندی Git را بررسی می‌کنیم.

نظرات خوانندگان

نویسنده: رضا

تاریخ: ۲۱:۲ ۱۳۹۱/۰۵/۱۶

من در مورد ترتیب modified و stage شک دارم .
وقتی یک فایل به پوشه پروژه اضافه میشه برای اینکه تغییراتش توسط Git دنبال و ثبت بشه باید وارد stage بشه یا به عبارتی Add بشه و در اولین commit بعد از اون به عنوان staged ثبت میشه . از این به بعد تغییرات در این فایل دنبال و در commit های بعدی به عنوان modified نشون داده میشه . درسته ؟ یا اشتباه متوجه شدم ؟

نویسنده: حسام امامی

تاریخ: ۲۱:۵۴ ۱۳۹۱/۰۵/۱۶

خیر به این صورت نیست تصور کنید شما پنج فایل درون working directory خود دارید همچنین دو فایل جدید نیز اضافه کردید تا زمانی که آن ها را با استفاده از دستور add به stage نیاورید git اقدامی برای ساخت سابقه برای آن فایل ها نمی کند به عنوان مثال سه فایل از پنج فایلی که قبلا وجود داشته تغییر کرده باشد و از این سه فایل تغییر کرده تنها دو تا و یکی از فایل های جدید به stage اضافه شده باشند و دستور commit اجرا شود تنها همان دو فایل تغییر کرده و فایل جدید موجود در stage در repository ذخیره می شوند
اما در مورد سوال شما می تونید فعلا به این صورت تصور کنید که بعد از commit فایل از روی stage حذف میشه (البته دستورات git در این زمینه متفاوت عمل می کنند و لزوما اینگونه نیست) بنابراین فایلی که قبلا commit شده و الان تغییر کرده و روی stage نیست وضعیت modified دارد

نویسنده: هوشنگ

تاریخ: ۰:۲۲ ۱۳۹۱/۰۵/۱۸

بصبرانه منتظر قسمت های بعدی اون هستم . میخوام هر چه سریعتر به قسمت GitExtension و Git Source Control Provider برسیم . کلی سوال واسم ایجاد شده .

برای نصب Git ابتدا به [msysgit](#) رفته و مطابق شکل زیر بر روی گزینه دانلود کلیک کنید. سپس در صفحه باز شده آخرین نسخه Git را دانلود نموده و فایل مربوطه را اجرا کنید:

of Git for Windows

entralized source code management

quite dependent on POSIX features
ie efforts of [a few contributors](#), this
it on Windows. Being solely driven by

environment that is based on the
naming scheme, let's have a look at

Links:

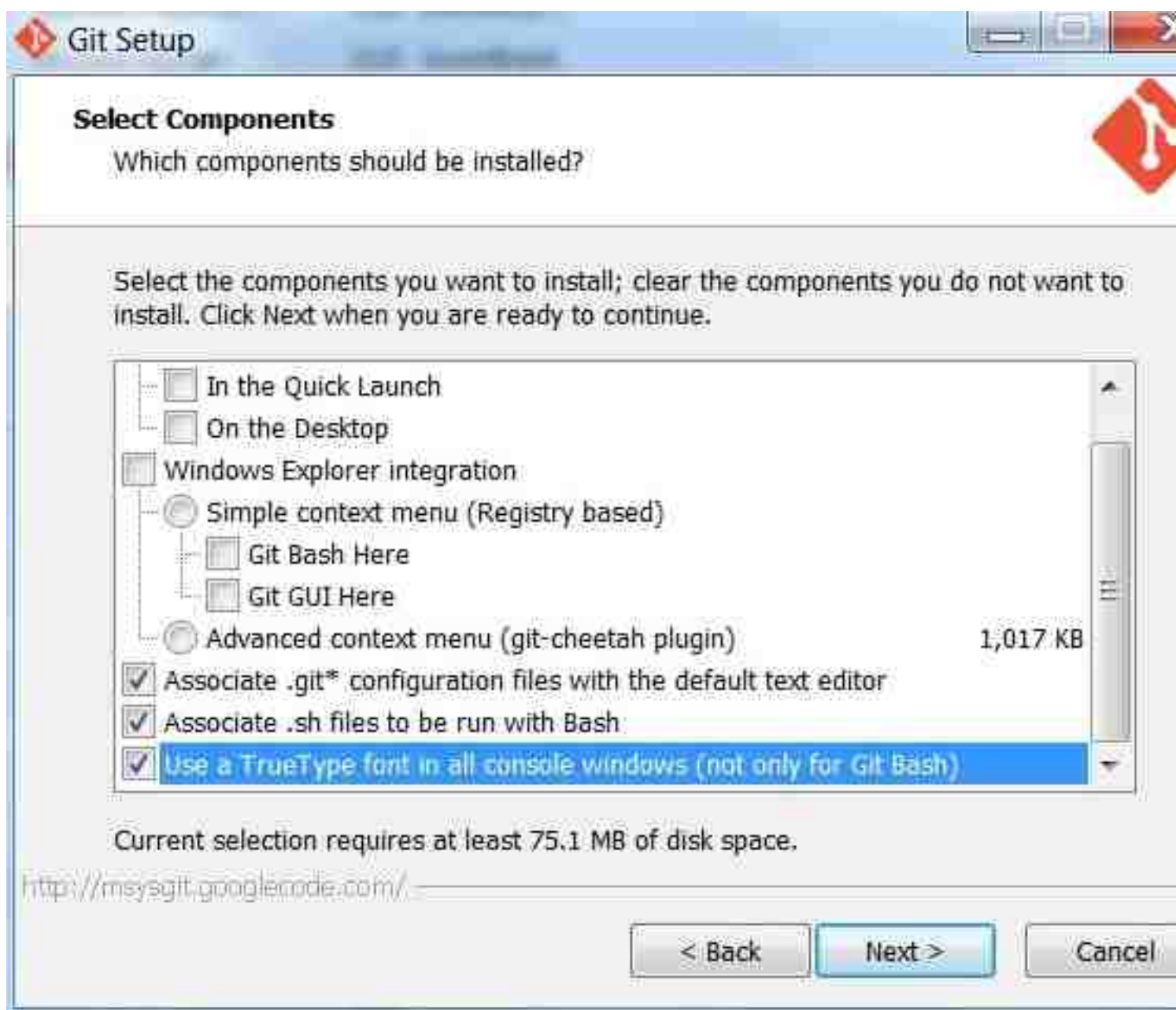
- [FAQ](#)
- [Homepage](#)
- [Wiki](#)
- [Downloads](#)
- [Downloads \(build environment\)](#)
- [Repository](#)
- [Repository \(build environment\)](#)
- [Mailing list](#)

msysGit

شروع نصب:



در این مرحله بخش Windows Explorer Integration اهمیت دارد. در صورت انتخاب این بخش، بعد از نصب، Git و Git Bash به منوی راست کلیک شما اضافه می‌شود. به این ترتیب با سرعت بیشتری می‌توانید به Git در یک پوشه خاص دسترسی داشته باشید.



در این مرحله از شما خواسته می‌شود تعیین کنید که آیا فقط می‌خواهید از طریق Git Bash با Git کار کنید یا با اضافه کردن فایل اجرایی Git به متغیرهای محلی ویندوز از طریق Command Prompt ویندوز نیز می‌خواهید به Git دسترسی داشته باشید. گزینه سوم هم Git و هم برخی از ابزارهای یونیکسی را به متغیرهای محلی اضافه می‌کند که سبب می‌شود شما یک خط فرمان قدرتمندتر در ویندوز داشته باشید. اما این کار ممکن است در برخی از برنامه‌های پیش فرض اختلال ایجاد کند بنابراین در انتخاب این گزینه احتیاط کنید.



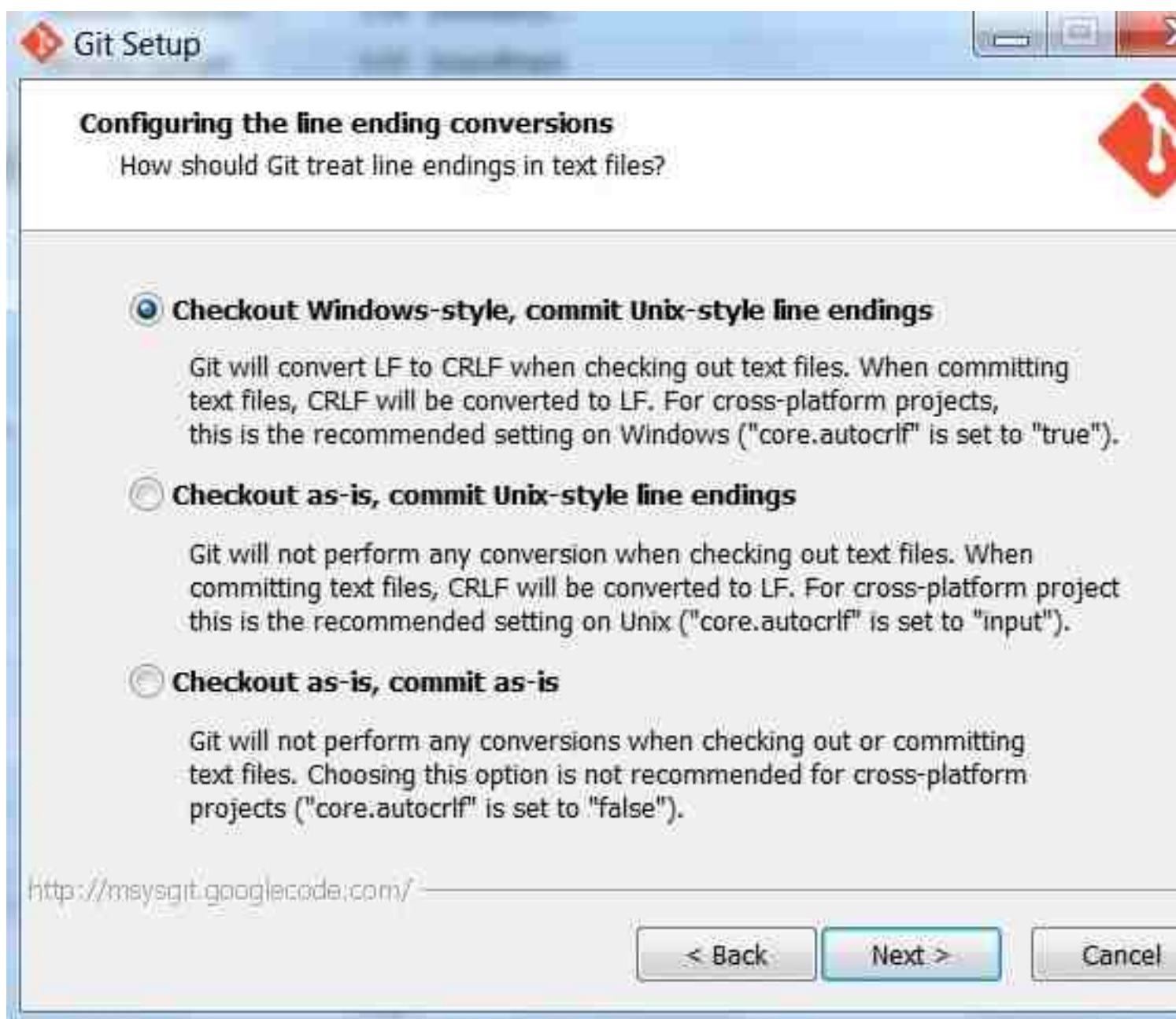
در این مرحله کاراکتری را که نشان دهنده انتهای خط است تعیین می‌کنید. این کاراکتر در ویندوز و یونیکس متفاوت است. بنابراین Git از شما می‌خواهد که برای حفظ سازگاری در محیط‌هایی که چند سیستمی هستند، آن را تعیین کنید.

گزینه اول به صورت فرمت یونیکس ذخیره و به شکل ویندوز بازیابی می‌شود (مناسب برای محیط ویندوز).

گزینه دوم ذخیره به فرمت یونیکسی است و مناسب محیط‌های یونیکس است.

و آخرین گزینه فایل را بدون تغییر ذخیره و بازیابی می‌کند (از این گزینه نیز می‌توان هم برای Unix و هم windows استفاده کرد).

بعد از این مرحله نصب آغاز می‌شود.



نکته: شما می‌توانید جهت دسترسی به یک محیط گرافیکی قوی از [gitextensions](http://gitextensions.org/) استفاده کنید. با دانلود این فایل، هم خود Git و هم GUI هایی برای کارهای مختلف، نظیر مشاهده تفاوت‌های دو فایل یا نمایش گرافیکی شاخه‌ها به سیستم شما اضافه می‌شود.

پیکربندی Git:

برای پیکربندی Git شما باید یک فایل config ایجاد کنید و با استفاده از دستوراتی که در ادامه می‌آید این تنظیمات را انجام دهید. البته پیکربندی Git از طریق ابزارهای گرافیکی که در محله قبل نصب کردید نیز امکان‌پذیر است.

Git دارای سه نوع دسترسی برای پیکربندی است:

سیستمی: این تنظیمات بر روی کل سیستمی که git برای آن نصب شده اعمال می‌شود. فایل gitconfig در مسیر program files/Git/etc/gitconfig قرار دارد و برای تغییر آن باید از سوئیچ --system استفاده نمود.

در سطح کاربر: فایل config در مسیر users/[username] برای این منظور است و تغییر این تنظیمات تنها بر روی همین کاربر اعمال می‌شود برای دسترسی به این فایل باید از سوئیچ --global استفاده کرد.

در سطح Repository: برای هر پوشه repository این فایل موجود است و اگر از دستور config بدون هیچ سوئیچی استفاده

کنیم تغییرات بر روی این فایل اعمال می‌شود.

نکته: معمولاً فایل پیکربندی git در سطح سیستم را تغییر نمی‌دهند.

دستورات پیکربندی:

همان‌طور که گفته شد هر Commit حاوی اطلاعات فردی است که آنرا انجام داده است. این اطلاعات را می‌توان به صورت زیر تنظیم کرد:
نام کاربر:

```
git config --global user.name "Hessam"
```

ایمیل کاربر:

```
git config --global user.email "hessam@localhost.com"
```

با استفاده از دستور زیر می‌توان تنظیماتی را که تا کنون انجام شده ببینیم:

```
git config --global --list
```

همچنین می‌توان ویرایشگر متن پیش فرضی برای git تعیین کرد. از این ویرایشگر می‌توان به عنوان مثال بعد از فرخوانی دستور commit استفاده نمود تا دلیل commit مشخص شود. در صورت تعیین این ویرایشگر، git آنرا خودکار باز می‌کند:

```
git config --global core.editor notepad
```

من در اینجا notepad را انتخاب کردم توجه کنید که مسیر ویرایشگر باید در متغیرهای محلی ویندوز باشد.
و در نهایت جهت نمایش بهتر پیام‌های git می‌توانیم تنظیم کنیم که آن‌ها را با رنگ‌های متفاوتی نمایش دهد:

```
git config --global color.ui auto
```

البته تنظیمات بیشتری را می‌توان در اینجا انجام داد، مانند تعیین برنامه پیش فرض برای نمایش اختلاف فایل‌ها و یا برنامه پیش فرض برای حل کردن مشکل conflict و غیره که این تنظیمات در همان بخش‌ها گفته خواهد شد.

در قسمت بعد دستورات اولیه کار با git به صورت محلی گفته خواهد شد.

نظرات خوانندگان

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۵/۱۹ ۱۹:۵

یک نکته. اگر به گوگل کد دسترسی ندارید، [از این آدرس](#) هم می‌توانید فایل‌ها را دریافت کنید.

نویسنده: پژمان
تاریخ: ۱۳۹۱/۰۶/۰۱ ۲۳:۲۲

ممنون. راهنمای نصب بسیار واضح و مفیدی بود.

نویسنده: اژدری
تاریخ: ۱۳۹۱/۰۶/۱۳ ۱۰:۴

با سلام و عرض خسته نباشید به همه‌ی دوستان و همکاران

سوالی که بنده داشتم این بود که چرا و به چه علتی با وجود ابزاری مثل 2012 , visual studio team foundation server باید با ابزاری مثل git کار کرد و البته با توجه به اینکه دوستان این سایت یا وبلاگ عموماً در حوضه دات نت هستند این سوال مهم‌تر هم میشه ، در مورد مطالب در خصوص git باید بگم طرز کار کردن با این ابزار بسیار پیچیده‌تر و غیر اصولی‌تر از tfs هست ، مثلاً اینکه خود فایل رو پس از تغییر نگهداری میکنه یک نقطه ضعفه ولی نویسنده مطلب از اون به عنوان نقطه قوت یاد کرده ، اگر فایل به صورت مجموعه تغییرات ذخیره بشه هم حجم اطلاعات ذخیره شده کاهش پیدا میکنه و هم منبع نگهداری سورس‌ها میتونه مثل ماشین زمان ما رو به جلو و عقب ببره و محدودیتی نخواهد داشت ، در هر حال با توجه به محصول میکروسافت بودن tfs و رایگان بودن git فکر کنم حتی مقایسه این دو حتی درست هم نباشه.

با تشکر از تمامی زحمات شما دوستان عزیز

نویسنده: حسام امامی
تاریخ: ۱۳۹۱/۰۶/۱۳ ۱۱:۳۱

اگر شما به سایت‌های مدیریت کدی نظیر github مراجعه کنید و تعداد کاربران و یا پروژه‌های قرار گرفته بر روی آن‌ها را در نظر بگیرید متوجه محبوبیت سیستم مدیریت کد git خواهید شد در مورد تفاوت‌های سیستم‌های CVS و DVCS در مقاله اول توضیحاتی داده شد و در مقاله بعد درباره نحوه ذخیره سازی اطلاعات که باعث افزایش سرعت چشمگیر در عملیات check-in و check-out می‌شود

در ضمن در git و در همه سیستم‌های مدیریت کد امکان دستیابی به کدهای قبل وجود دارد و به طور کلی این یکی از اهداف سیستم‌های مدیریت کد است.

خود من هم یک برنامه نویس دات نت هستم اما دلیلی ندارد که مجبور باشیم هر آنچه که میکروسافت ساخته را استفاده کنیم من با هر دو سیستم TFS و Git کار کردم و به شخصه استفاده و راه اندازی آن را از TFS ساده‌تر می‌بینم چون تنها یکی از کاربردهای TFS مدیریت کد است بنابراین شما به طور نسبی با سیستم پیچیده‌تری سرو کار خواهید داشت. اما در نهایت نیاز شما به معماری مورد استفاده در مدیریت کدهای خود تعیین کننده است اگر یک سیستم مدیریت کد توزیع شده لازم دارید بهترین انتخاب git است موفق باشید

نویسنده: امید
تاریخ: ۱۳۹۱/۱۲/۰۹ ۱۰:۵۵

سلام

من اولین بار هست که میخوام از کنترل ورژن‌ها استفاده کنم

اگره gitextensions رو نصب کنم نیازی به نصب msysgit نیست؟
آیا استفاده از gitextensions برای اولین تجربه و شروع کار با git و کلا کنترل ورژن انتخاب درستی هست؟ یا بهتره از msysgit استفاده کرد؟

نویسنده: ندا صابری
تاریخ: ۱۴:۲۸ ۱۳۹۲/۱۲/۲۶

سلام ممنون از مطلب خوبتون.
من تازه VS2013 نصب کردم و گزینه هایی برای کار با Git دیدم که [اینجا](#) در موردش توضیح داده شده. میخوام بدونم VS2013 خودش Git داره؟ لازم نیست دیگه [msysgit](#) رو نصب کنم؟

نویسنده: سعید قره داغی
تاریخ: ۱۶:۳۰ ۱۳۹۳/۰۵/۰۱

با سلام و عرض ادب اگر سرور مون لینوکس باشه ولی یوزرها ویندوزی باشن دیگه احتیاجی به CopSSH نیست؟
اصلا این CopSSH برای چی استفاده می کنن؟
بر اساس این لینک

<http://git-scm.com/book/en/Git-on-the-Server-The-Protocols>

خودش گفته که از Http پشتیبانی میکنن پس چه دلیلی به SSH هست تو ویندوز معادل CopSSH ریگان نرم افزاری وجود نداره؟
یه سوال دیگه اگر کلاینت ها ویندوزی باشن شما صلاح میدونین سرور گیت ، لینوکس باشه یا ویندوزی و کدوم راحت تر و بهتره ؟

نویسنده: وحید نصیری
تاریخ: ۱۷:۲۱ ۱۳۹۳/۰۵/۰۱

- OpenSSH کار مدیریت و اجرای دستورات کاربران راه دور سرور Git را انجام می دهد.
- در لینوکس [OpenSSH](#) هست. کار [CopSSH](#) (که دیگر رایگان نیست) ساده سازی نصب OpenSSH بر روی ویندوز است. البته OpenSSH را در ویندوز بدون نیاز به این ابزارهای جانبی، توسط [cygwin](#) می شود نصب کرد (اصل کار و درستش به این صورت است). شبیه CopSSH، مثلا [sshwindows](#) هم هست ولی بهتره وقت بگذارید روی cygwin.
- اگر ویندوزی می خواهید کار کنید و سرور Git راه اندازی کنید، از [Bonobo Git Server](#) استفاده کنید. [راهنمای نصب](#)
- همچنین [Bitvise SSH Server](#) هم برای ویندوز تهیه شده و [از آن هم می شود](#) جهت نصب سرور Git استفاده کرد.
- [لیست کاملتر](#) نصاب های سرور Git روی ویندوز

در قسمت قبل با چگونگی نصب و راه‌اندازی git آشنا شدیم، در ادامه با نحوه‌ی استفاده از git به صورت local آشنا خواهیم شد.

