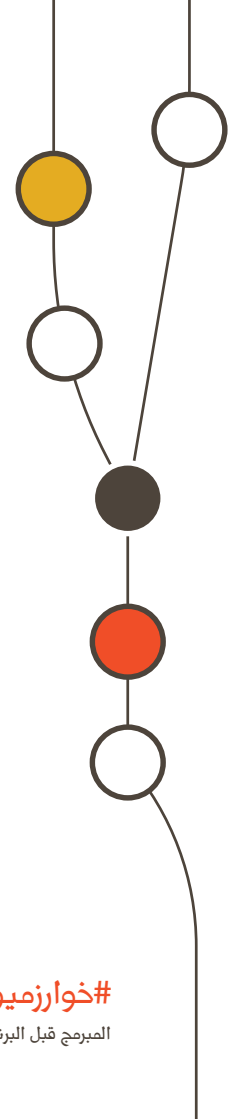


Git

المبرمجون وإدارة الشيفرات البرمجية



#خوارزميون هو مشروع يهدف إلى تقديم محتوى ومنتجات عربية وخدمات في مجال تعليم البرمجة بأسلوب مبتكر، وهو عبارة عن عمل مؤسسي (شركة ناشئة) يقوم عليه فريق عمل يهدف إلى تكوين قاعدة معرفية عربية ضخمة.

ينقسم المشروع إلى قسمين أساسيين وهما المجاني و التجاري، بحيث يهدف القسم المجاني إلى توفير مادة علمية مجانية للجميع تتضمن المحتوى المكتوب والمرئي وأي ابتكارات أخرى تتعلق بالمحتوى، بينما يتضمن القسم التجاري بيع منتجات رقمية وغير رقمية من خلال إرسالها للعملاء في جميع أنحاء العالم.

يحاول **#خوارزميون** سد الثغرة الموجودة في المحتوى العربي قدر الإمكان من خلال كتابة مادة علمية مرتبة وتناسب احتياجات المجتمع وبعيدة عن التكلفة والترجمة الحرفية.

نحن نؤمن بأن التعليم من حق الجميع ومن حق الجميع الحصول عليه بالمجان، وما يباع هنا ليس العلم كونه ليس ملكاً لشخص بعينه وإنما الجهد الذي قضي في تحضير تلك المواد وتجهيزها، فبدلاً من التفرغ لساعة أسبوعياً لعمل محتوى بسيط ومجاني، فقد تم بناء فريق عمل متكامل للتفرغ بشكل كامل لهذا المشروع وهم يعملون بوظيفة مستقلة خاصة فقط بهذا العمل، والعمل بهذه الطريقة سيضمن الاستدامة والتوسع وعدم توقف المشروع واعتماده على أفراد بعينهم كونه أصبح عملاً مؤسسياً وليس مجرد جهود فردية تتوقف بغياب أو تأخر من يقوم عليها.

يحاول **#خوارزميون** الدخول في تفاصيل المبرمج نفسه، ويحاول صناعة مجتمع قوي و فعال في هذا المجال، بالإضافة إلى ابتكار أفكار جديدة في مجال تعليم البرمجة

#خوارزميون

المبرمج قبل البرنامج

www.algorithmers.com

يشكر فريق عمل #خوارزميون المبرمج
سيف الحارثي لمساهمته في عملية المراجعة

@saifalharthi

البداية

لتحميل git، و تطبيق الأوامر الواردة في
الكتاب، قم بزيارة الموقع التالي

www.git-scm.com

مقدمة في مفهوم Git

يمكن النظر إلى Git على أنه نظام لإدارة الملفات بشكل مبسط، أي كنظرة أولية، و بعدها يمكننا النظر إلى طريقة إدارته لتلك الملفات و التي تمثل المحور الأساسي لطريقة عمله و هي أيضاً ما يميزه عن بقية الأنظمة الأخرى لإدارة الملفات.

ماهي الخطوات الأساسية التي يقوم بها المبرمج عند البدء في كتابة الشيفرة البرمجية الخاصة به؟

غالباً ما يقوم بإنشاء ملف أو أكثر، و من ثم يقوم بإجراء تعديلات مستمرة على الملف حتى ينتهي من الملف بالكامل. بعد ذلك تأتي التطويرات المستقبلية، فيعود المبرمج إلى الملف أو مجموعة الملفات ليقوم بالإضافة و الحذف و التعديل ثم الحفظ و هكذا.

الآن ماذا لو كان هناك برنامج يقوم بتسجيل كل تلك التعديلات بحيث يقوم بتخزين العملية أو العمليات على الملف و من قام بها و متى قام بها، بحيث يصبح لدينا سجلات **History** كاملة عن الملف و عن كل عملية تمت عليه و ماهي تلك العملية أو العمليات، بالإضافة إلى أنه بإمكانك العودة إلى نسخة معينة من ذلك الملف، أي بإمكانك العودة للملف قبل سنة أو قبل ستة أشهر أو أي وقت آخر، و كيف كان حينها ذلك الملف و بإمكانك إلغاء التعديلات التي تمت عليه و هكذا.

لو فرضنا أنك كنت تعمل في فريق عمل برمجي، و قمت أنت و أحد أفراد الفريق بالتعديل على نفس الملف و في نفس الوقت وكل منكما على جهازه، كيف يتم تخزين التعديلات دون أن تتأثر تعديلات كل منكما بما قام به الآخر؟. باختصار، هذه هي بعض الأدوار الأساسية التي وجد من أجلها **Git**.

يمكن النظر له بشكل مبسط على أنه «قاعدة بيانات Database» أو نظام ملفات «File System» يقوم بتخزين العمليات التي تقوم بها على ملف أو مجموعة ملفات، بحيث يمكنك العودة لتلك التعديلات في أي وقت.

إلتقط لي صورة من فضلك

عندما تمسك الكاميرا بيديك و تشير لزملائك بأن يستعدوا للتصوير، فأنت تلتقط الصورة لتعود إليها في وقتٍ ما لأغراض محددة سواء للذكرى أو لأمر رسمي أو أياً كان السبب، فالمهم أننا «نعود إليها».

تخيل أن لدينا مشروع برمجي مكون من ٧ ملفات، و قمت بإجراء تعديلات على ٤ ملفات، هنا تخيل أن **Git** هو الكاميرا، بحيث يلتقط صورة لوضع جميع الملفات بالتعديلات التي أجريت عليها، و يحتفظ بتلك الصورة، و بعدها أجريت تعديل على ملف واحد، و استدعيت **Git** ليقوم بالتقاط صورة لوضع الملفات مع التعديلات التي قمت بها و هكذا. في هذه الحالة، تخيل أنه و مع كل تعديل نلتقط صورة، و نقوم بتخزين تلك الصورة في مستودع خاص **Repository**، أي كأن **Git** يقول : خذ صورة لكامل المشروع بالوضع الحالي، و خزن في تلك الصورة التعديلات التي أجريت. بهذا الشكل، يصبح لدينا مستودع مليء بالصور أو اللقطات **Snapshots**، و بإمكاننا العودة لأي لقطة نريدها إن أردنا. تشمل الصورة أو اللقطة التي يتم إلتقاطها معلومات كثيرة مثل من قام بتلك التعديلات، وقت التعديل، و ماهي تفاصيل كل تعديل على كل ملف تم تعديله، و أشياء إضافية أيضاً.

تسمى اللقطة التي يتم أخذها بإسم Commit، بحيث يقوم المبرمج بعمل Commit للتعديلات التي قام بها حتى يتم تخزينها، و السؤال الآن، لو قمت بتخزين ٥٠ لقطة مختلفة للنظام، فكيف أعرف الصورة أو اللقطة التي أريد الرجوع إليها أو التعامل معها؟ ببساطة كل لقطة Commit يتم أخذها، يتم توليد رقم خاص لها و مميز Id، بحيث لا يتشابه بين لقطة و أخرى، و يتكون هذا الرقم من ٤٠ خانة من حروف و أرقام بالنظام الست عشري أو Hexadecimal بحيث يتم توليد هذا الرقم بإستخدام خوارزمية SHA-١.



```
git commit -m 'My message'
```

سناتي على تفصيل هذا الأمر بإذن الله تعالى، و لكن نود أن نوضح هنا أن هذا الأمر ببساطة هو الذي يلتقط الصورة أو حالة الملفات و الخيار m خاص بوضع رسالة يقوم المبرمج بوضعها لتحتفظ مع تلك الصورة، وغالباً ما يضع المبرمجين رسالة توضح التعديلات أو سبب التعديل أو الخطأ الذي تم إصلاحه أو أي رسالة توضح الهدف التعديلات.

ماذا نقصد بالمستودع Repository؟

كل ما في الأمر أن Git يحتاج إلى مكان لكي يخزن فيه قائمة التعديلات أو كما سبق و ذكرنا اللقطات أو الصور، وهنا سنتعرف على الآلية التي يستخدمها Git في تخزينه لتلك المحتويات.

بكل بساطة، يقوم Git بإنشاء مجلد مخفي بإسم **git**، ولاحظ النقطة في البداية، بداخل المجلد الرئيسي لمشروعك، بحيث يمثل هذا المجلد قاعدة البيانات **Database** و التي سيقوم Git بتخزين العمليات والتعديلات التي ستقوم بها على مشروعك فيها. يسمى هذا المجلد بالمستودع أو **Repository**. في حال أردت إنشاء مستودع شيفرة جديد لمشروعك، فكل ما عليك القيام به هو الذهاب لمجلد المشروع الخاص بك و كتابة الأمر التالي



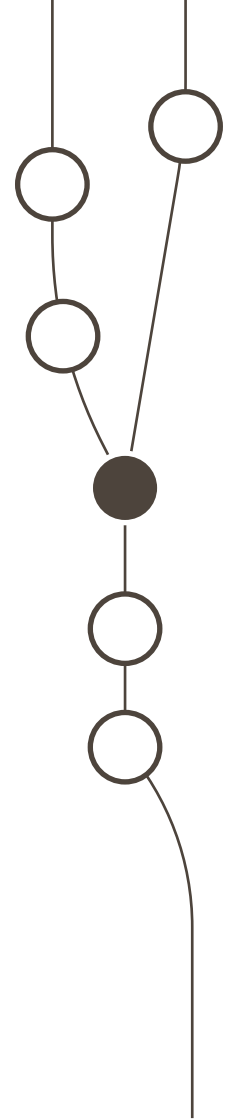
```
git init
```

الآن أصبح المشروع الخاص بك يملك مستودع شيفرة خاص به و هو جاهز للبدء في تخزين التعديلات التي ستجريها عليه، لكن ينبغي التنبيه هنا إلى أن إنشاء المستودع لا يقوم بتخزين أي ملفات أو تعديلات موجودة في مشروعك، أي أن **init** قامت بإنشاء مستودع فارغ.

لو فرضنا الآن أن مشروعك اسمه **book**، و ذهبت لمجلد المشروع **book**، فستجد بداخله مجلد بإسم **git**، وهو المستودع الذي تم إنشائه بواسطة **init**. ما ينبغي التنبيه له هنا، هو أنك في حال قمت بنسخ المجلد **git** الخاص بمشروعك فأنت فعلياً نسخت المشروع بكامل تعديلاته أيضاً.

يستخدم بعض المطورين واجهة **Command Line** لإدارة الشيفرة من خلال أوامر **Git**، و تجد بعضهم يستخدم تطبيقات **Desktop** معينة لإدارة الشيفرة، و في كل الحالات، كل ما تقوم به تطبيقات **Command Line** أو **Desktop** هو التعامل مع مجلد **git**. من خلال جلب المعلومات الموجود بداخله وعرضها و عرض التعديلات التي تمت و هكذا.

ما تقوم به برامج وتطبيقات سطح المكتب الخاصة بـ **Git** هو تنفيذ أوامر **Git** في الخلفية أي كأنها تعمل بطريقة **Command Line** في الخلفية، و لكن دون أن تشعر المستخدم بهذا الأمر.



مفهومي Tracked و Untracked قبل البدء بتخزين التعديلات

بعد إنشاء مستودع شيفرة جديد بداخل المجلد الرئيسي لمشروعك الخاص، هل سيتم إضافة جميع ملفات مشروعك لذلك المستودع، أم أنني سأختار من تلك الملفات ما أريد تخزينه و أُلقي ما لا أريد تخزينه و متابعتها؟، سنفصل هنا عن هذا الأمر.

دعنا نفرض أن لديك مشروع تقوم بتطويره بلغة PHP لموقعك الشخصي، و لديك ٣ ملفات هي `home.php` و `contact.php` و `admin.php` في مجلد المشروع المسمى `MySite`. بعد إنشاء مستودع الشيفرة بداخل هذا المجلد، جاء دور تسجيل الملفات المراد متابعتها من قبل `Git`، و في هذه الحالة يقوم `Git` بتصنيف الملفات إلى نوعين، وهما `Tracked` و `Untracked`.

مفهوم Tracked

ببساطة عندما يجد `Git` ملف معين في مشروعك مثل `home.php` في البداية، فسيراه على أنه `Untracked`، و ليس `Tracked`، و المقصود بـ `Tracked` هو أن `Git` سيقوم بمتابعة التعديلات التي تسير على هذا الملف.

مفهوم Untracked

في حال جعلت أحد الملفات على أنه `Untracked` فأنت تقول لـ `Git` لا تتابع هذا الملف، أي لا تتابع التعديلات التي تجري عليه و لا تقوم بتخزينه أو التعامل معه. هذه الحالة هي ما تكون عليها الملفات في البداية.

هذا يعني أن ملفات المشروع الثلاثة في البداية ستكون جميعها **Untracked** و سنحتاج إلى جعلها **Tracked** من خلال الأمر **add**. كالتالي



```
git add home.php
git add contact.php
git add admin.php
```

بهذا الأسلوب، أضفنا الملفات و أصبحت جميعها **Tracked**. أي سيتم إدارتها و مراقبتها من قبل **Git**. الآن قد يتبادر إلى ذهنك، هل سأقوم بتنفيذ هذا الأمر على كل ملف في مشروعتي لإضافته؟. الجواب ببساطة هو لا، فقد يكون لديك ١٠٠ ملف ومن غير المنطقي إضافتها جميعها بهذا الأسلوب، لذا بإمكانك استخدام الطريقة المختصرة لإضافة جميع ملفات المشروع كالتالي



```
git add .
```

من خلال النقطة، ستصبح جميع الملفات **Tracked**، و نود الإشارة هنا، إلى أن هذا الأمر أيضاً يضيف أي ملفات حالتها **Modified** إلى مرحلة **Staged** وسنأتي على تفصيل هذا الأمر بإذن الله تعالى.

الحالات الثلاث لأي ملف مدار بواسطة Git

كما تعلمنا من قبل، أن الملف إما أن يكون Tracked أو Untracked، سنتعلم هنا أن أي ملف تتم إدارته من قبل Git أي Tracked سيمر بثلاثة حالات أساسية هي Modified و Staged و Committed، وسنتحدث عنها هنا بشيء من التفصيل.

