.1 الإستعداد للبدء! - إدارة الإصدارات (Version Control)

**إدارة الإصدارات (Version Control)**

ما هو نظام إدارة الإصدارات، ولماذا عليك أن تهتم به؟ تمكنك هذه الأنظمة من تسحيل التغيرات التي تحدث على ملف أو مجموعة ملفات خلال الزمن بحيث يمكنك الرجوع الى مرحلة معينة (إصدار) لاحقاً.

فعلى سبيل المثال، ستستخدم في هذا الكتاب الكود المصدري للملفات في حين أنه تتم إدارتها من قبل نظام إدارة الإصدارات، ولكن في الحقيقة يمكنك استخدام هذه الأنظمة مع أي نوع من الملفات على حاسوبك.

إذا كنت تعمل كمصمم غرافيك أو ويب وتريد طريقة لمتابعة جميع الإصدارت والتعديلات التي تجريها على صورة أو قالب ما (الأمر الذي سيعجبك بالتأكيد)، فإن نظام إدارة الإصدارات (VCS) هو الحل الأمثل.

حيث يمكنك من إرجاع الملفات الى حالة كانت عليها سابقاً، أو ارجاع المشروع بأكمله لحالة سابقة، يمكنك أيضاً مقارنة التغيرات الحاصلة مع مرور الزمن، أو معرفة من قام بتعديل معين أدى الى خطأ ما، من قام بتقديم إقتراح ومتى قام بذلك، والمزيد.

إستخدامك لنظام إدارة الإصدارات يعني أيضاً أنه اذا قمت بخطأ ما في مرحلة من المراحل أو خسرت ملفات المشروع لسبب ما، يمكنك استرجاعها الى حالها بسهولة. كل هذه الميزات مقابل تعب خفيف منك.

[**أنظمة إدارة الإصدارات المحلية**](https://git-scm.com/book/ar/v1/%D8%A7%D9%84%D8%A5%D8%B3%D8%AA%D8%B9%D8%AF%D8%A7%D8%AF-%D9%84%D9%84%D8%A8%D8%AF%D8%A1%21-%D8%A5%D8%AF%D8%A7%D8%B1%D8%A9-%D8%A7%D9%84%D8%A5%D8%B5%D8%AF%D8%A7%D8%B1%D8%A7%D8%AA-Version-Control#%D8%A3%D9%86%D8%B8%D9%85%D8%A9-%D8%A5%D8%AF%D8%A7%D8%B1%D8%A9-%D8%A7%D9%84%D8%A5%D8%B5%D8%AF%D8%A7%D8%B1%D8%A7%D8%AA-%D8%A7%D9%84%D9%85%D8%AD%D9%84%D9%8A%D8%A9)

تقوم الطريقة التي يقوم بها معظم المستخدمين لإدارة إصداراة مشاريعهم على "نسخ الملفات" الى مكان آخر. يقوم المعظم بهذه الطريقة لأنها تبدوا وكأنها الحل الأسهل، ولكنها تجلب المشاكل والأخطاء بشكل لا يحتمل أيضاً. من السهل جداً أن تنسى بأي مجلد وضعت نسخة معينة أو أن تقوم بالتغيير أو الحذف عن طريق الخطا لملف ما.

لعلاج هذه المشكلة، قام المبرمجون بتطوير أنظمة إدارة إصدارات المحلية، حيث تستخدم قاعدة بيانات بسيطة تحفظ فيها التغيرات على الملفات في نظام إصدارات معين (إنظر الشكل 1-1).



الشكل 1-1. مخطط أنظمة إدارة الإصدارات المحلية.

أحد أشهر أنظمة إدارة الإصدارات هو نظام يدعى rcs، والذي مازال يتم توزيعه مع العديد من الحواسيب في يومنا هذا. حتى أن أنظمة Mac OS X الشهيرة تحوي نظام rcs مضمنة في حزمة برامج التطوير Developer Tools. يقوم عمل هذه الأداة ببساطة على حفظ مجموعات من الإصلاحات (Patch sets) (أي فروقات بين الملفات بمعنى آخر) من تغيير الى آخر بطريقة خاصة; يمكنها بعد ذلك اعادة تشكيل أي ملف بالطريقة التي كان عليها في أي مرحلة خلال حياة الملف عن طريق اضافة جميع هذه التغيرات.

[**أنظمة إدارة الإصدارات المركزية**](https://git-scm.com/book/ar/v1/%D8%A7%D9%84%D8%A5%D8%B3%D8%AA%D8%B9%D8%AF%D8%A7%D8%AF-%D9%84%D9%84%D8%A8%D8%AF%D8%A1%21-%D8%A5%D8%AF%D8%A7%D8%B1%D8%A9-%D8%A7%D9%84%D8%A5%D8%B5%D8%AF%D8%A7%D8%B1%D8%A7%D8%AA-Version-Control#%D8%A3%D9%86%D8%B8%D9%85%D8%A9-%D8%A5%D8%AF%D8%A7%D8%B1%D8%A9-%D8%A7%D9%84%D8%A5%D8%B5%D8%AF%D8%A7%D8%B1%D8%A7%D8%AA-%D8%A7%D9%84%D9%85%D8%B1%D9%83%D8%B2%D9%8A%D8%A9)

المشكلة التالية التي ظهرت هي الرغبة في التعاون والتشارك بين المطورين الذين يعملون على أنظمة أخرى.

لحل هذه المشكلة تم إنشاء أنظمة إدارة الإصدارات المركزية Centralized Version Control Systems. تقوم هذه الأنظمة، مثل CVS, Subversion و Perforce على مخدم Server واحد يحتوي على جميع الملفات، وعدد من المستخدمين Clients تقوم بطلب هذه الملفات من مكان وجودها المركزي. للعديد من السنوات، كانت هذه الأنظمة هي المسيطرة على عالم إدارة الإصدارات (انظر الشكل 1-2).



الشكل 1-2. مخطط أنظمة إدارة الإصدارات المركزية.

تقدم هذه الطريقة العديد من الأفضليات على أنظمة ادارة الإصدارات المحلية. فعلى سبيل المثال، جميع المشاركين في المشروع يعرف مالذي يقوم به المشارك الآخر الى حد معين. مدراء المشروع يستطيعون التحكم بمن يستطيع فعل ماذا في النظام العام; وبالطبع فإنه من الأسهل التعامل مع أنظمة إدارة الإصدارات المركزية على التعامل مع الأنظمة المحلية وقاعدات بياناتها من قبل كل مستخدم.

ولكن، ومن جهة ثانية، فإن لهذه الطريقة جانباً سيئاً أيضاً.

أهم هذه النقاط السيئة هي أنه هذه الأنظمة المركزية تقوم على مركز واحد للكود، أي إنه اذا حصل وتوقف المخدم لساعة، فلن يتمكن أحد من خفظ التغييرات أو القيام بأي تعديل على أي شيء يعمل عليه.

إذا حصل وأن تعطل القرص الأساسي والذي يحوي على قاعدة البيانات المركزية، ولم تعمل النسخ الإحتياطية المخبأة، فإنك ستخسر كل شي عن تاريخ عملك في المشروع.

تعاني أنظمة إدارة الإصدارات المحلية من هذه المشكلة أيضاً، وهي أنه عندما تكون جميع ملفات المشروع في مكان واحد فإنك على خطر من خساراة كل شيء!

[**أنظمة إدارة الإصدارات الموزعة**](https://git-scm.com/book/ar/v1/%D8%A7%D9%84%D8%A5%D8%B3%D8%AA%D8%B9%D8%AF%D8%A7%D8%AF-%D9%84%D9%84%D8%A8%D8%AF%D8%A1%21-%D8%A5%D8%AF%D8%A7%D8%B1%D8%A9-%D8%A7%D9%84%D8%A5%D8%B5%D8%AF%D8%A7%D8%B1%D8%A7%D8%AA-Version-Control#%D8%A3%D9%86%D8%B8%D9%85%D8%A9-%D8%A5%D8%AF%D8%A7%D8%B1%D8%A9-%D8%A7%D9%84%D8%A5%D8%B5%D8%AF%D8%A7%D8%B1%D8%A7%D8%AA-%D8%A7%D9%84%D9%85%D9%88%D8%B2%D8%B9%D8%A9)

وهنا تأتي أنظمة إدارة الإصدارات الموزعة لحل المشكلة. في هذه الأنظمة (مثل Git, Mercurial, Bazaar أو Darcs) يحصل المستخدمون ليس فقط على آخر نسخة من الملفات الموجود على المخدم، بل على كامل تاريخ النظام.

في هذه الحال، إن تعطل النظام، فإنه يمكن الحصول على نسخة من المشروع من أي مستخدم الى المخدم مرة أخرى.

أي أن كل عملية طلب للكود هي في الحقيقة عملية حفظ كاملة وشاملة لتاريخ المشروع (إنظ الشكل 1-3).



الشكل 1-3. مخطط أنظمة إدارة الإصدارات الموزعة.

وفوق هذا فإن معظم هذه الأنظمة يتعامل بشكل جيد جداً مع أكثر من نسخة خارجية للمشروع، أي يمكنك التعاون مع أكثر من مجموعة مختلفة من الأشخاص في طرق مختلفة وفي وقت واحد وعلى مشروع واحد. يمكنك هذا من تطوير أكثر من طريقة عمل واحدة مناسبة لك، الأمر الذي لم يكن متاحاً مع أنظمة إدارة الإصدارات المركزية

# 2 الإستعداد للبدء! - لمحة تاريخية عن Git

## لمحة تاريخية عن Git

كما تبدأ العديد من الأشياء الجميلة في الحياة، بدأت Git كنوع من التدمير المبدع المثير للجدل. الـ Linux Kernel المعروف هو برنامج مفتوح المصدر ذو إطار واسع نوعاً ما. طوال حياة هذا المشروع (من 1991-2002)، كان يتم تناقل التعديلات على شكل ملفات إصلاح و ملفات مؤرشفة (مضغوطة). في 2002، في 2002 بدأ المشروع باستخدام نظام إدارة إصارات موزعة DVCS يدعى BitKeeper .

في 2005، بدأت العلاقة بالإنهيار بين المجتمع المطور للـ Linux Kernel والمجتمع التجاري المطور لـ BitKeeper، وتم العدول عن توفير البرنامج بشكل مجاني.

أثار هذا التغيير الرغبة في المجتمع المطور لـ Linux (وبالتحديد لينوس تورفالدوس، منشيء Linux) الى تطوير نظامهم الخاص بناء على الدروس التي تعلموها عند استخدامهم لـ BitKeeper. وتم وضع أهداف لكي يحققها النظام الجديد مشمولة بـ:

* السرعة
* التصميم البسيط
* الدعم القوي للبرمجة الغير خطية (الكثير من أشجار التطوير الفرعي)
* التوزيع بشكل كامل
* القدرة على تحمل مشروعات ضخمة مثل الـ Linux Kernel بشكل فعال (السرعة وحجم المعلومات)

منذ ولادة Git في 2005، تطورت لكي تصبح ناضجة مع المحافظة على السهولة والمبادئ الأساسية التي وضعت عليها. حيث أنها سريعة بشكل لايصدق، و فعالة مع المشاريع الكبيرة، وتحوي على نظام تشجير ممتاز لدعم التطوير الغير خطي (انظر الفصل 3).