در ابتدای کار نیاز است تا repository خود را ایجاد کنیم. بدین منظور از طریق محیط command prompt به آدرس پوشه مورد نظر رفته و دستور git init را اجرا می‌کنیم. این کار سبب می‌شود تا پوشه git در داخل فولدر جاری ایجاد شود. این پوشه در واقع همان repository و پوشه جاری، همان working tree ما خواهند بود. حال با استفاده از یک ادیتور نظیر notepad یک فایل متنی جدید را با نام readme1.txt در پوشه ایجاد کنید (توجه کنید در working tree، نه در پوشه .git؛ محتویات این پوشه جز در مورد برخی فایل‌ها نباید توسط کاربر تغییر کند) اکنون دستور زیر را اجرا کنید:

```
git status
```

همانطور که می‌بینید git نشان می‌دهد فایلی در working tree وجود دارد که تغییرات آن دنبال نمی‌شود:

```
PS D:\gitSamples\1> git init
Initialized empty Git repository in D:/gitSamples/1/.git/
PS D:\gitSamples\1> git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       readme1.txt
nothing added to commit but untracked files present (use "git add" to track)
PS D:\gitSamples\1>
```

برای آن‌که این فایل را در repository ذخیره کنیم همانطور که قبلاً گفته شد باید ابتدا آن‌را به index اضافه کنیم این کار با استفاده از دستور زیر انجام می‌شود:

```
git add readme1.txt
```

حال اگر مجدداً دستور status را اجرا کنید می‌بینید که فایل به index یا همان stage اضافه شده‌است.

```
PS D:\gitSamples\1> git add .\readme1.txt
PS D:\gitSamples\1> git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   readme1.txt
#
```

اما توجه کنید که کار در این جا تمام نشده است برای آن که فایل در repository ذخیره شود باید از دستور commit استفاده کرد:

```
git commit
```

بعد از اجرای این دستور، git ادیتور پیش فرضی را که در پیکربندی قبلا تعیین کردید باز می کند تا شما بتوانید توضیحاتی درباره commit خود بنویسید. از این توضیحات بعدا می توان به عنوان راهنمایی جهت دنبال کردن تغییرات فایل ها استفاده نمود. می توان از دستور زیر به منظور اجرای commit و نوشتن پیام آن به صورت همزمان استفاده نمود:

```
git commit -m "commit descriptions"
```

بعد از اجرای دستور commit در صورتی که دستور status را اجرا نمایید خواهید دید که stage خالی شده و فایل readme1 در repository ذخیره شده است. در بعضی موارد می خواهیم چند فایل را همزمان به index اضافه کنیم در این مواقع می توان از دستور زیر استفاده کرد:

```
git add .
```

دستور فوق تمامی فایل های تغییر کرده و یا جدیدا اضافه شده در پوشه جاری را به stage اضافه می کند. فایل readme1.txt را باز کرده و در آن تغییری دلخواه را ایجاد کنید. با اجرای دستور status می بینید که git به شما نشان می دهد فایلی تغییر یافته است. بنابراین برای ثبت تغییرات باید فایل را به stage اضافه کرد. برای اضافه کردن فایل های آپدیت شده، علاوه بر دستور add که در بالا گفته شد از دستور زیر نیز می توان استفاده کرد:

```
git add -u
```

سپس دستور commit را اجرا کنید تا تغییرات در repository ثبت شود. با استفاده از دستور زیر می توان از دستورات commit، یک log تهیه کرد:

```
git log
```

همانطور که در شکل زیر می بینید، ما دارای دو دستور commit هستیم که هر کدام از این commit ها توسط یک کد SHA-1 منحصر به فرد مشخص شده است

```
PS D:\gitSamples\1> git log
commit ff86b2ec6c63ee6ca185fe237de5c0e132427c23
Author: Hessam1 <Hessam@localhost.com>
Date:   Mon Sep 3 00:02:26 2012 +0430

    readme1.txt is changed

commit 54ba3ff69862a105cea6db47d2c33d2d693957ef
Author: Hessam1 <Hessam@localhost.com>
Date:   Sun Sep 2 23:59:48 2012 +0430

    Ininit Command
PS D:\gitSamples\1> _
```

اگر می‌خواهید مشاهده تعداد commit‌های ثبت شده را در دستور log محدود کنید از دستورات زیر می‌توانید استفاده کنید:

```
git log --until [date]
git log --since [date]
git log -[number]
```

چگونگی حذف فایل‌ها:

تا اینجا با نحوه چگونگی ایجاد فایل‌های جدید و یا ویرایش فایل‌های قدیمی آشنا شدید. برای حذف یک فایل می‌توان به دو صورت عمل کرد:

(1) ابتدا فایل را مستقیماً حذف نموده، سپس با استفاده از دستور زیر ابتدا فایل حذف شده را به stage آورده و سپس آن را commit می‌کنیم:

```
git rm [filename]
```

(2) دستور فوق را نوشته و سپس آن را commit می‌کنیم. در این حالت خود git مدیریت حذف فایل را به عهده می‌گیرد و آن را حذف می‌کند.

چگونگی تغییر نام و یا جابجایی یک فایل:

برای تغییر نام و جابجایی یک فایل نیز مانند حذف، دو روش وجود دارد:

(۱) ابتدا فایل مورد نظر را تغییر نام داده و یا جابجا می‌کنیم. در این حالت اگر status بگیریم خواهیم دید که git به ما می‌گوید فایلی با نام قبلی حذف شده و فایلی با نام جدید اضافه شده است. یعنی git تشخیص نمی‌دهد که این دو فایل یکی هستند و تنها تغییر نام داده شده است. اما به محض آن‌که فایل اول را با دستور rm حذف و فایل دوم را با دستور add اضافه کنیم، git متوجه می‌شود که این دو فایل در واقع یک فایل تغییر نام یافته هستند. البته در صورتی‌که حداقل ۵۰ درصد فایل دوم با فایل اول شباهت داشته باشد، بعد از انجام عملیات فوق از دستور commit استفاده می‌کنیم.

(۲) در این روش از دستور زیر استفاده کرده و سپس commit را انجام می‌دهیم:

```
git mv [firstname][secondname]
```

در ادامه مثالی را برای هر دو روش مشاهده خواهید کرد:

روش اول :

```
PS D:\gitSamples\1> git status
# On branch 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:    readme1.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       readme2.txt
no changes added to commit (use "git add" and/or "git commit -a")
PS D:\gitSamples\1> git rm readme1.txt
rm 'readme1.txt'
PS D:\gitSamples\1> git add .\readme2.txt
PS D:\gitSamples\1> git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    readme1.txt -> readme2.txt
#
PS D:\gitSamples\1> git commit -m "readme1.txt is renamed to readme2.txt"
[master 3fc5772] readme1.txt is renamed to readme2.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename readme1.txt => readme2.txt (100%)
PS D:\gitSamples\1> git status
# On branch master
nothing to commit (working directory clean)
PS D:\gitSamples\1>
```

روش دوم :

```
PS D:\gitSamples\1> git mv .\readme2.txt .\readme1.txt
PS D:\gitSamples\1> git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    readme2.txt -> readme1.txt
#
PS D:\gitSamples\1> git commit -m "readme2.txt is changed to readme1.txt"
[master 83026d0] readme2.txt is changed to readme1.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename readme2.txt => readme1.txt (100%)
PS D:\gitSamples\1> git status
# On branch master
nothing to commit (working directory clean)
PS D:\gitSamples\1> _
```

نظرات خوانندگان

نویسنده: احمد احمدی
تاریخ: ۱۳:۴۰ ۱۳۹۱/۰۶/۲۱

برای خروج از نتیجه‌ی بعضی دستورات (مثل log) ، [کلید Q را بزنید](#) .

نویسنده: بابک
تاریخ: ۱:۲۹ ۱۳۹۱/۱۰/۲۲

باید همه فایل‌های پروژه رو تو قسمت stage وارد کنیم یا فقط فایل هایی که قراره تغییر بدیم؟

نویسنده: حسام امامی
تاریخ: ۱۵:۳۱ ۱۳۹۱/۱۰/۲۲

شما می‌توانید stage را به عنوان واسطی بین Working Directory و Repository تصور کنید بنابراین هر فایلی که می‌خواهد در Repository ذخیره شود ابتدا به stage آورده می‌شود.

نویسنده: بهار یگانه
تاریخ: ۱۰:۱۲ ۱۳۹۳/۱۱/۱۹

سلام. من git status که می‌زنم ، اسم یکسری فایل رو مینویسه و جلوش مینویسه مثلا " :

```
deleted:  foldername1/text1.txt
```

```
deleted:  foldername2/text2.txt
```

من قبلش از add . برای ثبت تغییرات استفاده کرده بودم .. حالا لازمه دستور rm رو هم واسه حذف فایلها استفاده کنم ؟

در قسمت قبل برخی از دستورات مورد نیاز برای کار با git به صورت محلی گفته شد. در اینجا به بخشی دیگر از این دستورات خواهیم پرداخت:

مشاهده تغییرات فایل‌ها:

در بسیاری از موارد نیاز است تا بتوانیم تفاوت فایل‌های موجود در working tree و فایل‌های موجود در stage و repository را دریابیم. بدین منظور می‌توان از دستورات زیر استفاده کرد:

```
git log
```

برای مشاهده تغییرات فایل‌ها بین دو commit دلخواه از کد زیر استفاده می‌کنیم:

```
git diff
```

تذکر: در اغلب موارد می‌توانید تنها از چند مقدار اول SHA-1 برای آدرس‌دهی استفاده نمود. چون معمولاً این کد به اندازه کافی دارای تغییرات است. البته کار کردن با کدهای SHA-1 ممکن است مشکل باشد؛ به همین جهت می‌توان از دستور زیر نیز برای مشاهده تغییرات استفاده نمود:

```
git diff HEAD~[number]..HEAD~[number]
```

توجه کنید که کلمه HEAD اشاره به وضعیت جاری head دارد و عدد number اختلاف آن را با وضعیت جاری مشخص می‌نماید. به عنوان مثال در شکل زیر ما می‌خواهیم اختلاف فایل‌ها را بین ۲ دستور commit با مقادیر 9da و e0e را مشخص نماییم. همانطور که ملاحظه می‌کنید اولی اشاره به وضعیت جاری head و دومی وضعیت قبلی head است. بنابراین ما از دستور زیر استفاده می‌کنیم:

```
git diff HEAD~1..HEAD
```



```
Hessam@HESSAM-PC /c/GitSamples/3 <master>
$ git log
commit 9da065830b32cce917f0fb0c088546068f8845b7
Author: Hessam <hessam@localhost.com>
Date:   Sun Sep 9 00:42:23 2012 +0430

    readme3.txt is changed

commit e0e1e1ce83deaf015b17855f6d6399c6420560fc
Author: Hessam <hessam@localhost.com>
Date:   Sun Sep 9 00:20:35 2012 +0430

    Initial Commit

Hessam@HESSAM-PC /c/GitSamples/3 <master>
$ git diff HEAD~1..HEAD
diff --git a/readme3.txt b/readme3.txt
index e6136f2..0639bc6 100644
--- a/readme3.txt
+++ b/readme3.txt
@@ -1,1 @@
-This is the third file that is created
\ No newline at end of file
+This is a first change on this file.
\ No newline at end of file
```

همچنین اگر بخواهیم اختلاف فایلی را در working tree و stage ببینیم، کافی است که از دستور زیر استفاده کنیم:

```
git diff --staged [filename]
```

در صورتی که در تنظیمات git، نرم افزار پیش فرضی را برای نمایش اختلاف فایل ها تعیین نکرده باشید، git اختلاف فایل ها را خود نمایش می دهد. اما از آنجایی که این نمایش چندان مطلوب نیست، بهتر است از دستور زیر برای تنظیم نمایش اختلاف فایل ها در نرم افزار دیگری استفاده کنید:

```
git config --global diff.external <path_to_wrapper_script>
```

تنظیمات مورد نیاز برای این کار در [اینجا](#) گفته شده است.

تذکر: راه حل ساده برای این منظور نصب git extension است که در آموزش نصب گفته شد.

تنظیم git برای صرف نظر کردن از برخی فایل ها:

اگر از دستوراتی نظیر add استفاده کنید متوجه خواهید شد در بعضی موارد نیازی ندارید که تمامی فایل های موجود در working tree به repository اضافه شوند. فایل ها در git به دو دسته تقسیم می شوند؛ برخی که در حال حاضر دنبال شده و برخی که git تغییرات آنها را دنبال نمی کند. در صورتیکه بخواهید فایلی که تغییرات آن دنبال نمی شود را به طور کلی حذف کنید، می توانید از دستور clean استفاده کنید. دو اصلاح کننده معروف این دستور -n برای نمایش آنکه چه فایل هایی حذف خواهند شد و -f برای اجبار در حذف آنها:

```
git clean -n [filename]
```

```
git clean -f [filename]
```

اما در برخی موارد نیاز است که فایل‌ها وجود داشته باشند، اما تنها git تغییرات آن‌ها را دنبال نکند، نه آنکه مانند دستور بالا آن‌ها را از working tree نیز حذف نماید. بدین منظور git از فایل بی‌نامی با پسوند gitignore. استفاده می‌کند این فایل از عبارات منظم به شکل بسیار محدودی پشتیبانی می‌کند. در ادامه برخی از دستوراتی را که می‌توان برای حذف برخی فایل‌ها در این فایل نوشت را مشاهده خواهید کرد:

۱ مجموعه: مثال [adgJHn]

۲ بازه: [0-9] یا [a-z]

۳ حذف یک دایرکتوری با نوشتن آدرس آن و قرار دادن / (البته توجه کنید که با این کار sub directory ها هنوز هم track خواهند شد)

می‌توان با استفاده از علامت ! برخی از فایل‌ها و یا دایرکتوری‌ها را مستثنی کرد
می‌توان این تنظیمات را در فایلی با نام دلخواه ذخیره کرد و سپس با استفاده از دستور زیر آن‌ها را به صورت global یا سراسری اعمال نمود:

```
git config global core.excludesfile [path and filename]
```

توجه کنید که git تغییرات پوشه‌های خالی را دنبال نمی‌کند بنابراین اگر قصد دارید پوشه‌ای در repository ذخیره شود یک فایل temp در آن ایجاد کنید
چند مثال:

اگر بخواهید فایل‌های باینری داخل فولدر bin در repository ذخیره نشوند این خط را در این فایل اضافه می‌کنیم:

```
bin/
```

هیچ فایلی با پسوند txt را در نظر نگیر:

```
*.txt
```

هیچ فایلی را با پسوند txt در فولدر bin در نظر نگیر

```
/bin/*.txt
```

هیچ فایلی با پسوند txt را در نظر نگیر به جز readme1.txt

```
*.txt  
!readme1.txt
```

توجه کنید که هر آنچه بین دو علامت # قرار گیرد به عنوان توضیح در نظر گرفته می‌شود

نظرات خوانندگان

نویسنده: وحید نصیری
تاریخ: ۹:۱۳ ۱۳۹۱/۰۶/۱۹

یک [gitignore](#) مفید برای VS.NET:

```
#OS junk files
[Tt]humbs.db
*.DS_Store

#Visual Studio files
*.[Oo]bj
*.user
*.aps
*.pch
*.vspssc
*.vssscc
*_i.c
*_p.c
*.ncb
*.suo
*.tlb
*.tlh
*.bak
*.[Cc]ache
*.ilk
*.log
*.lib
*.sbr
*.sdf
*.opensdf
*.unsuccessfulbuild
ipch/
obj/
[Bb]in
[Dd]ebug*/
[Rr]elease*/
Ankh.NoLoad

#MonoDevelop
*.pidb
*.userprefs

#Tooling
_ReSharper*/
*.resharper
[Tt]est[Rr]esult*
*.sass-cache

#Project files
[Bb]uild/

#Subversion files
.svn

# Office Temp Files
~$*

#NuGet
packages/

#ncrunch
*ncrunch*
*crunch*.local.xml

# visual studio database projects
*.dbmdl

#Test files
*.testsettings
```

دو تا - اول global در این دستور رو یادتون رفته بزارید. لطفا اصلاح کنید

```
git config --global core.excludesfile [path and filename]
```

در این مقاله با یکی از مهمترین ویژگی‌های git یعنی بازبازی تغییرات فایل‌ها، آشنا می‌شویم. اما در ابتدا نگاهی می‌کنیم به چگونگی ایجاد تغییر در آخرین commit:

تغییر آخرین commit:

در git این امکان وجود دارد که آخرین فرمان commit با استفاده از اصلاح‌کننده amend تغییر کند. علت تاکید بر روی آخرین دستور این است که git به دلیل ساختاری که دارد نمی‌تواند commit‌های قبل را تغییر دهد. اگر مقالات ابتدایی آموزش git را مطالعه کرده باشید، به خاطر دارید که هر commit دارای یک کد منحصر به فرد SHA-1 است، که این کد از هش کردن BLOB‌ها به همراه خود مقادیر commit یعنی مشخصات ایجاد کننده آن و از همه مهمتر SHA-1 پدر ایجاد می‌شود. در نتیجه تغییر commit‌یی که نقش برگ را ندارد، یعنی در ساختار درختی git دارای فرزند است، سبب می‌شود که SHA-1 آن تغییر کند. این تغییر، commit‌های فرزند را مجاب می‌کند برای حفظ صحت داده‌ها مقدار SHA-1 خود را تغییر دهند. به این ترتیب این تغییرات در کل repository پخش خواهد شد. به همین دلیل git جز آخرین commit امکان اصلاح دیگر commit‌ها را نخواهد داد. برای اصلاح آخرین commit کافی است دستور commit خود را با amend -- بیاورید

دستورات بازبازی فایل:

دستور checkout:

این فرمان یکی از مهمترین فرمان‌های git است که دارای دو کاربرد است:

- ۱) بازبازی فایلی از repository و یا stage
- ۲) تغییر شاخه (این مورد را در مقالات مربوط به branch بررسی خواهیم کرد)

با استفاده از این دستور می‌توان فایلی را از repository به stage یا working tree و یا هر دو بیاوریم. عملکرد این دستور با اصلاح کننده‌های گوناگون متفاوت خواهد بود. در ادامه روش‌های مختلف فراخوانی این دستور و کاربرد هر کدام آورده شده است:

در صورتی که بخواهیم فایلی را از محلی که head اکنون به آن اشاره می‌کند به working tree بیاوریم از دستور زیر استفاده می‌کنیم:

```
git checkout --[filename]
```

در حالت فوق فایل مستقیماً به working tree آورده شده و در stage قرار نمی‌گیرد

تذکر: -- در دستور بالا اختیاری بوده، اما استفاده از آن توصیه می‌شود. زیرا در صورتی که نام فایل به اشتباه وارد شود و یا فایل موجود نباشد، git اقدام به تعویض شاخه می‌کند. زیرا همانطور که گفته شد، این دستور کاربرد دوگانه دارد. در این حالت ممکن است به علت سهل انگاری مشکلاتی ایجاد شود علامت -- تاکید می‌کند که مقدار نوشته نام فایل است. حال اگر بخواهیم فایلی را از commit‌های قبل بازبازی کنیم، می‌توانیم از دستور زیر استفاده کنیم:

```
git checkout [SHA-1] [filename]
```

در این حالت فایل هم در stage و هم در working tree قرار می‌گیرد.

دستور reset:

در صورتی که بخواهید تعداد زیادی فایل را به وضعیت مشخصی در زمان قبل برگردانید، reset فرمان مناسبی خواهد بود. البته استفاده از این دستور باید با احتیاط کامل صورت گیرد. زیرا در صورت اشتباه، این امکان وجود دارد که دیگر نتوانید به بخشی از

سوابق فایل‌های خود دسترسی داشته باشید. بنابراین این دستور همان قدر که کاربردی است، به همان اندازه نیز خطرناک است. دستور reset را می‌توان به ۳ صورت اجرا نمود:

soft (۱)

mixed (حالت پیشفرض) (۲)

hard (۳)

(۱) در حالت soft تنها head به commit گفته شده منتقل می‌شود و working tree و همچنین stage تغییری نمی‌کند. دقیقا مانند آن که هد یک نوار خوان ویدئویی به جای آن که به آخرین محل ضبط اشاره کند، به عقب برگشته و به قسمتی در قبل برود. در این حالت در صورتی که دستور commit جدیدی ایجاد نشود که باعث پاک شدن commit‌های از آن جا به بعد شود، می‌توان با اجرای مجدد دستور reset و اشاره به آخرین commit، مجدداً head را به سر جای اول برگرداند. البته توجه کنید در صورتی که در هنگام برگرداندن head به commit‌های قبلی، فایل‌هایی تغییر کرده باشند، آن‌ها به صورت خودکار به stage اضافه می‌شوند.

(۲) در حالت mixed که پیش فرض این دستور نیز است، working tree بدون تغییر می‌ماند. اما stage تغییر کرده و دقیقا مانند وضعیت commit می‌شود.