١- حالة Modified

عند البدء بتعديل ملف معين أو مجموعة ملفات من مشروعك الخاص، فإن Git يعلم بذلك، وسيقوم بجعل حالة تلك الملفات على أنها **Modified**، أي تم تعديلها.

٢- حالة Staged

هذه المرحلة تحدد التعديلات التي سيتم تخزينها (قبل التخزين الفعلي)، فلو قمت بتعديل ٣ ملفات، فليس شرطاً أن تقوم بتخزين تلك التعديلات، بل بإمكانك أن تحدد الملفات المراد إلتقاط صورة التعديلات لها وتخزينها، و ينبغي التنبيه هنا إلى أنه في حال أردنا نقل ملف تم تعديله من حالة **Modified** إلى **Staged**، فإننا سنستخدم الأمر **add** بنفس الطريقة السابقة، بحيث نذكر اسم الملف أو نضع نقطة لنحدد جميع الملفات، أي كالتالي




```
git add home.php
```

ما نقصده بحالة **Staged**، أي أنه تم تحديد التعديلات التي سيتم إلتقاط صورة لها وتخزينها.

٣- حالة Committed

في هذه الحالة، يتم تخزين التعديلات بشكل فعلي في قاعدة بيانات Git وتعود الملفات التي تم تعديلها مسبقاً لتصبح حالتها **Unmodified**. أي لم يتم تعديلها منذ آخر **Commit** تم عليها، بكلامٍ آخر، يقوم Git بأخذ التعديلات التي تم حصرها في مرحلة **Staged** و من ثم تخزينها، أي إلتقاط الصورة الفعلية لحالة الملفات.

يتم تخزين الصورة النهائية من خلال الأمر التالي

```
 git commit -m 'reason here..'
```

من خلال الأمر السابق، سيتم التخزين الفعلي للتعديلات و تحفظ و تعود كما ذكرنا سابقاً حالة الملفات إلى **Unmodified**. و بعدها، و بمجرد التعديل على أي ملف من الملفات السابقة، سيتم تكرار نفس الخطوات السابقة، بحيث يمر الملف بحالة **Modified** ثم **Staged** ثم **Committed**.

ينبغي التنبيه هنا إلى أن فهم الحالات التي يمر بها الملف هو أمر جوهري، كونها هي الأمر الأساسي الذي يحدث بداخل **Git**. و بشكل متكرر طوال فترة تطوير المشروع البرمجي أو غير البرمجي الخاص بك.

الأقسام الثلاث لأي مشروع مدار بواسطة Git

تحدثنا عن الحالات التي يمر بها الملف الذي تتم إدارته من قبل Git، و الآن سنتحدث بطريقة توضح الصورة الكبيرة لأي مشروع برمجي يدار من خلال Git، بحيث سنوضح الأقسام التي يتواجد فيها الملف أو مجموعة الملفات أثناء العمليات المتكررة كالإضافة و الحذف و التعديل و غيرها.

1 - Working Directory

ببساطة هو مجلد المشروع الخاص بك و الذي تعمل عليه.

2 - Staging Area

هي المنطقة التي تنتقل إليها التعديلات التي قام بها المبرمج قبل تخزينها بشكل نهائي.

3 - .git Directory - Repository

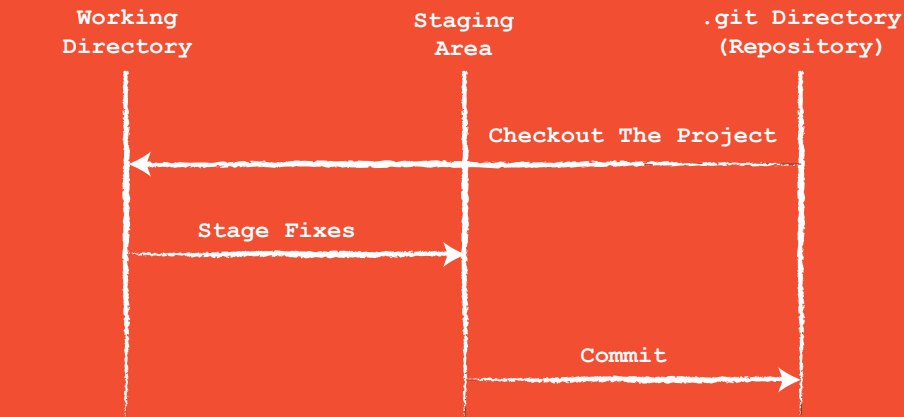
هي المنطقة التي تنتقل لها الملفات من منطقة Staging ليتم تخزينها بشكل دائم.

يمكننا تلخيص طريقة العمل أو ما يسمى بـ Git Workflow كالتالي

١ يقوم المبرمج أو المحرر بتعديل الملفات في Working Directory أو مجلد المشروع.

٢ بعد ذلك يقوم بإضافة الملفات التي تم تعديلها إلى منطقة Staging Area.

٣ يقوم بعمل commit، و التي بدورها تقوم بأخذ جميع التعديلات الموجودة في Staging Area ومن ثم تخزينها بشكل دائم في Git Directory.



Git Workflow

يوضح الشكل خطوات سير العمل في **Git** أو **Git Workflow**، بحيث يتم نقل التعديلات من **Working Directory** إلى **Staging Area** و من ثم يتم عمل **Commit** لنقلها إلى **Git Directory** أو ما يسمى بـ **Repository**.

في ما يخص **Checkout**، فهي العملية التي تقوم بها لكي تقوم بجلب إصدار أو نسخة من المشروع من مجلد **Git** إلى مجلد العمل **Working Directory** أي أن **Working Directory** هو نسخة واحدة من المشروع أي **Single Checkout**.

التعرف على أنواع الكائنات Object Types في Git

Git Objects تمثل فعلياً البيانات التي يتم تخزينها في قاعدة بيانات Git أو في نظام الملفات، و يعتبر Git Object هو الشيء الأساسي الذي بني من أجله Repository أو المستودع، بحيث يقوم المستودع بتخزين البيانات على شكل كائنات Objects. سنتعرف هنا على هذه الكائنات Objects بتفصيل.

في مستودع الشيفرة أو مجلد Git أو ما يسمى **git Directory**، يتم تخزين البيانات على شكل كائنات **Objects**، و يوجد عدد مختلف من الكائنات في **Git**، حيث يوجد ٤ أنواع أساسية يتم تخزينها بشكل أساسي في **Git Object Database** و المتواجد بداخل **git Directory**، وتمثل تلك الكائنات البيانات المراد تخزينها.

كل كائن من تلك الكائنات يتم تخزينه بعد عمل **compress** له أي (ضغطه) وتوليد رقم ست عشري طويل خاص به من خلال خوارزمية **SHA-1** وهي اختصار لـ **Secure Hash Algorithm**، بحيث يصبح ذلك الرقم مرجع يمكن استخدامه للرجوع لنفس الكائن، و يتكون ذلك الرقم من ٤٠ خانة.

الآن بإمكانك النظر للأمر على أن كل التعديلات التي نقوم بها على مشروع تتم إدارته بواسطة Git، سيتم تخزينها على شكل كائنات Objects منظمة بطريقة معينة، وكل كائن من تلك الكائنات له رقم مميز يمكننا من خلال الرجوع لذلك الكائن للتعامل معه، و من هنا يتضح لنا أن الأوامر التي نقوم بتنفيذها على مستودع Git، جزء كبير منها في الأصل يتعامل مع تلك الكائنات إما بتخزينها أو حذفها أو تعديلها أو استرجاعها.

أنواع الكائنات التي يتم تخزينها Git Object Types

يوجد ٤ أنواع من الكائنات **Objects** التي يتم تخزينها في **Git** وهي

١. **Blob Object** و يستخدم لتخزين محتويات الملف.
٢. **Tree Object** و يستخدم لتخزين المجلدات.
٣. **Commit Object** و تشبه **Tree** في كثير من الأمور، ولكن تقوم بتخزين مؤشر يشير إلى **Tree** معينة و تحتفظ بمعلومات إضافية مثل المؤلف و الرسالة و مؤشر **Parent Commit** و غيرها.
٤. **Tag Object** و يمكن التفكير فيه على أنه طريقة لإعطاء **Commit** معينة اسم مستعار يمكن الرجوع له في أي وقت.

سنقوم بالتفصيل في كل نوع من هذه الأنواع على حدى، و لكن الآن، مفهوم هذه الكائنات يتضح عندما تعلم أنه عندما تقوم بتخزين البيانات في مستودع **Git** فأنت تقوم بتخزينها على أحد تلك الأشكال التي تم ذكرها، و فهم كل نوع من تلك الأنواع يساعد على وضوح الصورة.

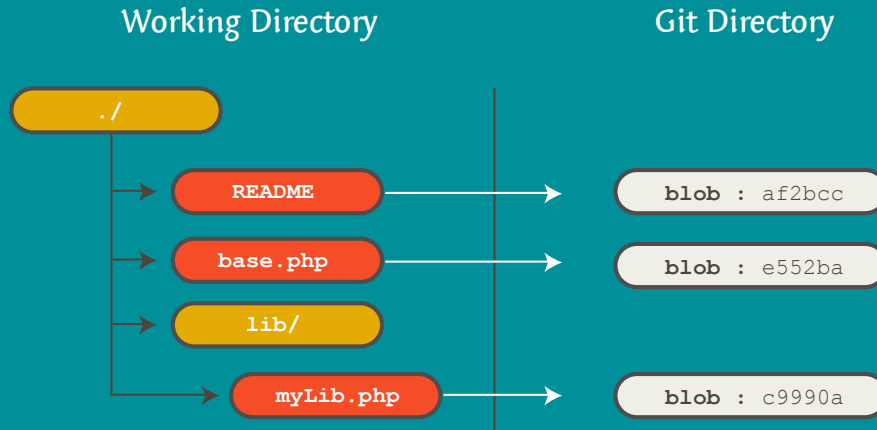
سنفرض أن لدينا مشروع معين في مجلد يحتوي على الملفات README و base. php و مجلد آخر بداخله اسمه lib يحتوي على ملف واحد بإسم myLib.php. فبالغة Git يسمى مجلد المشروع Working Directory، و سنوضح من خلال هذا المشروع أنواع الكائنات السابق ذكرها فيما يلي من صفحات.

النوع الأول : مفهوم Blob Object

يتم تخزين محتويات الملفات على هيئة كائن نوعه Blob، ويجب التنبيه هنا إلى أن ما يتم تخزينه هنا ليس الملف نفسه وإنما محتويات الملف، أي لا يتم تخزين التفاصيل الأخرى مثل اسم الملف وغيرها من التفاصيل المرتبطة به.

النوع الأول من الكائنات التي يقوم **Git** بتخزينها هو **Blob**، حيث أن أي ملف موجود في مشروعك البرمجي سيتم تخزين محتوياته على هيئة **Blob** و يكون له رقم مرجعي خاص به.

بما أن **Git** يقوم بتخزين المحتوى الخاص بالملف و ليس الملف نفسه، فهذا يعني أنه في حال كان المشروع يحتوي على ملفين أو أكثر بنفس المحتوى، فإن **Git** سيقوم بتخزين **Blob** واحد لهم جميعاً، وفي حال الإسترجاع يقوم بتوزيع **Blob** نفسها على الملفات المسترجعة. هذا الأمر يساعد كثيراً على توفير المساحة و العمل بفعالية كونه لا يوجد إلا نسخة واحدة من **Blob** مهما تعددت الملفات المتكررة بنفس المحتوى.



Git Blob Object

لاحظ كيف يتم تخزين الملفات في Git Directory، بحيث يظهر جزء بسيط من الرقم المسند لكل Blob Object و ليس كامل الرقم و المكون من ٤٠ خانة كما سبق و ذكرنا.

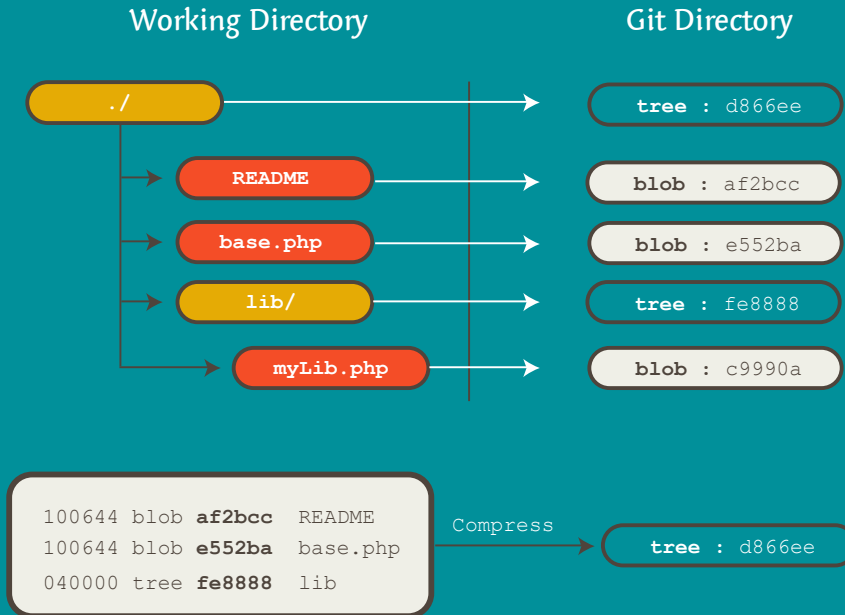
النوع الثاني : مفهوم Tree Object

كائن Tree هو عبارة عن طريقة لتمثيل مفهوم المجلدات Directories، فمثلاً يتم تخزين محتويات المجلد من خلال تخزين مؤشرات أو مراجع تشير إلى كائنات Blobs و كائنات Trees الموجودة بداخل Tree نفسها، سنتحدث هنا بتفصيل أكثر عن هذا الكائن.

النوع الثاني من الكائنات التي يقوم Git بتخزينها هو كائن Tree، وهنا نقول أنه يقوم بتخزين المجلدات، بحيث يكون لكل مجلد في مشروعك كائن من نوع Tree.

ببساطة، يقوم كائن Tree بتخزين Blob أو أكثر و Tree أو أكثر، أي يمكن النظر لها بنفس مفهوم المجلد مع الملفات، فالمجلد يمكن أن يحتوي على أكثر من ملف و مجلد بداخله، وكذلك كائن Tree يحتوي بداخله أكثر من Tree أو Blob.

فعلياً، كائن Tree ليس أكثر من قائمة تحتوي بيانات توضح ماهي كائنات Blobs و كائنات Trees الموجودة بداخلها، بالإضافة إلى رقم كل كائن من تلك الكائنات.



يوضح الشكل على اليسار محتويات المجلد الأساسي Working Directory و كيف تظهر في الأسفل على أنها مجرد قائمة تحتوي على نفس المجلد في الأعلى، وهذا يقودنا إلى أن Tree ليس إلا كائن يحتوي بداخله على قائمة بالكائنات الأخرى التي يحتويها وتفصيلها الأخرى، مع التنبيه على أن الرقم الأول من اليسار في القائمة يسمى Mode.