3 الإستعداد للبدء! - أوليات Git

**أوليات Git**

لنتحدث الآن عن Git بشكل سريع. هذا القسم هام جداً لك، لأنك إذا عرفت ماهي Git وماهيا أوليات عملها، سيكون من السهل عليك استخدام Git بشكل أسهل وأفضل. وخلال تعلمك لـ Git حاول أن تفرغ ذهنك من المعلومات السابقة عن أنظمة إدارة الإصدارات الأخرى، مثل Subversion أو Perforce; سيجنبك هذا من الخلط بين المعلومات عند استخدام هذه الأداة.

تقوم Git بالتعامل وتخزين المعلومات بشكل مختلف تماماً عن الأنظمة الأخرى، وبالرغم من أن واجهة الإستخدام بسيطة نسبياً; سيكون فهمك لهذه الإختلافات سيبعد عنك الحيرة عند الإستخدام.

[**لفطات، وليست تغيرات**](https://git-scm.com/book/ar/v1/%D8%A7%D9%84%D8%A5%D8%B3%D8%AA%D8%B9%D8%AF%D8%A7%D8%AF-%D9%84%D9%84%D8%A8%D8%AF%D8%A1%21-%D8%A3%D9%88%D9%84%D9%8A%D8%A7%D8%AA-Git#%D9%84%D9%81%D8%B7%D8%A7%D8%AA%D8%8C-%D9%88%D9%84%D9%8A%D8%B3%D8%AA-%D8%AA%D8%BA%D9%8A%D8%B1%D8%A7%D8%AA) **Snapshots, Not Differences**

الفرق الرئيسي بين Git وأي نظام إدارة إصدارات آخر (Subversion وأصدقاءه) هو الطريقة التي تتعامل بها Git مع المعلومات.

تقوم معظم هذه الأنظمة بتخزين المعلومات كقائمة من التغيرات القائمة على الملفات. هذه الأنظمة (مثل CVS، Subversion، Perforce، Bazaar وغيرها) تتعامل مع المعلومات التي تحفظها كمجموعة ملفات والتغيرات القائمة عليها مع مرور الوقت، كما هو موضح في الشكل 1-4.



الشكل 1-4. الأنظمة الاخرى تقوم بحفظ معلومات التغيرات الحاصلة على كل ملف وكأنها الإصدار الأول.

الأمر مختلف مع Git. حيث تعامل Git المعلومات المخزنة على أنها "لقطات" من نظام ملفات مصغر.

ففي كل مرة تقوم بـ Commit أو تحفظ حالة المشروع، تقوم Git بأخذ صورة عن جميع الملفات في تلك اللحظة وتخزن رابطاً الى تلك اللقطة.

ومن أجل السرعة، إذا لم يتغير الملف، لا تقوم Git بحفظة، بل تحفظ رابطاً عن الملف الأسبق منه المطابق. تتعامل Git مع المعلومات كما هو موضح في الشكل 1-5.



الشكل 1-5. تحفظ Git المعلومات على أنها "لفطات".

وبالطبع، هذا الإختلاف بين Git وبين جميع أنظمة إدارة الإصدارات الأخرى تقريباً. يجعل هذا Git أشبه بنظام ملفات مصغر مزود بأدوات خارقة مبنية عليه، عوضاً عن نظام إدارة إصدارات عادي. سنقوم بشرح الفوائد التي تحصل عليها عندما تتعامل مع المعلومات بهذه الطريقة في الفصل الثالث "التشعيب في Git".

[**جميع العمليات هي عمليات داخلية تقريباً**](https://git-scm.com/book/ar/v1/%D8%A7%D9%84%D8%A5%D8%B3%D8%AA%D8%B9%D8%AF%D8%A7%D8%AF-%D9%84%D9%84%D8%A8%D8%AF%D8%A1%21-%D8%A3%D9%88%D9%84%D9%8A%D8%A7%D8%AA-Git#%D8%AC%D9%85%D9%8A%D8%B9-%D8%A7%D9%84%D8%B9%D9%85%D9%84%D9%8A%D8%A7%D8%AA-%D9%87%D9%8A-%D8%B9%D9%85%D9%84%D9%8A%D8%A7%D8%AA-%D8%AF%D8%A7%D8%AE%D9%84%D9%8A%D8%A9-%D8%AA%D9%82%D8%B1%D9%8A%D8%A8%D8%A7%D9%8B)

أغلب العمليات في Git لا تحتاج سوى ملفات ومصادر داخلية لتعمل - أي بشكل عام، لا توجد معلومات تحتاجها من حاسوب آخر من الشبكة. إذا كنت قد استخدمت أحد أنظمة إداراة الإصدارات الأخرى والتي تعتمد عملياتها بشكل كبير على الشبكة فإن هذا سيجعلك تعتقد أن آلهة السرعة قد باركت Git وأعطتها هذه السرعة الجبارة.

ستجد أنك تملك كامل تاريخ مشروعك موجود أمامك مباشرة، وأغلب العمليات أقرب ما تكون الى الآنية.

فعلى سبيل المثال، لكي تستعرض تاريخ مشروعك، لن تحتاج Git الى الذهاب الى مخدم معين للحصول عليه وعرضه، بل تقوم بقراءته مباشرة من حاسوبك. اي انك ستحصل على تاريخ مشروعك بشكل مباشر. اذا أردت استعراض التغيرات الحاصلة بين الإصدار الحالي لملف ما واصدار الشهر السابق، تستطيع Git النظر الى الملف من الشهر السابق وحساب الفروقات بينهما مباشرة، عوضاً عن طلب هذه المعلومات من مخدم خارجي.

هذا يعني أيضاً أن الأمور التي لا تستطيع فعلها عندما تكون مفصولاً عن الإنترنت أو عن الشبكة الداخلية قليلة جداً.

إذا كنت في طائرة أو في القطار ولم تستطيع الإرتباط بالشبكة بشكل صحيح، يمكنك إكمال عملك بشكل طبيعي. في الأنظمة الأخرى يكون هذا الأمر مستحيلاً! أو من الممكن لك أن تعدل الملفات ولكن دون عمليات Commit لتغييراتك. في نظام Perforce على سبيل المثال، لا يمكنك فعل الكثير حينما تكون مفصولاً عن المخدم الأساسي; وفي Subversion أو CVS، يمكنك التعديل على الملفات لكن بدون عمليات Commit لتعديلاتك، قد تعتقد بأن هذا ليس بالأمر الكبير، ولكن ستتفاجئ بحجم الفرق الذي يصنعه.

[**Git تحمل المصداقية**](https://git-scm.com/book/ar/v1/%D8%A7%D9%84%D8%A5%D8%B3%D8%AA%D8%B9%D8%AF%D8%A7%D8%AF-%D9%84%D9%84%D8%A8%D8%AF%D8%A1%21-%D8%A3%D9%88%D9%84%D9%8A%D8%A7%D8%AA-Git#Git-%D8%AA%D8%AD%D9%85%D9%84-%D8%A7%D9%84%D9%85%D8%B5%D8%AF%D8%A7%D9%82%D9%8A%D8%A9)

كل شيء في Git سيتم وضع Check-sum عليه قبل تخزينة حيث ستتم الإشارة اليه لاحقاً بهذه الـ checksum. هذا يعني أنه من المستحيل أن يتم تغيير أي ملف أو مجلد دون أن يتم اعلام Git بهذا التغيير. تم بناء هذه الخاصية بشكل أساسي في فلسفة وطريقة عمل Git. من المستحيل أن تخسر معلوماتك بشكل فجائي أو أن ينعطب معلف ما دون أن تعرف Git بذلك.

تدعى الآلية التي تستخدمها Git لعملية الـ checksum هذه باسم SHA-1 hash. وهي عبارة عن قيمة نصية من 40 محرف ستاعشري (0-9 و a-f) ويتم حسابها بناء على محتوى الملف أو المجلد الذي تتبعه Git. يبدو الـ hash من نوع SHA-1 كما يلي:

24b9da6552252987aa493b52f8696cd6d3b00373

سترى أن هذه القيم ستكون موجودة في كل مكان في Git ذلك لأنها تستخدمها بشكل كثيف. في الحقيقة، تستخدم Git هذه القيم للدلالة على الملفات في قاعدة بياناتها.

[**Git تضيف المعلومات فقط!**](https://git-scm.com/book/ar/v1/%D8%A7%D9%84%D8%A5%D8%B3%D8%AA%D8%B9%D8%AF%D8%A7%D8%AF-%D9%84%D9%84%D8%A8%D8%AF%D8%A1%21-%D8%A3%D9%88%D9%84%D9%8A%D8%A7%D8%AA-Git#Git-%D8%AA%D8%B6%D9%8A%D9%81-%D8%A7%D9%84%D9%85%D8%B9%D9%84%D9%88%D9%85%D8%A7%D8%AA-%D9%81%D9%82%D8%B7!)

عندما تقوم بعملياتك في Git، ستقوم Git في أغلب الأحيان بإضافة معلومات فقط الى قاعدة البيانات. ومن الصعب جداً القيام بعملية بحيث لا يمكنك العودة الى الوراء فيها أو محيها من قاعدة البيانات بشكل نهائي. ولكن، وكما هو الأمر في أغلب نظم إدارة الإصدارات، من الممكن أن تخسر المعلومات قبل القيام بعملية Commit للتغيرات الحاصلة; ولكن بعد عملية الـ commit في Git، من الصعب جداً خسارة المعلومات، وخاصة اذا كنت تقوم بعملية push للتعديلات من قاعدة بياناتك الى repository اخرى بشكل منظم.

سيضيف هذا الكثير من المتعة لإستخدامك لـ Git، لأننا نعرف أنه بإمكاننا القيام بالتجارب دون خطر تعريض المعلومات للخطر أو الضياع. للحصول على معلومات معمقة أكثر عن كيفية حفظ Git للمعلومات وكيفية استرجاعها انظر فقرة "Under the Cover" في الفصل التاسع.