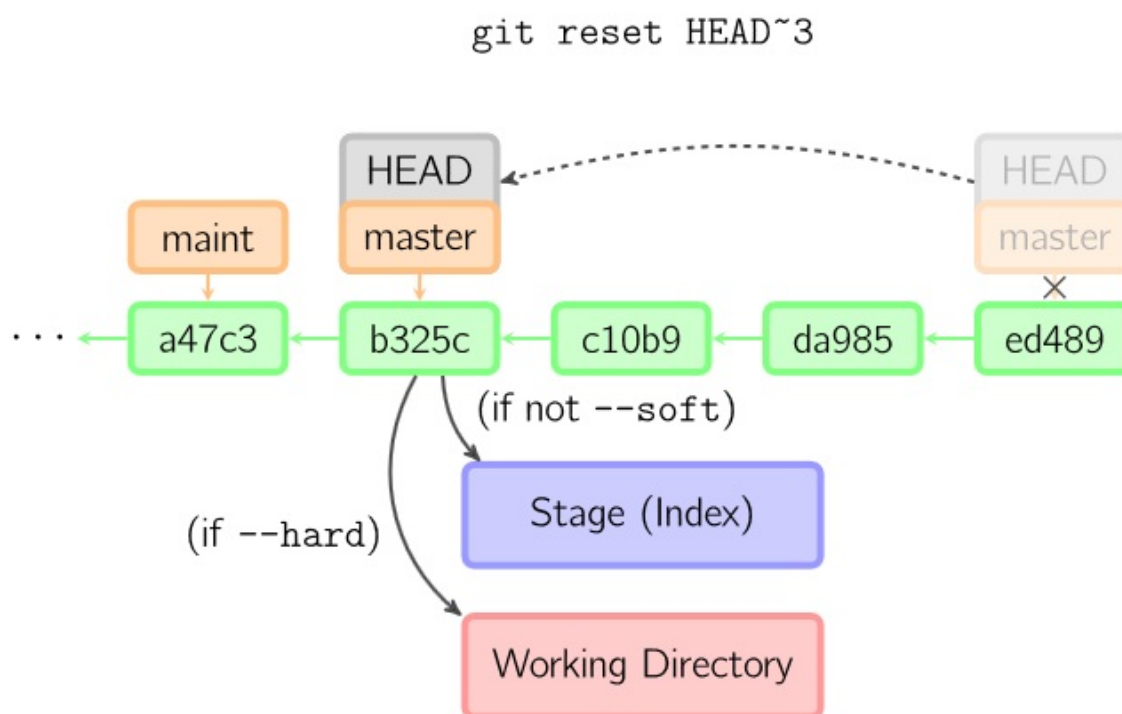
(۳) در این حالت هم working tree و هم stage تغییر می‌کند و عینا وضعیت commit را می‌گیرند که اکنون head به آن اشاره می‌کند. استفاده از این اصلاح کننده بسیار خطرناک‌تر از موارد قبل است.

در هر یک از موارد فوق تا زمانی که دستور commit جدیدی را اجرا نکرده باشید، می‌توانید به وضعیت قبل برگردید. اما اگر commit جدید اجرا شود دیگر امکان بازگشت به commit‌های صورت گرفته بعد از آن وجود ندارد.

نکته مهم:

علیرغم آن که می‌توان به commit‌های گذشته در صورت عدم داشتن commit جدید مراجعه کرد، اما یک اشکال فنی وجود دارد و آن این است که شما نمی‌توانید SHA-1‌های آن commit‌ها را با دستوراتی نظیر log ببینید. بنابراین بهتر است مقدار آن‌ها را قبل از اجرای دستور، ذخیره و تا اطمینان از وضعیت فعلی در محلی نگه دارید.

شکل زیر نمایانگر وضعیت‌های مختلف دستور reset در هنگام بازگشت به سه commit قبل نسبت به وضعیت فعلی Head است:



عنوان: #8 آموزش سیستم مدیریت کد Git

نویسنده: حسام امامی

تاریخ: ۲۰:۳۹ ۱۳۹۱/۰۷/۰۲

آدرس: www.dotnettips.info

برچسب‌ها: Git

در این بخش به بررسی چگونگی ایجاد branch ها و همچنین نحوه‌ی merge کردن آن‌ها خواهیم پرداخت.

Branch:

در این [مقاله](#) به بررسی شاخه‌ها و همچنین ضرورت ایجاد آن‌ها پرداخته شده است. جهت ایجاد یک شاخه می‌توان از دستور زیر استفاده کرد:

```
git branch [branch name]
```

توجه کنید که دستور فوق تنها یک شاخه را ایجاد می‌کند؛ اما همچنان git در شاخه جاری باقی می‌ماند. همچنین جهت مشاهده شاخه‌های ایجاد شده از دستور زیر استفاده می‌شود:

```
git branch
```

شاخه جاری، با یک علامت * در کنار آن مشخص می‌شود:



```
Hessam@HESSAM-PC
$ git branch
a1
a2
* master
```

در حالت پیش‌فرض، تمامی عملیات در git، در شاخه master انجام می‌گیرد. برای تعویض و رجوع به شاخه ایجاد شده می‌توان از دستور checkout استفاده کرد. همانطور که قبلاً گفته شد، یکی دیگر از کاربردهای این دستور تعویض شاخه‌ها است:

```
git checkout [branch name]
```

همچنین می‌توان به صورت همزمان هم شاخه جدید ایجاد کرد و هم به این شاخه جدید سوئیچ نمود:

```
git checkout -b [branch name]
```

تذکر:

در صورتیکه working tree تقریباً clean نباشد، یعنی تغییراتی در فایل‌ها صورت گرفته باشد که این تغییرات هنوز در repository ذخیره نشده باشند، git امکان تعویض شاخه را نخواهد داد. علت تقریباً به این جهت است که در مواردی git می‌تواند برخی تفاوتها را نادیده بگیرد؛ مثلاً اگر فایلی در شاخه‌ی دیگر وجود نداشته باشد. در این حالت سه راهکار پیش روی کاربر است:

(۱) حذف تغییرات

(۲) ذخیره تغییرات در repository

(۳) استفاده از stash

دو مورد نخست مشخص هستند و استفاده از stash در ادامه همین مقاله آورده شده است.

برای حذف یک شاخه ایجاد شده از دستور زیر استفاده می‌شود:

```
git branch -d [branch name]
```

در این حالت نباید در شاخه‌ای باشیم که قصد حذف آن را داریم. همچنین اگر تغییرات در شاخه والد موجود نباشند، git هشدار می‌دهد. این هشدار مبنی بر آن‌که «شاخه دارای تغییراتی است که در صورت حذف آن از بین می‌روند» به کاربر می‌دهد. در این حالت اگر مسر به انجام حذف باشیم، دستور فوق را این بار با -D به کار می‌بریم. بنابراین جهت جلوگیری از اشتباه بهتر است دستور حذف ابتدا با d انجام شود و در صورت نیاز از D استفاده شود. برای تغییر نام یک شاخه از دستور زیر استفاده می‌شود:

```
git branch -m [old name][new name]
```

ادغام شاخه‌ها:

معمولا بعد از آن‌که ویرایش فایل‌ها در یک شاخه به پایان رسید و فایل‌های نهایی تولید شدند، باید این فایل‌ها را در شاخه‌ای دیگر مثلا master قرار داد. برای این منظور، از دستور merge استفاده می‌شود. در هنگام merge باید در شاخه مقصد قرار داشت؛ یعنی در همان شاخه‌ای که قرار است فایل‌های شاخه‌ای دیگر با آن ادغام شوند. برای ادغام یک شاخه به شاخه دیگر از دستور زیر استفاده می‌شود:

```
git merge [branch name]
```

نکته مهم:

در git دو نوع ادغام وجود دارد:

fast forward (۱)

real merge (۲)

حالت اول زمانی اتفاق می‌افتد که در شاخه والد، commit جدیدی ثبت نشده باشد. در این حالت در هنگام merge، اشاره‌گر آخرین فرزند والد، به اولین commit در شاخه‌ی فرزند اشاره می‌کند و دقیقا مانند یک زنجیر دو شاخه به هم متصل می‌شوند. اما اگر در شاخه والد بعد از تشکیل شاخه فرزند commit هایی صورت گرفته باشد، ما یک real merge خواهیم داشت.

تداخل یا conflict:

در هنگام merge کردن شاخه‌ها گاهی این مساله به وجود می‌آید که فایل‌هایی که قرار است تغییرات آن‌ها با هم ادغام شوند، به گونه‌ای ویرایش شده‌اند که git نمی‌تواند عمل merge را انجام دهد. به عنوان مثال تصور کنید فایلی دارای ۱۰ خط است. در شاخه والد خطوط ۱ و ۴ و در شاخه فرزند خطوط ۲ و ۴ ویرایش شده‌اند. git برای ادغام فایل، برای خطوط ۱ و ۲ دچار مشکلی نیست؛ زیرا خط یک را از شاخه والد و خط ۲ را از شاخه فرزند بر می‌دارد. اما برای خط ۴ چه کار کند؟ git نمی‌تواند تصمیم بگیرد که داده نهایی از خط شماره ۴ فرزند است و یا والد. به همین جهت در این‌جا ما یک merge conflict داریم. برای رفع این مشکل یا می‌توان با استفاده از دستور زیر از انجام merge صرف‌نظر کرد:

```
git merge --abort
```

و یا به صورت دستی و یا با استفاده از برخی از ابزارهای موجود، اقدام به رفع دوگانگی فایل‌ها کرد. بعد از رفع conflict ها با دستور:

```
git merge --continue
```

می‌توان ادامه ادغام را خواستار شد.

:Stash

در هنگام توضیح چگونگی تعویض شاخه‌ها، به مطلبی به نام stash اشاره شد. Stash در واقع مکان جدایی در git است که از آن به عنوان محلی جهت ذخیره‌سازی موقت تغییرات استفاده می‌شود. عملکرد stash مانند commit می‌باشد. با این تفاوت که SHA-1 منحصر به فردی برای آن در نظر گرفته نمی‌شود. بنابراین stash محلی است که به طور موقت می‌تواند تغییرات فایل‌ها را ذخیره کند.

برای ایجاد یک stash از دستور زیر استفاده می‌شود:

```
git stash save "[stash name]"
```

همچنین جهت مشاهده تمامی stash‌های ذخیره شده از دستور زیر می‌توان استفاده کرد:

```
git stash list
```

در صورت اجرای این دستور، همانطور که در شکل زیر مشخص است، هر stash توسط یک شماره به صورت:

```
stash@{number}
```

مشخص می‌شود.

```
$ git stash list
stash@{0}: On a2: stash for readme4
```

برای مشاهده تغییرات در یک stash از دستور زیر استفاده می‌شود:

```
git stash show stashes@{[number]}
```

همچنین در صورتیکه جزئیات بیشتری مورد نیاز باشد، می‌توان p- را قبل از شماره stash به دستور فوق اضافه کرد. در صورتیکه بخواهید stash ایجاد شده را حذف کنید، می‌توانید از دستور زیر استفاده کنید:

```
git stash Drop [stash name]
```

همچنین می‌توان با دستور زیر کل stash‌های موجود را حذف نمود:

```
git stash clear
```

برای اعمال تغییرات با استفاده از stash می‌توان از دو دستور استفاده کرد:

۱) pop : در این حالت همانند ساختار پشته، آخرین stash اعمال و از لیست stash‌ها حذف می‌شود.

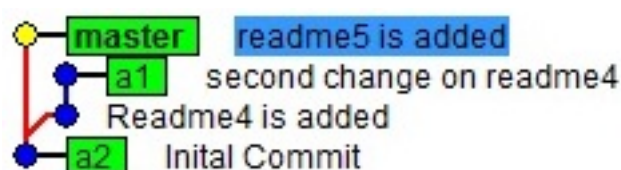
۲) apply : در این دستور، در صورتیکه شماره stash ذکر شود، آن stash اعمال می‌شود. در غیر این صورت، آخرین stash استفاده خواهد شد. تفاوت این دستور با دستور فوق در این است که در اینجا stash بعد از استفاده حذف نمی‌گردد.

دستور rebase:

عملکرد این دستور برای بسیاری از افراد چندان واضح و مشخص نیست و نمی‌توانند تفاوت آن را با دستور merge به خوبی دریابند. برای درک بهتر این موضوع سناریوی زیر را در نظر بگیرید:

تصور کنید شما در حال توسعه یک برنامه هستید و هر از چندگاهی نیاز پیدا می‌کنید تا باگ‌های ایجاد شده در برخی از فایل‌های قبلی خود را رفع کنید. برای این منظور شما برای هر فایل، شاخه‌ای جدید ایجاد کرده و طی چند مرحله، هر فایل را اصلاح می‌کنید. سپس شاخه ایجاد شده را در شاخه اصلی ادغام می‌کنید. حال تصور کنید که تعداد این فایل‌ها افزایش یافته و مثلاً به چند صد عدد برسد. در این حالت شما دارای تعداد زیادی شاخه هستید که تا حدود زیادی سوابق فایل‌های شما را دچار پیچیدگی می‌کنند. در این حالت شاید بهتر باشد که دارای یک فایل سابقه خطی باشیم. بدین معنا که بعد از merge سوابق، شاخه اصلی شما به گونه‌ای در خواهد آمد که انگار هیچ وقت شاخه‌های اضافی وجود نداشته‌اند و تمام تغییرات برای هر فایل پشت سر هم و در شاخه اصلی اتفاق افتاده‌اند. برای این منظور می‌توانید از دستور rebase استفاده کنید.

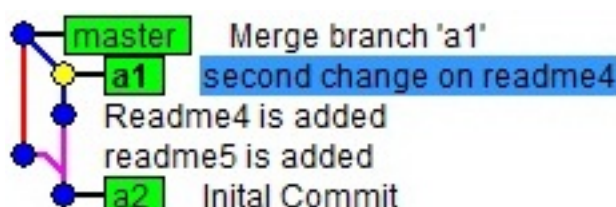
به مثال‌های زیر و شکل شاخه‌ها بعد از اعمال دستورات merge و rebase توجه کنید :



در شاخه master فایل readme5 اضافه شده و در شاخه a2 فایل readme4 اضافه شده و بعد تغییری در آن ذخیره شده است

```
$ git log --oneline
d6ed8ef Merge branch 'a1'
e01afc2 readme5 is added
5182a64 second change on readme4
31d9419 Readme4 is added
183b405 Initial Commit
```

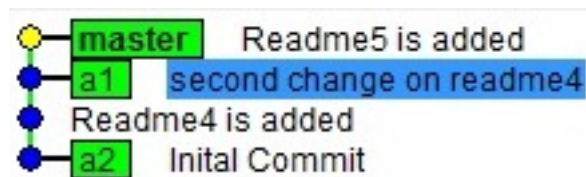
شاخه a1 در master ادغام شده است



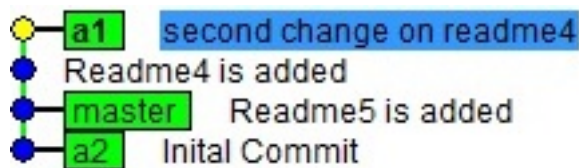
شکل درختی شاخه‌ها پس از ادغام

```
$ git log --oneline
1144826 second change on readme4
4cf90f6 Readme4 is added
184b399 Readme5 is added
183b405 Inital Commit
```

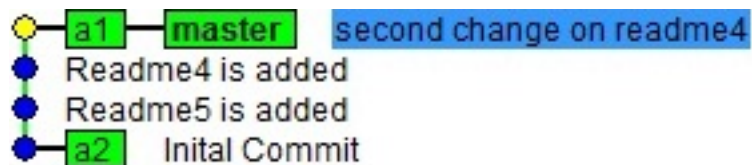
در شکل فوق از دستور rebase استفاده شده است



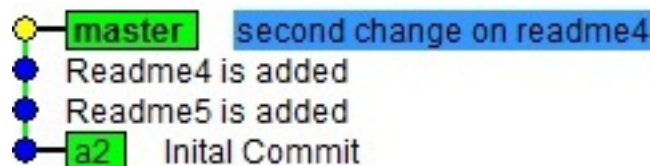
شکل شاخه‌ها بعد از اعمال rebase



همانطور که مشاهده می‌شود با سوئیچ به شاخه master هنوز head در محل قبلی خود است



با اعمال دستور ادغام، head به محل آخرین commit منتقل می‌شود



اکنون می‌توان شاخه a1 را حذف کرد. همانطور که دیده می‌شود، به نظر می‌رسد این شاخه هیچگاه وجود نداشته است.

تذکر:

بعد از انجام دستور rebase باید از دستور merge استفاده کرد. زیرا هر شاخه برای خود head جداگانه‌ای دارد. بعد از اجرای این فرمان، هنوز head در شاخه مقصد به آخرین فرمان خود اشاره می‌کند. در آخرین فرمان، شاخه‌ای اضافه شده، بنابراین اجرای دستور merge حالت fast forward را پیاده می‌کند و head به آخرین commit منتقل می‌شود.

تذکر:

همانطور که مشاهده کردید، دستور rebase به صورت فوق سوابق شاخه را از بین می‌برد. بنابراین نباید از این دستور برای شاخه‌های عمومی یعنی آنهایی که دیگران تغییرات آنها را دنبال می‌کنند استفاده کرد. شکل استفاده از این دستور به صورت زیر است:

```
git rebase [destination branch]
```

یا

```
git rebase [destination][source]
```

همانند دستور merge این دستور نیز ممکن است سبب ایجاد تداخل شود و برای رفع این موضوع باید مانند merge عمل کرد؛ این دستور نیز دارای دو اصلاح کننده --abort و --continue می‌باشد

تذکر مهم :

به تفاوت محل درج ادغام‌ها در merge و rebase توجه کنید.

دستور cherry-pick :

با استفاده از این فرمان می‌توان یک یا چند commit را از شاخه‌ای برداشته و در شاخه‌ی دیگری اعمال کنیم. در واقع دستور cherry-pick همانند بخشی از دستور rebase است. با این تفاوت که rebase در واقع چندین cherry-pick را یکجا انجام می‌دهد. البته در cherry-pick هر commit بدون تغییر باقی می‌ماند. بیشترین کاربرد این دستور برای اعمال patch و رفع باگ‌ها در یک شاخه است. این دستور به صورت زیر استفاده می‌شود:

```
git cherry-pick [branch name]
```

تا اینجا هر آنچه درباره git آموختیم در رابطه با عملکرد git به صورت محلی بود. اما یکی از ویژگی‌های سیستم‌های توزیع شده، امکان استفاده از آن‌ها به صورت remote می‌باشد.

در مورد git تفاوت چندانی بین سرورها و کلاینت‌ها وجود ندارد. تنها تفاوت، نحوه‌ی پیکربندی سرور است که این امکان را می‌دهد تا چندین کلاینت به صورت همزمان به آن متصل شده و با repository آن کار کنند. اما عملاً تفاوتی بین repository موجود در کلاینت و سرور نیست.

تذکر ۱: در این مقاله از وب سایت [github](https://github.com) برای توضیح مثال‌ها استفاده شده است. github قدیمی‌ترین و قدرتمندترین وب سایت برای مدیریت repository های git است. اما اجباری در انتخاب آن نیست؛ زیرا انتخاب‌های فراوانی از جمله [bitbucket](https://bitbucket.org/) نیز وجود دارد.

تذکر ۲: نام مستعار origin اجباری نیست؛ اما از آن جهت که نام پیش فرض است، در اکثر مثال‌ها و توضیحات استفاده شده است.

قبل از شروع مبحث بهتر است کمی درباره‌ی پروتکل‌های ارتباطی پشتیبانی شده توسط git صحبت کنیم:
git از ۴ نوع پروتکل پشتیبانی می‌کند:

(۱) http(s): پروتکل http با پورت ۸۰ و https با پورت ۴۴۳ کار می‌کند و معمولاً فایروال‌ها مشکلی با این پروتکل‌ها ندارند. از هر دوی آن‌ها می‌توان برای عملیات نوشتن و یا خواندن استفاده نمود و می‌توان آن‌ها را به گونه‌ای تنظیم کرد که برای برقراری ارتباط نیاز به تأیید هویت داشته باشند.

(۲) git: پروتکلی فقط خواندنی است که به صورت anonymous و بر روی پورت ۹۴۱۸ کار می‌کند. شکل استفاده از آن به صورت زیر است و معمولاً در github کاربرد فراوانی دارد:

```
git://github.com/[username]/[repositoryname].git
```

(۳) ssh: همان پروتکل استفاده شده در یونیکس است که بر اساس مقادیر کلیدهای عمومی و خصوصی تعیین هویت را انجام می‌دهد. شکل استفاده از آن به صورت زیر است و بر روی پورت ۲۲ کار می‌کند و امکان نوشتن و خواندن را می‌دهد:

```
git@github.com:[username]/[repositoryname].git
```

(۴) file: تنها استفاده محلی دارد و امکان نوشتن و خواندن را می‌دهد.

نحوه‌ی عملکرد git به صورت remote:

به طور کلی هر برنامه‌نویس نیاز به دو نوع از دستورات دارد تا همواره repository محلی با remote هماهنگ باشد:

(۱) بتواند به طریقی داده‌های موجود در repository محلی خود را به سمت سرور بفرستد.

(۲) این امکان را داشته باشد تا repository محلی خود را با استفاده از repository در سمت سرور آپدیت نماید تا از آخرین تغییراتی که توسط بقیه اعضای گروه صورت گرفته است آگاهی یابد.