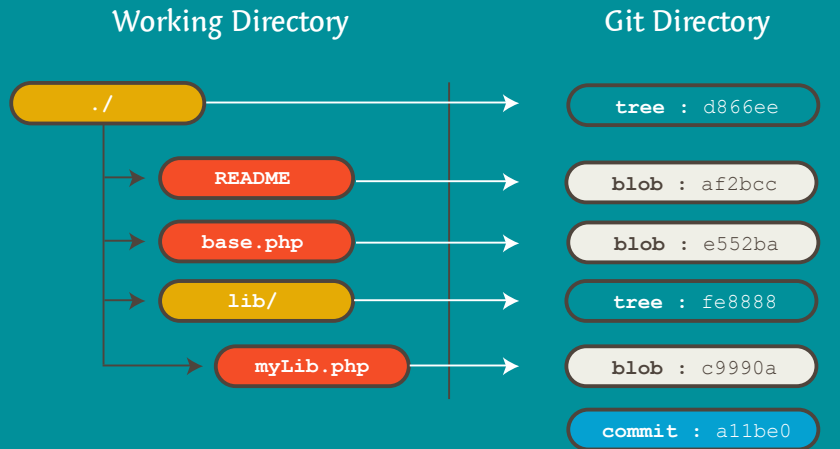
Git Tree Object

النوع الثالث : مفهوم Commit Object

ذكرنا فيما سبق أننا نقوم بتخزين لقطة أو صورة لحالة المستودع، يعتبر كائن Commit من الكائنات البسيطة، وهو المسئول عن تخزين بيانات تلك اللقطة، فهو مشابه إلى حد كبير لكائن Tree. سنتحدث هنا بتفصيل أكثر عن هذا الكائن.

النوع الثالث من الكائنات التي يقوم Git بتخزينها هو Commit، وهنا نقول أنه مشابه لمفهوم Tree نوعاً ما، لكنه يمثل مؤشر يشير إلى Tree Object، بحيث يحتفظ بذلك المؤشر، ويحتفظ بمعلومات المؤلف أو الشخص الذي نفذ التعديلات مثل اسمه و بريده الإلكتروني وكذلك يحتفظ هذا الكائن بالمعلومات الزمنية التي تمت فيها العملية، وأخيراً يحتفظ بالرسالة التي كتبها المبرمج ليوضح التعديلات أو العمليات التي قام به.

بكلام آخر، يمكن النظر إلى هذا الكائن على أنه مجرد مؤشر يشير إلى كائن Tree بحيث يحتفظ بمعلومات عن كائن Tree ومعلومات أخرى إضافية.



الشكل الموجود على اليسار يوضح كائن Commit و التفاصيل التي يحتويها، مع التنبيه على أن كائن Commit يأخذ أيضاً رقم SHA-1 خاص به.



Git Commit Object

النوع الرابع : مفهوم Tag Object

عدد العمليات التي تجري على المستودع البرمجي قد يكون كبير جداً و حتى في حالاته المتوسطة، سيحتوي على عدد commits وتفاصيل لعمليات كثيرة جداً، و مع أنه بإمكاننا العودة لـ commit معين بدلالة رقمها، إلا أن Git يوفر طريقة مختصرة لوضع علامات معينة بطريقة بسيطة يمكن الرجوع لها بطريقة أسهل.

النوع الرابع من الكائنات التي يقوم Git بتخزينها كائن Tag، و يمكن النظر له على أنه "علامة"، أو إشارة مثل الإشارة التي تضعها على سطر معين في كتاب أثناء القراءة لكي تعود لهذا السطر بسهولة عند فتح الصفحة، فكائن Tag يقوم بالإشارة إلى Commit معينة و يضع لها اسم مستعار واضح، أي يمكن الرجوع لكائن Commit من خلال رقمه أو من خلال اسم Tag الذي تم وضعه لها.

يقوم كائن Tag بتخزين عدد من المعلومات مثل رقم commit التي يشير إليها و اسم أو عنوان Tag الذي تم وضعه، بالإضافة إلى Tagger وهو الشخص الذي قام بإنشاء كائن Tag نفسه بالإضافة إلى بريده، و معلومات أخرى مثل الرسالة.

أنواع Tags في Git

Lightweight Tag - ١

يمكن النظر له على أنه ليس أكثر من مجرد مؤشر يشير إلى Commit معينه. و يمكن إستخدامه للأشياء المؤقتة و التي لا تحتاج إلى حفظ معلومات إضافية عن Tag.

Annotated Tag - ٢

هنا يقوم Git بتخزين Object كامل في قاعدة بياناته، و يحتوي على معلومات مختلفة مثل Tagger و بريده الإلكتروني بالإضافة إلى رسالة معينة قد تستخدم لتوضيح سبب وضعه، و غالباً ما ينصح بإستخدام هذا النوع كونه يحتوي على تفاصيل أكثر.

```
object 05744e
type commit

tag v0.1
tagger Abdullah Eid
      <me@algorithmers.com> 1205624433

this is my v0.1 tag
```

tag : 0b8c19

Git Tag Object

Git في Lightweight Tags و Annotated Tags

ذكرنا سابقاً أن لدينا نوعين من Tags وهما Annotated Tag و Lightweight Tag، وذكرنا أن Tag ماهو إلا علامة يقوم بوضعها المبرمج على Commit معينة لتشكيل اسم مختصر لها يساعدنا على العودة إليها بسهولة. سنتحدث هنا عن إنشاء Tag و كيفية التعامل معه.

ذكرنا أن Tag ماهو إلا عبارة عن مؤشر يشير إلى Commit معينة بغرض تسهيل العودة لها، فمثلاً بإمكاننا عمل Tag ليحدد أننا وصلنا لإصدار معين من المشروع ووضع علامة على آخر Commit تشير إلى ذلك الإصدار قبل أن نضيف Commit أخرى، وهذا بدوره سيساعدنا على معرفة الإصدارات، و بإمكانك عرض قائمة Tags الموجودة في مستودع معين من خلال استخدام الأمر التالي



```
git tag
```

بإمكانك أيضاً البحث عن Tag أو مجموعة Tags من خلال استخدام pattern أو صيغة معينة، ولتوضيح الأمر لاحظ الأمر التالي



```
git tag -l "v1.7*"
```

ببساطة، أخبرناه هنا بأننا نريد البحث عن أي Tag (لاحظ النجمة) يبدأ بـ v1.7، وهذا الأمر قد يفيد في حال كان لديك عدد كبير جداً من Tags وهذا ما يحدث في المشاريع الكبيرة، و ليس شرطاً أن نقوم بتسمية Tag لإصدار معين، بل بإمكاننا وضع الاسم الذي نريد، ولكن الأفضل أن يكون ذا مدلول واضح لسبب التسمية.

إنشاء Annotated Tag

لإنشاء **Annotated Tag**، فأسهل طريقة ستكون من خلال استخدام الخيار **-a** كما يلي

```
 git tag -a v1.8.0 -m 'version 1.8'
```

الآن تم إنشاء **Tag** بإسم **v1.8.0** و أما بالنسبة للرسالة التي تأتي بعد **-m** فهي رسالة يتم تخزينها مع **Tag** وهي تشبه فكرة الرسالة التي تأتي مع **Commit**.

إنشاء Lightweight Tag

لإنشاء **Lightweight Tag**، لا تقم باستخدام **-a** و **-s** و **-m** أثناء الإنشاء كالتالي

```
 git tag v1.8.0
```

لرؤية تفاصيل أكثر عن **Tag** بإمكانك استخدام الأمر **git show** كالتالي

```
 git show v1.8.0
```

بهذا الأمر، ستري تفاصيل أكثر عن محتوى أي **Tag** من كلا النوعين.

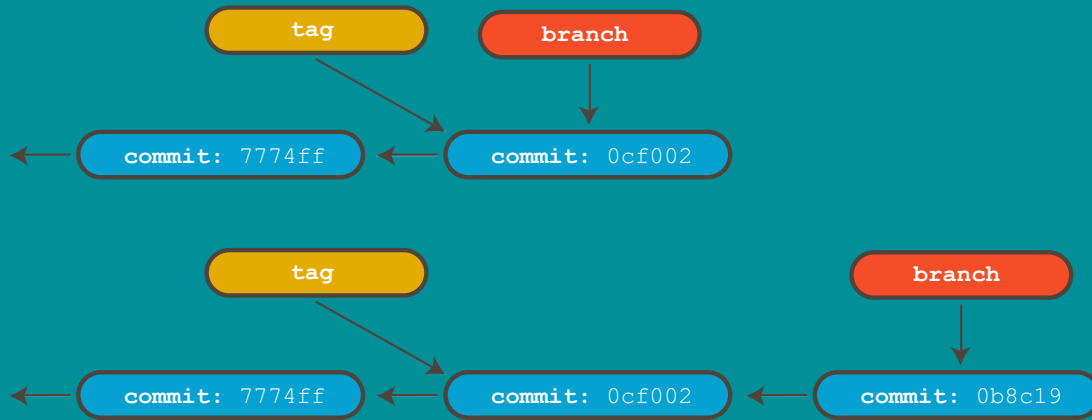
مفهوم المراجع أو المؤشرات References في Git

تقوم قاعدة بيانات Git بتخزين عدد مختلف من البيانات، بالإضافة إلى الكائنات الأربعة الأساسية التي يتم تخزينها، يقوم Git بتخزين بيانات أخرى، وفهمنا لتلك البيانات سيساعدنا على فهم كيفية عمل Git بشكل أسهل.

بالإضافة إلى الكائنات الأساسية الأربعة التي ذكرناها من قبل، و التي يتم تخزينها بشكل ثابت غير قابل للتغيير، يقوم Git بتخزين ما يسمى أيضاً بـ **References** أو المراجع أو المؤشرات، وهي عبارة عن مؤشرات بسيطة تشير إلى **Commit** معينة بحيث تكون قابلة للتغيير لتشير إلى **Commit** أخرى.

يمكن النظر إلى تلك المؤشرات على أنها مثل Tag بالضبط و لكنها قابلة للتغيير لتشير إلى Commit أخرى على عكس Tag

من الأمثلة على المراجع أو المؤشرات **References**، ما يسمى بـ **Branch** و **Remote** (سنحدث عنه لاحقاً)، فـ **Branch** هو مجرد ملف يتم تخزينه في مجلد **git/refs/heads**، حيث يحتوي ذلك الملف على **SHA-1** و التي تمثل أحدث **Commit** أو آخر **Commit** من ذلك التفرع **Branch**. لذا عندما تقوم بعمل **Commit** في كل مرة، فإن مؤشر **Branch** يتغير ليشير لآخر **Commit** قمت بها (راجع الصور المتحركة المرفقة مع الكتاب)، أي أن مؤشر **Branch** يتحرك في كل **Commit** نقوم بها، و المقصود بكلمة "يتحرك"، أي أنه يستبدل رقم **Commit** الذي يشير إليه أو **SHA-1** الخاص بها برقم الجديدة.



Git Branch Reference

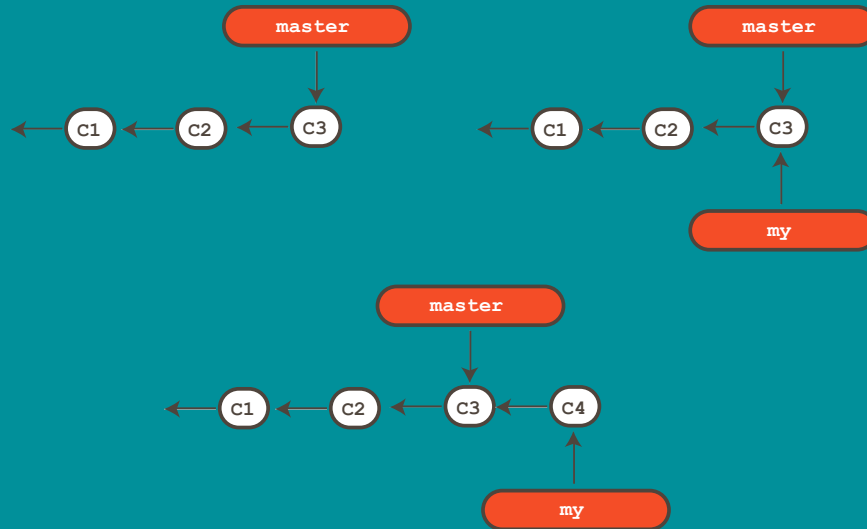
لاحظ كيف يتحرك مؤشر branch في الجزء الأعلى من الصورة ليشير إلى Commit أخرى في الجزء الموجود في الأسفل من الصورة

مفهوم Branching في Git

لو أتينا للترجمة الحرفية لكلمة Branching فسنجد أنها تعني "تفرع"، وهنا نقول أنه بإمكانك عمل مسارات من خلال Branching، أي يستطيع أي مطور أن يقوم بإنشاء "تفرع" أو "مسار" ليقوم بتعديل أشياء معينة أو تطويرها، بحيث يضيف في ذلك المسار أي عدد يريده من التعديلات أو Commits.

من المعلوم أننا عند التخزين في Git فنحن نقوم بإنشاء Commit أو أكثر، و يكون هناك مؤشر Branch يشير إلى أحدث (آخر) Commit، و بشكل افتراضي، يكون اسمه master. يتحرك مؤشر master مع كل commit جديدة كما ذكرنا، و السؤال الآن، هل بإمكانني إنشاء Branch جديد آخر غير master؟.

ببساطة، نعم، يمكن إنشاء أكثر من تفرع، أو Branch أو مسار، ولكن الآن، ما الذي سيحدث في حال قمنا بهذا الأمر؟، هنا سنجد أن لدينا أكثر من Branch أو مسار أثناء تطوير المشروع و كل مطور أو فريق قد يعمل في مسار معين ضمن نفس المشروع.



لاحظ في الرسم (من اليسار لليمين) كيف أن master يشير إلى C3 والتي تمثل آخر Commit تم إضافتها، و بعدها قمنا بإنشاء Branch جديد اسمه my والذي يشير في البداية إلى آخر Commit، ثم بدأنا التطوير في فرع my وبقي فرع master كما هو لم يتغير.

الآن قد يأتي مطور ويعمل على مسار master و آخر على مسار my، وهنا سنجد أن التطوير سيذهب في مسارات مختلفة، فقد يكون هناك مسار لتعديل خطأ، ومسار لإضافة خاصية، ومسار آخر لعمل نسخة جديدة (معدلة كلياً) من كامل المشروع وهكذا.