[**الحالات الثلاثة**](https://git-scm.com/book/ar/v1/%D8%A7%D9%84%D8%A5%D8%B3%D8%AA%D8%B9%D8%AF%D8%A7%D8%AF-%D9%84%D9%84%D8%A8%D8%AF%D8%A1%21-%D8%A3%D9%88%D9%84%D9%8A%D8%A7%D8%AA-Git#%D8%A7%D9%84%D8%AD%D8%A7%D9%84%D8%A7%D8%AA-%D8%A7%D9%84%D8%AB%D9%84%D8%A7%D8%AB%D8%A9)

والآن عليك الإنتباه قليلاً. الأمر الأساسي الذي عليك أن تتذكره في Git اذا أردت أن تكمك تعلمك بسلاسة هو أنه في Git توجد ثلاث حالات أساسية يمكن للملفاتك أن تكون عليها: commited ، معدلة أو مهيئة للعملية commit (أو staged). نعني بـ commitedf بأن المعلومات تم تخزينها بآمان في قاعدة بياناتك الخاصة. "معدل" تعني أنك قمت ببعض التعديلات على الملف ولكنك لم تقبل بعملية Commit عليه بعد. أما "مهيأ (Staged)" فتعني بأنك حددت ملفاً قمت بتعديله بإصداره الحالية ليتم حفظ في عملية الـ commit التي ستقوم بها.

يقودنا هذا الى الحديث عن الأقسام الثلاثة الرئيسية في أي مشروع Git: المجلد الخاص بـ Git، مجلد العمل و مكان التهييئ (staging area).



الشكل 1-6. المجلد الخاص بـ Git، مجلد العمل و مكان التهييئ (staging area).

المجلد الخاص بـ Git هو المكان الذي تحفظ فيه Git المعلومات الخاصة بها عن مشروعك. هذا هو المكان الأكثر أهمية في Git، وهو الذي يتم نسخه عندما تنسخ الـ repository من حاسوب آخر.

مجلد العمل هو النسخة الحالية المسحوبة من مشروعك. الملفات التي سيتم سحبها من قاعدة بيانات Git للمشروع وتجهيزها لك لتقوم بتعديلها.

مكان التهييئ (staging area)، عادة ما تكون موجودة في مجلد Git لمشروعك، وتحتوي على المعلومات التي سيتم عملية commit عليها. قد يشار الى هذا المكان أيضاً باسم الفهرس (index).

خطوات العمل الإعتيادية في Git غالباً ما تكون كالتالي:

1. تقوم بالتعديلات على الملفات في مجلد العمل.
2. تقوم بوضع هذه الملفات في مكان التهييئ (staging area)، حيث تقوم بإضافة اللقطات الى الـ staging area.
3. تقوم بعملية Commit، يتم أخذ الملفات المهيئة من مكان التهيئة Staging Area وتقوم بتخزين هذه الللقطة بشكل نهائي في مجلد عمل Git.

إذا كان هناك إصدار معين لملف، فسيكون Commited. إذا كان معدل ومضاف إلى مكان التهييئ staging area فهو مهيئ staged. If a particular version of a file is in the git directory, it’s considered committed. If it’s modified but has been added to the staging area, it is staged. And if it was changed since it was checked out but has not been staged, it is modified. In Chapter 2, you’ll learn more about these states and how you can either take advantage of them or skip the staged part entirely.

# .4 الإستعداد للبدء! - تصيب Git

## تصيب Git

والآن لنبدأ باستخدام Git! ولكن أولاً علينا تنصيبها على نظامنا. يمكنك الحصول على Git بأكثر من طريقة. الطريقتين الأكثر شهرة هي إما أن تنصبها يدوياً من الكود المصدري أو أن تحصل على تنصيب مهيئ لنظامك مباشرة.

### [التنصيب من الكود المصدري](https://git-scm.com/book/ar/v1/%D8%A7%D9%84%D8%A5%D8%B3%D8%AA%D8%B9%D8%AF%D8%A7%D8%AF-%D9%84%D9%84%D8%A8%D8%AF%D8%A1%21-%D8%AA%D8%B5%D9%8A%D8%A8-Git#%D8%A7%D9%84%D8%AA%D9%86%D8%B5%D9%8A%D8%A8-%D9%85%D9%86-%D8%A7%D9%84%D9%83%D9%88%D8%AF-%D8%A7%D9%84%D9%85%D8%B5%D8%AF%D8%B1%D9%8A)

بالطبع، من المفضل أن تقوم بتنصيب Git من الكود المصدري (أي أن تقوم ببناء Git يدوياً، إن استطعت) وذلك لأنك ستحصل دائماً على آخر نسخة منها. كل اصدار جديد من Git غالباً ما يحوي على تحسينات في واجهة الإستخدام. بالإضافة إلى أنك ستحصل على اصدار قديم اذا كنت تستخدم احدى توزيعات لينكس المختلفة. ولذلك، إذا لم تكن على توزيعة حديثاً جداً من المفضل أن تقوم بتنصيب مباشرة من الكود المصدري لـ Git.

لتنصيب Git، عليك أن تتحق من أن نظامك يحوي على المكتبات التالية، والتي تعتمد عليها Git، وهي: curl, zlip, openssl, expat و libiconv. على سبيل المثال، اذا كنت على نظام يحوي على أداة yum (مثل فيدورا Fedora) أو apt-get (مثل ديبيان Debian)، يمكنك تشغيل أحد الأوامر التالية للتأكد من تواجد هذه المكتبات:

$ yum install curl-devel expat-devel gettext-devel \

openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \

libz-dev

بعد تأكدك من المكتبات المطلوبة، يمكنك الحصول على آخر نسخة من الكود المصدري لـ Git من الموقع:

http://git-scm.com/download

ثم، يمكنك البناء والتنصيب:

$ tar -zxf git-1.6.0.5.tar.gz

$ cd git-1.6.0.5

$ make prefix=/usr/local all

$ sudo make prefix=/usr/local install

بعد ذلك، ستتمكن من الحصول على تحديثات على الكود المصدري لـ Git من خلال Git نفسها :)

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

### [التنصيب على لينكس](https://git-scm.com/book/ar/v1/%D8%A7%D9%84%D8%A5%D8%B3%D8%AA%D8%B9%D8%AF%D8%A7%D8%AF-%D9%84%D9%84%D8%A8%D8%AF%D8%A1%21-%D8%AA%D8%B5%D9%8A%D8%A8-Git#%D8%A7%D9%84%D8%AA%D9%86%D8%B5%D9%8A%D8%A8-%D8%B9%D9%84%D9%89-%D9%84%D9%8A%D9%86%D9%83%D8%B3)

إذا أردت تنصيب Git على نظام لينكس دون بناءه يدوياً، يمكنك ذلك من خلال أداة النصيب الخاصة بنظامك والتي تأتي مع نظامك. على سبيل المثال اذا كنت تستخدم نظام فيدورا Fedora، يمكنك استخدام yum:

$ yum install git-core

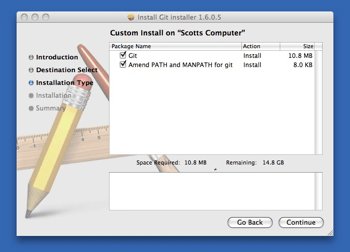
أو اذا كنت على نظام مبنى على ديبيان Debian مثل أوبونتو Ubuntu، يمكنك استخدام apt-get كالتالي:

$ apt-get install git

### [التنصيب على نظام الماك أو اس](https://git-scm.com/book/ar/v1/%D8%A7%D9%84%D8%A5%D8%B3%D8%AA%D8%B9%D8%AF%D8%A7%D8%AF-%D9%84%D9%84%D8%A8%D8%AF%D8%A1%21-%D8%AA%D8%B5%D9%8A%D8%A8-Git#%D8%A7%D9%84%D8%AA%D9%86%D8%B5%D9%8A%D8%A8-%D8%B9%D9%84%D9%89-%D9%86%D8%B8%D8%A7%D9%85-%D8%A7%D9%84%D9%85%D8%A7%D9%83-%D8%A3%D9%88-%D8%A7%D8%B3)

هناك طريقتين للتنصيب على ماك أو اس، الأسهل هي استخدام الواجهة الرسومية للتنصيب، والتي يمكنك تحميلها من صفحة المشروع على غوغل كود (انظر الشكل 1-7):

http://sourceforge.net/projects/git-osx-installer/



الشكل 1-7. تنصيب Git على ماك أو اس.

والطريقة الأكثر شهرة للحصول على Git هي من خلال MacPorts، ("http://macports.org"). يمكنك تشغيل الأمر التالي اذا كنت قد نصبت MacPorts من قبل:

$ sudo port install git-core +svn +doc +bash\_completion +gitweb

يمكنك عدم تنصيب أي من الأمور الإضافية مع Git، ولكنك ستحتاج الى +svn اذا كنت ستستخدم Git على repository تعتمد على نظام Subversion (انظر الفصل 8).

### [التنصيب على ويندوز](https://git-scm.com/book/ar/v1/%D8%A7%D9%84%D8%A5%D8%B3%D8%AA%D8%B9%D8%AF%D8%A7%D8%AF-%D9%84%D9%84%D8%A8%D8%AF%D8%A1%21-%D8%AA%D8%B5%D9%8A%D8%A8-Git#%D8%A7%D9%84%D8%AA%D9%86%D8%B5%D9%8A%D8%A8-%D8%B9%D9%84%D9%89-%D9%88%D9%8A%D9%86%D8%AF%D9%88%D8%B2)

يمكنك تنصيب Git على نظام ويندوز بسهولة. أحد أسهل الطرق هو استخدام مشروع msysGit. يمكنك تنصيب البرنامج من صفحة المشروع على غوغل كود:

http://msysgit.github.com/

بعد التنصيب سيكون لدين نسختين من الأداة للـ command-line في ويندوز (بالإضافة الى أداة SSH والتي ستستفيد منها لاحقاً) والأداة بالواجهة الرسومية الإعتيادية.

# .5 الإستعداد للبدء! - إعداد Git لأول مرة

## إعداد Git لأول مرة

والآن وبعد أن حصلت على Git على نظامك، سيتوجب عليك تخصيص بعض الخيارات لتناسب بيئتك. سيكون عليك القيام بهذه التعديلات مرة واحدة فقط; حتى بعد قيامك بتحديث نسختك من Git. ولكن بالطبع يمكنك تغيير هذه الإعدادات في أي وقت تريد.

تأتي Git مرفقة بأداة تدعى git config والتي تمكنك تعديل الخيارات (المتغيرات) التي تتحكم بطريقة عمل Git. يتم حفظ هذه المتغيرات في أحد ثلاث أمكنة مختلفة:

* في ملف /etc/gitconfig: يحتوي على قيم لجميع المستخديم على نظامك لجميع الـ repositories أيضاً. اذا قمت بوضع إضافة --system عند تشغيل الأمر git config، سيتم تعديل الخيارات على هذا الملف بالتحديد.
* في ملف ~/.gitconfig: وهو مخصص للمستخدم الخاص بك فقط. سيتم تغيير الإعدادات في هذا الملف بالتحديد عن طريق إضافة --global الى أمر git config.
* ملف الخيارات الموجود في مجلد عمل Git (الموجود في .git/config) في أي repository تعمل عليها، حيث تكون هذه الإعدادات مخصصة لهذه الـ repository فقط. عليك أن تعلم أيضاً بأن الإعدادات الموجودة في أي ملف سيتم تفضيلها (أي استخدامها) على المعلومات الموجودة في الملف الأعم، أي الإعدادات الموجودة في ملف الإعدادات .git/config ستتفوق على تلك الموجودة في /etc/gitconfig.