طریقه رفتار git برای کار با repository های remote به صورت زیر است:

هنگامی که کاربر قصد دارد تا repository یا شاخه‌ای از آن را به سمت سرور بفرستد، git ابتدا یک شاخه با نام همان شاخه به اضافه origin/ ایجاد می‌کند. مثلاً برای شاخه master، آن نام به صورت زیر می‌شود:

```
origin/master
```

عملکرد این شاخه دقیقاً مانند دیگر شاخه‌های git است؛ با این تفاوت که امکان check-in یا out برای این نوع شاخه‌ها وجود

ندارد. زیرا git باید این شاخه‌ها را با شاخه‌ها متناظرشان در remote هماهنگ نگه دارد.
از این پس این شاخه‌ی ایجاد شده، به عنوان واسطی بین شاخه محلی و شاخه راه دور عمل می‌کند.

:cloning

با استفاده از دستور clone می‌توان یک repository در سمت سرور را به طور کامل در سمت کلاینت کپی کرد. به عنوان مثال repository مربوط به کتابخانه jquery از وب سایت github به صورت زیر است:

```
git clone https://github.com/jquery/jquery.git
```

همچنین می‌توان با استفاده از دستور زیر پوشه‌ای با نامی متفاوت را برای repository محلی انتخاب نمود:

```
git clone [URL][directory name]
```

اضافه کردن یک remote repository:

برای آن‌که بتوان تغییرات یک remote repository را به repository محلی منتقل نمود، ابتدا باید آن را به لیست repositoryهای ریموت که در فایل config ذخیره می‌شود به شکل زیر اضافه نمود:

```
git remote add [alias][URL]
```

در دستور فوق، برای repository باید یک نام مستعار تعریف کرد و در بخش URL باید آدرسی که سرور به وسیله آن امکان دریافت اطلاعات را به ما می‌دهد، نوشت. البته این بستگی به نوع پروتکل انتخابی دارد. به عنوان مثال:

```
git remote add origin https://github.com/jquery/jquery.git
```

اگر بخواهیم لیست repositoryهایی که به صورت remote اضافه شده‌اند را مشاهده کنیم، از دستور زیر استفاده می‌کنیم:

```
git remote
```

در صورتی‌که دستور فوق را با v- تایپ کنید اطلاعات کامل‌تری در رابطه با repositoryها مشاهده خواهید کرد.
همچنین برای حذف یک remote repository از دستور زیر استفاده می‌کنیم:

```
git remote [alias] -rm
```

در صورتی‌که بخواهید لیستی از شاخه‌های remote را مشاهده کنید کافیست از دستور زیر استفاده کنید:

```
git branch -r
```

همچنین می‌توان از دستور زیر برای نمایش تمامی شاخه‌ها استفاده کرد:

```
git branch -a
```

:fetch

برای دریافت اطلاعات از دستور زیر استفاده می‌کنیم:

```
git fetch [alias][alias/branch name]
```

در صورتی که تنها یک repository باشد می توان از نوشتن نام مستعار صرف نظر نمود. همچنین اگر شاخه یا شاخه های مورد نظر به صورت track شده باشند، می توان قسمت دوم دستور فوق را نیز ننوشت. اگر بعد از اجرای دستور فوق، بر روی یک شاخه log بگیرید، خواهید دید که تغییرات در شاخه محلی اعمال نشده است زیرا دستور فوق تنها داده ها را بر روی شاخه [origin/[branchname] ذخیره کرده است. برای آپدیت شدن شاخه اصلی باید با استفاده از دستور merge آن را در شاخه مورد نظر ادغام کرد.

:pulling

چون کاربرد دو دستور fetch و merge به صورت پشت سر هم زیاد است git دو دستور فوق را با استفاده از pull انجام می دهد:

```
pull [alias][remote branch name ]
```

اگر دو مقدار فوق را برای دستور pull تعیین نکنید، ممکن است در هنگام اجرای دستور فوق با خطایی مواجه شوید مبنی بر اینکه git نمی داند دقیقا شاخه ریموت را با کدام شاخه محلی باید ادغام کند. این مشکل زمانی پیش می آید که برای شاخه ریموت یک شاخه محلی متناظر وجود نداشته باشد. برای ایجاد تناظر بین دو شاخه ریموت و لوکال در گذشته باید فایل config را تغییر می دادیم، اما نسخه جدید git دستوری را برای آن دارد:

```
git branch --set-upstream [local brnach][alias/branch name]
```

با اجرای این دستور از این پس شاخه محلی تغییرات شاخه remote را دنبال می کند.

:pushing

با استفاده از push می توان تغییرات ایجاد شده را به remote repository انتقال داد:

```
git push -u [alias][branch name ]
```

وجود -u در اینجا بدین معنا است که ما می خواهیم تغییرات repository در سمت سرور دنبال شود. در صورت استفاده نکردن از -u بایستی برای push هر بار مقادیر داخل کروشه ها را بنویسیم. در صورتی که بعدا بخواهیم، می توان توسط همان دستوری که در قسمت pull گفته شد دو شاخه را به هم وابسته کنیم.

:tag

همانطور که قبلا گفته شد تگ ها برای نشانه گذاری و دسترسی راحت تر به commit ها هستند. برای ایجاد یک تگ از دستور زیر استفاده می شود:

```
git tag [tag name]
```

همچنین می توان با -a برای تگ پیامی نوشت و یا با -s آن را امضا کرد. برای مشاهده تگ ها از دستور زیر استفاده می شود:

```
git tag
```

در حالت پیش فرض git تگ ها را push نمی کند. برای push کردن تگ ها باید دستور push را با اصلاح کننده --tags به کاربرید.

نظرات خوانندگان

نویسنده: reza

تاریخ: ۱۵:۵۵ ۱۳۹۱/۰۷/۲۰

اگر بخواهیم بر روی سرور خودمان راه اندازی کنیم چه راه حلی وجود دارد ؟

نویسنده: حسام امامی

تاریخ: ۰:۲۲ ۱۳۹۱/۰۷/۲۱

میشه از git daemon استفاده کرد که در واقع یک سرور برای repository های git است

توی این مقاله از CopSSH استفاده کرده [codeproject](#)

از scm-manager هم میتونید استفاده کنید

در ضمن همانطور که گفتم از پروتکل فایل هم پشتیبانی میشه یعنی میتونید در شبکه شیر کنید

نویسنده: neo

تاریخ: ۰:۰ ۱۳۹۱/۰۸/۱۸

با سلام خدمت شما آقای امامی.

من در قسمت pushing وقتی میخواهم این دستور را git push origin master اجرا کنم یه یوزر و پسورد میخواهد؟ این قسمت

را میشه بگید چطوری است؟

دوستدار شما علیرضا از شهر ممقان.

نویسنده: حسام امامی

تاریخ: ۲۱:۵۴ ۱۳۹۱/۰۸/۲۰

با سلام

دقیقا متوجه منظور شما نشدم

اگر شما قصد داشته باشید که به یک سرور راه دور repository را push کنید بسته به نوع پروتکل از شما شناسه کاربری و رمز

عبوری می‌خواهد که همان سایت به شما داده است

موفق باشید

نویسنده: سام ناصری

تاریخ: ۸:۲۱ ۱۳۹۱/۱۲/۲۹

فکر کنم بشود در محیط LAN با استفاده از آدرسهای شبکه محلی repository ها را بین کاربران به اشتراک گذاشت. مثلاً من

repository خودم را share کنم و همکارم هم repository خودش را. بعد هر کدام repository اون یکی را به remote ها اضافه کند.

همچنین میتوان یک repository از نوع bare در سرور شبکه LAN تعریف کرد و همه به آن به عنوان repository مادر متصل

شوند.

البته من خیلی مطمئن نیستم. منتظرم تا ببینم نظر حسام در اینباره چیه؟

نویسنده: حسن

تاریخ: ۲۳:۵۰ ۱۳۹۱/۱۲/۲۹

بله. امکان تهیه [Shared Repository](#) هست. [این مطلب](#) هم مفید است.

نویسنده: kish

تاریخ: ۱۵:۳۱ ۱۳۹۳/۰۳/۲۰

سلام

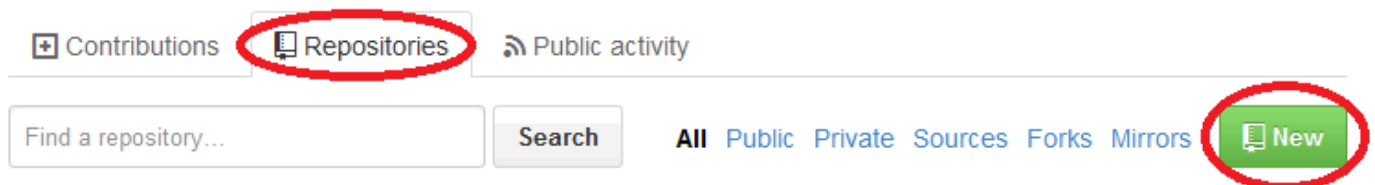
در webstorm تحت LAN چه جوری می‌تونم از git استفاده کنم؟

با تشکر


از نگارش 2012 ویژوال استودیو، امکان کار با مخازن Git، به صورت یکپارچه و توکار و بدون نیاز به ابزارهای جانبی، توسط آن فراهم شده‌است. در ادامه قصد داریم به کمک این ویژگی توکار، نحوه‌ی ارسال یک پروژه‌ی از پیش موجود VS.NET را برای اولین بار به GitHub بررسی کنیم.

تنظیمات مقدماتی GitHub

در ابتدا نیاز است یک مخزن کد خالی را در GitHub ایجاد کنید. برای این منظور به برگه‌ی Repositories در اکانت GitHub خود مراجعه کرده و بر روی دکمه‌ی New کلیک کنید:



سپس در صفحه‌ی بعدی، نام پروژه را به همراه توضیحاتی وارد نمائید و بر روی دکمه‌ی Create repository کلیک کنید. در اینجا سایر گزینه‌ها را انتخاب نکنید. نیازی به انتخاب گزینه‌ی READ ME و یا انتخاب مجوز و غیره نیست. تمام این کارها را در سمت پروژه‌ی اصلی می‌توان انجام داد و یا VS.NET فایل‌های ignore را به صورت خودکار ایجاد می‌کند. در اینجا صرفاً هدف، ایجاد یک مخزن کد خالی است.

Owner  **VahidN** / **Repository name**

Great repository names are short and memorable. Need inspiration? How about [turnt-octo-bugfixes](#).

Description (optional)

☒ **Public**
Anyone can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.


☐ **Initialize this repository with a README**
This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally.

Add .gitignore: **None** | Add a license: **None** ⓘ

Create repository

از اطلاعات صفحه‌ی بعدی، تنها به آدرس مخصوص GitHub آن نیاز داریم. از این آدرس در VS.NET برای ارسال اطلاعات به سرور استفاده خواهیم کرد:

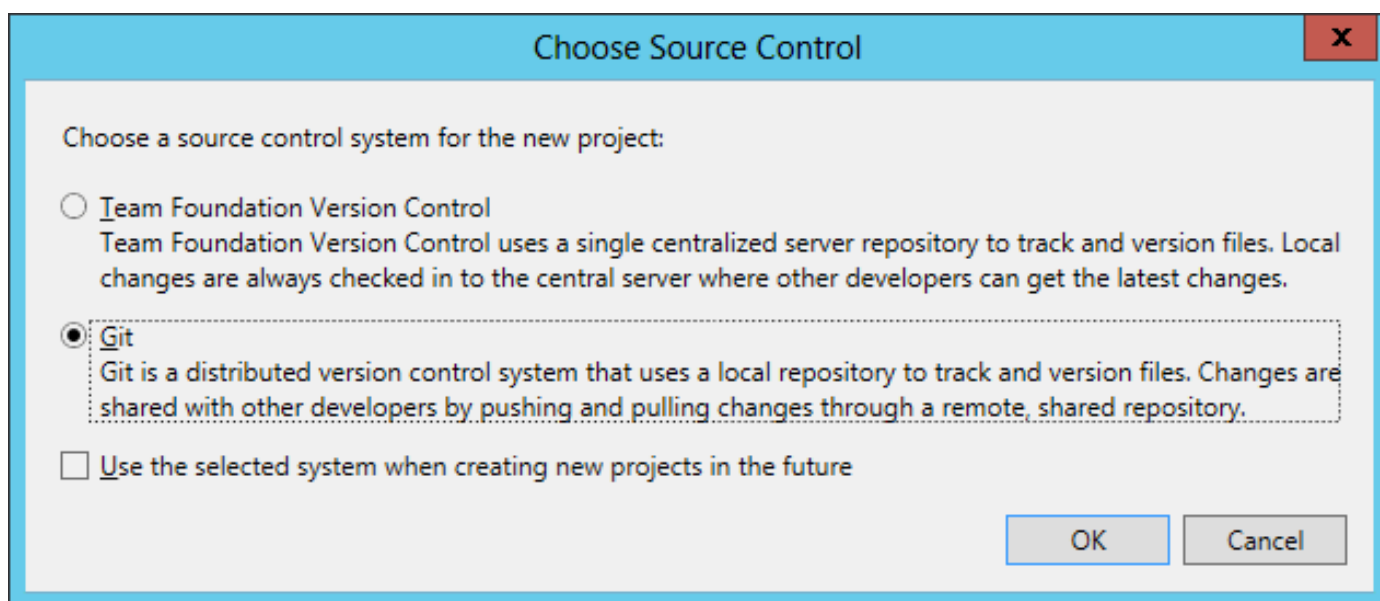
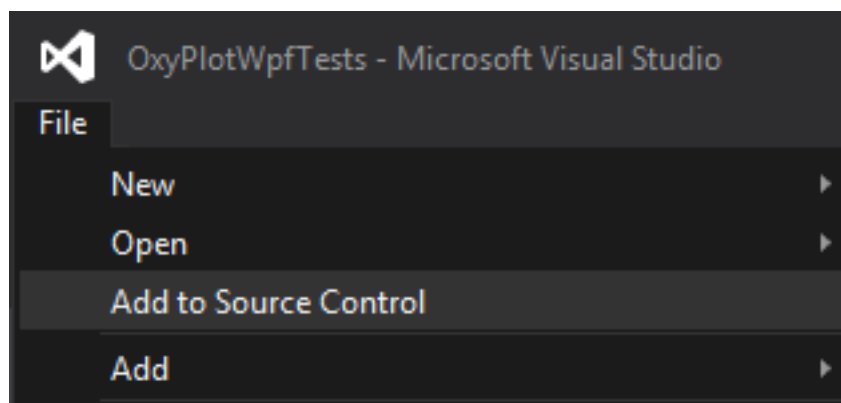
Quick setup — if you've done this kind of thing before

 **Set up in Desktop** or **HTTPS** **SSH**

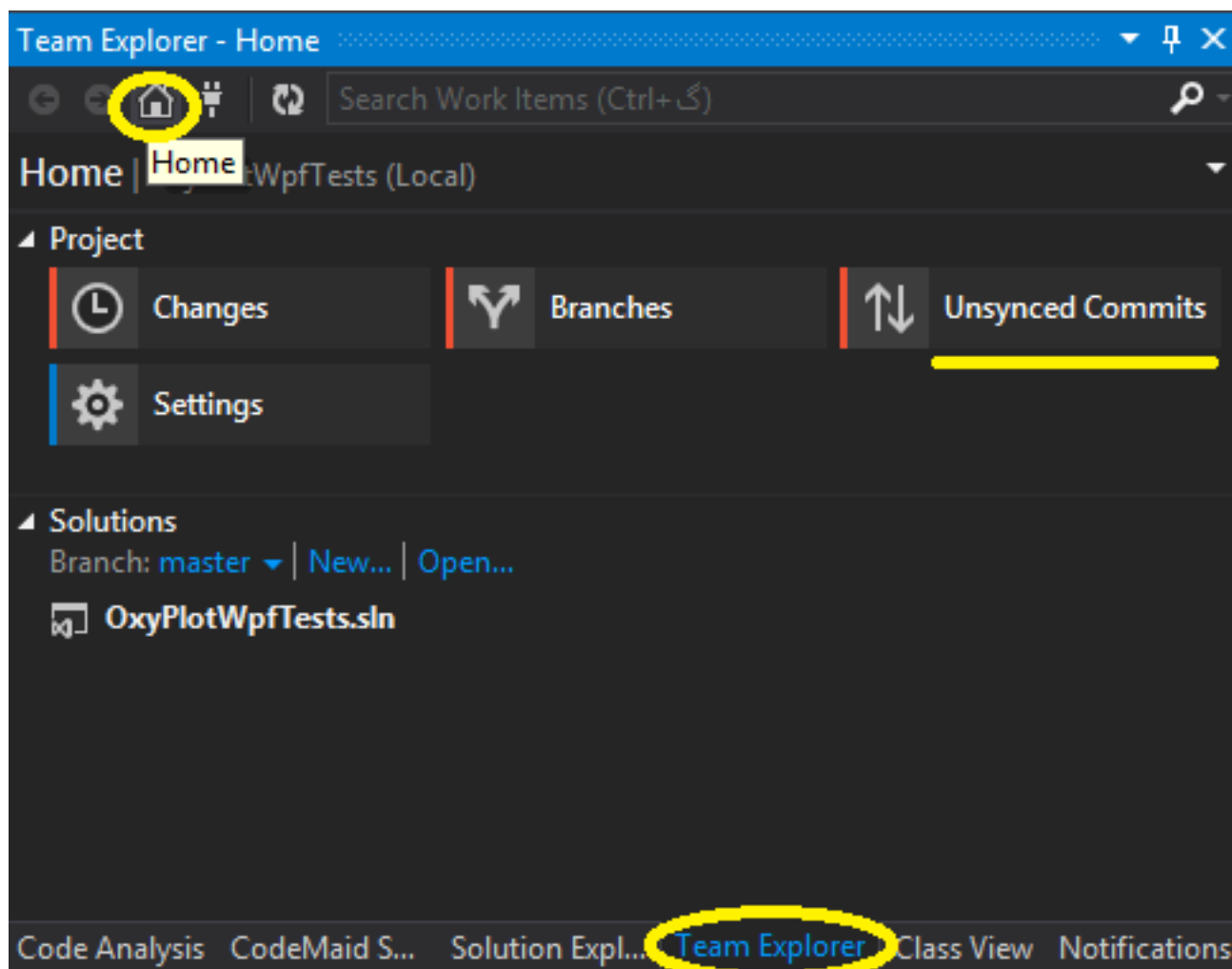
We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

تنظیمات VS.NET برای ارسال پروژه به مخزن GitHub

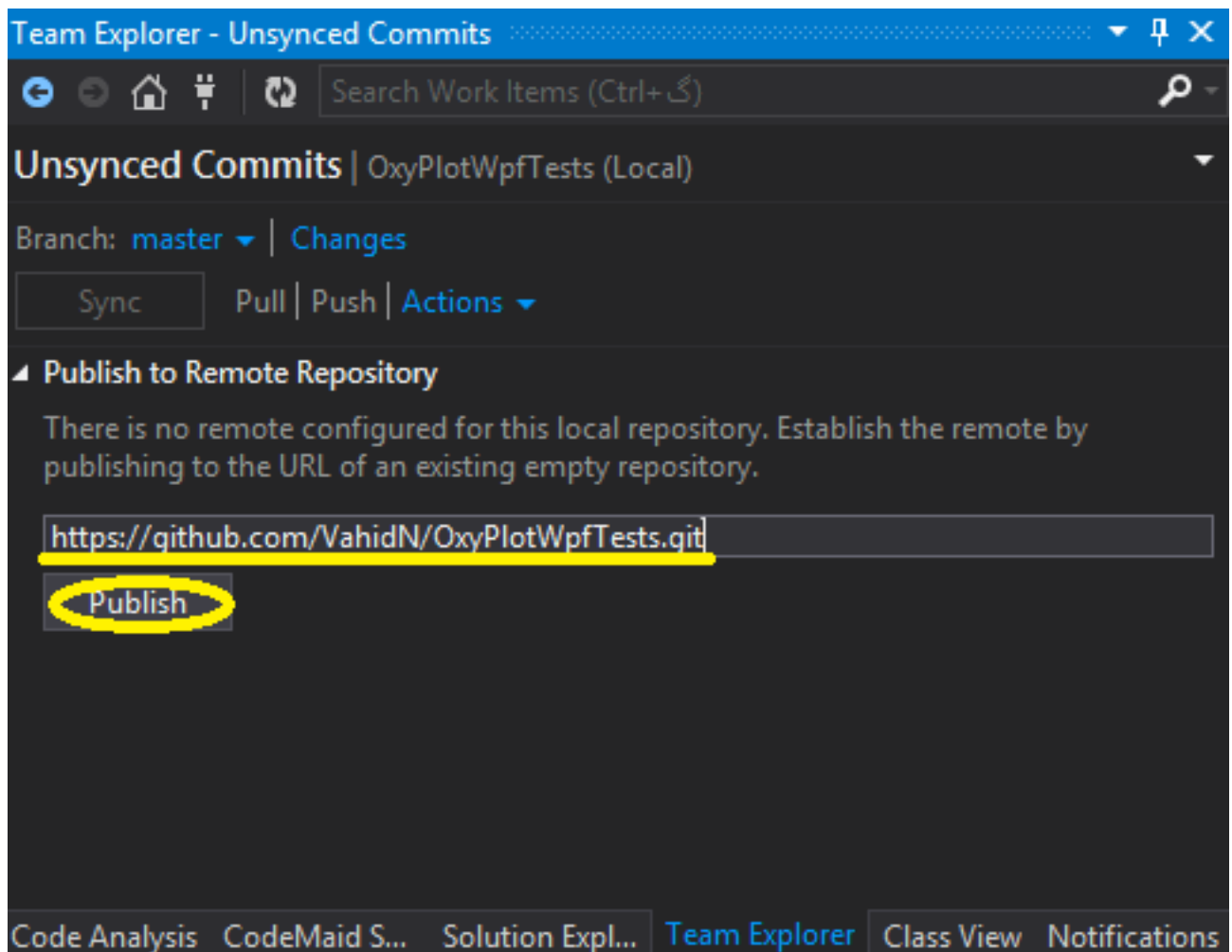
پس از ایجاد یک مخزن کد خالی در GitHub، اکنون می‌توانیم پروژه‌ی خود را به آن ارسال کنیم. برای این منظور از منوی `File`، گزینه‌ی `Add to source control` را انتخاب کنید و در صفحه‌ی باز شده، گزینه‌ی `Git` را انتخاب نمایید:



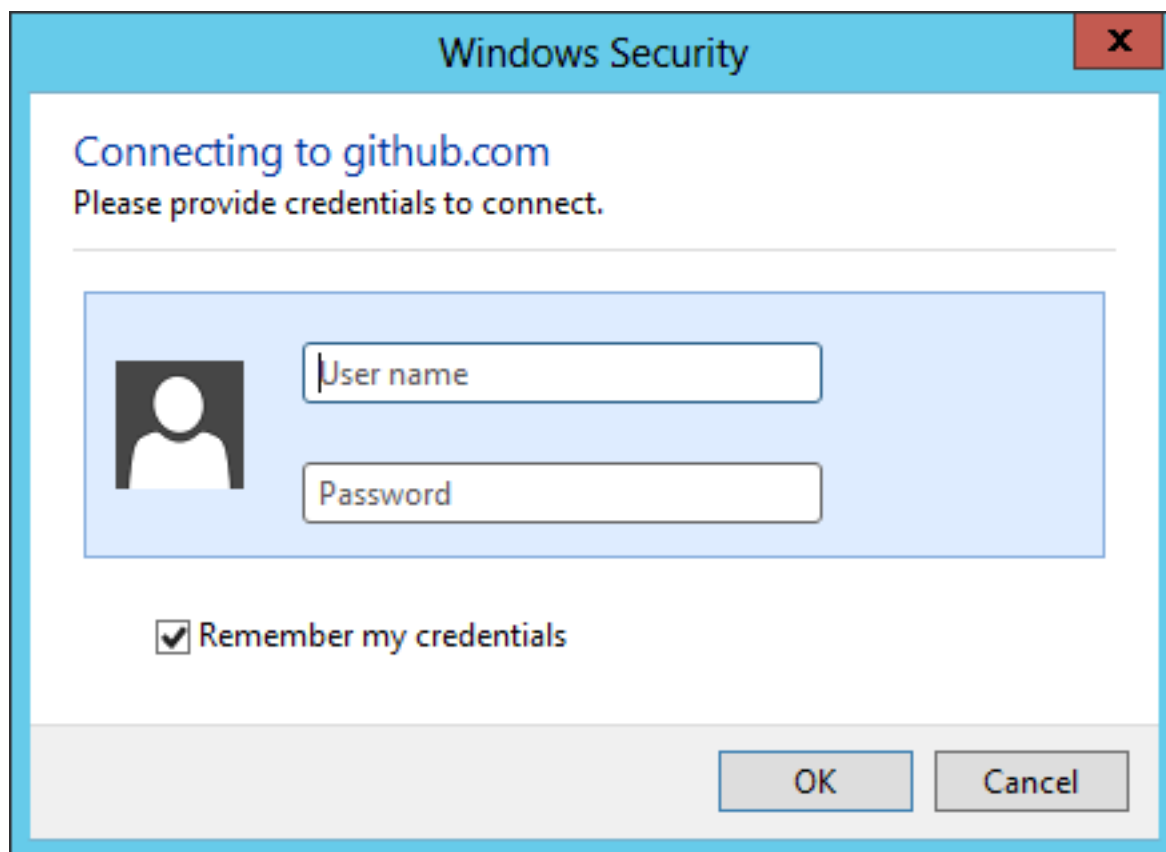
سپس در کنار برگه‌ی Solution Explorer، برگه‌ی Team Explorer را انتخاب کنید. در اینجا بر روی دکمه‌ی Home در نوار ابزار آن کلیک کرده و سپس بر روی دکمه‌ی Unsynced commits کلیک نمایید.



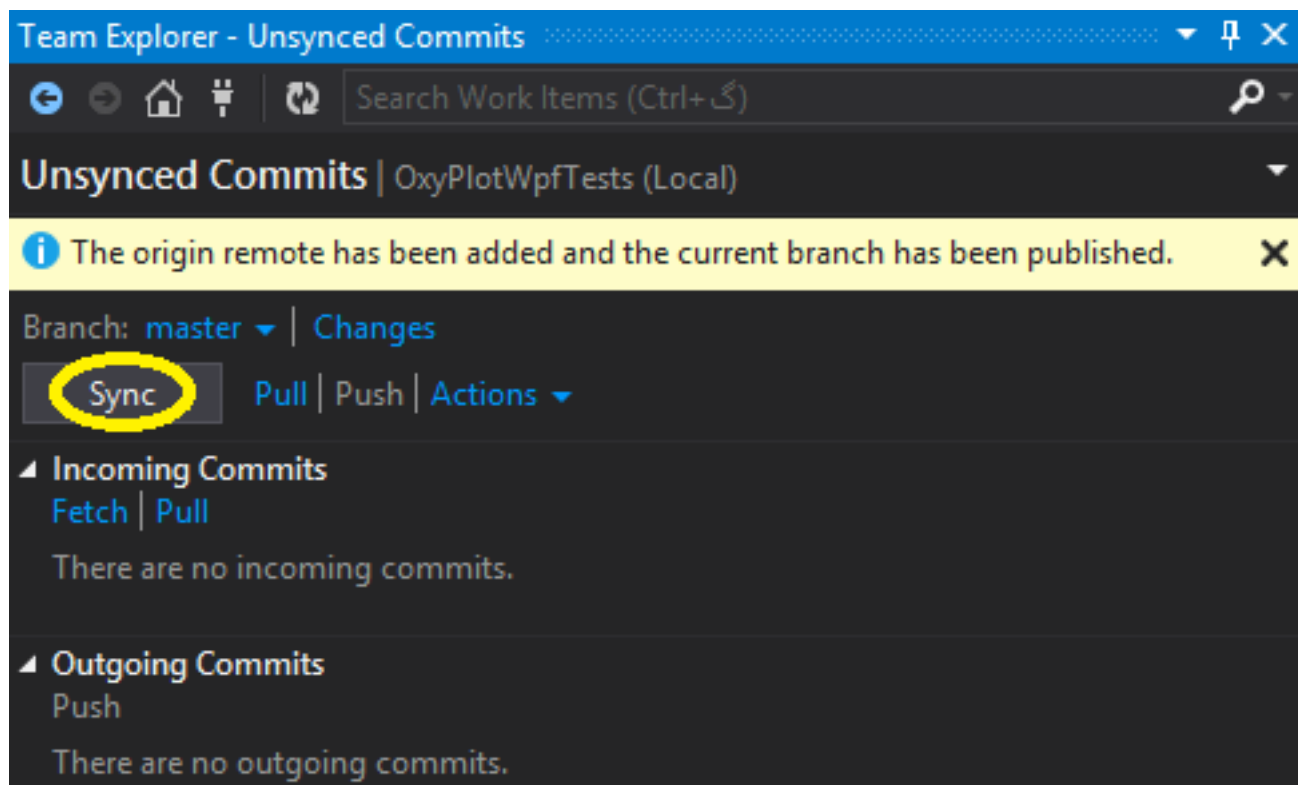
در ادامه در صفحه‌ی باز شده، همان آدرس مخصوص مخزن کد جدید را در GitHub وارد کرده و بر روی دکمه‌ی Publish کلیک کنید:



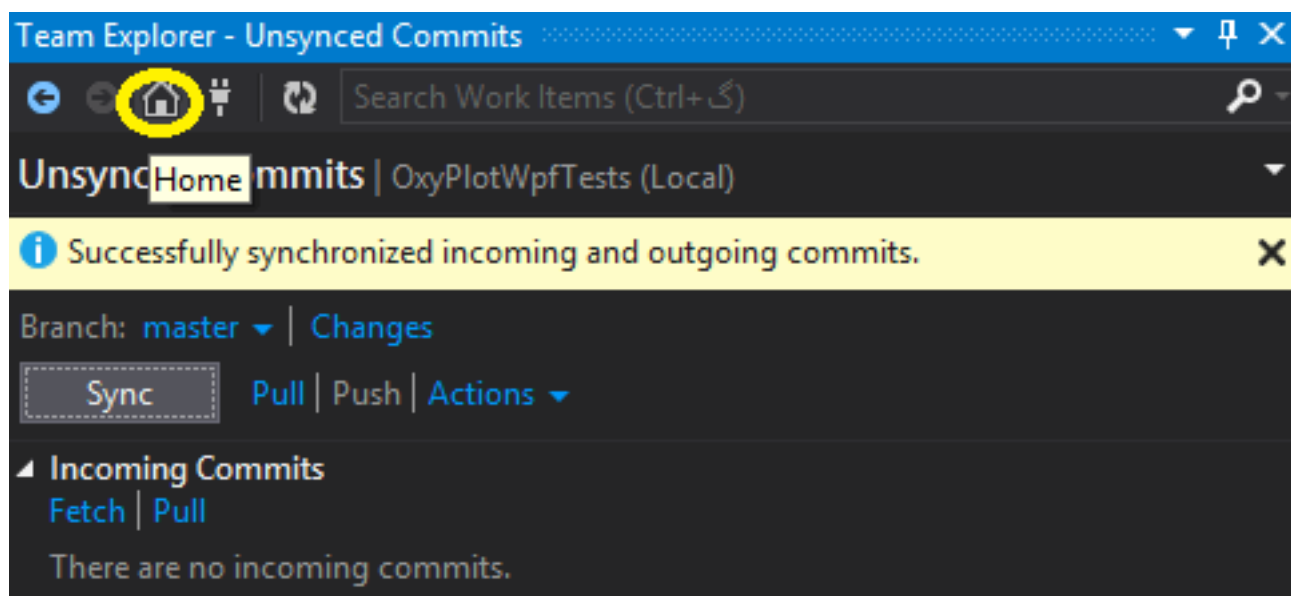
در اینجا بلافاصله صفحه‌ی لاگینی ظاهر می‌شود که باید در آن مشخصات اکانت GitHub خود را وارد نمایید:



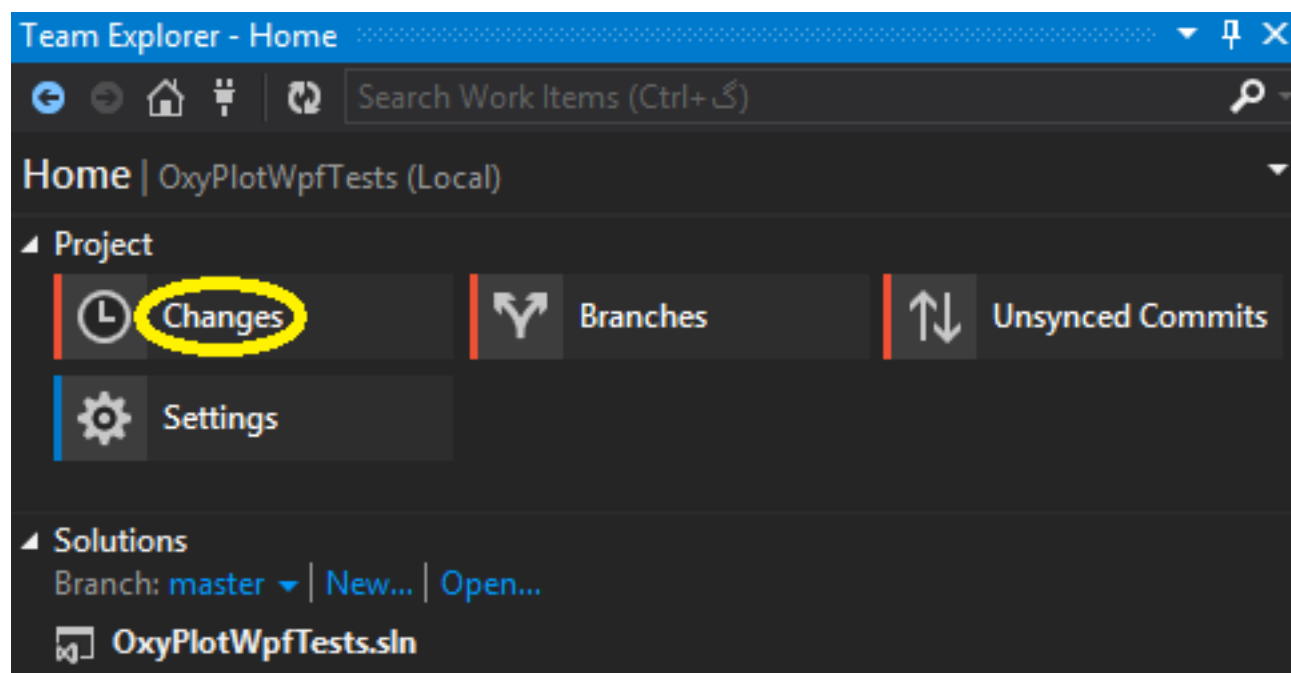
به این ترتیب عملیات Publish اولیه انجام شده و تصویر ذیل نمایان خواهد شد:



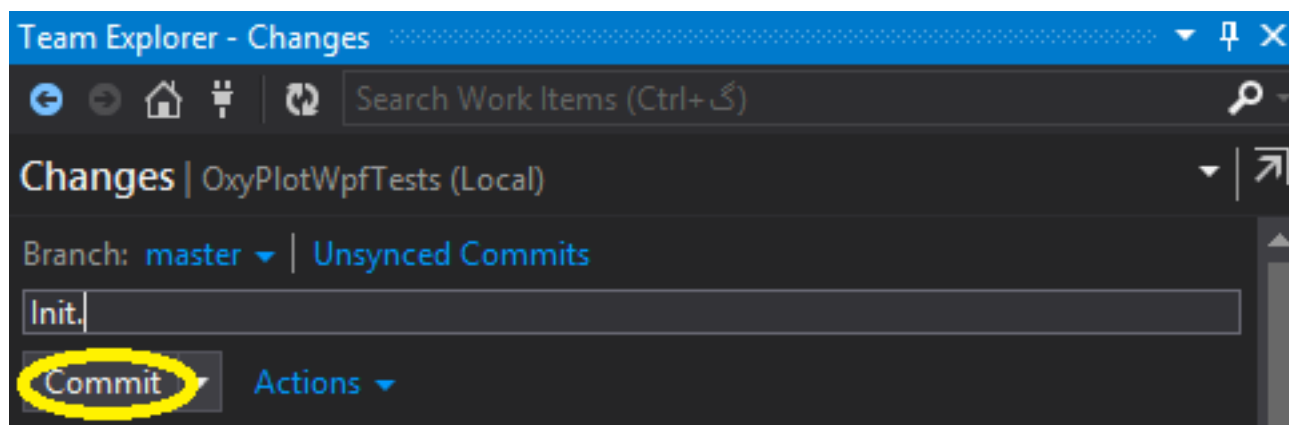
در اینجا بر روی دکمه‌ی Sync کلیک کنید. به این ترتیب مخزن کد GitHub به پروژه‌ی جاری متصل خواهد شد:



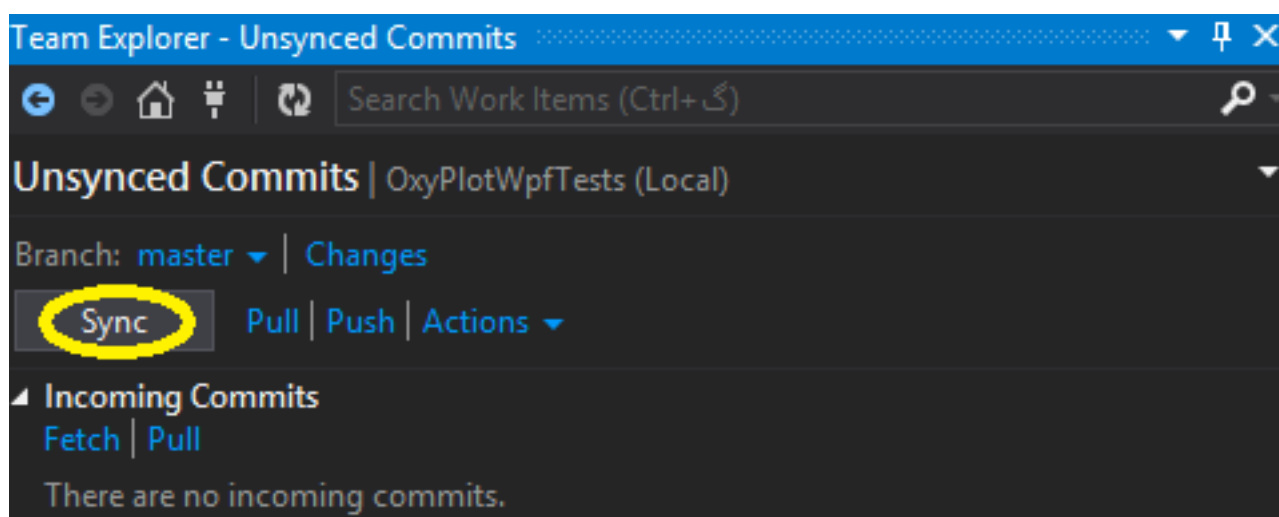
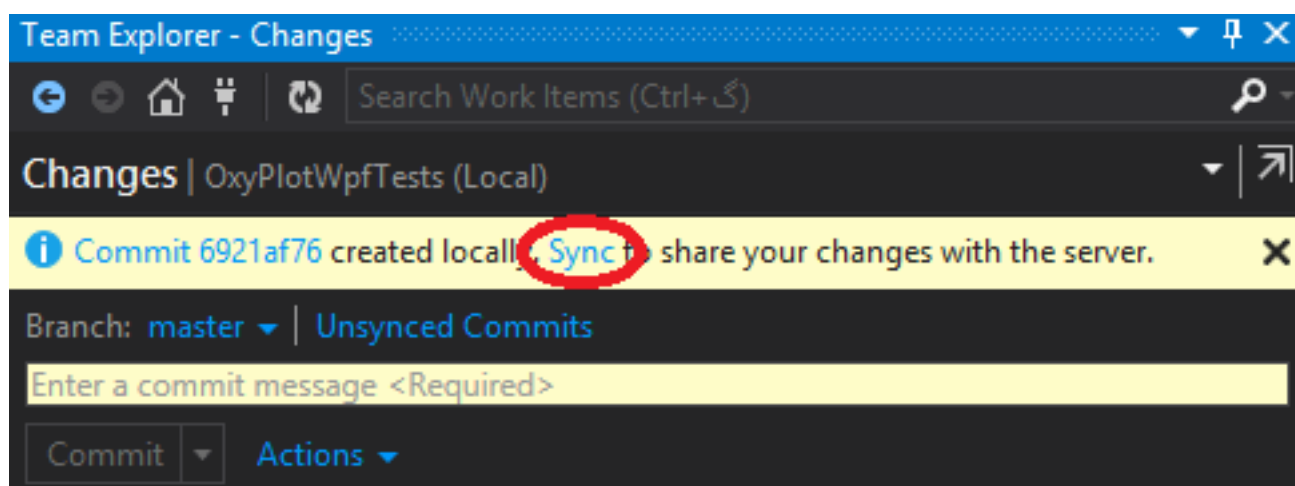
سپس نیاز است فایل‌های موجود را به مخزن کد GitHub ارسال کرد. بنابراین پس از مشاهده‌ی پیام موفقیت آمیز بودن عملیات همگام‌سازی، بر روی دکمه‌ی Home در نوار ابزار کلیک کرده و اینبار گزینه‌ی Changes را انتخاب کنید:



در اینجا پیام اولین ارسال را وارد کرده و سپس بر روی دکمه‌ی Commit کلیک کنید:



پس از مشاهده‌ی پیام موفقیت آمیز بودن commit محلی، نیاز است تا آنرا با سرور نیز هماهنگ کرد. به همین جهت در اینجا بر روی لینک Sync کلیک کرده و در صفحه‌ی بعدی بر روی دکمه‌ی Sync کلیک کنید:



اندکی صبر کنید تا فایل‌ها به سرور ارسال شوند. اکنون اگر به GitHub مراجعه کنید، فایل‌های ارسالی قابل مشاهده هستند:

An OxyPlot Sample — Edit

2 commits 1 branch 0 releases 1 contributor

branch: master OxyPlotWpfTests / +

Init.

VahidN authored 3 minutes ago latest commit 6921af7608

OxyPlotWpfTests	Init.	3 minutes ago
.gitattributes	Initial commit to add default .gitignore and .gitAttribute files.	18 minutes ago
.gitignore	Initial commit to add default .gitignore and .gitAttribute files.	18 minutes ago
OxyPlotWpfTests.sln	Init.	3 minutes ago

We recommend **adding a README** to this repository to help give people an overview of your project. [Add a README](#)

اعمال تغییرات بر روی پروژه‌ی محلی و ارسال به سرور

در ادامه می‌خواهیم دو فایل README.md و LICENSE.md را به پروژه اضافه کنیم. پس از افزودن آن‌ها، یا هر تغییر دیگری در پروژه، اینبار برای ارسال تغییرات به سرور، تنها کافی است به برگه‌ی Team explorer مراجعه کرده و ابتدا بر روی دکمه‌ی Home کلیک کرد تا منوی انتخاب گزینه‌های آن ظاهر شود. در اینجا تنها کافی است گزینه‌ی Changes را انتخاب و دقیقاً همان مراحل عنوان شده‌ی پیشین را تکرار کرد. ابتدا ورود پیام Commit و سپس Commit. در ادامه Sync محلی و سپس Sync با سرور.