Git Branch

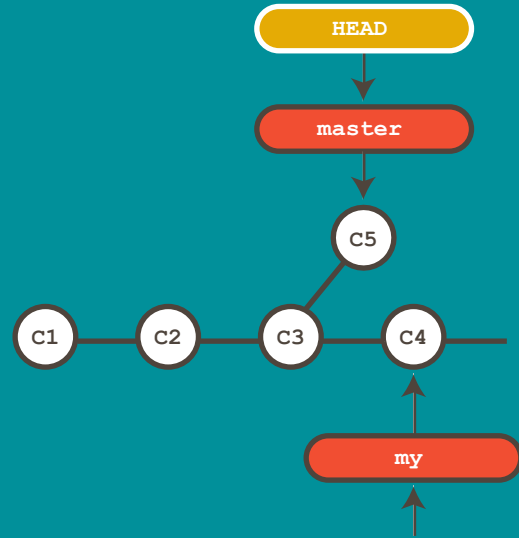
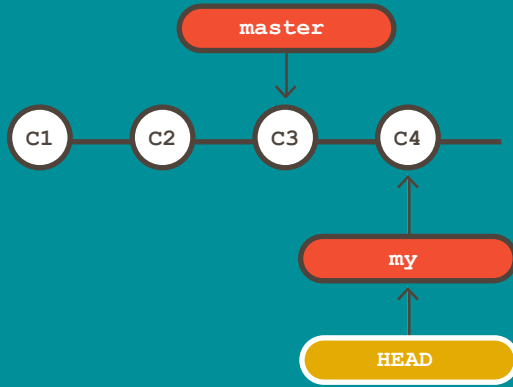
مفهوم مؤشر HEAD في Git

بعد أن يقوم المبرمج بإنشاء أكثر من تفرع أو Branching، يجب أن يكون هناك Branch أساسي يتم العمل عليه، وذلك لأنه في حال قمنا بعمل Commit، ففي أي Branch سيتم إضافتها؟.

كما أن Branch هو مؤشر يشير إلى Commit، بنفس الأسلوب نقول أن HEAD هو مؤشر يشير إلى Branch الحالي الذي يتم العمل عليه، لذا، يستخدم مؤشر HEAD للتنقل ما بين Branch و Branch.

لتوضيح الأمر، لو كان مؤشر HEAD يشير إلى master، فأني Commit نقوم بها سيشير إليها مؤشر master و سيتحرك معها، أما في حال قمنا بتغيير مؤشر HEAD ليشير إلى my، فعندها نقول أن أي Commit جديدة سيشير لها المؤشر my و سيتحرك معها.

لاحظ في الرسم التالي و أنظر من اليسار كيف أن المؤشر HEAD كان يشير إلى my، و في هذه الحالة إذا قمنا بعمل أي Commit فسيتحرك my مع آخر Commit أضفناها. أما بالنسبة للشكل الآخر على اليمين، فقد قمنا بتغيير المؤشر ليشير إلى master، و بعد ذلك قمنا بعمل Commit وهي C5، ولاحظ بعدها كيف تفرغ الموضوع من C3 إلى C5، وكيف أن المؤشر master هو من تحرك (راجع الصور المتحركة المرفقة مع الكتاب).



Git HEAD

يمكن النظر للموضوع ببساطة على أن مؤشر HEAD هو الذي يحدد Branch الذي يتحرك أو الذي نعمل عليه حالياً. ويمكن الانتقال من Branch إلى Branch ونقل مؤشر HEAD من Branch إلى آخر من خلال استخدام `checkout`.

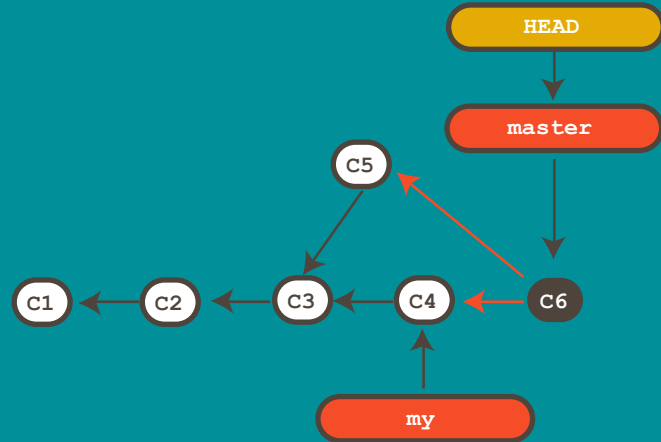
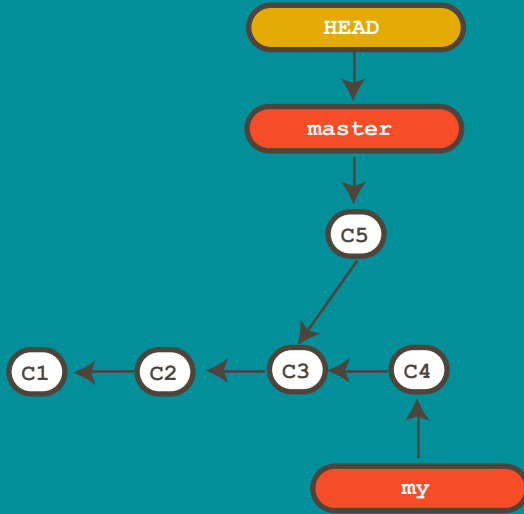
مفهوم الدمج Merge في Git

تعرفنا سابقاً على مفهوم Branching، وكيف يوفر لنا Git آلية للتطوير من خلال عمل تفرع و التطوير من خلاله، بحيث يستطيع مطور أو أكثر أن يعملوا على Branch معين دون التأثير على أي عمل آخر في نفس المشروع، سنتحدث هنا عن آلية Merge وهي كيفية دمج تلك التفرعات مع بعضها البعض.

افترض أن لديك Branch أساسي master و Branch آخر يعمل عليه أحد المطورين معك بإسم my ليقوم بإضافة خاصية جديدة دون أن يؤثر على العمل الأساسي. فلنفرض أن المبرمج أنتهى من تطوير تلك الخاصية، هنا نقول أنه أصبح جاهزاً لدمج Merge العمل الذي قام به في my مع العمل الأساسي الموجود في master، وهنا تسمى هذه العملية بعملية دمج أو Merge في Git.

ببساطة، لاحظ الرسم التالي، في الخطوة الأولى كان master (ويسمى في هذه الحالة Current Branch) يشير إلى C5، وكان my يشير إلى C4، وكل منهما يسير في مسار تطوير خاص به.

عندما قمنا بعملية الدمج في الخطوة التي تليها، لاحظ C6 والتي قامت بشكلٍ عام بالربط بين المسارين السابقين من خلال احتوائها على أكثر من Parent، فأصبح لها C4 و C5.



Git Merge

الآن لو قمنا بإضافة Commit جديدة، و لنقل C7 فستأتي مباشرة بعد C6 و ستشير إلى C6 كـ Parent لها. نخلص هنا إلى أن Merge هو عملية لدمج مسارات التطوير التي يقوم بها المطورين على نفس مستودع الشيفرة.

أصبح master الآن يشير إلى أحدث Commit وهي C6، ولو عدنا للرسم لوجدنا أن هناك سهمين باللون الأحمر وهما فعلياً من يقومان بربط المسار أو التفرع أو Branch المسمى master بـ my، وهنا أصبح مسار التطوير واحد.

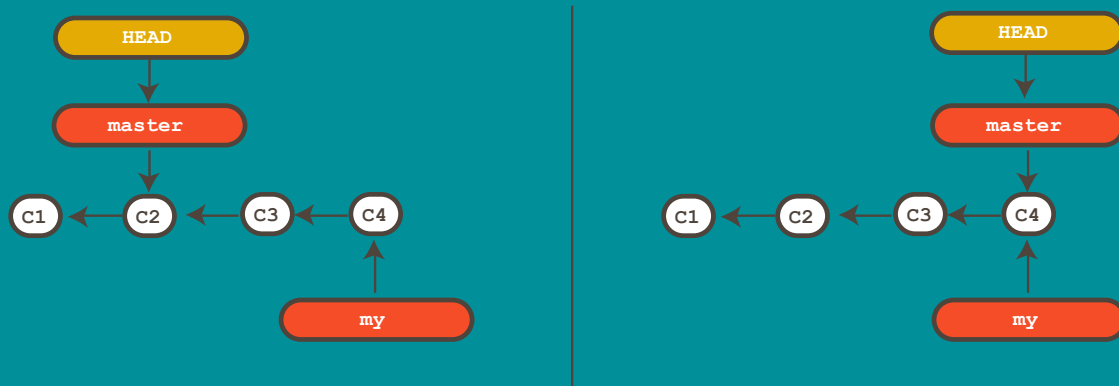
الدمج Merge و أسلوب fast-forward

تختلف التفرعات Branching التي تنتج من مستودع لأخر بناءً على العمليات التي حدثت عليه، وعند دمج تلك التفرعات فإن Git يستخدم أكثر من طريقة أو خوارزمية لإجراء عملية الدمج، سنتحدث هنا عن أسلوب أو طريقة fast-forward المستخدمة في عملية الدمج.

يعتمد أسلوب الدمج بإستخدام fast-forward على طريقة بسيطة للغاية، فكل ما يقوم به Git هو تحريك مؤشر Current Branch أو التفرع الحالي، و في حالتنا هنا دعنا نقول master، ليشير إلى نفس المكان الذي يشير إليه Target Branch، و في حالتنا هنا my.

لاحظ الصورة التالية، فلو نظرنا إليها من اليسار إلى اليمين، فسنجد master يشير إلى C2 و my يشير إلى C4، و لو دققت النظر، فستجد أننا لو قمنا بتحريك مؤشر master ليشير إلى my فسنسير في مسار خطي و لن ندخل في مسارات مختلفة، وهذه هي القاعدة التي يقوم عليها أسلوب أو خوارزمية fast-forward.

ببساطة، يظهر أسلوب fast-forward عندما يكون الإنتقال بشكل خطي في مسار واحد دون الدخول في مسارات أخرى غير مستقيمة أثناء الإنتقال، وهذا يعني أن Git يقوم بتحريك المؤشر للأمام فقط لإجراء عملية الدمج بحيث يشير كلا المؤشرين إلى نفس المكان.



Git Fast-Forward Merge

لاحظ الرسم الموجود في الأعلى، ففي الخطوة الثانية الموجودة في يمين الشكل قمنا بإجراء عملية الدمج، و إذا دققنا، فلن نجد سوى أن Git قام بتحريك مؤشر master ليشير إلى C4.

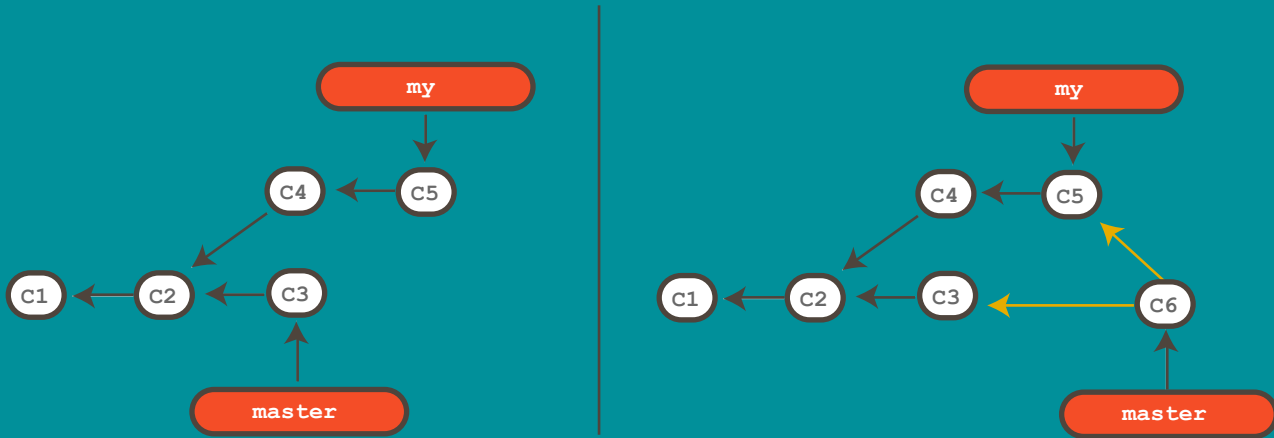
نقطة مهمة يجب الإنتباه لها، وهي أن هناك Branch يسمى Current وهو الذي يشير إليه HEAD و نعمل عليه حالياً، و Branch يسمى Target وهو المراد دمجها أو عمل Merge له مع Current Branch، وهذا يوضح لنا أننا يجب أن نكون على علم "بمن سيدمج مع من".

الدمج Merge و أسلوب 3-way

عند عمل Merge فأسلوب fast-forward يحدث عندما يكون هناك مسار خطي من Current Branch إلى Target Branch، وهذا الأمر ليس دائماً الحدوث، ففي المشاريع الكبيرة تكون التفرعات متباعدة ومتفرعة بشكل معقد، لذا سنتحدث هنا عن أسلوب آخر يقوم به Git لإجراء عملية الدمج Merge.

عند العمل على المشاريع المتوسطة والكبيرة، ستجد أن التفرعات أو Branching التي يقوم بها المطورون كثيرة و قد تكون معقدة أيضاً، وحيث أن fast-forward مفضل لدى المطورين في حال أرادوا إصلاح Bug معين كونه بإمكانك عمل Branch لإصلاح الخطأ يلي Current Branch بشكل مباشر، وهنا يمكن تنفيذ fast-forward كون مسار التفرع خطي.

بالنسبة للمشاريع الكبيرة، قد يتم إنشاء Branch و العمل عليه لفترة طويلة جداً مثل إضافة خاصية للتطبيق أو إعادة إصلاح عدد من الأمور الأساسية، وقد يعمل جزء من الفريق على هذا الأمر لمدة شهر مثلاً، بينما يتم العمل على التفرع الأساسي في نفس الوقت، وهنا قد نجد أن هناك مسافة بعيدة جداً وغير خطية بين Current Branch و Target Branch، وفي هذه الحالة لا يمكن تطبيق fast-forward، لذا يلجأ Git إلى ما يسمى بأسلوب 3-Way Merge.



Git 3-Way Merge

يعتمد أسلوب 3-Way على إنشاء Commit تقوم بربط طرفي التفرعين ومن ثم تحريك Current Branch لها، ولو لاحظنا الصورة الموجودة في الأعلى، فسنجد من اليسار أن master يشير إلى C3 و my يشير إلى C5، ومسار التطوير غير خطي، وعندما إجراء الدمج في الخطوة الثانية، لاحظ كيف تم إنشاء Commit بإسم C6 تربط ما بين الطرفين، ومن ثم تم تحريك مؤشر master لها. تسمى C6 في هذه الحالة بـ Merge Commit.

مفهوم Merge Conflict في Git

يقوم Git بعمل جيد تجاه إدارة المشروع الذي نعمل عليه، لكن هناك حالات قد تتسبب في جعله لا يستطيع أن يأخذ القرار المناسب فيها، و التي بدورها تحتاج تدخل منا كمطورين لإدارة أو تحديد المطلوب بشكل مباشر، سنتحدث هنا عن Merge Conflict.