على نظام ويندوز، ستقوم Git بتفحص .gitconfig في مجلد الـ $HOME (عادة تكون C:\Documents and Settings\$USER). ستقوم Git أيضاً بتفقد ملف /etc/gitconfig، والذي يكون مبنياً على مكان وجود MSys، والذي تحدده أنت عندما تقوم بتنصيب Git على نظامك.

### [شخصيتك](https://git-scm.com/book/ar/v1/%D8%A7%D9%84%D8%A5%D8%B3%D8%AA%D8%B9%D8%AF%D8%A7%D8%AF-%D9%84%D9%84%D8%A8%D8%AF%D8%A1%21-%D8%A5%D8%B9%D8%AF%D8%A7%D8%AF-Git-%D9%84%D8%A3%D9%88%D9%84-%D9%85%D8%B1%D8%A9#%D8%B4%D8%AE%D8%B5%D9%8A%D8%AA%D9%83)

أول شيء عليك فعله بعد تنصيبك لـ Git هو اعداد اسمك وبريدك الالكتروني. أهمية هذا الأمر تكمن في أن كل عملية Commit في Git ستستخدم هذه المعلومات وسيتم لصقها بشكل غير قابل للتغيير في كل عملياتك:

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

وطبعاً، اذا أردت أن تتحدد هذه المعلومات على كامل النظام، عليك اضافة '--global' الى الأمر. واذا أردت تجاوز هذا الأمر وتحديد معلومات مختلفة لمشروع معين، عليك تشغيل هذا الأمر بدون '--global' داخل مشروعك.

### [محرر النصوص](https://git-scm.com/book/ar/v1/%D8%A7%D9%84%D8%A5%D8%B3%D8%AA%D8%B9%D8%AF%D8%A7%D8%AF-%D9%84%D9%84%D8%A8%D8%AF%D8%A1%21-%D8%A5%D8%B9%D8%AF%D8%A7%D8%AF-Git-%D9%84%D8%A3%D9%88%D9%84-%D9%85%D8%B1%D8%A9#%D9%85%D8%AD%D8%B1%D8%B1-%D8%A7%D9%84%D9%86%D8%B5%D9%88%D8%B5)

والآن وبعد اعداد شخصيتك، يمكنك تعيين محرر النصوص الإفتراضي والذي ستستخدمه Git عندما تحتاج منك الى ادخال رسالة ما. افتراضياً، ستستخدم Git المحرر الإفتراضي المحدد للنظام، والذي يكون عادة Vi أو Vim. إذا أردت استخدام محرر آخر، مثل Emacs، يمكن كتابة مايلي: ء $ git config --global core.editor emacs

### [أداة عرض الإختلافات Diff Tool](https://git-scm.com/book/ar/v1/%D8%A7%D9%84%D8%A5%D8%B3%D8%AA%D8%B9%D8%AF%D8%A7%D8%AF-%D9%84%D9%84%D8%A8%D8%AF%D8%A1%21-%D8%A5%D8%B9%D8%AF%D8%A7%D8%AF-Git-%D9%84%D8%A3%D9%88%D9%84-%D9%85%D8%B1%D8%A9#%D8%A3%D8%AF%D8%A7%D8%A9-%D8%B9%D8%B1%D8%B6-%D8%A7%D9%84%D8%A5%D8%AE%D8%AA%D9%84%D8%A7%D9%81%D8%A7%D8%AA-Diff-Tool)

أمر آخر قد تهتم بتعديله عن الخيار الإفتراضي هو أداة عرض الإختلافات Diff tool والتي ستستخدمها لحل التعارضات بين الإصدارات عن الدمج. فعلى سبيل المثال، لإستخدام أداة vimdiff:

$ git config --global merge.tool vimdiff

تقبل Git كلا من: kdiff3، tkdiff، meld، xxdiff، emerge، vimdiff، gvimdiff، ecmerge و opendiff كأدوات فعالة، يمكنك إضافة خيارات أخرى خاصة أيضاً، انظر الفصل السابع لمعلومات اخرى عن كيفية القيام بهذا.

### [تغيير الإعدادات](https://git-scm.com/book/ar/v1/%D8%A7%D9%84%D8%A5%D8%B3%D8%AA%D8%B9%D8%AF%D8%A7%D8%AF-%D9%84%D9%84%D8%A8%D8%AF%D8%A1%21-%D8%A5%D8%B9%D8%AF%D8%A7%D8%AF-Git-%D9%84%D8%A3%D9%88%D9%84-%D9%85%D8%B1%D8%A9#%D8%AA%D8%BA%D9%8A%D9%8A%D8%B1-%D8%A7%D9%84%D8%A5%D8%B9%D8%AF%D8%A7%D8%AF%D8%A7%D8%AA)

اذا أردت القاء نظرة على إعداداتك، يمكنك استخدام أمر 'git config --list' لعرض قائمة بكافة الخيارات التي أعددتها في Git:

$ git config --list

user.name=Scott Chacon

user.email=schacon@gmail.com

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

...

من الممكن أن ترى هذه المفاتيح أكثر من مرة، ذلك لأن Git تقوم بقراءة هذه المفاتيح من الملفات المختلفة (/etc/gitconfig و ~/.gitconfig, على سبيل المثال). في هذه الحاله، تقوم Git باستخدام آخر قيمة موجودة لكل مفتاح مختلف.

يمكنك أيضاً تفحص القيمة التي "تتعامل" معها Git لأي مفتاح مختلف عن طريق الأمر git config {key}:

$ git config user.name

Scott Chacon

# 6 الإستعداد للبدء! - الحصول على المساعدة

## الحصول على المساعدة

هناك أكثر من طريقة للحصول على المساعدة أثناء استخدامك لـ Git، يمكنك كتابة أحد الأوامر التالية:

$ git help <verb>

$ git <verb> --help

$ man git-<verb>

على سبيل المثال، يمكنك الحصول على المساعدة لاستخدام أمر config عن طريق الأمر التالي:

$ git help config

تكمن روعة هذه الأوامر بأنه يمكنك الوصول اليها من أي مكان، حتى ولو لم تكن موصولاً بالشبكة. إذا لم تكن صفحات المساعدة السابقة أو هذا الكتاب كافية بالنسبة اليك يمكنك الذهاب الى قناة #git أو #github على شبكة IRC Freenode (irc.freenode.net). عادة ما تكون هذه القنوات مليئة بالعديد من الخبراء بكيفية عمل Git ومستعدين للمساعدة.

# .7 الإستعداد للبدء! - الملخص

## الملخص

لقد حصلت حتى الآن على معلومات أولية عن نظام Git وماهي اختلفاته عن أنظمة ادارة الإصدارات المركزية الآخرى. يجب أن تكون قد حصلت على نسخة من Git تعمل على نظامك. والآن حان الوقت لتعلم بعض مبادئ استخدام Git.

# Chapter 2

# مبادئ Git

إذا كان هناك فصل واحد عليك قراءته لكي تبدأ بإستخدام Git، فعليك بهذا الفصل! يغطي هذا الفصل جميع الأوامر الأساسية التي عليك معرفتها لكي تتمكن من القيام بأغلب الأمور أثناء استخدامك لـ Git. في نهاية هذا الفصل يجب أن تكون قادراً على انشاء واعداد الـ repository لمشروعك وعلى تحديد الملفات التي ستتم متابعتها والتي ستترك، وعلى تهييئ التغييرات لعمل commit عليها. ستتعلم أيضاً كيف تعد Git لكي تتجاهل بعض أنواع الملفات، كيف تقوم بالتراجع عن الأخطاء التي سترتكبها بسرعة وبسهولة، كيف تتصفح تاريخ مشروعك وكيف تعرض التغيرات بين الـ commits، وكيف تنشر وتسحب (push & pull) التغيرات من الـ repositories البعيدة عنك.

# 1 مبادئ Git - الحصول على repository لمشروعك بـ Git

## الحصول على repository لمشروعك بـ Git

يمكنك انشاء مشروع بـ git بإحدى طريقتين. الأولى تكمن في أن تبدأ بناء على مشروع أسبق أو مجلد على سبيل المثال يحوي ملفات وأن تستورد مافيه الى مشروعك الجديد. أو أن تستنسخ repository بـ git جاهزة من مخدم ما.

### [انشاء Repository في مجلد موجود مسبقاً](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-%D8%A7%D9%84%D8%AD%D8%B5%D9%88%D9%84-%D8%B9%D9%84%D9%89-repository-%D9%84%D9%85%D8%B4%D8%B1%D9%88%D8%B9%D9%83-%D8%A8%D9%80-Git#%D8%A7%D9%86%D8%B4%D8%A7%D8%A1-Repository-%D9%81%D9%8A-%D9%85%D8%AC%D9%84%D8%AF-%D9%85%D9%88%D8%AC%D9%88%D8%AF-%D9%85%D8%B3%D8%A8%D9%82%D8%A7%D9%8B)

للبد بمتابعة تغيرات مشروع مسبق لديك بـ git، عليك الذهاب الى مجلد مشروعك ثم كتابة:

$ git init

يقوم هذا بانشاء مجلد فرعية جديد تحت اسم .git يحوي جميع المعلومات والملفات المطلوبة - هيكل repository بـ git. حتى الآن، لن يتم متابعة أي تغيرات تحصل على مشروعك (انظر الفصل التاسع لمعلومات أخرى عن الملفات الموجودة في مجلد .git الذي أنشأته قبل قليل).

اذا أردت بمتابعة اصدارات الملفات الموجودة، يجب عليك البدء باخبار Git بهذه الملفات ومن ثم اجراء commit أولي. للقيام بهذا عليك استخدام أمر git add بالعدد الكافي بالملفات التي تريد، ومن ثم تتبعها أمر commit:

$ git add \*.c

$ git add README

$ git commit –m 'initial project version'

سنقوم بشرح هذه الأوامر في الفقرة القادمة، ولكن، الآن أصبحت التغيرات في مشروعك تتم متابعتها و commit أولي.