نظرات خوانندگان

نویسنده:

سیروس

تاریخ:

۱۸:۴۵ ۱۳۹۳/۱۰/۲۵

میخواستم بدونم برای پروژه‌های که نمیخواهیم کد اون در دسترس عموم قرار بگیره مانند پروژه‌های شرکت‌های برنامه نویسی، آیا Github قابل استفاده و اطمینان هست؟ و همینکه مخزن ما بصورت خصوصی باشه، کافیه؟

نویسنده:

وحید نصیری

تاریخ:

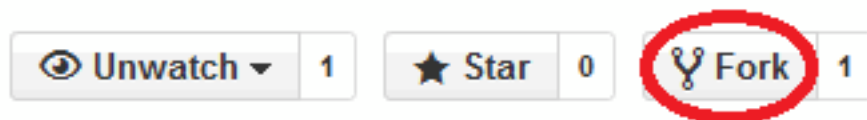
۱۸:۵۷ ۱۳۹۳/۱۰/۲۵

GitHub امکان تهیه مخزن کد خصوصی هم دارد ولی [رایگان نیست](#) . سایت [BitBucket](#) امکان ایجاد مخزن کد خصوصی رایگان را دارد؛ البته با محدودیت حداکثر 5 کاربر تعریف شده‌ی برای کار با یک مخزن.

فرض کنید برای رفع باگی در پروژه‌ای از GitHub، ایده‌ای دارید. روند کاری اعلام آن، روش‌های مختلفی می‌تواند داشته باشند؛ از باز کردن یک Issue جدید تا فرستادن یک فایل zip و غیره. اما روش استاندارد مشارکت در پروژه‌های Git، ارسال یک PR یا Pull Request است. در ادامه نحوه‌ی انجام این کار را به کمک امکانات توکار VS.NET بررسی خواهیم کرد.

ایجاد یک Fork جدید در GitHub

برای ارسال تغییرات انجام شده بر روی یک پروژه، نیاز است به صاحب یا مسئول آن مخزن در GitHub مراجعه و سپس درخواست دسترسی اعمال تغییرات را نمود. در این حالت، احتمال اینکه جواب منفی دریافت کنید، بسیار زیاد است. جهت مدیریت یک چنین مواردی، قابلیت به نام ایجاد یک Fork پیش بینی شده است.

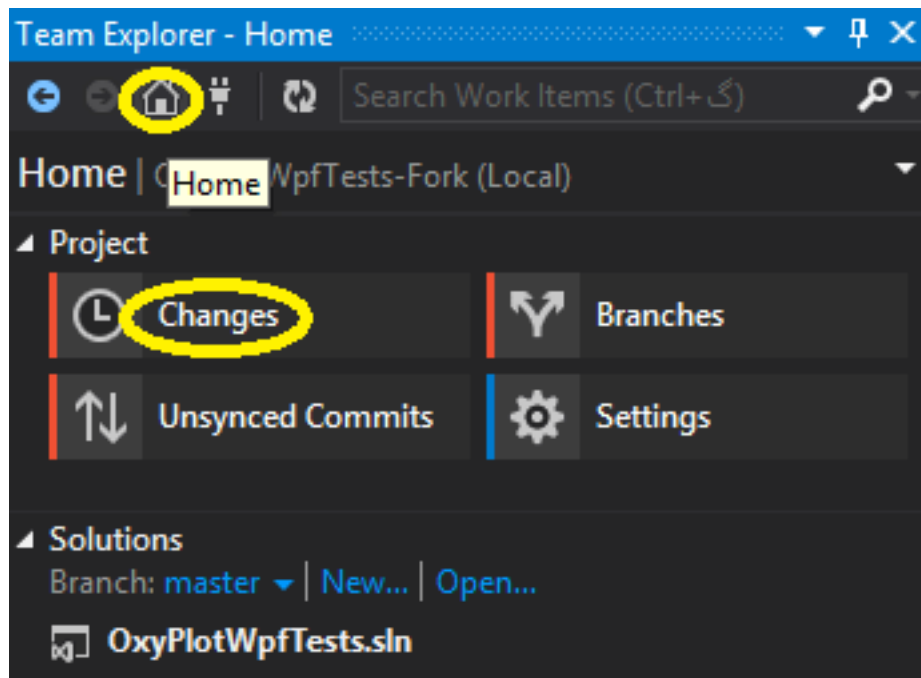


در بالای هر مخزن کد در GitHub، یک دکمه به نام Fork موجود است. بر روی آن که کلیک کنید، یک کپی از آن پروژه را به مجموعه‌ی مخزن‌های کد شما در GitHub اضافه می‌کند. بدیهی است در این حالت، مجوز ارسال تغییرات خود را به GitHub و در اکانت خود خواهید داشت. نحوه‌ی اطلاع رسانی این تغییرات به صاحب اصلی این مخزن کد، ارسال همان PR یا Pull Request است.

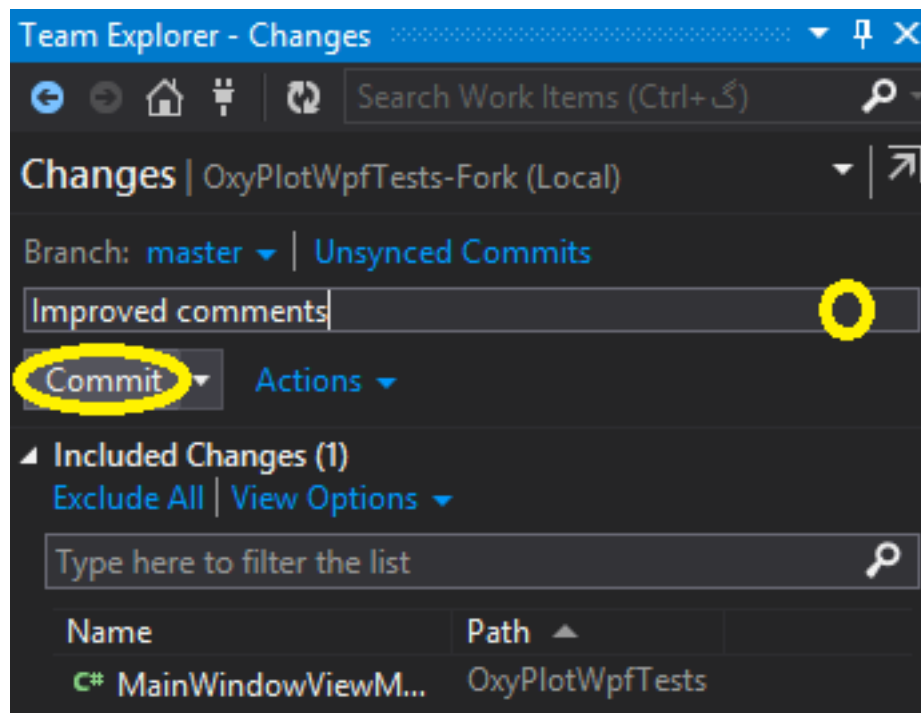
دریافت مخزن کد Fork شده از GitHub به کمک Visual Studio

پس از اینکه Fork جدیدی را از پروژه‌ای موجود ایجاد کردیم، نیاز است یک Clone یا کپی مطابق اصل آن را جهت اعمال تغییرات محلی، تهیه کنیم. برای اینکار VS.NET را گشوده و به برگه‌ی Team Explorer آن که در کنار Solution Explorer قرار دارد، مراجعه کنید.

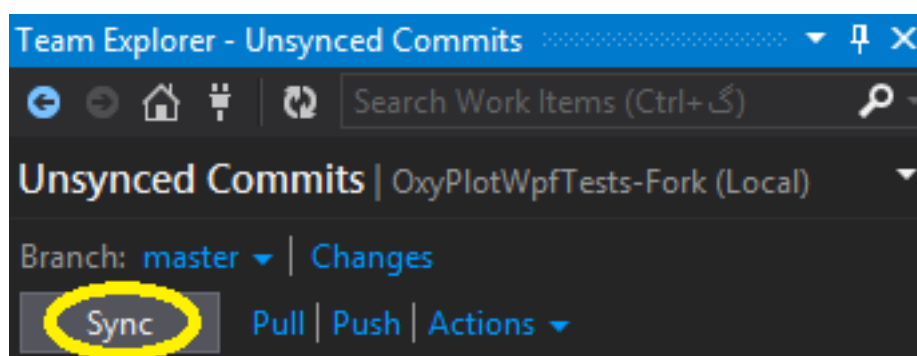
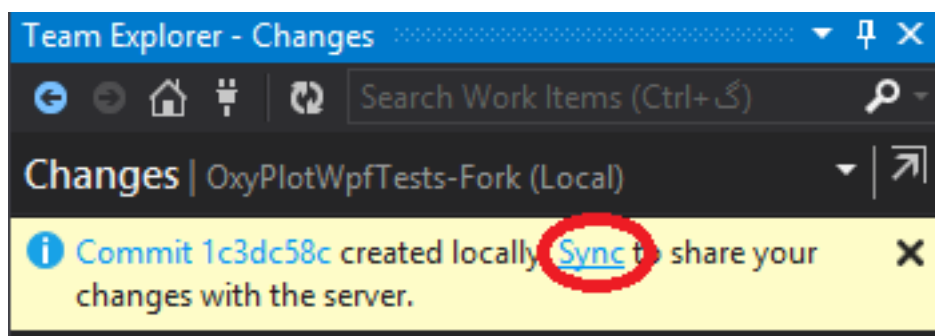
Changes را انتخاب نمایید:



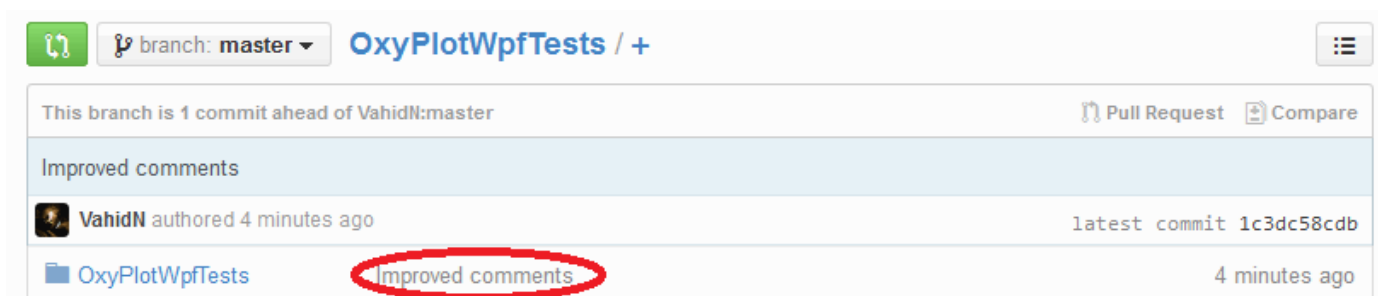
در اینجا توضیحاتی را نوشته و سپس بر روی دکمه‌ی Commit کلیک کنید.



پس از هماهنگ‌سازی محلی، اکنون نوبت به هماهنگ‌سازی این تغییرات با مخزن کد GitHub است. بنابراین بر روی لینک Sync در پیام ظاهر شده کلیک کنید و در صفحه‌ی بعدی نیز بر روی دکمه‌ی Sync کلیک نمایید:



اکنون اگر به پروژه‌ی GitHub خود مراجعه کنید، این تغییر جدید قابل مشاهده‌است:



مطلع سازی صاحب اصلی مخزن کد از تغییرات انجام شده

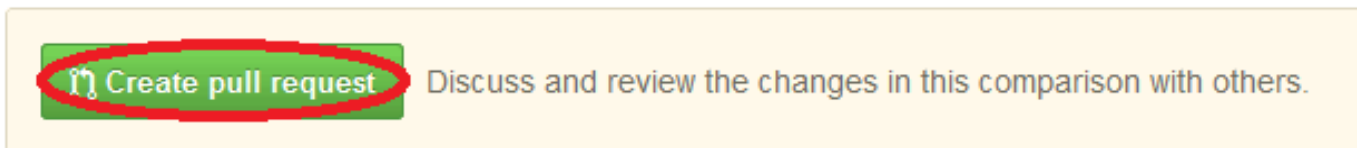
تا اینجا کسی از تغییرات جدید انجام شده‌ی توسط ما باخبر نیست. برای اطلاع رسانی در مورد این تغییرات، به مخزن کد Fork شده که اکنون تغییرات جدید به آن ارسال شده‌اند، مراجعه کنید. سپس در کنار صفحه بر روی لینک Pull request کلیک نمائید:



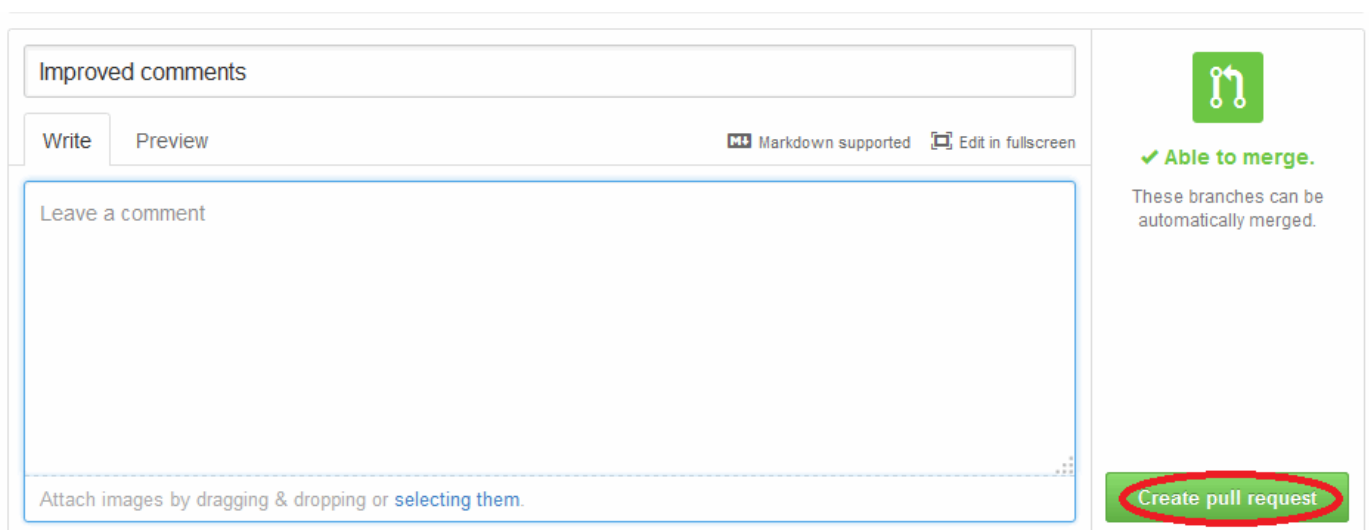
در اینجا بر روی دکمه‌ی New pull request کلیک کنید:



در ادامه تغییرات ارسال شما نمایش داده خواهند شد. آن‌ها را بررسی کرده و مجدداً بر روی دکمه‌ی Create pull request کلیک کنید:

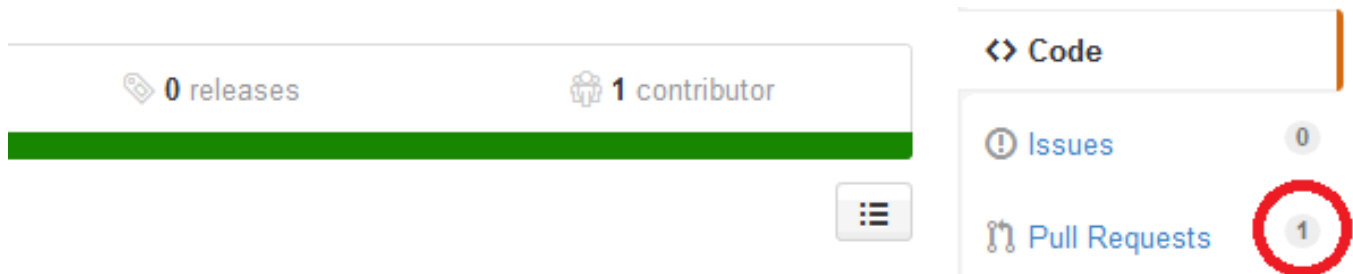


در اینجا عنوان و توضیحاتی را وارد کرده و سپس بر روی دکمه‌ی Create pull request کلیک نمایید:



یکی سازی تغییرات با مخزن اصلی

اکنون صاحب اصلی مخزن کد یک ایمیل را دریافت خواهد کرد؛ همچنین اگر به مخزن کد خود مراجعه نماید، آمار Pull requests دریافتی قابل مشاهده است:

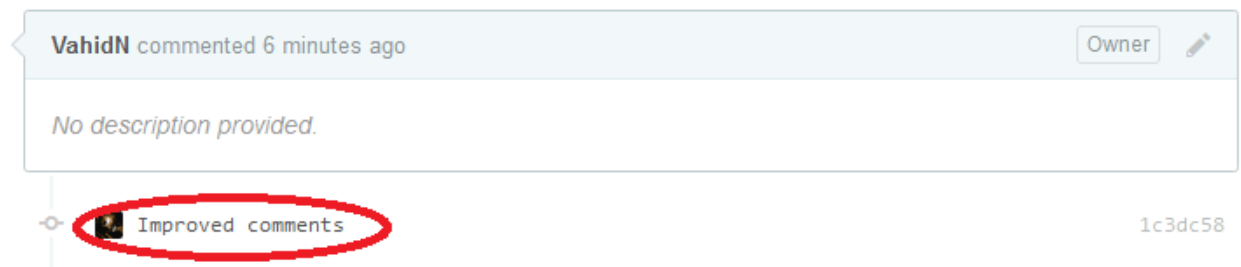


پس از انتخاب یکی از آن‌ها، لینکی برای بررسی تغییرات انجام شده و همچنین دکمه‌ای برای یکی سازی آن‌ها با پروژه‌ی اصلی وجود دارد:

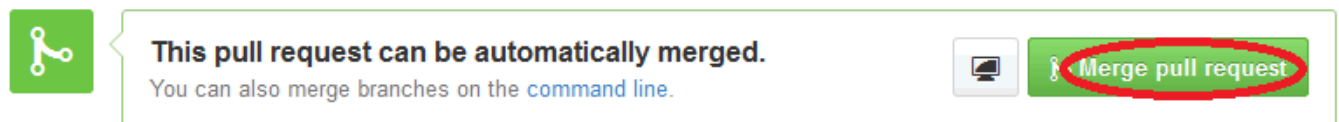
Improved comments #1

Open VahidN wants to merge 1 commit into `VahidN:master` from `DotNETTips:master`

Conversation 0 Commits 1 Files changed 1

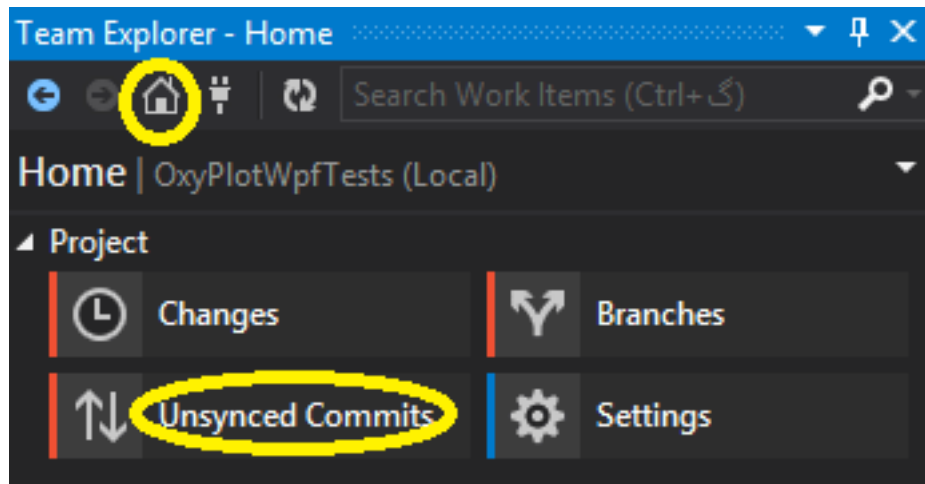


Add more commits by pushing to the **master** branch on **DotNETTips/OxyPlotWpfTests**.