ذكرنا سابقاً أننا نستطيع عمل Merge أو دمج بين تفرعين Branches، ولم نذكر أنه قد تحدث بعض المشاكل في هذا الأمر أو أثناء عملية الدمج، لذا تخيل معي أن لدينا master branch و my branch، و كل منهما قام بتعديل ملف اسمه base.php، عندها، وعندما نقوم بعملية الدمج، ستحدث مشكلة كون Git دمج تلك التعديلات على نفس الملف، فقد يكون هناك من قام بحذف بعض الأمور من الملف، بينما الآخر أبقى تلك الأمور و أضاف شيفرات أخرى، وهنا لا يملك Git القدرة على معرفة المطلوب بالضبط فيحدث ما يسمى بـ Merge Conflict.

عندما تواجه مشكلة Merge Conflict، قم بعمل git status و سيعطيك Git تفاصيل أكثر عن المشكلة و أين تكمن، وهذا سيساعدنا على التعرف على الملف المراد المراد معالجته و في أي Branch تكمن تلك المشكلة، و تسمى عملية معالجة المشكلة أثناء حدوث Merge Conflict بإسم Resolving Conflicts.

بعد إجراء التعديلات، كل ماتحتاجه هو إجراء الروتين الإعتيادي الذي تقوم به، من خلال عمل add على conflicted file(s) لتخبر Git أنه تمت معالجة المشكلة Resolved. بعد ذلك بإمكانك تنفيذ git commit وذلك لإنشاء merge commit (سبق و تحدثنا عنها).

نود التنبيه هنا إلى أن Merge Conflict لا يمكن أن يظهر في حال كان الدمج بأسلوب fast-forward، ويظهر فقط في حالة كان الدمج من خلال أسلوب 3-Way Merge

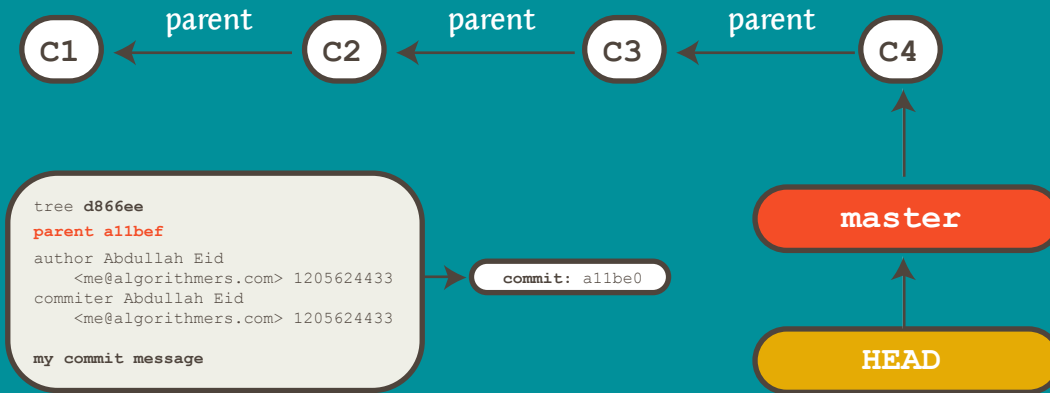
مفهوم مؤشر Parent في Git

يعتبر مؤشر Parent من المؤشرات المهمة في فهم آلية عمل Git، بل و يدخل في صميم التركيب العام للهيكل الذي يقوم ببنائه Git لتسيير عمليات التخزين و الإسترجاع، سنتحدث هنا عن هذا المؤشر بتفصيل.

لو دققنا النظر، لوجدنا أن أغلب العمل الذي نقوم به هو تخزين Commit أو أكثر، مما ينتج عنه ما يسمى بـ History أو تاريخ العمليات التي نقوم بها و التعديلات التي تم إجرائها.

لو قلت لك الآن كيف يستطيع Git الرجوع بسهولة للعمليات السابقة التي تم تخزينها من قبل من خلال Commit؟، ببساطة، كل Commit تحتوي على مؤشر داخلي اسمه Parent، بحيث يشير هذا المؤشر إلى Commit التي قبله، و بهذا الأسلوب نجد أنه في حال كانت كل Commit تشير إلى التي تسبقها، فسنحصل في النهاية على سلسلة مترابطة تمكننا من الرجوع للوراء للحد الذي نريده، وهذا ما يستخدمه Git فعلياً للتنقل وإجراء العمليات مثل جلب Commit سابقة و غيرها.

الرسم التالي يوضح مؤشر Parent بين كل Commit و Commit، بحيث لا تحتوي أول Commit على مؤشر Parent كونه لا يوجد شيء قبلها.



Git Parent

هنا يمكننا القول بأن أي Commit جديدة نقوم بإضافتها، سيقوم Git بأخذ آخر Commit قبل الإضافة ووضعها كـ Parent لها، أي أن Git يقوم بربط Commit الأخيرة بـ Commit السابقة، وهذا يفسر عدم وجود مؤشر Parent لأول Commit.

لاحظ في التفاصيل الخاصة بـ Commit أو محتواها، فستجد أن مؤشر Parent ماهو إلا إشارة إلى رقم Commit التي تسبقها، وقد تم تحديدها باللون البرتقالي، و بالنسبة لأول Commit فلن تجد هذا السطر من ضمن المعلومات التي تم تخزينها.

للتفكير، ماذا يعني وجود أكثر من مؤشر Parent لـ Commit معينة؟

التراجع عن عمل Staging من خلال reset

أثناء التطوير قد تقوم بعمل add لملف أو مجموعة ملفات، وهذا بدوره سيجعلها Staged أي أصبحت جاهزة لحفظها و تخزينها بشكل دائم من خلال عمل commit، في هذه المرحلة قد تود لسبب ما عمل Unstage لها أو إرجاعها إلى ما قبل add، هنا يأتي دور reset.

يوفر Git طريقة سهلة لإرجاع التعديلات من Staging Area، فعند عمل التعديلات التي نريدها و استخدام add، فنحن نقوم بنقلها إلى Staging Area لتصبح جاهزة للتخزين بشكل دائم في Git من خلال commit، وهنا قد يحتاج المبرمج إلى إعادتها من تلك المنطقة لسبب ما، أي ما قبل add، بكلام آخر، يريد المبرمج التراجع و لإبريد التخزين بتلك الطريقة أو الحالة التي عليها التعديلات التي قام بها، وهنا يأتي دور الأمر reset، ولتوضيح الأمر تخيل أننا قمنا بما يلي



```
git add myCode.c
git status
```

بعد عمل status ستجد شرح مختصر أيضاً لكيفية لتراجع من ضمن المخرجات، و لكي نقوم بعمل Unstage لملف myCode.c، فسنقوم بما يلي



```
git reset HEAD myCode.c
```

لاحظ هنا كيف قمنا بالتراجع من خلال reset عن الملف myCode.c، و من الأمور التي ينبغي التنبيه لها هنا هو أن الأمر reset يملك أحد الخيارات المسماة `--head`، و استخدام هذا الخيار قد يكون خطير للغاية كونه قد يتسبب في التراجع عن التعديلات التي قمت بها ليس فقط Staging Area بل من Working Directory.

بعد اجراء الأمر reset، تتحول حالة الملف أو مجموعة الملفات من Unstaged إلى Modified، أي بكلام آخر، الإتجاه الأساسي هو إنتقال الملفات من حالة Modified إلى حالة Staged، هنا تحتاج فقط إلى التفكير في عكس هذه العملية بحيث تنتقل الملفات من حالة Staged لتصبح Modified أو Unstaged.

git checkout و العودة للنسخة ما قبل التعديل

قد تقوم بإجراء تعديلات على ملف أو مجموعة ملفات، و بعد فترة، تكتشف لأي سبب من الأسباب أنك تريد إلغاء كل تلك التعديلات و العودة إلى الوضع التي كانت عليه تلك الملفات قبل إجراء تلك التعديلات الأخيرة، سنتحدث هنا عن كيفية إلغاء التعديلات التي قمت بها و العودة بالملف أو الملفات كما كانت قبل التعديلات.

تخيل أنك تقوم بالتعديل على ملف برمجي معين بإسم `file.java`، و بعد عدد من التعديلات التي قمت بها على الملف، وجدت أنه من الأفضل إلغاء كل تلك التعديلات و العودة للنسخة التي كنت عليها قبل البدء في التعديل، أي تحويل حالته من `Modified` إلى `Unmodified`، هنا يوفر لك `Git` أسلوب مبسط لإجراء تلك العملية من خلال استخدام `checkout`. عندما تصبح حالة الملف `Modified`، وتريد إلغاء كل تلك التعديلات ليعود لشكله الأصلي، وفي حالتنا هنا `file.java`، قم بتنفيذ الأمر التالي



```
git checkout -- file.java
```

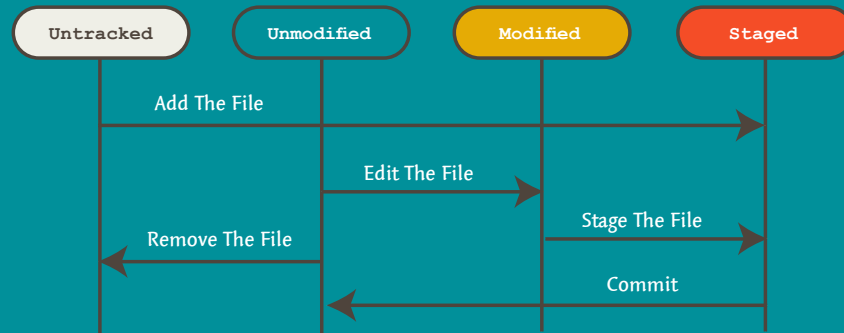
بهذا الأمر كأنك تقول: أرجع لي الملف `file.java` إلى حالته التي كان عليها منذ آخر `commit` قمت على المستودع وقبل التعديلات الخاصة بي. يمكن النظر لأمر `checkout` في هذه الحالة و كأنه يمثل "إلغاء الأمر" أو "تراجع" عن ما حدث من تعديلات. ينبغي التنبيه هنا إلى أن أمر `checkout` يعتبر من الأوامر الخطيرة، فهو يقوم بإلغاء التعديلات من خلال جلب النسخة السابقة من المستودع و إزالة النسخة التي تحتوي على التعديلات، أي سيستبدلها بالنسخة الأخيرة الموجودة في المستودع قبل التعديلات التي قمت بها، لذا من الأفضل أن لا تستخدم هذا الأمر إلا و أنت متأكد مما تقوم به، كونه سيكلفك الكثير في حال الخطأ.

يستخدم الأمر `checkout` أيضاً للتنقل من `Branch` لأخر، أي يستخدم لتحديد `Current Branch`.

يوفر `Git` طرق و أساليب متقدمة أيضاً في حال أردت إبعاد التعديلات التي قمت بها دون إلغائها بشكل نهائي، أي الإحتفاظ بالتعديلات مع بقاء الملف بشكله السابق، من خلال ما يسمى بـ `Stashing`، و سنأتي عن الحديث عنه بإذن الله تعالى.

Git Status و التعرف على وضع المستودع Repository الحالي

كما ذكرنا سابقاً أن الملف يمر بعدد من المراحل من خلال Git، لذا، نحتاج غالباً التعرف على حالة الملفات و حالة المستودع الحالية، و ماهي الأقسام التي تتواجد فيها الملفات التي يتم تعديلها في المشروع، أي هل هي معدلة Modified و أنها Staged أو Unmodified، وغيرها من الأمور، هذا ما سنتحدث عنه هنا.



Git Life Cycle

توضح الصورة الموجودة في الأعلى ما يسمى بـ **Git Life Cycle** أو دورة حياة الملف خلال التعديلات التي تجري عليه من خلال **Git**، وهنا نجد أن الملف يمر بأربع مراحل (حالات)، **Untracked** و **Unmodified** و **Modified** و **Staged**، لذا نحن نحتاج أثناء العمل على المشروع البرمجي إلى وجود أداة أو أمر يساعدنا على معرفة الحالة التي تتواجد ملفات المشروع عليها، وهذا هو دور **status**. بإمكانك معرفة حالة الملفات في المشروع كالتالي



```
git status
```

يعطيك الأمر **status** تفاصيل عن حالة الملفات كما تلخصها الصورة السابقة بشكلٍ مفصل، و بإمكانك الحصول على تقرير مختصر عن حالة الملفات من خلال استخدام التالي



```
git status --short
```

سيعطيك هذا الأمر تقرير مختصر حول حالة المشروع و التعديلات الحالية، و بإمكانك تنفيذ نفس الأمر من خلال استخدام **-s** كالتالي



```
git status -s
```

ستجد في النتائج رموز بجانب الملفات مثل **??** وهي تعني ملف جديد **Untracked**، و الملفات التي تم إضافتها إلى **Staging Area** سيكون بجانبها الرمز **A**، و أما التي يكون الرمز **M** بجانبها فتعني **Modified**. تتكون الرموز التي تأتي بجوار الملف من خانتين اليمنى ويسرى، قد تجد رمزين بجانب الملف، أو رمز واحد يأتي على اليمين أو على اليسار، حيث تمثل الخانة اليسرى حالة **Staging Area** و أما اليمين **Working Directory**.

مفهوم ملف gitignore. و الملفات التي يتم تجاهلها

ذكرنا أن الملف يكون بين حالتين، Tracked و Untracked و أي ملف جديد يتم وضعه في مجلد المشروع سيراه Git على أنه Untracked و سيظل يخبرنا بأن لدينا ملف جديد و يجب أن نضيفه ليصبح Tracked، ولكن أحياناً يكون هناك ملفات لا تريد رفعها أو جعل Git يتعامل معها من الأساس، وهنا يأتي دور ملف gitignore. في Git.

بعض المشاريع البرمجية عند عمل run لها فإنها ستقوم بتوليد ملفات خاصة مثل مشاريع C و C++ وغيرها داخل مجلد المشروع، وهذه الملفات ليس لها علاقة بالتطوير، ولكن Git سيراه على أنها ملفات جديدة، وعندما تقوم بعمل status لها، ستجد أنها موجودة، و أن Git يعطيك تنبيه على أنها Untracked و يجب عليك إضافتها، و في حقيقة الأمر، أنت تود منه أن لا يتعامل معها ولا يضعها في مستودع المشروع حتى لو كانت موجودة.