### [استنساخ repository موجودة مسبقاً](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-%D8%A7%D9%84%D8%AD%D8%B5%D9%88%D9%84-%D8%B9%D9%84%D9%89-repository-%D9%84%D9%85%D8%B4%D8%B1%D9%88%D8%B9%D9%83-%D8%A8%D9%80-Git#%D8%A7%D8%B3%D8%AA%D9%86%D8%B3%D8%A7%D8%AE-repository-%D9%85%D9%88%D8%AC%D9%88%D8%AF%D8%A9-%D9%85%D8%B3%D8%A8%D9%82%D8%A7%D9%8B)

للقيام باستنساخ repository بـ git موجودة مسبقاً - مشروع تريد المساهمة فيه على سبيل المثال - الأمر الذي ستحتاجه هو git clone. اذا كنت قد استخدمت أحد أنظمة إدارة الإصدارات الأخرى مثل Subversion مثلاً، ستلاحظ الإختلاف بين الأمر clone في git وأمر checkout. هناك فرق مهم، عندما تقوم git بعملية clone فإنك ستحصل على نسخة من كامل المعلومات تقريباً الموجودة على مخدم الـ repository. جميع إصدارات الملفات كلها في تاريخ حياة المشروع. أي، وبمعنى آخر، اذا حدث عطل أو ضرر ما لملفات المشروع الموجودة على المخدم الأساسي يمكنك استخدام أي واحدة من النسخ لاسترجاع المشروع للحالة التي كان عليها عند استنساخه (من الممكن أن تخسر بعض الروابط من طرف المخدم، ولكن جميع معلومات الإصدارات ستكون موجودة- انظر الفصل الرابع لمعلومات اخرى).

لاستنساخ repository يمكنك استخدام الأمر 'git clone [url]'. فعلى سبيل المثال، لكي تقوم باستنساخ مكتبة Git للغة Ruby والتي تدعى Grit، يمكنك القيام بالتالي:

$ git clone git://github.com/schacon/grit.git

سيقوم هذا بانشاء مجلد جديد باسم "grit"، وتجهيز مجلد '.git' في داخله، سيقوم أيضاً بسحب جميع المعلومات من الـ repository، ويجهز نسخة جاهزة لكي تعمل عليها لآخر نسخة. اذا دخلت على مجلد 'grit' الجديد ستجد جميع ملفات المشروع. يمكنك بالطبع استنساخ المشروع لمجلد بإسم آخر، إليك مثال لكيفية القيام بهذا:

$ git clone git://github.com/schacon/grit.git mygrit

سيقوم هذا الأمر بذات الأمر ولكن سيتم وضع المشروع بمجلد جديد باسم mygrit.

هناك عدد من البروتوكولات المختلفة التي يمكنك اسستعمالها لنقل المعلومات في git. المثال السابق يستعمل بروتوكول 'git://'، ولكن من الممكن أن تجد أيضاً استخداماً لـ 'http(s)://' أو 'user@server:/path.git'، والتي تستعمل بروتوكول SSH في النقل. في الفصل الرابع من الكتاب ستتعرف على الخيارات المتوفرة للتواصل مع الـ repository الخاصة بك وميزات ومساوئ كل منها.

2 مبادئ Git - تسجيل التعديلات في الـ repository

**تسجيل التعديلات في الـ repository**

لديك repository أصلي ونسخة لتعمل عليها من ملفات المشروع. عليك أن تقوم ببعض التعديلات ثم تعمل commit لهذه التعديلات في repository الخاص بك في كل مرة يصل فيها المشروع إلى نقطة تود تسجيلها.

تذكر أنه كل ملف في مجلد العمل يمكن أن يكون في إحدى الحالتين فقط: مٌتَتَبّع tracked أو غير مٌتَتَبّع untracked. الملفات المٌتتبّعة هي ملفات كانت في أخر snapshot ويمكن إلغاء التعديلات عليها أو التعديل عليها أو وضعه في حالة staged (جاهز من أجل commit). الملفات غير المُتتبّعة هي كل الملفات الآخرى - أي ملف في مجلد العمل لم يكن موجوداً في آخر snapshot وليس معلماً بأنه staged. عندما تقوم باستنساخ repository جميع ملفاتك تكون بحالة متتبّعة tracked و غير معدلة unmodified لأنك قمت للتو بعمل check out ولم تقم بأي تعديل.

عندما تعدل الملفات، سيقوم git بتأشيرهم على أنهم modified، لأنك قمت بتغيرهم عن آخر commit. تقوم بعمل stage لهذه الملفات المعدلة ثم تقوم بعمل commit لجميع التغيرات في منطقة stage، وتتكرر العملية. يوضع الشكل 2-1 دورة العملية.



الشكل 2-1. دورة حالة الملفات.

[**تفقد حالة ملفاتك**](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-%D8%AA%D8%B3%D8%AC%D9%8A%D9%84-%D8%A7%D9%84%D8%AA%D8%B9%D8%AF%D9%8A%D9%84%D8%A7%D8%AA-%D9%81%D9%8A-%D8%A7%D9%84%D9%80-repository#%D8%AA%D9%81%D9%82%D8%AF-%D8%AD%D8%A7%D9%84%D8%A9-%D9%85%D9%84%D9%81%D8%A7%D8%AA%D9%83)

باستخدام الأمر git status يمكننا معرفة حالة الملفات لدينا. إذا قمت بتشغيل هذا الأمر مباشرة بعد قيامك بعمل clone يجب أن ترى شيئاً يشبه التالي:

$ git status

# On branch master

nothing to commit, working directory clean

وهذا يعني أنه لديك مجلد عمل نظيف - بمعنى آخر، لايوجد أي ملفات معدلة أو ملفات غير مُتتبّعة. كما أنّ هذا الأمر يخبرك بأي فرع branch أنت تعمل. حالياً، دائماً هو master، وهو الافتراضي؛ في الفصل المقبل سنمر على الأفرع و المرجعيات references بالتفصيل.

لنقل بأنك قمت بإضافة ملف جديد على مشروعك، وليكن ملف README بسيط. إذا لم يكن الملف موجوداً مسبقاً، وقمت بتنفيذ الأمر git status سترى الملف غير مُتتبّعاً كما يلي:

$ vim README

$ git status

# On branch master

# Untracked files:

# (use "git add <file>..." to include in what will be committed)

#

# README

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

يمكنك ملاحظة أنّ ملفك الجديد README غير مُتتبّع، فهو تحت تبويب "untracked files" في خرج الأمر. ويعني ذلك أنّ git يرى ملفاً جديداً على commit السابقة؛ علماً أنّ git لن يقوم بإضافة هذا الملف إلى الملفات المتتبعة إلا إذا قمت بطلب ذلك بشكل مباشر، والهدف من ذلك من أجل حماية المشروع من الضم الخاطئ لملفات binary أو أي ملفات لا تود بإضافتها. إلاّ أنّك ترغب في إضافة README إلى الملفات المتتبّعة. وسنقوم بذلك حالاً.

[**تتبع الملفات الجديدة**](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-%D8%AA%D8%B3%D8%AC%D9%8A%D9%84-%D8%A7%D9%84%D8%AA%D8%B9%D8%AF%D9%8A%D9%84%D8%A7%D8%AA-%D9%81%D9%8A-%D8%A7%D9%84%D9%80-repository#%D8%AA%D8%AA%D8%A8%D8%B9-%D8%A7%D9%84%D9%85%D9%84%D9%81%D8%A7%D8%AA-%D8%A7%D9%84%D8%AC%D8%AF%D9%8A%D8%AF%D8%A9)

للقيام بتتبع ملف جديد عليه استخدام الأمر git add . مثلاً لنقم بتتبع الملف الجديد README:

$ git add README

إذا قمنا بتنفيذ الأمر git status مرة أخرى سنلاحظ أن الملف README أصبح متتبعاً وجاهزاً staged للقيام بعملية commit:

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

#

نستطيع معرفة بأن الملف staged من خلال ملاحظته تحت بند "changes to be comitted". إذا قمت بعمل commit في هذه اللحظة، سيقوم git بإضافة النسخة الحالية من الملف إلى snapshot. تذكر عندما قمنا بعمل git init سابقاً، ثم قمنا بإضافة الملفات عن طريق git add، كان ذلك للقيام ببدء تتبع الملفات في مجلد المشروع. يقبل الأمر git add مسار لملف أو لمجلد؛ فإذا كان المسار لمجلد سيقوم git بإضافة جميع الملفات والمجلدات ضمنه بشكل تعاودي recursively.

[**تجهيز الملفات المعدلة**](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-%D8%AA%D8%B3%D8%AC%D9%8A%D9%84-%D8%A7%D9%84%D8%AA%D8%B9%D8%AF%D9%8A%D9%84%D8%A7%D8%AA-%D9%81%D9%8A-%D8%A7%D9%84%D9%80-repository#%D8%AA%D8%AC%D9%87%D9%8A%D8%B2-%D8%A7%D9%84%D9%85%D9%84%D9%81%D8%A7%D8%AA-%D8%A7%D9%84%D9%85%D8%B9%D8%AF%D9%84%D8%A9)

لنقم بالتعديل على ملف قمنا بإضافته سابقاً. إذا قمنا مثلاً بالتعديل على ملف متتبع مسبقاً يدعى benchmarks.rb وقمنا بتنفيذ الأمر git status، سيكون الخرج مشابهاً للخرج التالي:

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: benchmarks.rb

#

يظهر الملف benchmarks.rb تحت بند "changes not staged for commit" وهذا يعني أن الملف المُتتبّع قد خضع لعملية تعديل لكنه لم يخضع للتجهيز من أجل commit. للقيام بتجهيزه (أو تأشيره للإضافة إلى commit الجديد) يجب علينا تنفيذ الأمر git add (لاحظ بأنّه أمر متعدد الوظائف - نستطيع استخدامه لتتبع الملفات الجديدة، تجهيز الملفات من أجل commit، والقيام بأمور أخرى مثل حل الاعتراضات في حال القيام بدمج merge). لنقم بتنفيذ git add الآن لوضع benchmarks.rb بحالة staged، ومن ثم لنقم بتنفيذ الأمر git status لنرى ما الذي قد تغيّر:

$ git add benchmarks.rb

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

# modified: benchmarks.rb

#

كلا الملفين الآن جاهز للإدخال بعملية commit المقبلة. في هذه النقطة، لنفرض أنك تود القيام بتعديل على ملف benchmarks.rb قبل القيام بعملية commit، ستقوم بفتح الملف والتعديل عليه وأنك جاهز للقيام بعملية commit. لكن قبل ذلك، دعنا نقوم بتنفيذ الأمر git status مرة إضافية:

$ vim benchmarks.rb

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

# modified: benchmarks.rb

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: benchmarks.rb

#

ما الذي يحصل؟ كيف أصبح benchmarks.rb موجوداً تحت التبويبين staged و unstaged؟ لقد اتضح لنا أنّ git يقوم بتجهيز الملف على حالته عند قيامك بتنفيذ الأمر git add. إذا قمت بعمل commit الآن، ستكون نسخة benchmarks.rb كما كانت عند قيامك بتنفيذ الأمر git add وليس النسخة الجديدة التي حصلنا عليها بعد قيامنا بتعديل الملف. لذا إذا قمنا بتعديل ملف قبل قيامنا بتنفيذ git add، وقبل القيام بعملية commit، علينا تجهيز الملف مرة أخرى لعملية commit، وذلك بتنفيذ الأمر git add مرة جديدة:

$ git add benchmarks.rb

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

# modified: benchmarks.rb

#

[**تجاهل الملفات**](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-%D8%AA%D8%B3%D8%AC%D9%8A%D9%84-%D8%A7%D9%84%D8%AA%D8%B9%D8%AF%D9%8A%D9%84%D8%A7%D8%AA-%D9%81%D9%8A-%D8%A7%D9%84%D9%80-repository#%D8%AA%D8%AC%D8%A7%D9%87%D9%84-%D8%A7%D9%84%D9%85%D9%84%D9%81%D8%A7%D8%AA)

غالباً ما تود من git تجاهل صنف من الملفات بحث لا يقوم بإضافتها تلقائياً أو لا يظهرها بأنّها غير متتبعة. تكون هذه الملفات عادة ملفات مولدة بشكل تلقائي مثل ملفات log و الملفات الوسيطة التي تولدها أدوات التطوير لديك. في مثل هذه الحالات/ يمكن إنشاء ملف .gitignore يحوي على أنماط لأسماء الملفات التي نرغب بتجاهلها. هذا مثال عمّا قد يحتويه ملف .gitignore:

$ cat .gitignore

\*.[oa]

\*~

أول سطر يقوم بتوجيه git إلى تجاهل أي ملفات ذات لواحق من النوع o أو a - ملفات الكائنات وملفات الأرشيف وهي ملفات وسيطة تولدها أدوات بناء الكود عادة. السطر الثاني يوجه git إلى تجاهل أي ملفات تنتهي بالرمز (~) والتي تكون ملفات مؤقتة عادةً تستخدمها بعض برامج تحرير الكود. قد ترغب أيضاً بإضافة مجلدات log و tmp أو pid؛ أو حتى ملفات التوثيق تلقائية التوليد (من الكود عادة)، وغيرها. ينصح بإضافة ملف .gitignore في بداية إنشاء repository حتى نتجنب إضافة بعض الملفات عن طريق الخطأ وتلويث respository.

قواعد الأنماط التي يمكن وضعها ضمن ملف .gitignore هي كالتالي:

* الأسطر التي تبدأ بالرمز (#) يتم تجاهلها.
* أنماط glob القياسية تعمل.
* يمكن إنهاء النمط برمز (/) للدلالة على أنه يستهدف مجلداً.
* يمكن نفي نمط ما عن طريق وضع علامة التعجب (!) في بداية السطر قبل النمط.

أنماط Glob عبارة عن نسخة مبسطة من Regular Expressions يتم استخدامها ضمن واجهة الأوامر shell. رمز النجمة (\*) يطابق صفر-محرفاً أو أكثر. [abc] تطابق أي محارف ضمن الأقواس المربعة في هذه الحالة تكون a b c؛ علامة الاستفهام (?) تطابق محرفاً واحداً فقط؛ بينما تطابق الأقواس المربعة التي تحوي على محارف مفصولة بإشارة hyphen أي محرفاً يقع في المجال بين محرف البداية والنهاية - مثلا [0-9] يطابق محارف الأرقام بين 0 و 9 ضمناً.

مثال عن ملف .gitignore:

# a comment – this is ignored

# no .a files

\*.a

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

!lib.a

# only ignore the root TODO file, not subdir/TODO

/TODO

# ignore all files in the build/ directory

build/

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

doc/\*.txt

[**مشاهدة التغيّرات المجهّزة والتغيّرات غير المجهّزة**](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-%D8%AA%D8%B3%D8%AC%D9%8A%D9%84-%D8%A7%D9%84%D8%AA%D8%B9%D8%AF%D9%8A%D9%84%D8%A7%D8%AA-%D9%81%D9%8A-%D8%A7%D9%84%D9%80-repository#%D9%85%D8%B4%D8%A7%D9%87%D8%AF%D8%A9-%D8%A7%D9%84%D8%AA%D8%BA%D9%8A%D9%91%D8%B1%D8%A7%D8%AA-%D8%A7%D9%84%D9%85%D8%AC%D9%87%D9%91%D8%B2%D8%A9-%D9%88%D8%A7%D9%84%D8%AA%D8%BA%D9%8A%D9%91%D8%B1%D8%A7%D8%AA-%D8%BA%D9%8A%D8%B1-%D8%A7%D9%84%D9%85%D8%AC%D9%87%D9%)

إذا لم تكتف بالمعلومات التي يقدمها لك أمر git status يمكنك استخدام أمر git diff للحصول على معلومات تفصيلية حول التغيرات التي طرأت على الملفات. سنقوم لاحقاً بالتعمق في هذا الأمر، لكن الآن سنكتفي بالإشارة إلى الاستخدامات الغالبة له؛ حيث أنك ستستخدمه غالباً للحصول على أجوبة على هذين السؤالين: ما الذي قمنا بالتعديل عليه ولم نجهزه بعد لعملية commit؟ ما الملفات التي أصبحت جاهزة للدخول في عملية commit المقبلة؟ بالرغم من أنه يمكننا أن نحصل على هذه المعلومات باستخدام أمر git status إلا أنّ أمر git diff يوضح لنا التغيرات التي جرت على مستوى السطر والحرف - ما الذي قمنا بإضافته وما الذي أزلناه!

لنقل أنك قمت بالتعديل على ملف README مرة أخرى وأشرته للإضافة إلى عملية commit وقمت بالتعديل على ملف benchmarks.rb ولم تضفه إلى قائمة الملفات الجاهزة لعملية commit؛ إذا قمت بتنفيذ الأمر status ستشاهد مرة أخرى شيئاً من الشكل:

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: benchmarks.rb

#

لترى ما قمت بالتعديل عليه ولم تجهزه للإضافة نفذ الأمر git diff بدون إضافات:

$ git diff

diff --git a/benchmarks.rb b/benchmarks.rb

index 3cb747f..da65585 100644

--- a/benchmarks.rb

+++ b/benchmarks.rb

@@ -36,6 +36,10 @@ def main

@commit.parents[0].parents[0].parents[0]

end

+ run\_code(x, 'commits 1') do

+ git.commits.size

+ end

+

run\_code(x, 'commits 2') do

log = git.commits('master', 15)

log.size

يقوم هذا الأمر بعمل مقارنة بين الملفات ضمن مجلد العمل والملفات الموجودة في منطقة التعديلات المجهزة للإضافة staging area. وتخبرنا نتيجته بالتعديلات التي أجريناها ولم نقم بتجهيزها للإضافة إلى عملية commit المقبلة.

لمشاهدة الملفات ذات الحالة staged والتي ستدخل في عملية commit المقبلة، يمكن استخدام الأمر git diff --cached (بالنسبة للإصدارات 1.6.1 وما بعد من git يمكنك أيضاً استخدام الأمر git diff --staged )، كالتالي:

$ git diff --cached

diff --git a/README b/README

new file mode 100644

index 0000000..03902a1

--- /dev/null

+++ b/README2

@@ -0,0 +1,5 @@

+grit

+ by Tom Preston-Werner, Chris Wanstrath

+ http://github.com/mojombo/grit

+

+Grit is a Ruby library for extracting information from a Git repository

من الجدير بالذكر أن أمر git diff لوحده لا يقوم بعرض جميع التعديلات التي تمت من آخر commit - فهو يقوم بعرض فقط التعديلات التي لم تؤشر على أنها staged. ويمكن أن يسبب ذلك بعض الإرباك، حيث أنك إذا قمت بإضافة جميع التعديلات إلى قائمة staged فلن يقوم بعرض أي شي في خرج تنفيذه.

كمثال أيضاً، إذا قمنا بإضافة benchmarks.rb إلى قائمة staged ومن ثم قمنا بالتعديل عليه من جديد، يمكننا استخدام أمر git diff للحصول على لائحة بالتغييرات التي حصلت ولم تضف إلى قائمة staged كالتالي:

$ git add benchmarks.rb

$ echo '# test line' >> benchmarks.rb

$ git status

# On branch master

#

# Changes to be committed:

#

# modified: benchmarks.rb

#

# Changes not staged for commit:

#

# modified: benchmarks.rb

#

$ git diff

diff --git a/benchmarks.rb b/benchmarks.rb

index e445e28..86b2f7c 100644

--- a/benchmarks.rb

+++ b/benchmarks.rb

@@ -127,3 +127,4 @@ end

main()

##pp Grit::GitRuby.cache\_client.stats

+# test line

وباستخدام git diff --cached نتمكن من رؤية ما تم تجهيزه للإضافة إلى عملية commit القادمة:

$ git diff --cached

diff --git a/benchmarks.rb b/benchmarks.rb

index 3cb747f..e445e28 100644

--- a/benchmarks.rb

+++ b/benchmarks.rb

@@ -36,6 +36,10 @@ def main

@commit.parents[0].parents[0].parents[0]

end

+ run\_code(x, 'commits 1') do

+ git.commits.size

+ end

+

run\_code(x, 'commits 2') do

log = git.commits('master', 15)

log.size

**القيام بعملية (اعتماد) commit للتغيّرات**

بعد اكمال تجهيز الملفات التي ترغب بإضافتها إلى النسخة snapshot الجديدة، يمكنك تنفيذ أمر commit ليتم اعتماد التعديلات التي أجريتها في سجل git. تذكر أنّ أي ملف لم يتم تجهيزه - سواء لم تقم بإضافته باستخدام الأمر git add بعد إنشاءه أو التعديل عليه - لن يدخل في هذه الإعتمادية، وستبقى على أنها ملفات تم تعديلها في مجلد العمل. أبسط طريقة لاعتماد التعديلات هي القيام بأمر git commit كالتالي:

$ git commit

تنفيذ هذا الأمر سيطلب منا إدخال رسالة عملية commit - عادة ما يتم فتح محرر النصوص المشار إليه بمتغير البيئة $EDITOR. يمكنك تهيئته عن طريق الأمر git config --global core.editor كما شاهدنا في الفصل الأول.

يقوم محرر النصوص بعرض هذه الشاشة (مثالنا باستخدام VIM):

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

# modified: benchmarks.rb

~

~

~

".git/COMMIT\_EDITMSG" 10L, 283C

يمكنك ملاحظة أن رسالة عملية commit تحوي على خرج آخر عملية git status على شكل تعليقات بالإضافة إلى سطر فارغ في بداية الملف. يمكنك إزالة هذه التعليقات، أو يمكنك تركها لمساعدتك بتذكر ما قمت باعتماد تعديلاته. يمكنك الحصول على معلومات أكثر تفصلياً إذا قمت بتمرير الخيار -v إلى الأمر git commit. حيث يقوم ذلك بإضافة خرج أمر git diff إلى رسالة commit على شكل تعليقات أيضاً. عند إغلاق المحرر يقوم git بإنشاء commit ويتجاهل التعليقات.