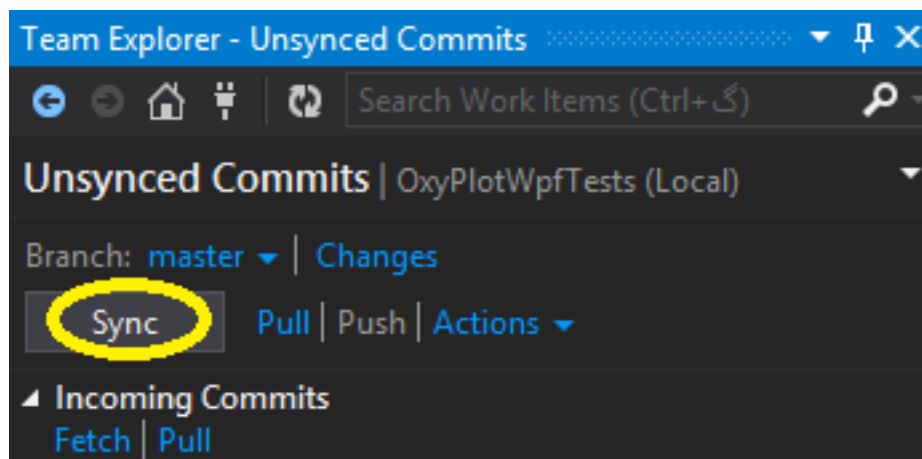


دریافت این تغییرات در مخزن کد محلی توسط صاحب اصلی پروژه

اکنون که این تغییرات با پروژه‌ی اصلی Merge و یکی شده‌اند، صاحب اصلی پروژه جهت تهیه‌ی یک کپی محلی و بهبود یا تغییر آن‌ها می‌تواند به صورت ذیل عمل کند:



ابتدا به برگه‌ی Team explorer مراجعه کرده و بر روی دکمه‌ی Home آن کلیک کنید. سپس گزینه‌ی Unsynced commits را انتخاب نمایید. در صفحه‌ی باز شده بر روی دکمه‌ی Sync کلیک نمایید. به این ترتیب آخرین تغییرات را از مخزن کد GitHub به صورت خودکار دریافت خواهید کرد:



نظرات خوانندگان

نویسنده: بهزاد شیرانی
تاریخ: ۱۳۹۳/۱۱/۲۶ ۱۲:۱۸

چطور می‌تونیم سورس خودمون رو با آخرین تغییرات انجام شده روی سورس اصلی sync کنیم؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۱/۲۶ ۱۲:۴۹

باید عملیات [pull commits](#) انجام شود؛ در همان برگه‌ی Unsynced commits .

در [این صفحه](#) یک برنامه مختص ویندوز قرار داده شده است که شعار آن بدین شکل است: "کار با گیت هاب تا بحال تا این حد آسان نبوده است". موقعی که فایل را دانلود کنید، بعد از اجرا، شروع به دانلود و نصب برنامه اصلی خواهد کرد که در حال حاضر حجم فعلی آن حدود 45 مگابایت است. بعد از اینکه برنامه را نصب کرده و آن را اجرا کنید، از شما درخواست اطلاعات لاگین را می‌کند. اطلاعات ورود به GitHub را وارد کنید تا با اکانت شما در سایت ارتباط برقرار کند و خود را با آن سینک نماید. برای ایجاد یک repository جدید می‌توانید از دکمه‌ی Add، که در بالا سمت چپ قرار دارد استفاده کنید. در اولین کادر متنی، یک نام و در دومین کادر، متن مسیر ذخیره پروژه را اختصاص دهید. در قسمت git ignore می‌توانید مشخص کنید که چه فایل‌هایی توسط سیستم گیت ردیابی نشوند و در زمان سینک کردن یا انتشار محتوا، به سیستم گیت اضافه نشوند. این گزینه را می‌توانید none انتخاب کنید تا شاید بعدا بخواهید دستی آن را تغییر دهید. ولی با این حال این گزینه شامل قالب‌های از پیش آماده‌ای است که ممکن است کار را برای شما راحت کند. مثلا گزینه‌ی پیش فرض Windows، در مورد فایل‌هایی با پسوند doc یا docx و ... می‌باشد. برای اطلاع از روش کار این فایل، مطالب [اینجا](#) را مطالعه فرمایید.

The screenshot shows the GitHub 'Create repository' interface. At the top, there's a '+ master' dropdown. Below it are two tabs: 'Create' (active) and 'Clone'. The form contains three input fields: 'Name' with the value 'dotnettips', 'Local path' with the value 'E:\GitHub\dotnettips', and 'Git ignore' with a dropdown menu showing 'Windows'. To the right of the 'Local path' field is a 'Browse' button. At the bottom of the form is a large blue button with a checkmark icon and the text 'Create repository'.

در صورتیکه فایل‌های شما برای انشار نهایی آماده هستند، پروژه خود را در لیست سمت چپ برنامه انتخاب کنید تا در بالا و سمت راست برنامه، گزینه‌ی Publish Repository دیده شود و با انتخاب آن، یک نام را که قبلا وارد کرده اید و یک توضیح مختصر را از شما می‌خواهد. به صورت پیش فرض انتشارها عمومی و رایگان هستند. در صورتی که اگر بخواهید این انتشار را تنها برای خود و به صورت اختصاصی انجام دهید، باید هزینه آن را پرداخت کنید.

Publish Repository

GitHub

Enterprise

Name

dotnettips

Description

a tutorial for 'github for windows'

yeganehaym

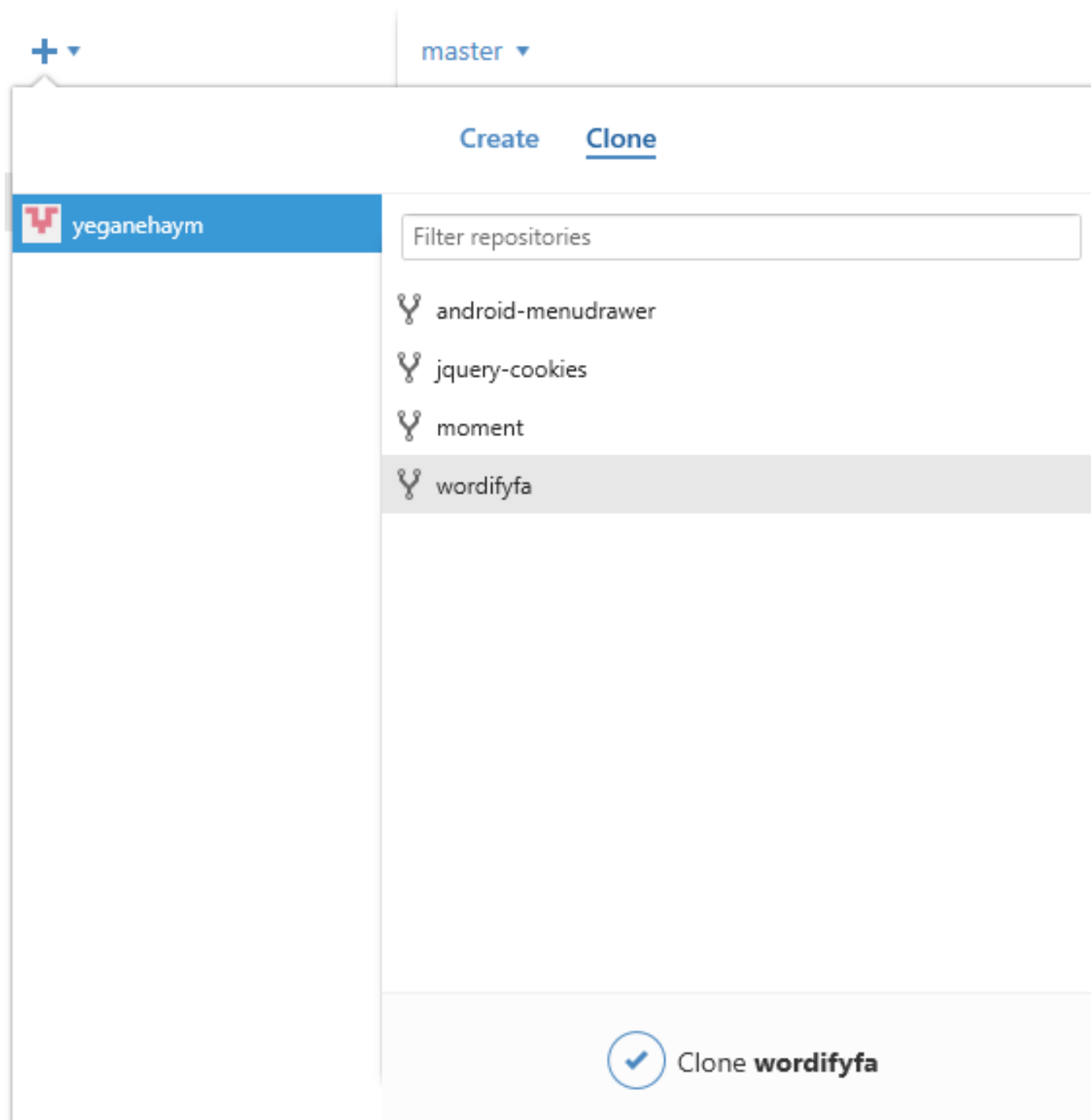
☐ Private Repository

Private repositories require a [micro plan](#) for \$7/month

✓

Publish dotnettips

در صورتیکه دوست دارید در پروژه‌ای مشارکت داشته باشید، ابتدا پروژه مورد نظر را در سایت گیت هاب Fork کنید و سپس از طریق گزینه‌ی Add در برنامه عمل کنید و اینبار در سربرگ‌های بالا، به جای Create گزینه‌ی Clone را انتخاب نمایید. در این حالت لیستی از پروژه‌های Fork شده نمایش داده می‌شوند و با انتخاب هر کدام، پروژه بر روی سیستم شما کیی خواهد شد.



بعد از انتخاب گزینه‌ی Clone، از شما محل ذخیره‌ی پروژه را خواهد پرسید و بعد از تایید آن، مقدار زمان کمی برای کپی کردن پروژه خواهد خواست. پس از آن لیستی از همه‌ی تغییرات و مشارکت‌ها به شما نمایش داده می‌شود و در صورتیکه دوست دارید به تغییری در قبل برگردید تا کارتان را از آن شروع کنید، می‌توانید از گزینه‌ی Revert استفاده کنید. برای یادگیری سایر اصطلاحات فنی گیت و گیت‌هاب می‌توانید از [مسیرهای آموزشی آن](#) استفاده کنید.

History

negative numbers support
2 months ago by Salman Arab Ameri

Merge pull request #2 from MBehtemam/...
2 months ago by Salman Arab Ameri

adding support for Nodejs
2 months ago by Mohammad Bagher Ehtemam

Merge pull request #1 from MBehtemam/...
2 months ago by Salman Arab Ameri

JSLint Correction
2 months ago by Mohammad Bagher Ehtemam

Update README.md
2 months ago by Mohammad Bagher Ehtemam

Update README.md
2 months ago by Mohammad Bagher Ehtemam

formatting added to readme
2 months ago by Salman Arab Ameri

project files added
2 months ago by Salman Arab Ameri

Added .gitattributes & .gitignore files
2 months ago by Salman Arab Ameri

negative numbers support

Salman Arab Ameri
f6126ed

GitHub
Revert
Collapse all

negative numbers support added. also index file encoding corrected.

index.html
15
wordifyfa.js
12

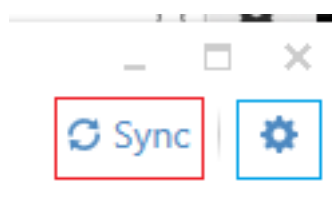
```

... -9,6 +9,11 @@ var wordifyfa = function (num, level) {
9 9     if (num === null) {
10 10         return "";
11 11     }
12 + // convert negative number to positive and get wordify value
13 + if (num<0) {
14 +     num = num * -1;
15 +     return "منفی " + wordifyfa(num, level);
16 + }
12 17     if (num === 0) {
13 18         if (level === 0) {
14 19             return "صفر";
... -56,9 +61,12 @@ var wordifyRialsInTomans = function (num) {
56 61     'use strict';
57 62     if (num >= 10) {
58 63         num = parseInt(num / 10, 10);
64 +     } else if (num<=-10) {
65 +         num = parseInt(num/10,10);
59 66     } else {
60 -         num = 0;
61 -     }
67 +         num=0;
68 +     }
69 + }
62 70     return wordifyfa(num, 0) + " تومان";
63 71 };

```

حال با خیال راحت روی پروژه کار کنید و تغییرات را روی آن اعمال کنید و بعد از اینکه کارتان تمام شد، دوباره به برنامه باز گردید و پروژه را در لیست انتخاب کرده و در سمت راست بالای صفحه، گزینه Sync Now را انتخاب کنید تا مشارکت جدید شما به سیستم گیت هاب اعمال شود و حالا اگر به صفحه‌ی پروژه در سایت گیت هاب بروید، می‌بینید که شما به عنوان یک مشارکت کننده‌ی جدید اضافه شده‌اید. پس با هر بار تغییر نسخه‌ی پروژه می‌توانید آن را با سیستم گیت سینک نمایید.

گزینه‌ی تنظیمات که در کنار عبارت Sync Now قرار دارد و با رنگ آبی در شکل مشخص شده است نیز به شما اجازه‌ی تغییر فایل‌های تنظیماتی از قبیل gitignore یا gitattribute را می‌دهد.



در صورتی که برای پروژه‌ای در گیت هاب شاخه‌ها یا branches تعریف شده باشند، در اینجا هم می‌توانید شاخه‌ی مورد نظر را انتخاب کنید:

ammeep/httpclient-extension ▼

Branches

 Manage

Filter or create new

ammeep/fix-convention-tests

ammeep/httpclient-extension ✓

ammeep/statistics-api

bump-perpage-parameter

dont-pull-down-comments

get-content-spike

hahmed/search-api

master

niik/support-etags-through-wininet

release-docs

shiftkey/rework-build-script

shiftkey/symbol-server-support



3 months ago by Brendan Forster

در مطلب « [نحوه مشارکت در پروژه‌های GitHub به کمک Visual Studio](#) » با مفهوم pull request آشنا شدیم. اما ... یک pull request خوب چه خصوصیتی دارد و فرهنگ ارسال یک PR خوب چیست؟

اخلاق مشارکت در یک پروژه سورس باز

بعضی از توسعه دهنده‌ها در حین مشارکت در یک پروژه سورس باز، برای مثال جهت افزودن قابلیت جدید و یا رفع مشکلی، ابتدا سعی می‌کنند تا کدهای فعلی را برای خودشان «قابل فهم‌تر» کنند. این قابل فهم‌تر کردن پروژه، شامل تغییر نام متغیرها و متدهای فعلی، انتقال کدهای موجود به فایل‌هایی دیگر یا حتی یکی کردن چندین فایل با هم، مرتب سازی متدهای یک کلاس بر اساس حروف الفباء و امثال آن می‌شود. این کارها را نباید در حین مشارکت و توسعه‌ی پروژه‌های سورس باز دیگران انجام دهید! اگر هدفتان رفع مشکلی است یا افزودن قابلیت جدید، باید نحوه‌ی کدنویسی فعلی را حفظ کنید. از این جهت که نگهدارنده‌ی اصلی پروژه، پیش از شما این کار را شروع کرده‌است و زمانیکه شما به پروژه‌ای دیگر رجوع خواهید کرد، باز نیز باید همین کار را ادامه دهید. اگر refactoring گسترده‌ی شما به هر نحوی سبب بهبود پروژه‌ی اصلی می‌شود، ابتدا این مورد را با مسئول اصلی پروژه مطرح کنید. اگر او قبول کرد، سپس اقدام به چنین کاری نمائید.

بحث در مورد تغییرات پیش از ارسال PR

قبل از اینکه PR ایی را ارسال کنید، بهتر است یک issue یا ticket جدید را باز کرده و در مورد آن بحث کنید یا توضیح دهید. در این حالت ممکن است توضیحات بهتری را در مورد سازگار سازی تغییرات خود با کدهای فعلی دریافت کنید.

Pull request ها را کوچک نگه‌دارید

برای اینکه شانس قبول شدن PR خود را بالا ببرید، حجم و تمرکز آن را کوچک نگه دارید. بسیاری از توسعه دهنده‌های سورس باز اگر با یک PR حجم روبرو شوند، آن را رد می‌کنند چون مشکل اصلی، مدت زمان بالایی است که باید جهت بررسی این PR اختصاص داد. هرچقدر حجم آن بیشتر باشد، زمان بیشتری را خواهد برد.

فقط یک کار را انجام دهید

شبهه به اصل تک مسئولیتی کلاس‌ها، یک PR نیز باید تنها یک کار را انجام دهد و بر روی یک موضوع خاص تمرکز داشته باشد. فرض کنید PR ایی را ارسال کرده‌اید که سه مشکل A، B و C را برطرف می‌کند. از دیدگاه مسئول اصلی پروژه، موارد A و C قابل قبول هستند؛ اما نه مورد C مطرح شده. در این حالت کل PR شما برگشت خواهد خورد. به همین جهت بهتر است بجای یک PR، سه PR مختلف و مجزا را جهت رفع مشکلات A، B و C ارسال کنید.

سازگاری تغییرات ارسالی را بررسی کنید

حداقل کاری را که پیش از ارسال PR باید انجام دهید این است که بررسی کنید آیا این تغییرات قابل Build هستند یا خیر. همچنین اگر پروژه دارای یک سری Unit tests است، حتماً آن‌ها را یکبار بررسی کنید تا مطمئن شوید جای دیگری را به هم نریخته‌اید. ضمناً وجود این تست‌ها به صورت ضمنی به این معنا است که تغییرات جدید شما نیز باید به همراه تست‌های مرتبطی باشند تا پذیرفته شوند.

PR ایی را بر روی شاخه‌ی master ارسال نکنید

پس از اینکه یک fork از پروژه‌ای سورس باز را ایجاد کردید و سپس آنرا clone نمودید تا به صورت Local بتوانید با آن کار کنید، فراموش نکنید که در همینجا باید یک branch و انشعاب جدید را جهت کار بر روی ویژگی مدنظر خود ایجاد کنید (برای مثال feature-X, fix-Y). بسیاری از پروژه‌های سورس باز به هیچ عنوان PRهای کار شده‌ی بر روی انشعاب master را قبول نمی‌کنند.

برای مطالعه بیشتر

[Open Source Contribution Etiquette](#)

[ten tips for better Pull Requests](#)

[Getting a Pull Request Accepted](#)

[Optimize Your Pull-request](#)

نظرات خوانندگان

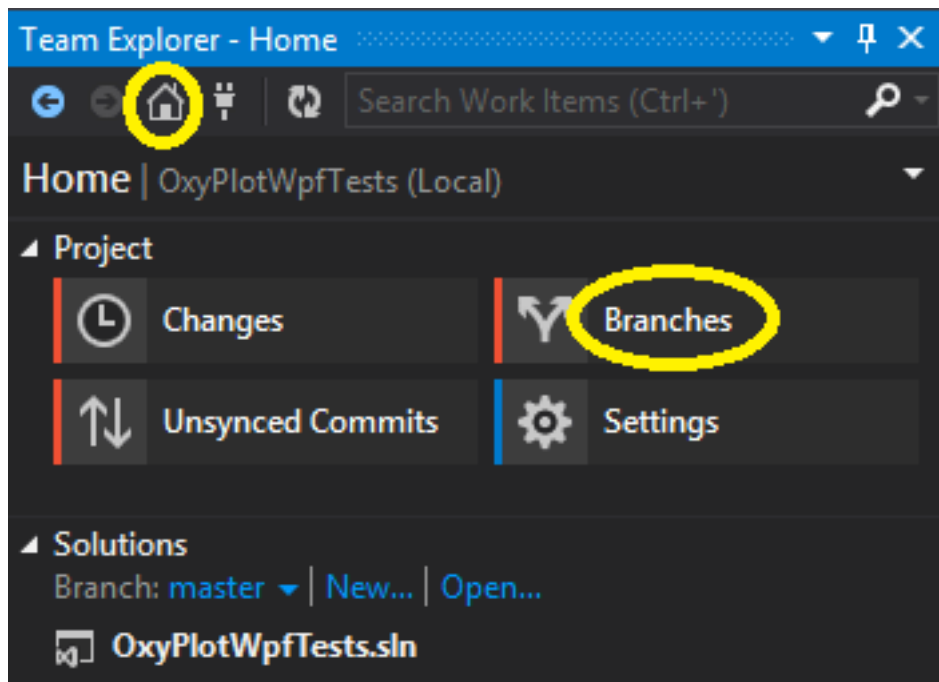
نویسنده: جلال
تاریخ: ۱۹:۱۱ ۱۳۹۳/۱۲/۱۴

کاش زودتر خونده بودمش P:

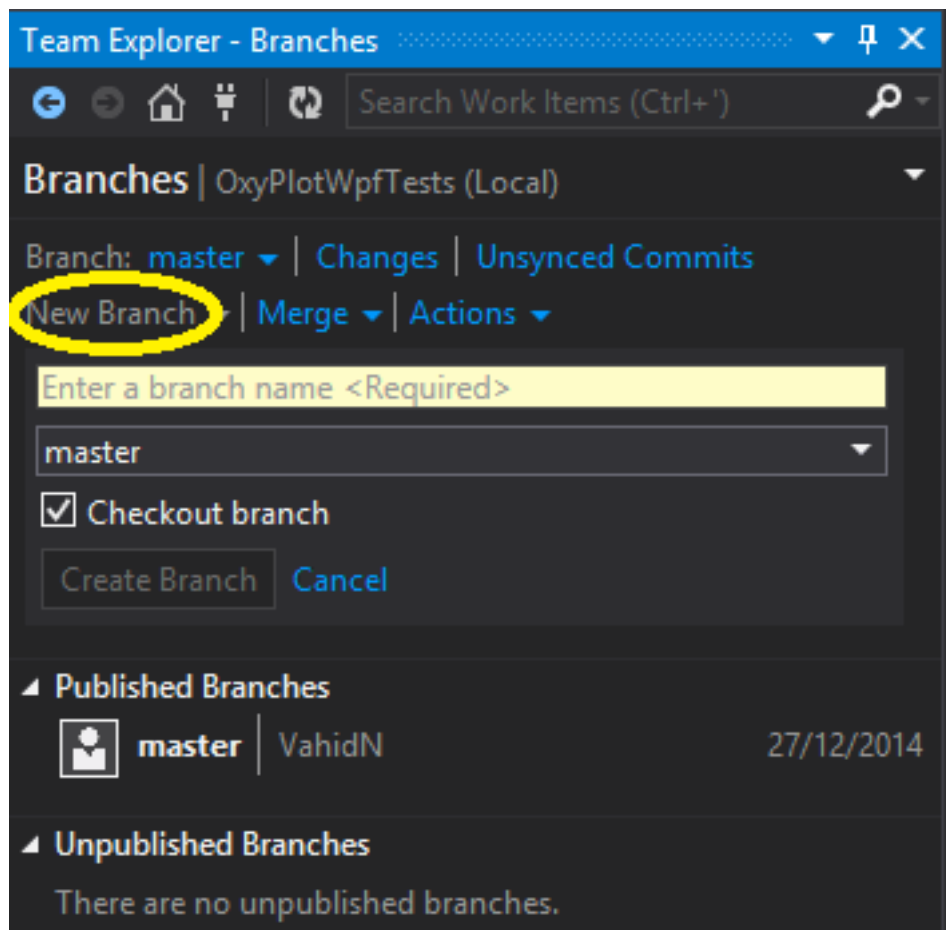
در مطلب «[آشنایی با ساختار یک Pull Request خوب](#)» عنوان شد که قابلیت‌های جدید و یا رفع مشکلات را در شاخه‌ی اصلی کار نکنید. اما ... چگونه؟

ایجاد یک شاخه‌ی جدید در Visual Studio و انتشار آن

به برگه‌ی Team explorer مراجعه کرده و سپس گزینه‌ی Branches آن را انتخاب کنید:



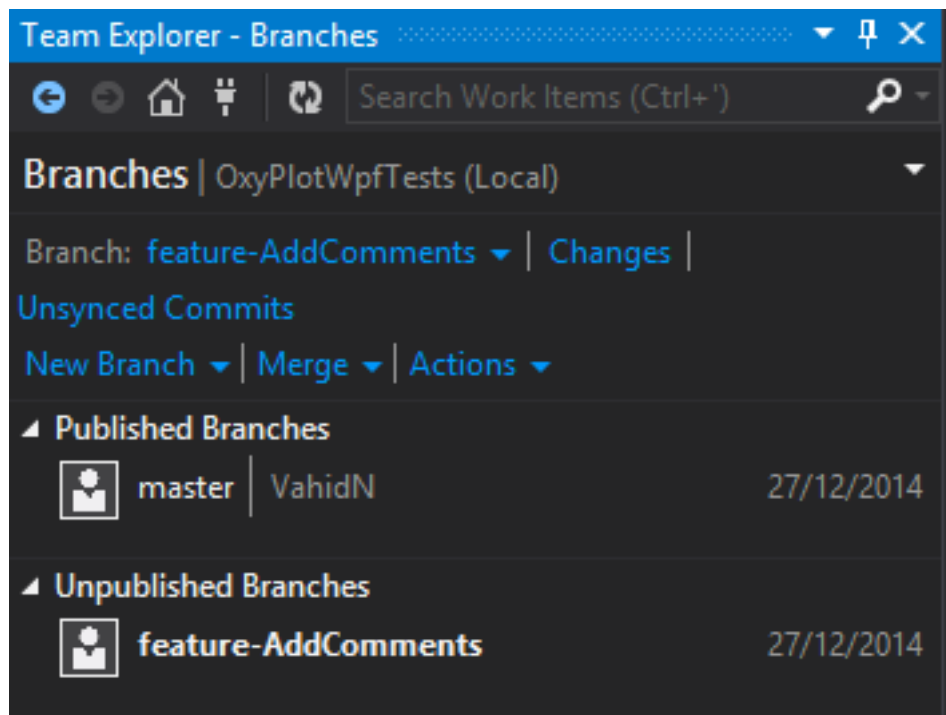
در برگه‌ی باز شده، انشعاب و شاخه‌ی جاری با فونت ضخیم نمایش داده می‌شود. برای مثال در اینجا، انشعاب کاری همان master است:



برای ایجاد یک شاخه‌ی جدید، بر روی لینک new branch کلیک کنید تا بتوان نامی را برای این منظور وارد کرد. بهتر است از نام‌های با مفهومی مانند feature-X یا fix-Y استفاده کنید (افزودن قابلیت X و یا رفع مشکل Y) و در آخر بر روی دکمه‌ی Create branch کلیک کنید.

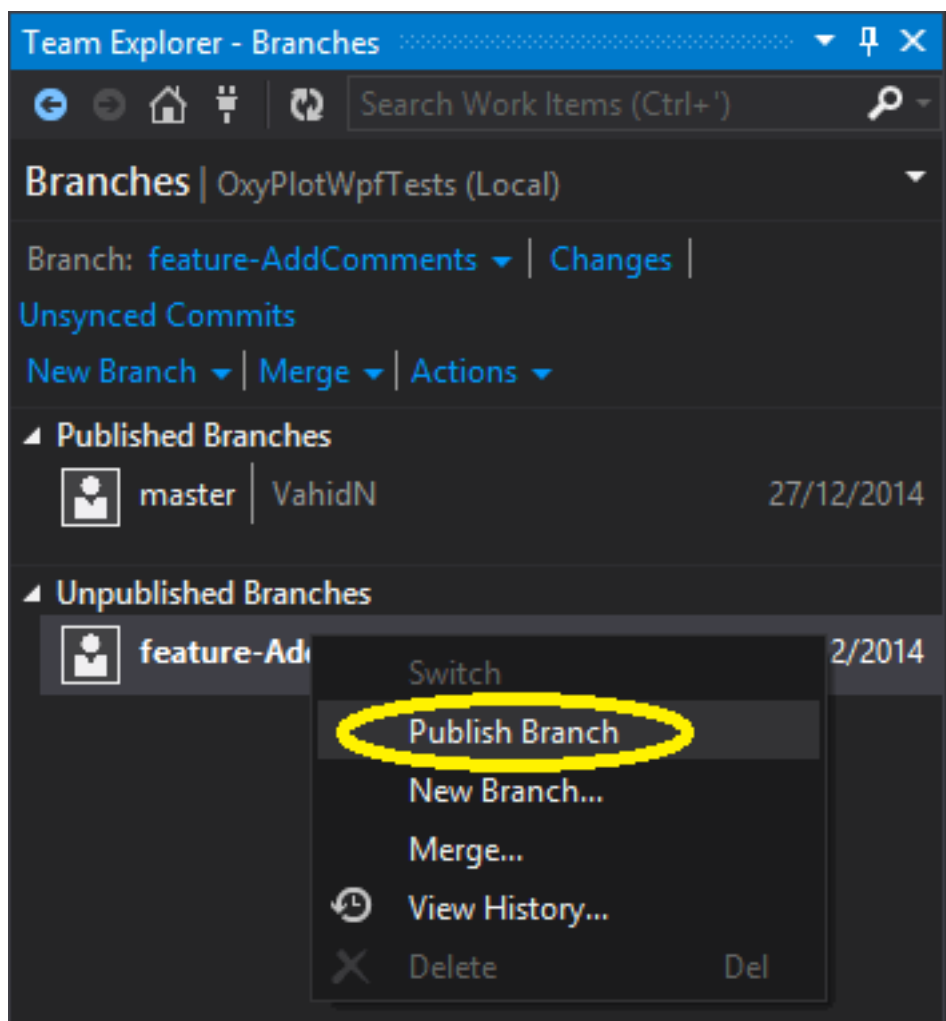
در اینجا می‌توان مشخص کرد که انشعاب ایجاد شده باید بر اساس کدام انشعاب فعلی نیز تهیه شود (دراپ داون ذیل قسمتی که می‌توان نام انشعاب را وارد کرد). برای مثال پروژه‌های میکروسافت در GitHub، دارای سه شاخه‌ی master، dev و release هستند. شاخه‌ی dev (یا توسعه) جایی است که انشعابات pull requests را قبول خواهند کرد. بنابراین بر اساس ساختار و طراحی پروژه‌ی جاری به این موضوع نیز باید دقت داشت.

پس از ایجاد شاخه‌ی جدید، تصویر ذیل نمایان خواهد شد:



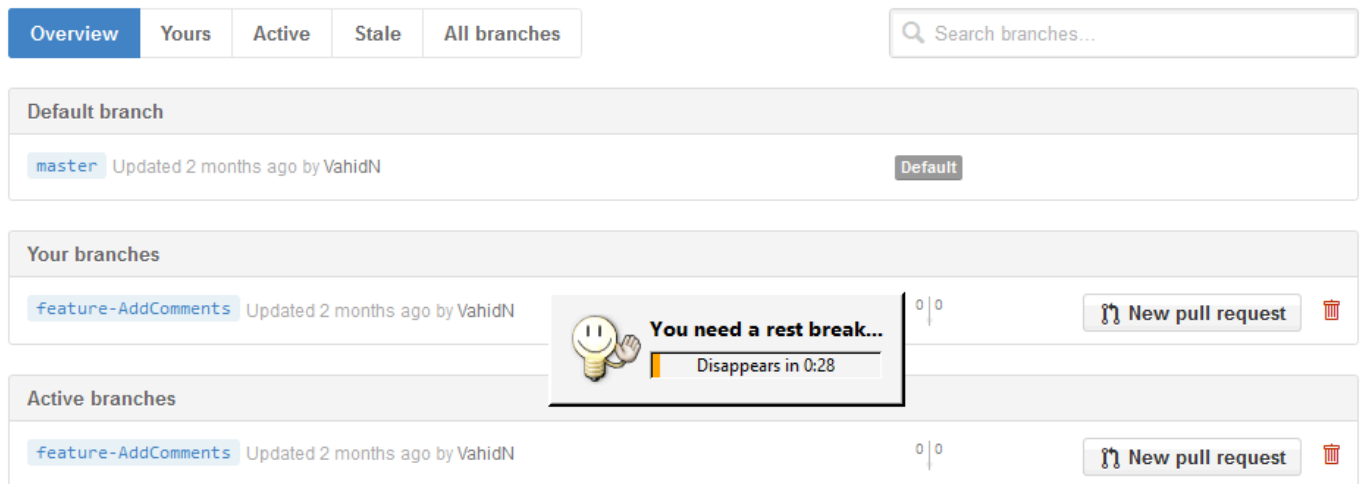
همانطور که ملاحظه می‌کنید، اینبار شاخه‌ی جدید ایجاد شده به صورت **bold** و ضخیم نمایش داده شده‌است. این **bold** بودن به معنای شاخه‌ی کاری جاری بودن است. همچنین این شاخه در قسمت **unpublished branches** قرار دارد. بنابراین کلیه‌ی تغییرات واقع شده‌ی در آن، محلی بوده و هنوز با سرور هماهنگ نشده‌اند.

برای انتشار و **publish** این شاخه، تنها کافی است تا بر روی آن کلیک راست کرده و گزینه‌ی **publish branch** را انتخاب کنیم:



این انتشار سبب نمایش لیستی از تغییرات جدید در برگه‌ی branches پروژه، در GitHub خواهد شد:

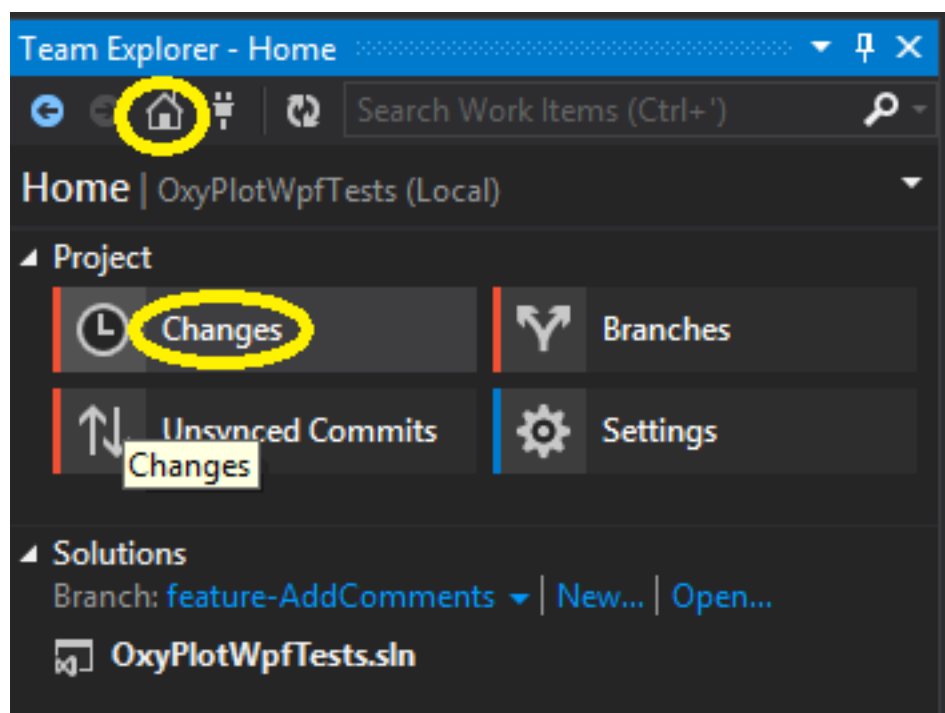




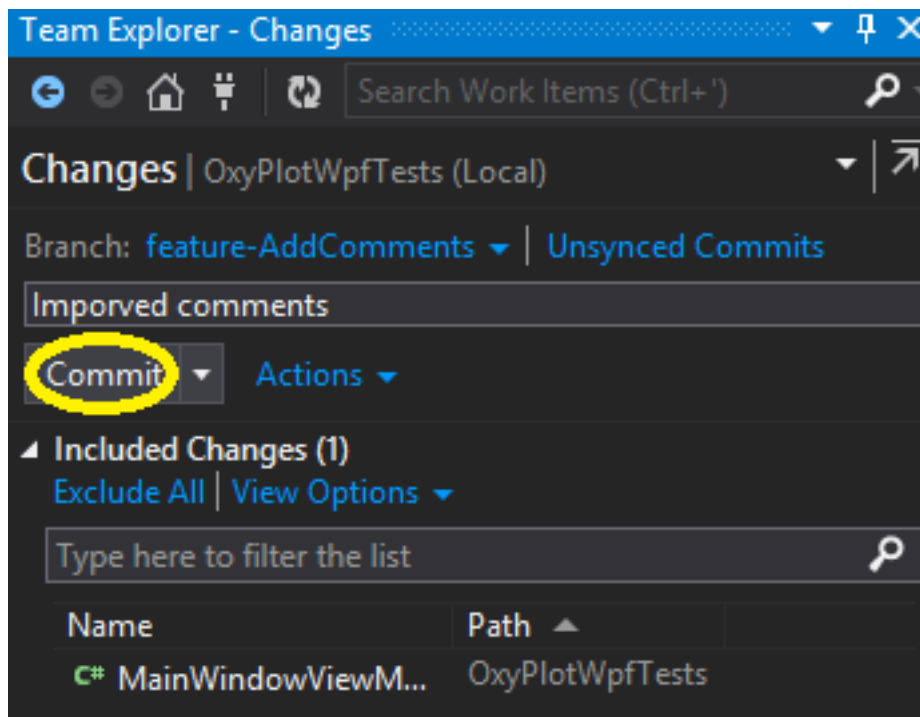
یک نکته: برای تغییر branch فعال جاری، فقط کافی است در برگه‌ی branches در ویژوال استودیو، دوبار بر روی لینک نام آن شاخه کلیک کنید تا به صورت bold ظاهر شود.

ارسال تغییرات انجام شده‌ی در Branch به سرور

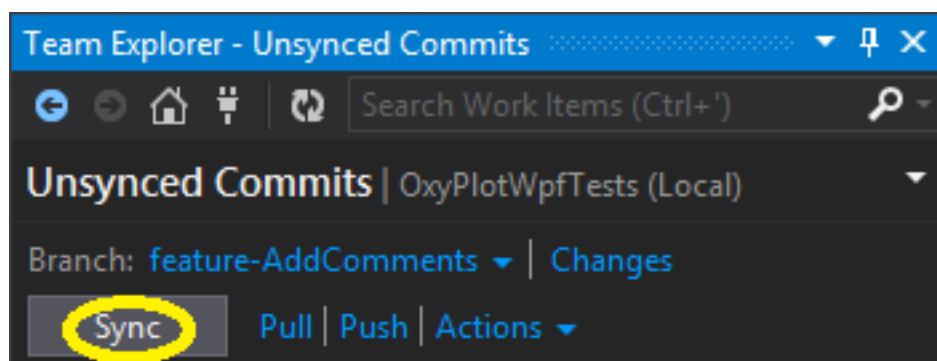
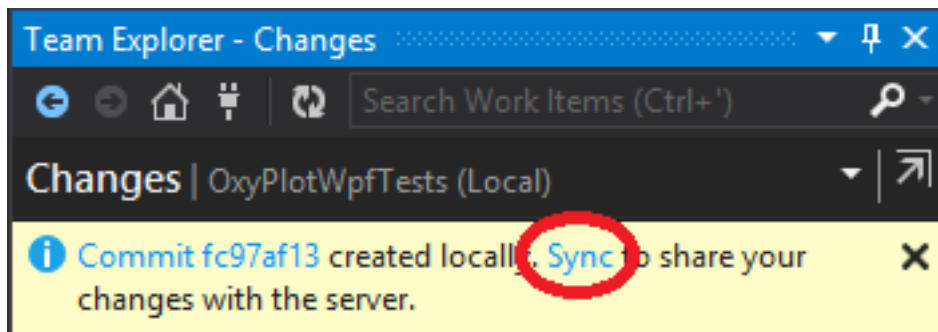
پس از کار بر روی شاخه‌ی جدید ایجاد شده، اکنون نوبت به ارسال و هماهنگ سازی این تغییرات با سرور است. این مورد نیز همانند قبل بوده و ابتدا باید به برگه‌ی Home و گزینه‌ی changes آن مراجعه کرد:



و سپس تغییرات را به همراه توضیحی commit کرد:

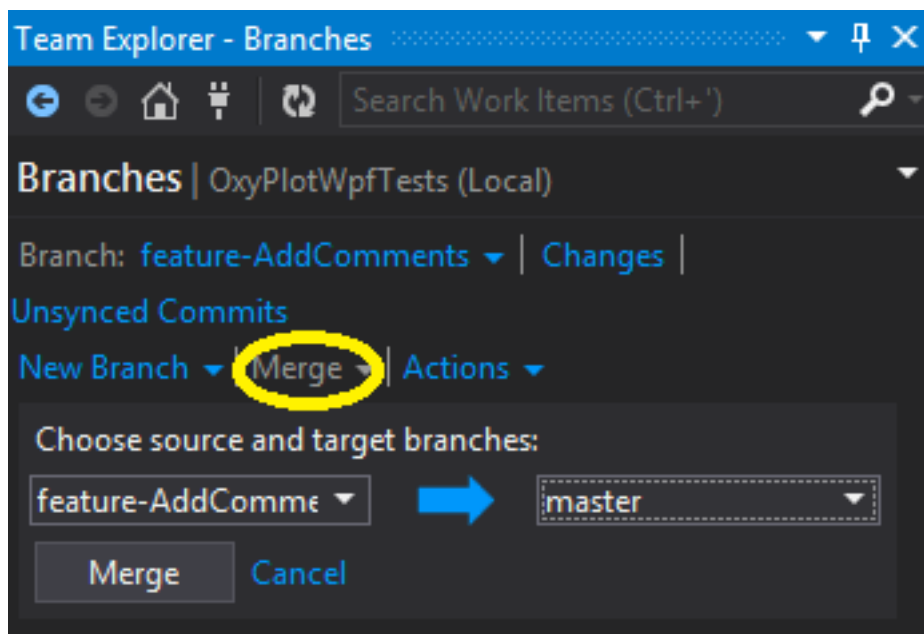


اینکار سبب sync محلی می‌شود. سپس بر روی لینک sync کلیک نمایید و تغییرات را با سرور هماهنگ کنید.



یکی کردن تغییرات شاخه‌ی جدید با شاخه‌ی اصلی

هرچند این تغییرات به سرور ارسال شده‌اند، اما چون در یک انشعاب کاری دیگر قرار دارند، با انشعاب اصلی یکی نخواهند شد. برای انجام عملیات merge، ابتدا به برگه‌ی Home و سپس گزینه‌ی branches مراجعه کنید. در ادامه بر روی لینک merge کلیک نمائید (تصاویر اول و دوم بحث).



در اینجا می‌خواهیم اطلاعات موجود در شاخه‌ی افزودن توضیحات را با شاخه‌ی اصلی یکی کنیم (انتخاب منبع و مقصد). سپس بر روی دکمه‌ی merge کلیک نمائید.

اکنون برای ارسال این تغییرات به سرور، به برگه‌ی Home و سپس گزینه‌ی unsynced commits مراجعه کرده و بر روی دکمه‌ی sync کلیک نمائید تا تغییرات یکی شده به سرور ارسال شوند.