في هذه الحالة يمكنك إنشاء ملف اسمه gitignore. (لاحظ أن النقطة من ضمن الملف) بحيث تضع فيه الملفات التي تريد تجاهلها إما بشكل مباشر من خلال وضع أسماء تلك الملفات أو من خلال تحديد أنواع الملفات التي لا تريد دمجها أو وضعها أو متابعتها من قبل Git. الآن لو قمنا بإنشاء ملف gitignore. ومن ثم قمنا بكتابة التالي بداخله (ولاتنسى وضعه في مجلد المشروع)



```
*.gif
*.pdf
dump.php
lib.a
*~
```

في هذه الحالة أنت تقول تجاهل أي ملف امتداده gif و أي ملف امتداده pdf و كذلك تجاهل ملف dump.php و ملف lib.a و أي ملف ينتهي اسمه بعلامة ~. هنا نقول ببساطة أنها مجرد قائمة تجاهل للملفات التي لا تريد إقحامها في المشروع، فمثلاً لو كنت تعمل مع فريق عمل برمجي و كنت متبرعاً للمشروع، أي كمشروع مفتوح على GitHub، فإن عدم الإهتمام بملف gitignore. سيجعل من العملية معقدة نوعاً ما، فستجد أنك قمت برفع ملفات ليس لها علاقة بالمشروع و إنما هي ملفات إضافية لا يجب أن تتواجد في مستودع الشيفرة نفسه.

تستطيع تجاهل مجلد بكامل محتوياته أيضاً إن أردت من خلال تحديد ذلك المجلد ضمن قائمة التجاهل في الملف، بالإضافة إلى ذلك يمكنك استخدام Glob Patterns و يمكنك النظر إليها على أنها نسخة مبسطة من Regular Expressions لتحديد قواعد عامة للملفات و المجلدات التي يجب تجاهلها.

من الأمور الأخرى في الملف استخدام الرمز ! في البداية وذلك لإستثناء ملف معين، فمثلاً لو ذكرت في الملف أنك تريد تجاهل جميع الصور من نوع png ولكنك تريد التعامل مع صورة واحدة هي my.png، عندها بإمكانك وضع اسم الصورة يسبقه !.

Git rm وحذف الملفات من Git

في مشاريعك البرمجية، ستحتاج عاجلاً أم آجلاً إلى حذف ملف من Git، إما لأنك قمت بتقسيم محتواه على ملفات أخرى، أو أنه لم يعد مناسباً لإصدار المشروع الذي تعمل عليه أو أنه كان السبب في الكثير من المشاكل برمجياً، أيّاً كان السبب، بإمكانك حذف الملف من Git، وهذا ما سنتحدث عنه هنا.

لحذف ملف من Git، يجب عليك أولاً حذفه من الملفات التي تكون حالتها **Tracked** ثم بعد ذلك تقوم بعمل **commit**، لذا بإمكانك حذف ملف وإلغاء متابعته من Git من خلال الأمر التالي

```
 git rm myFile.py
```

ينبغي التنبيه هنا إلى أنه في حالة قمت بحذف الملف بشكل مباشر من مجلد المشروع الخاص بك ودون استخدام الأمر **git rm**، فعندها ستكون حالته في حال استخدمت **git status** هي

```
 Changes not staged for commit
```

و قمت بعدها بتنفيذ الأمر **git rm** فستصبح حالته عند تنفيذ **git status** كالتالي

```
 Changes to be committed
```

بعد ذلك لو قمت بعمل **commit**، فسيتم حذف الملف و لن تجده ضمن مجلد المشروع الذي تعمل عليه.

هناك حالات تود فيها حذف الملف من **Git** كمتابعة فقط، أي من **Staged Area** و لكن لا تريد حذفه من مجلد المشروع نفسه، أي فقط حذف المتابعة مع بقاء الملف نفسه. في هذه الحالة بإمكانك استخدام **--cached** مع **git rm** كالتالي



```
git rm --cached myFile.py
```

بإمكانك أيضاً استخدام **Glob Patterns** و هي نسخة مبسطة من **Regular Expressions** كما ذكرنا من قبل لتحديد مجموعة من الملفات المراد حذفها، فلو أردنا مثلاً حذف جميع الملفات من النوع **txt** من المجلد **settings**، فسنقوم بالتالي



```
git rm settings/*.txt
```

git mv و نقل و إعادة تسمية الملفات في Git

من العمليات المتكررة التي يقوم بها المبرمج أثناء العمل على مشروعه البرمجي، نقل الملفات بين المجلدات و كذلك إعادة تسمية الملفات، يوفر Git هذه الإمكانيات بطريقة سلسة، لذا سنتحدث هنا عن كيفية تنفيذ تلك العمليات من خلال Git.

بإمكانك نقل الملفات من مجلد إلى مجلد، أي من مكان لآخر داخل مشروعك بإستخدام **git mv**، فلو فرضنا أن لديك ملف بإسم **base.rb** و تريد نقله إلى مجلد موجود بإسم **lib** بداخل مشروعك، عندها بإمكانك تنفيذ الأمر التالي



```
git mv base.rb lib/base.rb
```

عندما نقوم بعمل **move** أو نقل ملف معين (مثل ماحدث في الأعلى)، فكأننا نخبر **Git** عن أمرين هما

١. تم حذف الملف **base.rb**.

٢. تم إضافة الملف **lib/base.rb**.

كلا الأمرين السابقين، الحذف و الإضافة، يتم نقلهما لمرحلة **Stage** ليكونا جاهزين لعمل **commit** في خطوة واحدة من خلال الأمر السابق.

يمكن النظر إلى الأمر **git mv** على أنه طريقة لإعادة تسمية الملفات **rename**، وهذا ما ستجده في المخرجات عن تنفيذ الأمر، لذا افرض أن لدينا ملف بإسم **core.java** و نريد إعادة تسميته إلى **base.java** عندها سنقوم بالآتي



```
git mv core.java base.java
```

هذا الأمر فعلياً هو مكافئ لتنفيذ ثلاثة أوامر كالتالي



```
mv core.java base.java  
git rm core.java  
git add base.java
```

يتضح لدينا هنا أن الميزة في الطريقة الأولى هي أنها اختصرت الكثير من خلال أمر واحد، فبدلاً من نقل الملف و حذفه من **git** ثم إضافة النسخة الجديدة، **git mv** قامت بكل تلك الخطوات.

سجل العمليات على Git و استخدام أمر git log

بعد القيام بعدد من التعديلات المختلفة و عمل أكثر من commits على المستودع الخاص بك، أو بعد القيام بعمل clone لمستودع موجود مسبقاً، ستحتاج غالباً إلى رؤية سجل يوضح الأشياء التي حدثت على مستودع الشيفرة من خلال رؤية تفاصيل على commits التي حدثت وغيرها، هنا سنتحدث عن هذا الأمر.

لرؤية التفاصيل السابقة للمستودع الذي تعمل عليه فكل ما عليك القيام به هو تنفيذ الأمر التالي



git log

سيقوم هذا الأمر بعرض عدد من commits التي تمت على المستودع الذي تعمل عليه، بالإضافة إلى تفاصيل كل commit من خلال عرض رقمها ومن قام بها و التاريخ و الرسالة التي توضح سبب أو وصف commit. بعد تنفيذ الأمر، قد ترى شيء مشابه لما يلي



```
commit 41821545ea65edabdbccac1a3e82c618dbd509ffb
Author: Abdullah Eid <me@algorithms.com>
Date:   Sun May 29 13:50:07 2016 +0300
```

Closure type hint has removed

لاحظ المعلومات التي ذكرناها عن commit و سيظهر لك عدد منها حسب العمليات التي تمت على مستودع الشيفرة نفسه، وهنا يظهر جلياً الرقم و من قام بالعملية و بريده الإلكتروني و تاريخ تنفيذ العملية، بالإضافة إلى الرسالة الأخيرة التي توضح ملخص و سبب لهذه العملية. ننبه هنا إلى أن الرموز الكثيرة التي تأتي بعد كلمة commit تسمى SHA-1 checksum.

تستطيع استخدام **-n** مع الأمر حيث يمثل **n** عدد وذلك لكي تحدد عدد معلومات **commits** التي ستظهر لك، فلو أردنا ٢ فقط، فسننفذ التالي

```
git log -2
```

أحد الخيارات المتاحة كذلك هو الخيار **-p** وهو من الخيارات التي تساعدك على معرفة تفاصيل أكثر عن كل **commit** بحيث تعرض تفاصيل عن التعديلات التي تمت كالتالي

```
git log -p
```

بإمكانك كذلك رؤية عدد من الإحصائيات بشكل مختصر في حال استخدمت الخيار **--stat** كالتالي

```
git log --stat
```

هناك تفاصيل كثيرة حول **log** و سنأتي على ذكر المهم منها في المواضيع المتقدمة، أما الآن فيكفي التعامل مع الخيارات التي ذكرناها كبداية و من ثم سيزداد الإحتياج للخيارات الأخرى كلما تقدمنا في المستوى. قبل ذلك سنتحدث عن خيار **--pretty** و خيارات تحديد المخرجات زمنياً.

git log و التعامل مع خيار --pretty

عند عرض سجل المعلومات السابقة و العمليات التي تمت على مستودع الشيفرة Repository، فإن Git يقوم بعرضها بطريقة افتراضية معينة، و قد لا تتناسب طريقة العرض تلك من الطريقة التي نود القراءة بها، يقدم خيار --pretty طريقة ممتازة لتنسيق طريقة العرض و تحديد المعلومات المراد عرضها بالإضافة لخيارات أخرى.

يمكنك تحديد عدد من القيم المختلفة لخاصية --pretty وأحد تلك القيم هو **oneline** بحيث يقوم بعرض المعلومات بطريقة مبسطة تشمل **SHA-1** و الرسالة في سطر واحد، ويتم تنفيذ الأمر كالتالي



```
git log --pretty=oneline
```

من ضمن القيم المتاحة غير **oneline**، بإمكانك استخدام **short** و **full** و **fuller** و كلها تعرض بنفس الطريقة باختلاف الإختصار أو الزيادة في المعلومات التي يتم عرضها. قد تحتاج أحياناً إلى طريقة خاصة بك لعرض المعلومات، بالإضافة إلى تحديد أشياء معينة لعرضها وليس كل شيء، هنا يأتي دور القيمة **format**، بحيث تساعدك على تحديد طريقة العرض التي تريدها و المعلومات التي تريد وضعها كالتالي



```
git log --pretty=format:"%h - %an, %ar"
```

لاحظ كيف وضعنا تنسيق معين للبيانات وحددنا البيانات التي ستعرض في النص الذي يلي **format**، و في هذا النص ضع أي شكل تريده، فمثلاً قمنا بوضع - و فاصلة و كل تلك الأشياء فقط لتحديد الطريقة التي سيعرض بها كل سطر. أما بالنسبة لمعنى تلك الرموز مثل **%h** وغيرها، ففي ما يلي عرض ببعض أهم الرموز المتاحة و معنى كل منها

%H و يعني **Commit hash** وهو الرقم الذي يأتي مع **commit**.
%h نفس السابق ولكن يعرض بطريقة مختصرة أي عدد محدد من الأرقام.
%an أي **Author Name** أو من قام بعمل التعديلات.
%ae أي **Author Email** أو بريد من قام بالتعديلات.
%ar أي **Author Date** وهي تاريخ إضافة التعديلات.
%s أي **Subject** وهي الرسالة أو النص الذي يوضح سبب التعديلات.

هذه بعض وليست كل الخيارات المتاحة للإستخدام ولكن اخترناها كونها أهم الخيارات التي يجب استخدامها في البداية.

git log و التعامل مع سجل المخرجات زمنياً و نصياً

تخيل أنك كنت تقوم بتطوير أحد المشاريع الكبيرة لفترة من الزمن، و لنقل ستة أشهر، عندها قد يبدو من الصعب عمل log لجميع تلك التعديلات التي تمت على مستودع الشيفرة خلال كل تلك الأشهر، لذا يوفر لك Git طريقة استعراض زمنية، بحيث يمكنك تحديد فترة زمنية معينة لترى التعديلات التي جرت فيها.

يوفر Git طريقة مرنة جداً لتحديد المخرجات زمنياً من خلال عدد من الخيارات و لتوضيح الصورة، بإمكانك مثلاً تحديد المخرجات خلال أسبوعين من خلال تنفيذ الأمر التالي



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

ننوه هنا إلى أنه بإمكانك تحديد المدة الزمنية بطريقة مختلفة مثل الأيام و الأشهر أو حتى خلال تاريخ محدد مثل "2016-07-24"، بالإضافة إلى أنه بإمكانك تحديده على الشكل "2 years 1 day 3 minutes ago".

يتيح خيار -S أيضاً طريقة رائعة للبحث في التعديلات من خلال البحث عن نص معين في ملفات الشيفرة البرمجية الخاصة بك، وسيقوم Git بجلب جميع commits التي وجد فيها ذلك النص إما بالحذف أو التعديل، أي بكلام آخر، تجلب commits التي في تعديلاتها نص معين، و لتوضيح الأمر، لاحظ كيف يتم تنفيذ هذا الأمر



```
git log -S myFunction
```

ببساطة تبحث هنا عن أي commits تم التعامل فيها مع دالة myFunction.

يوجد عدد كبير من الخيارات الزمنية و الغير زمنية المتاحة لتحديد المخرجات و وفق تلك المعايير مثل المخرجات حسب فترة أو المخرجات وفق مؤلف معين أو المخرجات قبل أو بعد فترة معينة، لذا، فيما يلي سرد لأهم (وليس كل) تلك الخيارات التي تساعدك على تحديد المخرجات وفق المعايير التي تريدها

n وتعني عرض عدد محدد **n** من المخرجات.

--after، **--since** التعديلات بعد تاريخ معين.

--before، **--until** التعديلات قبل تاريخ معين.

--author جلب المخرجات التي تطابق المؤلف.

هذه ليست كامل الخيارات المتاحة و إنما أهمها، وهنا بإمكانك التحكم في طريقة عرض المخرجات زمنياً من خلال تحديد الفترة التي تريدها بأكثر من أسلوب مرّن، وهذا أيضاً سيجعل عملية المراجعة فعالة، كوني لا أحتاج للبحث بين قائمة ضخمة من التعديلات التي لا أريدها أصلاً، وستوفر هذه الأدوات وقتاً وجهداً في المشاريع الكبيرة كونها تعطي المخرجات المطلوبة.

التراجع عن التعديلات بعد تخزينها في مستودع Git

أثناء التطوير، ستقوم بالكثير من التعديلات، و أحياناً قد تقوم ببعض التعديلات التي تود التراجع عنها، أو قد تقوم بعمل commit و تنسى أحد الملفات معها و الذي كان من المفترض أن يكون مضافاً في ملف gitignore. لتجاهله، ولكنك حفظته بالخطأ، يوفر Git طرق مختلفة للتراجع عن العمليات التي تقوم بها.

يوفر Git طريقة مرنة جداً للتراجع عن العمليات و التعديلات التي تقوم بها، لكن ينبغي التنبيه هنا إلى التعامل بحذر مع هذا الأمر، لأنه ليس دائماً بإمكانك التراجع، وقد تخسر الكثير من البيانات في حال قمت بتنفيذ هذا الأمر بطريقة خاطئة.