علماً أنّه يمكنك كتابة رسالة الاعتمادية مباشرة من خلال تمرير الخيار -m إلى الأمر git commit على الشكل التالي:

$ git commit -m "Story 182: Fix benchmarks for speed"

[master]: created 463dc4f: "Fix benchmarks for speed"

2 files changed, 3 insertions(+), 0 deletions(-)

create mode 100644 README

مبروك، لقد قمت بعمل أول commit لك! يعطيك خرج العملية معلومات عنها: في أي فرع branch تم الإعتماد (هنا master)، ما قيمة هاش SHA-1 الخاصة بالعملية ( هنا 463dc4f)، عدد الملفات التي تغيّرت، بالإضافة إلى إحصاءات حول الأسطر التي أضيفت وأزيلت في هذه العملية.

تذكر أنّ عملية commit تأخذ صورة عن الملفات في قائمة staged. أي شيء لم تقم بإضافته إلى هذه القائمة ما زال في مجلد العمل بحالة "معدل" modified؛ يمكنك القيام بإضافتهم من خلال عملية commit جديدة إلى التأريخ في git. نستنتج أنّه في كل عملية commit يقوم git بأخذ "صورة" snapshot عن المشروع يمكننا العودة لها لاحقاً أو مقارنتها أو غير ذلك..

[**تجاوز منطقة التجهيز Staging Area**](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-%D8%AA%D8%B3%D8%AC%D9%8A%D9%84-%D8%A7%D9%84%D8%AA%D8%B9%D8%AF%D9%8A%D9%84%D8%A7%D8%AA-%D9%81%D9%8A-%D8%A7%D9%84%D9%80-repository#%D8%AA%D8%AC%D8%A7%D9%88%D8%B2-%D9%85%D9%86%D8%B7%D9%82%D8%A9-%D8%A7%D9%84%D8%AA%D8%AC%D9%87%D9%8A%D8%B2-Staging-Area)

منطقة التجهيز تكون أحياناً معقدة أكثر مما تحتاج في عملك إلا أنّها مفيدة لعمل commits تماماً كما تودهم أن يكونوا. إذا أردت تجاوز منطقة التجهيز، يوفر git اختصاراً بسيطاً لذلك. باستخدام الأمر git commit -a يقوم git بإضافة الملفات المتتبعة إلى منطقة التجهيز بشكل تلقائي، كأنك قمت بعمل git add:

$ git status

# On branch master

#

# Changes not staged for commit:

#

# modified: benchmarks.rb

#

$ git commit -a -m 'added new benchmarks'

[master 83e38c7] added new benchmarks

1 files changed, 5 insertions(+), 0 deletions(-)

لاحظ بأنك لاتحتاج إلى تنفيذ الأمر git add على ملف benchmark.rb قبل القيام بعملية commit.

[**إزالة الملفات**](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-%D8%AA%D8%B3%D8%AC%D9%8A%D9%84-%D8%A7%D9%84%D8%AA%D8%B9%D8%AF%D9%8A%D9%84%D8%A7%D8%AA-%D9%81%D9%8A-%D8%A7%D9%84%D9%80-repository#%D8%A5%D8%B2%D8%A7%D9%84%D8%A9-%D8%A7%D9%84%D9%85%D9%84%D9%81%D8%A7%D8%AA)

لحذف ملف من git، يجب عليك إزالته من قائمة الملفات المتتبعة (وبشكل أدق، إزالته من منطقة التجهيز) ومن ثم القيام بعملية commit. الأمر git rm يقوم بعمل ذلك كما يقوم بحذف الملف من مجلد العمل الخاص بك لذا لن تراه بعد الآن في قائمة الملفات غير المتتبعة في المرة المقبلة.

إذا قمت بإزالة الملف من مجلد العمل، يظهر تحت بند "تعديلات غير مجهزة للاعتماد" (أي **unstaged**) من خرج الأمر git status:

$ rm grit.gemspec

$ git status

# On branch master

#

# Changes not staged for commit:

# (use "git add/rm <file>..." to update what will be committed)

#

# deleted: grit.gemspec

#

فإذا قمت بتنفيذ أمر git rm يقوم بتجهيز عملية الحذف ليتم اعتمادها:

$ git rm grit.gemspec

rm 'grit.gemspec'

$ git status

# On branch master

#

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# deleted: grit.gemspec

#

وفي المرة المقبلة التي ستقوم فيها بعمل commit، سيتم إزالة الملف من قائمة التتبع. إذا قمت مسبقاً بتعديل الملف وإضفته إلى الفهرس، يجب عليه تنفيذ عملية الحذف قسرياً وذلك بإضافة الخيار -f. يتم اتباع هذه الطريقة بغية حماية البيانات من أية عمليات حذف "عرضية" لملفات لم يتم تسجيلها ضمن git ولايمكن استعادتها بعد ذلك.

أما إذا أردت إزالة الملف من منطقة التجهيز مع الحفاظ عليه في مجلد العمل (أي إزالته من تتبع git مع بقاءه على وسيطة التخزين) - وهو شيء مفيد إذا قمت بنسيان إضافة شيء ما إلى ملف .gitignore وأضفته عن طريق الخطأ ؛ كملف log كبير مثلاً - استخدم الخيار --cached مع الأمر git rm كالتالي:

$ git rm --cached readme.txt

يمكنك تمرير أسماء ملفات، مجلدات، وأنماط glob للأمر git rm. وهذا يعني بأنه يمكننا عمل أشياء كالتالي:

$ git rm log/\\*.log

لاحظ الشرطة العكسية (\) قبل رمز النجمة \*. إنّها ضرورية لأن git يقوم بعمل توسعة الأسماء الخاصة به بالإضافة إلى التوسعة الخاصة بسطر الأوامر. هذا الأمر يقوم بحذف كافة الملفات التي تملك اللاحقة .log في مجلد /log. أو يمكنك عمل شيء كالتالي:

$ git rm \\*~

وهذا الأمر يقوم بحذف كافة الملفات المنتهية بالمحرف ~.

[**نقل الملفات**](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-%D8%AA%D8%B3%D8%AC%D9%8A%D9%84-%D8%A7%D9%84%D8%AA%D8%B9%D8%AF%D9%8A%D9%84%D8%A7%D8%AA-%D9%81%D9%8A-%D8%A7%D9%84%D9%80-repository#%D9%86%D9%82%D9%84-%D8%A7%D9%84%D9%85%D9%84%D9%81%D8%A7%D8%AA)

على خلاف أغلب أنظمة إدارة الإصدارات VCS الأخرى، لا يقوم git بتعقب حركة الملفات بشكل صريح. إذا قمت بإعادة تسمية ملف ضمن git، لايتم تسجيل أي بيانات وصفية metadata في git تقوم بأنك قمت بإعادة تسمية الملف. لكن، git ذكي جداً في استعياب ذلك - وسنقوم بمناقشة الأمر بعد قليل.

إذا أردت القيام بإعادة تسمية ملف يمكنك استخدام أمر mv في git كالتالي:

$ git mv file\_from file\_to

إذا قمنا بتنفيذ الأمر والنظر إلى خرج أمر git status:

$ git mv README.txt README

$ git status

# On branch master

# Your branch is ahead of 'origin/master' by 1 commit.

#

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# renamed: README.txt -> README

#

وهو مكافئ للقيام بتنفيذ الأوامر التالي على التسلسل:

$ mv README.txt README

$ git rm README.txt

$ git add README

يدرك git بأنك قمت بعملية إعادة تسمية، لذا فالإختلاف الوحيد بين الطريقتين هو أنّ git mv عبارة عن أمر واحد، وليس ثلاثة أوامر. يمكن الاستفادة من هذه الخاصة باستخدام أية أدوات للقيام بعمليات إعادة التسمية، واستخدام add/rm قبل القيام بعملية commit.

# 3 مبادئ Git - مراجعة تأريخ عمليات commit

## مراجعة تأريخ عمليات commit

بعد قيامك بعدد من عمليات الاعتماد commit، أو استنساخ repository بسجل تأريخ، ربما ستود إلقاء نظرة على ما جرى. أبسط وأقوى أداة لعمل ذلك هي الأمر git log.

هذه الأمثلة تستخدم مشروع بسيط جداً يدعى simplegit أقوم باستخدامه في عمليات العرض بأغلب الأحيان. للحصول على المشروع قم باستنساخه عن موقع github كالتالي:

git clone git://github.com/schacon/simplegit-progit.git

عندما تقوم بعمل git log ضمن المشروع، سيظهر لديك خرج مشابه للتالي:

$ git log

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 10:31:28 2008 -0700

first commit

بتنفيذ الأمر بدون بارمترات، يقوم git بعرض عمليات commit في repository بترتيب زمني معكوس - من الأحدث إلى الأقدم. كما يقوم بعرض بجانب كل commit هاش SHA-1 checksum الخاص بها، اسم وبريد الكاتب الإلكتروني، تاريخ الاعتماد، ورسالة الاعتماد.

يمكن إرفاق الأمر git log بعدد كبير من الخيارات للحصول على المعلومات التي نريدها بالضبط.. تجد في الأسفل مثالاً عن أكثر الخيارات استخداماً.

أحد أهم الخيارات هو -p، والذي يقوم بإظهار الفوارق المستحدثة بين عمليات commit المختلفة. يمكن أيضاً استخدام الخيار -2 ليحد خرج النتيجة إلى آخر عمليتين:

$ git log –p -2

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

diff --git a/Rakefile b/Rakefile

index a874b73..8f94139 100644

--- a/Rakefile

+++ b/Rakefile

@@ -5,7 +5,7 @@ require 'rake/gempackagetask'

spec = Gem::Specification.new do |s|

- s.version = "0.1.0"

+ s.version = "0.1.1"

s.author = "Scott Chacon"

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

removed unnecessary test code

diff --git a/lib/simplegit.rb b/lib/simplegit.rb

index a0a60ae..47c6340 100644

--- a/lib/simplegit.rb

+++ b/lib/simplegit.rb

@@ -18,8 +18,3 @@ class SimpleGit

end

end

-

-if $0 == \_\_FILE\_\_

- git = SimpleGit.new

- puts git.show

-end

\ No newline at end of file

هذا الخيار يعرض نفس المعلومات بالإضافة خرج diff بعده مباشرة. فهو مهم جداً من أجل مراجعة الكود واستعراض ما الذي تغير بشكل سريع ضمن سلسلة من عمليات commit. يمكنك أيضاً استخدام خيارات للتلخيص مع أمر git log. على سبيل المثال، إذا أردت رؤية بعض الإحصاءات المختصرة لكل commit، يمكنك استخدام الخيار stat:

$ git log --stat

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

Rakefile | 2 +-

1 files changed, 1 insertions(+), 1 deletions(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

removed unnecessary test code

lib/simplegit.rb | 5 -----

1 files changed, 0 insertions(+), 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 10:31:28 2008 -0700

first commit

README | 6 ++++++

Rakefile | 23 +++++++++++++++++++++++

lib/simplegit.rb | 25 +++++++++++++++++++++++++

3 files changed, 54 insertions(+), 0 deletions(-)