تخيل أنك قمت بعمل commit و من ثم نسيت إضافة أحد الملفات لها، أو قمت بعمل commit و أخطأت في كتابة الرسالة وأردت أن تعيد صياغتها مثلاً، أو لأي سببٍ ما، أردت أن تتراجع عن الأمر، عندها نقول أنه في حالة أردت المحاولة مرة أخرى، فقم بتنفيذ commit مع الخيار amend - كالتالي

```
 git commit --amend
```

في هذه الحالة و كأنك تقوم بمحاولة عمل commit مرة أخرى للمرة السابقة. في حالة لم تقم بأي تعديلات بعد أول commit قمت به و الذي تنوي التراجع عنها، عندها ستكون Snapshot أو اللقطة السابقة مطابقة للجديدة بإستثناء ما قد تغيره في الرسالة التي تقوم بوضعها مع commit.

دعنا الآن نوضح الصورة بمثال بسيط، تخيل أنك قمت بعمل commit و بعد ذلك وجدت أنك نسيت أحد الملفات، و لنقل file.cpp، و أردت أن تضيفه، عندها يمكننا القيام بالتالي بعد commit التي قمنا بها



```
git commit -m 'initial commit'  
git add file.cpp  
git commit --amend
```

لاحظ هنا أننا قمنا بعمل **commit**، و بعد ذلك تذكرنا أننا نسينا الملف **file.cpp**، ومن ثم قمنا بإضافته، و بعد ذلك وبدلاً من إنشاء **commit** جديد، قمنا بعمل **commit** مع الخيار **--amend**، وهنا سننتهي بـ **commit** واحدة، بحيث تستبدل **commit** الثانية النتيجة الخاصة بالأولى. بالإضافة إلى ما تم ذكره، يوجد طرق أخرى للتراجع أيضاً مثل استخدام أمر **reset**، لكن سنكتفي بهذا القدر هنا كبداية، وسنأتي على تفصيل هذه الأمور في المواضيع المتقدمة.

العمل مع Remote Repository

من المبادئ الأساسية التي يقوم عليها Git هو مبدأ التعاون Collaboration، ولذا فإن العمل على نسخة خاصة بك على جهازك أمر جيد، لكن لن يتعدى الأمر كونه عمل فردي، لذا ماذا لو كان لدينا مستودع أو Repository موجود على Server، يستطيع عدد من المطورين التعامل معه من أي مكان؟، هذا ما سنتحدث عنه هنا.

ببساطة، مستودع الشيفرة، يكون إما موجود على جهازك أو موجود على Server أو كلا الحالتين معاً. و Remote Repository ليس أكثر من Repository عادي متواجد على جهاز آخر، ويكون الوصول له إما للقراءة فقط Read Only أو يكون لك صلاحية القراءة و الكتابة Read/Write.

يقوم المبرمج بإنشاء مستودع الشيفرة على جهازه، و بعد ذلك يبحث عن موقع (Server) لإستضافة مستودع الشيفرة الخاصة به، ثم يقوم برفع نسخة من ذلك المستودع إلى ذلك الموقع، و من ثم يقوم بالتعديلات على النسخة المحلية (الموجودة على جهازه)، وبعد الإنتهاء يقوم بعمل Push وهي ببساطة إضافة التعديلات على المستودع الموجود عن بعد بحيث يصبح المستودع الموجود لدى المطور مطابق للمستودع الموجود عن بعد.

الآن لو أتى أحد المطورين للعمل معك، فسيقوم بعمل Clone للمستودع الموجود على السيرفر ويحصل على نسخته لبدء التطوير، والآن تبدأ عملية التعاون من هنا، فسيقوم هو بعمل التعديلات ومن ثم Push وهكذا، و في حال اختلفت النسخة الموجودة على الموقع عن أحدكم، فبإمكانك عمل Fetch لجلب آخر نسخة تم تطويرها، ومن ثم البدء بعمل التطويرات عليها و عمل Push وهكذا، وهذا الأمر يشرح الصورة العامة لعملية التعاون وسنأتي على تفصيل هذا الأمر بإذن الله تعالى.

ما يميز Git في هذه الحالة أن كل مطور لديه نسخة من المستودع، وفي حال حدثت مشكلة في الموقع أو Server الذي يحتوي على المستودع الأساسي، عندها يمكن رفع أي نسخة من أي مطور في حال كانت تحتوي على آخر التعديلات التي كانت في المستودع الموجود على الموقع.

من أشهر المواقع التي تقدم لك خدمة إستضافة المستودع الخاص بك هما موقعي GitHub و BitBucket، بحيث يوفران لك مستودعات مجانية وخاصة إن أردت، وقد يتميز BitBucket حتى تاريخ كتابة هذا النص بأنه يوفر مستودعات خاصة بشكل مجاني وهذا ما لا يوفره GitHub، حيث تحتاج إلى الدفع مقابل المستودعات الخاصة أو Private Repositories، والمقصود بها، المستودعات التي لا يراها أي شخص وإنما تحتاج لأذن للوصول لها وهي على عكس المستودعات العامة أو Public Repositories و التي يمكن للجميع رؤيتها و معاينة الشيفرة الخاصة بالمشروع المتواجد في المستودع.

بإختصار، يقوم Git بعمل مستودعات موزعة أو Distributed Repositories، بحيث يملك كل مطور نسخته الخاصة بالإضافة إلى نسخة رئيسية على موقع، و يتم التركيز على تطوير النسخة الرئيسية الموجودة في ذلك الموقع من خلال تصحيح الأخطاء و إضافة الخصائص و غيرها، وفي حال أراد أي مطور الحصول على أحدث نسخة، يقوم بنسخ أو عمل Clone للنسخة الرئيسية.

إضافة Remote Repositories و استخدام git remote add

تستطيع العمل على أكثر من مشروع برمجي، لذا قد تجد بعض المبرمجين يعملون على أكثر من مشروع، والتي بدورها تحتاج إلى أكثر من مستودع، لذا بإمكانك إضافة أي عدد من المستودعات الموجودة على سيرفرات مختلفة، سنتحدث هنا عن كيفية إضافة معلومات Server معين لبدء العمل معه.

تعتبر إضافة Remote Repositories هي عملية إضافة رابط الموقع أو Server URL المتواجد عليه ذلك المستودع وذلك لبدء التعامل معه، و بإمكانك إضافة Remote Repository في Git من خلال `git remote add`، و التي بدورها ستقوم بتخزين مؤشر يشير إلى ذلك المستودع، ومن ثم يمكننا التعامل مع ذلك المؤشر أو الاسم، و لتوضيح الأمر، افرض أننا نريد إضافة مستودع موجود على GitHub ونريد إعطائه اسم (مؤشر) `calc`، فسيكون الأمر كالتالي



```
git remote add calc https://github.com/algorithms/calc
```

لاحظ هنا كيف تم إضافة المستودع الموجود على GitHub و ربطه بالإسم `calc` كإسم مختصر للمستودع نفسه بشكل يدوي. عند العمل مع `clone` فستجد أن GitHub يستخدم الاسم الافتراضي `origin` لتسمية المستودع عندما تقوم بنسخه على جهازك بشكل تلقائي. الآن لو أردنا معرفة المستودعات التي نتعامل معها عن بعد فإمكاننا استخدام الأمر التالي



```
git remote -v
```

سيظهر هذا الأمر قائمة بالمستودعات أو Remote Repositories التي نتعامل معها و أسمائها، بالإضافة إلى الأسماء المختصرة (المؤشرات) التي وضعناها.

بإمكانك الحصول على قائمة بالأسماء المستعارة أو المؤشرات التي تشير لتلك المستودعات بدون التفاصيل الأخرى التي ترافقها، مثل الرابط وغيرها عن طريقة استخدام الأمر التالي



```
git remote
```

سيظهر هذا الأمر قائمة بالمستودعات أو [Remote Repositories](#) التي نتعامل معها و أسمائها، بالإضافة إلى الأسماء المختصرة (المؤشرات) التي وضعناها.

مستودعات الشيفرة الموجودة مسبقاً و مفهوم Cloning

أحياناً تريد التعامل مع مشروع موجود مسبقاً، فمثلاً قد تجد مشروع على GitHub و تريد المشاركة في هذا المشروع، أو تكون أنضمت حديثاً لأحد فرق العمل في شركةٍ ما و تريد بدء العمل معهم، عندها كل ما تحتاجه هو عمل نسخ للمستودع Repository أو ما يسمى بـ Clone، وذلك لكي تحصل على نسختك التي ستعمل عليها.

بلغة بسيطة، هناك مشروع برمجي قائم، و تريد أنت الحصول على نسخة من المستودع الخاص به لتبدأ العمل، لو راجعت الصورة التالية، فستلاحظ وجود سهم يتجه من اليمين ليسار تحت مسمى Checkout The Project، بحيث تتم عملية نسخ المستودع Repository إلى مجلد المطور Working Directory، وهذا ما يحدث بالضبط عندما نقوم بعمل Clone. يستخدم git الأمر clone لنسخ مستودع شيفرة و جلبه إلى Working Directory ليعمل عليه المبرمج كالتالي

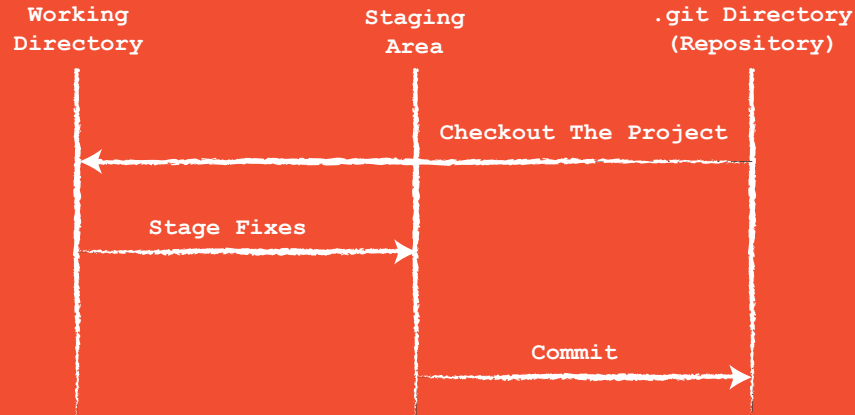


```
git clone https://github.com/algorithms/my.git
```

ببساطة، يتم نسخ المستودع my.git الموجود في الرابط و تحميله لديك، و سيتم إنشاء مجلد بإسم المستودع نفسه بإسم my وبداخله المشروع كامل و كذلك مجلد Git. بإمكانك تحديد اسم خاص بالمجلد إذا لم تكن تريد الافتراضي مثل my من خلال تحديد الاسم في نفس الأمر كالتالي



```
git clone https://github.com/algorithms/my.git proj
```



Git Workflow

لاحظ كلمة `proj` في آخر الأمر، وهي تعني أنه سيتم وضع النسخة في مجلد باسم `proj` عوضاً عن المجلد الافتراضي الذي يكون غالباً باسم المستودع نفسه، و في حالتنا هنا `.my`.

بعد ذلك سيصبح المطور جاهز لبدء العمل على النسخة الخاصة به و اجراء التعديلات، وستتعرف فيما بعد كيف نقوم بالعملية العكسية من خلال رفع التعديلات من مجلد المبرمج إلى المستودع الموجود على الموقع أو الموجودة على السيرفر من خلال التعامل مع أحد الأوامر التي توفرها Git وهو أمر `push`.

جلب بيانات Remote Repositories و استخدام git fetch

بعد إضافة Remote Repository، تحتاج كمبرمج إلى جلب البيانات الموجودة في ذلك المستودع، سواء كانت تلك البيانات بكاملها، أو بيانات جزئية جديدة تمت إضافتها من خلال المطورين الذين يعملون على المشروع، سنتحدث هنا عن `fetch` وكيفية جلب البيانات منها.

يوفر **Git** الأمر `fetch` وذلك لجلب البيانات الموجودة في **Remote Repository** أو المستودع الموجود على موقع أو **Server** ما، و بعد تنفيذ الأمر، سيتم تحميل البيانات أو التعديلات الجديدة التي قام بها المطورون و التي لا تتوفر لديك، وستحصل على جميع التفرعات أو **Branches** الموجودة هناك، والتي بإمكانك عمل دمج أو **Merge** لها مع العمل الخاص بك أو المستودع المتواجد لديك. بإمكانك تنفيذ أمر `fetch` من خلال الصيغة العامة التالية



```
git fetch [remote-name]
```

ببساطة، يحتاج أمر `fetch` إلى اسم المستودع (الاسم المستعار الذي قمنا بإعطائه له عندما قمنا بإضافة المستودع من خلال `git add remote`) لكي يجلب البيانات من ذلك المستودع، وعندما نقول (يجلب) فنحن نقصد ببساطة أنه يقوم بعمل **Download** لمحتويات المستودع الموجود على الموقع أو **Server** على جهاز المطور.

في حال قمت بعمل **clone** لمستودع بعيد، فإن الأمر سيقوم بشكل تلقائي بإضافة **Remote Repository** لذلك المستودع أي الذي قمت بعمل **clone** له بشكل تلقائي تحت اسم **origin**. أي لو أردنا جلب البيانات من ذلك المستودع فسنكتب التالي



```
git fetch origin
```

بهذا الأمر، سيبحث **Git** عن الرابط الخاص بالموقع أو **Server** المرتبط بإسم **origin** و من ثم سيقوم بجلب البيانات منه.

أحد الأمور المهمة التي يجب أن تتنبه لها حول أمر **fetch** هو أنه يقوم فقط بجلب البيانات من السيرفر أو الموقع إلى المستودع الخاص بك أو **Local Repository** ولكن لا يقوم بدمج **Merge** المحتويات الجديدة التي قام بجلبها مع المحتوى الموجود لديك في ذلك المستودع، لذا، يجب عليك عمل الدمج أو **Merge** بشكل يدوي **Manually**.

بإمكانك إختصار خطوتي جلب البيانات **Fetch** ودمجها **Merge** من خلال خطوة واحدة فقط و ذلك بإستخدام أمر **git pull**.

رفع بيانات Local Repositories و استخدام git push

يسمى المستودع الذي تعمل عليه في جهازك بإسم Local Repository، لذا، يقوم المبرمج بإجراء التعديلات على المشروع و حفظها فيه، وعندما يريد رفع التعديلات على Remote Repository فسيستخدم أمر push، سنتحدث هنا عن كيفية رفع التعديلات إلى Remote Repository.

يوفر Git الأمر **push** و ذلك لرفع البيانات الموجودة في **Local Repository** أو المستودع الموجود على جهاز المبرمج، و بعد تنفيذ الأمر، سيتم رفع البيانات أو التعديلات الجديدة التي قام بها المطور إلى مستودع الشيفرة الموجود على السيرفر، و بإمكانك تنفيذ أمر **push** من خلال الصيغة العامة التالية

```
$ git push [remote-name] [branch-name]
```

كما تلاحظ، يحتاج الأمر **push** إلى تحديد **remote-name** وهو الاسم المستعار للمستودع الموجود على السيرفر أو **Remote Repository**، و بما أن كل **Repository** قد يحتوي على أكثر من **Branch**، فهنا نقوم بتحديد **branch-name** الذي نريد رفع البيانات له.

لو قمنا بعمل **clone** كما ذكرنا سابقاً، فبشكل تلقائي سيتم تسمية **Remote Repository** بإسم **origin** و سيكون **Branch** بإسم **master**، لذا، إذا أردنا رفع البيانات أو التعديلات على ما تم ذكره حتى الآن، فسيكون الأمر على الشكل التالي



```
git push origin master
```

ما يظهر في الأمر ببساطة هو : قم برفع البيانات للمستودع **origin** و التفرع **master**.

يجب الإنتباه

لديك صلاحية الكتابة أو ما تسمى بـ `Write Access`، فهناك مستودعات تستطيع جلب المشروع منها لكن لا يحق لك رفع التعديلات التي قمت بها، و هذا النوع من المشاريع قد تجلب النسخة للتعديل عليها لأعمال خاصة بك مثلاً أو لتطوير نسخة معينة من النظام لتتوافق مع إحتياجاتك.

نقطة أخرى مهمة، وهي في حال قام أحد أعضاء الفريق بعمل `clone` معك و من ثم قام برفع التعديلات على المستودع الموجود على الموقع من خلال `push`، فعندها إذا قمت بعمل `push` فسيتم رفضه `Reject`، وذلك لأنه يجب عليك جلب `fetch` التعديلات التي قام بها المطور على آخر نسخة موجودة على المستودع الموجود على الموقع، و من ثم دمجها مع المستودع الموجود لديك و تنسيقها مع التعديلات التي قمت بها، وبعدها بإمكانك عمل `push`.

الحصول على تفاصيل Remote Repository و git remote show

أحياناً يحتاج المطور إلى معرفة تفاصيل أكثر عن مستودع الشيفرة الذي يعمل عليه، وتحديدًا مستودع الشيفرة الموجود على السيرفر، يوفر Git آلية بسيطة لإعطاء تفاصيل أكثر عن Remote Repository، وهذا ما سنتحدث عنه هنا.

لمعرفة تفاصيل أكثر حول Remote Repository معين، فبإمكانك استخدام الأمر `git remote show`، والذي من خلاله تستطيع عرض تفاصيل عن المستودع مثل قائمة الروابط `URLs` و `Remote Branches` وغيرها من المعلومات الأخرى التي ستساعدك على إتخاذ إجراء معين، وفيما يلي الشكل العام للأمر.



```
git remote show [remote-name]
```

لاحظ هنا أن الأمر يطلب منك الإسم المستعار للمستودع، وبما أن لدينا مستودع سابق قمنا بالتعامل معه وهو `origin`، عندها سيصبح الأمر كالتالي



```
git remote show origin
```

تنفيذ هذا الأمر سيعرض الكثير من التفاصيل عن مستودع الشيفرة المسمى `origin`، ولمعرفة التفاصيل و كيف يتم عرضها، لاحظ المحتوى الموجود في الصفحة التالية. لاحظ كمية المعلومات و التفاصيل التي تم عرضها، فمثلاً توضح التفاصيل أن مؤشر `HEAD` يشير حالياً إلى `master`، بالإضافة إلى قائمة بـ `Bmahces` الموجودة مثل `master` و `dev-branch`، وأيضاً هناك تفاصيل أخرى لم تظهر هنا وهي تظهر حسب نوع المستودع، فمثلاً قد ترى مستودعات الشيفرة الغير موجود لديك وهكذا.



```
git remote show origin
```

```
* remote origin
```

```
Fetch URL: https://github.com/algorithmers/my
```

```
Push URL: https://github.com/algorithmers/my
```

```
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)
```

إعادة تسمية Rename أو حذف Remove الاسم المختصر لـ Remote Repository

تعرفنا سابقاً على كيفية إضافة Remote Repository وكيفية وضع اسم مستعار له، بالإضافة إلى ذلك، تستطيع من خلال Git أن تعيد تسمية الاسم المستعار بالإضافة إلى إمكانية حذفه بشكل كلي، سنتحدث هنا عن هذان الأمران.

تستطيع من خلال **Git** إعادة تسمية **Rename** الاسم المختصر الذي قمت بإضافته لمستودع شيفرة موجود على **Server** و ذلك من خلال استخدام الأمر **remote rename** وهذا الأمر قد يكون مفيد في حال الخطأ في التسمية أو في حال أراد المطور أن يعطي اسماً أكثر دقة، و الصيغة العامة له كما يلي



```
git remote rename [old-remote-name] [new-remote-name]
```

على سبيل المثال، ماذا لو كان لدينا مستودع شيفرة على **Server** و قمنا بتسميته بالإسم المستعار **dev** و أردنا فيما بعد أن نعيد تسميته إلى **devrepo**، عندها سنقوم بتنفيذ الأمر كما يلي



```
git remote rename dev devrepo
```

لاحظ كيف جاء الاسم القديم أولاً ثم أتى بعده الاسم الجديد، وبعد تنفيذ هذا الأمر سيتغير اسم المؤشر للمستودع و سيتم التعامل معه في بقية الأوامر فيما بعد مثل **push** على أنه **devrepo** و ليس **dev**.

نوه هنا إلى أن هذا التغيير سيؤثر على أسماء التفرعات **Branching** أيضاً، فلو أخذنا **master branch** على سبيل المثال، فنقول أنه قبل التغيير كان يشار له على أنه **dev/master**، سنجد أنه بعد التغيير سيشار له على أنه **devrepo/master**.

نأتي الآن إلى نقطة أخرى، وهي في حال أردت حذف المستودع البعيد من جهازك، والأسباب كثيرة لمثل هذا الأمر، فقد يكون أحد الأسباب هو أنه تم تغيير السيرفر الذي يتواجد عليه المشروع الرئيسي، أو أنه لم يعد أحد من المطورين المتبرعين **Contributors** يهتم بإضافة شيء للمشروع، أو لأي سبب آخر، فإن **Git** في هذه الحالة يوفر لك الأمر **remote rm** والذي من خلاله تستطيع إجراء عملية الحذف، ولو أردنا الآن حذف المؤشر **devrepo** الذي قمنا بإعادة تسميته فيما سبق، فيمكننا تنفيذ ذلك من خلال ما يلي



```
git remote rm devrepo
```

بعد تنفيذ هذا الأمر، نقول أنه تم حذف المؤشر، ولن نستطيع تنفيذ الأوامر الأخرى عليه مثل **push**، وهنا ننوه أن حذف المؤشر لا تتأثر به مؤشرات المستودعات الأخرى لديك كونها منعزلة تماماً.

إختصار الأوامر و التعامل مع Git Aliases

عند العمل على Git لفترات طويلة، سيجد المبرمج نفسه أمام خطوات مكررة أو أوامر مكررة يقوم بها من مشروع لآخر أو في المشروع نفسه، و قد تكون هذه العملية مملة لبعض المبرمجين خصوصاً مع الأوامر الطويلة، لذا توفر لك Git آلية بسيطة لإختصار تلك الأوامر، سنتحدث هنا عن هذا الأمر.

يفضل العديد من المطورين التعامل مع الإختصارات، لذا تجد بعض المبرمجين قد لا يستخدمون الفأرة **Mouse** أثناء البرمجة و كتابة الشيفرات بل يركزون على الوصول للأشياء من خلال لوحة المفاتيح **Keyboard** مثل النسخ و اللصق والإختصارات البرمجية وذلك كون تلك العمليات تتكرر باستمرار من فترة لفترة. ستواجه هذا الأمر كثيراً في **Git**، فستجد نفسك أمام عدد من الأوامر التي تتكرر باستمرار و ستحتاج حينها لإيجاد طريقة تتجاوز بها هذا التكرار.

توفر لك **Git** آلية تسمى **Git Aliases** وهي عبارة عن طريقة لوضع أسماء مستعارة أو مختصرة لأوامر كاملة أو إختصار لجزء معين من أمر وذلك من خلال الأمر **git config**. لتوضيح الأمر، تخيل أننا نريد إختصار الأمر **status** إلى حرفين فقط هما **st**، عندها يمكننا فعل ذلك من خلال تنفيذ الأمر التالي



```
git config --global alias.st status
```

الآن في حال أردنا معرفة حالة المستودع، يمكننا إختصار الأمر **git status** إلى ما يلي



```
git st
```

بكل بساطة إذا لم يجد **git** الأمر **st** لديه فسيبحث عنه في الإختصارات، وكما كتبنا سابقاً **alias.st**، فسيصبح هذا الأمر إختصاراً. قد لا تظهر الفائدة من الإختصار هنا، لكنها ستكون ذات فائدة عند التعامل مع الأوامر الطويلة و المتكررة بشكل مستمر. يفيد استخدام **Aliases** أيضاً في الإختصار بطريقة تمكّنك من صناعة أوامر خاصة فيك تظن أنه من المناسب وجودها في **Git**.

لتوضيح هذا الأمر لاحظ ما يلي



```
git log --pretty=oneline
```

بإمكاننا الآن جعل كامل هذا السطر على أنه أمر بسيط في Git كالتالي



```
git config --global alias.oneline 'log --pretty=oneline'
```

الآن بإمكاننا تنفيذ الأمر السابق كالتالي



```
git oneline
```

git stash و حفظ حالة المستودع Repository الحالية

بما أن المبرمجين يعملون على أكثر من Branch في المستودع غالباً، فمن الطبيعي أن ينتقل المبرمج ما بين Branch و آخر، ولكن المشكلة تكمن في كون المبرمج أحياناً يريد أن ينتقل من Branch إلى آخر دون أن يقوم بعمل Commit للتعديلات الحالية، هذا ما أتى من أجله Stash.

عندما تعمل على Branch معين، فالملفات التي تتواجد فيه قد تكون بحالات مختلفة في نفس الوقت، فقد تكون هناك ملفات **Modified** و أخرى **Staged** وغيرها من الحالات، لذا، أثناء التطوير قد تمر بحالة، لا يكون فيها المستودع جاهز لعمل **Commit** عليه لكون العمل الذي تقوم به في المنتصف مثلاً و لم يكتمل، وتحتاج إلى أن تنتقل إلى Branch آخر قبل أن تكمل العمل السابق، وهنا ليس من الجيد عمل **Commit**، لذا سنقوم بعمل **Stash** وذلك لحفظ حالة التفرع **Branch** على ماهي عليه حتى تعود إليها مرة أخرى و تكمل العمل دون أن تحفظ أي **Commit**.

كل ما يتم عمله هو حفظ الحالة، أي كأنك تقول، قم بحفظ حالة **Branch** الحالي لكي أعود لها في أي وقت. لتوضيح الأمر، تخيل أنك قمت بعمل بعض التعديلات على ملف معين ولم تنتهي من التعديلات، و أردت الانتقال لتفرع آخر لمعالجة أمر طارئ، يمكننا قبل ذلك القيام بالأمر التالي



git stash

الآن قمت بحفظ الوضع الحالي لنعود له في أي وقت، ولو قمت بعمل **git status** فستجد المجلد الذي تعمل عليه **clean** ولا يوجد تعديلات لكي تقوم بعمل **Commit** لها. بإمكانك الآن الانتقال لأي **Branch** وبدء العمل.

عندما تقوم بعمل **Stash** فأنت تقوم بتخزين الحالة في **Stack** ولهذا بإمكانك تخزين أكثر من حالة، لذا بإمكانك استخدام **stash list** لمعرفة قائمة الحالات التي قمت بتخزينها لكي تساعدك في الرجوع للحالة التي تريدها، وبإمكانك استخدام الأمر كالتالي



git stash list

هنا ستظهر قائمة بالحالات التي قمت بتخزينها من قبل و بإمكانك الرجوع لأي منها، أي عمل **Reapply** من خلال ما يلي



```
git stash apply
```

في هذه الحالة سيقوم **Git** بجلب أحدث حالة قمت بتخزينها، و إذا أردت العودة لأحد الحالات المخزنة مسبقاً، فبإمكانك استخدام الاسم الذي يظهر مع تلك الحالة عند القيام بتنفيذ أمر **stash list**، و للعودة مثلاً لحالة معينة نقوم بتنفيذ التالي من خلال استخدام اسم الحالة.



```
git stash apply stash@{2}
```

git clean و تنظيف مجلد المشروع Working Directory

قد تحتاج أحياناً إلى إزالة جميع الملفات و المجلدات التي تكون حالتها Untracked في مشروعك، أي الملفات الموجودة في مجلد المشروع و لا يتم إستخدامها في أي أمر يخص المشروع. سنتحدث هنا عن كيفية التعامل مع هذا الأمر في Git.

يوفر لك Git الأمر **clean** والذي بدوره يساعدك على تنظيف و إزالة الملفات أو المجلدات الزائدة أو التي لا تحتاج إليها، و المقصود هنا، هو الملفات أو المجلدات التي تكون حالتها **Untracked** و غير موجودة في ملف **gitignore**. ولتنفيذ الأمر، قم بكتابة السطر التالي



```
git clean -f -d
```

بعد تنفيذ هذا الأمر سيتم حذف الملفات و كذلك سيتم حذف الملفات من المجلدات الفرعية، و بالنسبة للخيار **-f**، فهو يعني **force**، أي "really do this"، وهو لتأكيد عملية الحذف. أما بالنسبة للخيار **-d**، فيقصد به حذف الملفات و المجلدات التي تكون **Untracked**.

ينبغي التنبيه هنا إلى أن هذا الأمر خطير للغاية و قد يتسبب في ضياع الكثير من الملفات و المجلدات، و ينبغي التعامل معه بحذر. لهذا السبب يوفر لك Git خيار إضافي هو **-n**، بحيث يعطيك صورة عن ما سيتم حذفه فعلياً قبل حذفه بشكل فعلي، لذا دعنا ننفذ ما سبق و نستبدل **-n** مكان **-f** كما يوضح الأمر التالي



```
git clean -n -d
```

بهذا الأمر، وكأنك تقول: أرني ماذا ستفعل أو ماهي الملفات و المجلدات التي ستقوم بحذفها، و أنتبه هنا إلى أن **-f** هو من يقوم بفرض عملية الحذف.

ذكرنا أن `git clean` يحذف الملفات و المجلدات التي تكون حالتها `Untracked` وغير موجودة في `gitignore`، لكن هناك إستثناء يمكن أن تقوم به، وهو في حال أردت حذف الملفات و المجلدات الموجودة أيضاً في `gitignore`، فعندها بإمكانك استخدام الخيار `-x` مع الأمر كما يلي



```
git clean -f -d -x
```

أو للتحقق مما سيتم حذفه قبل حذفه بشكل فعلي كما يلي



```
git clean -n -d -x
```

من الأساليب التي تتيحها `Git` أيضاً، التنظيف و الحذف من خلال الأسلوب التفاعلي، بحيث يمكنك تنفيذ هذا الأمر من خلال الخيار `-i` كما يلي



```
git clean -x -i
```

انتظر المزيد في الإصدار ١.١

www.algothmers.com