كما ترى باستخدام الخيار --stat يتم عرض قائمة من الملفات المعدلة، عدد الملفات التي تغيرت، وعدد الأسطر التي أضيفت أو أزيلت. كما يضع ملخصاً للمعلومات في النهاية. يوجد خيار مفيد آخر وهو --pretty. هذا الخيار يغير خرج السل إلى صيغ غير الافتراضية. يوجد بعضها مركب مسبقاً. مثلاً oneline يقوم بطباعة كل commit على سطر لوحدها، وهذا أمر مفيد في حال كنت تنظر إلى العديد من عمليات commit. بالإضافة إلى short، full و fuller والتي تعرض خرجاً تقريباً بنفس الصيغة مع معلومات أكثر أو أقل:

$ git log --pretty=oneline

ca82a6dff817ec66f44342007202690a93763949 changed the version number

085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test code

a11bef06a3f659402fe7563abf99ad00de2209e6 first commit

أكثر الخيارات أهمية هو format، والذي يسمح لك بتحديد صيغة الخرج بشكل صريح بما يتناسب مع إعرابها آلياً - حيث أنك تعرف أنّه لن يتغير عند التحديث إلى git:

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

ca82a6d - Scott Chacon, 11 months ago : changed the version number

085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code

a11bef0 - Scott Chacon, 11 months ago : first commit

الجدول 2-1 يعرض بعض الخيارات المفيدة التي يمكن إضافتها إلى الصيغة.

| **الخيار** | **وصف الخرج** |
| --- | --- |
| %H | Commit هاش |
| %h | الهاش الاعتمادي المختصر |
| %T | هاش الشجرة |
| %t | هاش الشجرة المختصر |
| %P | هاشات الوالد |
| %p | هاشات الوالد المختصرة |
| %an | اسم الكاتب |
| %ae | بريد الكاتب الإلكتروني |
| %ad | تاريخ الكاتب (يراعي الصيغة –date= option) |
| %ar | تاريخ الكاتب - نسبياً |
| %cn | اسم منفذ الاعتماد |
| %ce | بريد منفذ الاعتماد الإلكتروني |
| %cd | تاريخ منفذ الاعتماد |
| %cr | تاريخ منفذ الاعتماد - نسبياً |
| %s | العنوان |

قد تتسائل عن الاختلاف بين **الكاتب** AUTHOR و **منفذ الاعتماد** COMMITTER. الكاتب هو الشخص الذي كتب العمل بداية، بينما منفذ الاعتماد هو آخر شخص طبق العمل. لذا، إذا أرسلت باتشاً إلى مشروع وأحد المطورين الرئيسين قام بتطبيق الباتش، تأخذان كلاكما الاعتمادية - أنت ككاتب وهو كمنفذ اعتماد.

وتكون هذه الخيارات مفيدة أكثر بالترافق مع الخيار --graph. حيث يضيف هذا الأمر غراف ASCII بسيط يوضح تأريخ الأفرع و الدمج، لاحظ التالي:

$ git log --pretty=format:"%h %s" --graph

\* 2d3acf9 ignore errors from SIGCHLD on trap

\* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit

|\

| \* 420eac9 Added a method for getting the current branch.

\* | 30e367c timeout code and tests

\* | 5a09431 add timeout protection to grit

\* | e1193f8 support for heads with slashes in them

|/

\* d6016bc require time for xmlschema

\* 11d191e Merge branch 'defunkt' into local

يوجد مجموعة من الخيارات البسيطة مع أمر git log. يوضح الجدول 2-2 قائمة من الخيارات التي قمنا بتغطيتها حتى الآن وبعض صيغ التنسيق الشائعة والتي قد تكون مفيدة، وبجانبها شرح مبسط عن التغيير الذي تجريه على الخرج.

| **الخيار** | **الوصف** |
| --- | --- |
| -p | يظهر الباتش المدخل مع كل عملية commit. |
| --stat | يظهر إحصاءات حول التعديلات التي حصلت على الملفات مع كل عملية commit. |
| --shortstat | Display only the changed/insertions/deletions line from the --stat command. |
| --name-only | يظهر قائمة الملفات المعدلة. |
| --name-status | يظهر قائمة الملفات المعدلة عن طريق الإضافة والحذف وغير ذلك. |
| --abbrev-commit | يظهر أول مجموعة من هاش SHA-1 بدلاً عن كامل الأحرف 40. |
| --relative-date | Display the date in a relative format (for example, “2 weeks ago”) instead of using the full date format. |
| --graph | عرض غراف ASCII مع الخرج يظهر عمليات التفريع و الدمج. |
| --pretty | Show commits in an alternate format. Options include oneline, short, full, fuller, and format (where you specify your own format). |

### [تحديد خرج السجل](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-%D9%85%D8%B1%D8%A7%D8%AC%D8%B9%D8%A9-%D8%AA%D8%A3%D8%B1%D9%8A%D8%AE-%D8%B9%D9%85%D9%84%D9%8A%D8%A7%D8%AA-commit#%D8%AA%D8%AD%D8%AF%D9%8A%D8%AF-%D8%AE%D8%B1%D8%AC-%D8%A7%D9%84%D8%B3%D8%AC%D9%84)

بالإضافة تحديد شكل الخرج، يسمح git بأخذ مجموعة جزئية فقط من الخرج. وقد قمت بمشاهدة ذلك مسبقاً - الخيار -2 المستخدم لعرض آخر عمليتي commit. في الواقع يمكنك استخدام -<n> حيث n هي عدد صحيح لعرض آخر n عملية commit. في الحقيقة، لن تستخدمها على الغالب، لأن git يقوم بتمرير كامل الخرج لبرنامج تصفح لتتمكن من تصفح عمليات commit صفحة صفحة.

من الخيارات الهامة جداً، المحددات الزمنية مثل --since و --until. على سبيل المثال، هذا الأمر يعطيك قائمة بعمليات commit التي حصلت في آخر أسبوعين:

$ git log --since=2.weeks

ويعمل هذا الأمر مع أنواع كثيرة من الصيغ - يمكنك تحديد تاريخ محدد ("2008-01-15") أو تاريخ نسبي مثل "2 years 1 day 3 minutes ago".

يكطمط أيضاً تصفية قائمة عمليات commit من خلال محددات البحث. مثلاً خيار --author يقوم بتصفية النتائج وفق كاتب محدد، و --grep يقوم بالفلترة عن طريق كلمة مفتاحية. (لاحظ أنّه إذا قمت بإضافة خياراي author و grep، يجب عليك إضافة الخيار --all-match أو سيقوم git بعمل مطابقة مع أحدهما فقط).

آخر الخيارات الهامة التي يمكن تمريرها إلى الأمر git log كمصناف هي المسار. إذا قمت بتحديد مجلد أو اسم ملف، يمكنك تحديد الخرج إلى عمليات commit التي آثرت في هذا المسار. ودائماً ما يكون آخر خيار يوضع في log ويسبق بداشين double dashes (--) ليفصل المسارات عن الخيارات.

في الجدول 2-3 - نعرض الخيارات التي يمكن إضافتها مع log.

| **خيار** | **وصف** |
| --- | --- |
| -(n) | يظهر آن n عملية commit. |
| --since, --after | تحديد عمليات commit بعد التاريخ الذي يتلوها. |
| --until, --before | تحديد عمليات commit حتى التاريخ الذي يتلوها. |
| --author | فقط أظهر عمليات commit التابعة لهذا الكاتب. |
| --committer | فقط أظهر عمليات commit التابعة لهذا المعتمد. |

على سبيل المثال، إذا أردت رؤية أي عمليات commit عدلت ملفات الاختبار في تأريخ الكود المصدري والتي قام Junio Hamano بعملها ولم يتم دمجها في شهر أوكتوبر 2008، يمكن كتابة هذا الأمر:

$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \

--before="2008-11-01" --no-merges -- t/

5610e3b - Fix testcase failure when extended attribute

acd3b9e - Enhance hold\_lock\_file\_for\_{update,append}()

f563754 - demonstrate breakage of detached checkout wi

d1a43f2 - reset --hard/read-tree --reset -u: remove un

51a94af - Fix "checkout --track -b newbranch" on detac

b0ad11e - pull: allow "git pull origin $something:$cur

من بين 20 ألف عملية commit تقريباً في تأريخ git الخاص بهذا الكود المصدري، أظهر الأمر فقط عمليات الاعتماد الستة المطابقة لمحددات البحث.

### [Using a GUI to Visualize History](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-%D9%85%D8%B1%D8%A7%D8%AC%D8%B9%D8%A9-%D8%AA%D8%A3%D8%B1%D9%8A%D8%AE-%D8%B9%D9%85%D9%84%D9%8A%D8%A7%D8%AA-commit#Using-a-GUI-to-Visualize-History)

If you like to use a more graphical tool to visualize your commit history, you may want to take a look at a Tcl/Tk program called gitk that is distributed with Git. Gitk is basically a visual git log tool, and it accepts nearly all the filtering options that git log does. If you type gitk on the command line in your project, you should see something like Figure 2-2.

Figure 2-2. The gitk history visualizer.

You can see the commit history in the top half of the window along with a nice ancestry graph. The diff viewer in the bottom half of the window shows you the changes introduced at any commit you click.

# 4 مبادئ بوابة - التراجع عن الأشياء

## التراجع عن الأشياء

في أي مرحلة ، قد ترغب في التراجع عن شيء ما. هنا ، سنراجع بعض الأدوات الأساسية للتراجع عن التغييرات التي أجريتها. كن حذرًا ، لأنه لا يمكنك دائمًا التراجع عن بعض هذه التراجع. هذه واحدة من المناطق القليلة في Git حيث قد تفقد بعض العمل إذا قمت بذلك بشكل خاطئ.

### [تغيير التزامك الأخير](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Undoing-Things#Changing-Your-Last-Commit)

تحدث إحدى عمليات التراجع الشائعة عندما تلتزم مبكرًا وربما تنسى إضافة بعض الملفات أو تخبط في رسالة الالتزام. إذا كنت ترغب في تجربة هذا الالتزام مرة أخرى ، يمكنك تشغيل الالتزام باستخدام --amendالخيار:

$ git commit --amend

يأخذ هذا الأمر منطقة التدريج الخاصة بك ويستخدمها للالتزام. إذا لم تقم بإجراء أي تغييرات منذ آخر التزام (على سبيل المثال ، يمكنك تشغيل هذا الأمر فورًا بعد الالتزام السابق) ، فستظهر لقطةتك تمامًا وستكون كل شيء ستغيره هي رسالة الالتزام.

يتم تشغيل محرر التزام الرسالة نفسه ، ولكنه يحتوي بالفعل على رسالة الالتزام السابق. يمكنك تعديل الرسالة كما هي دائمًا ، لكنها تحل محل التزامك السابق.

على سبيل المثال ، إذا قمت بالالتزام ثم أدركت أنك نسيت إجراء التغييرات في ملف تريد إضافته إلى هذا الالتزام ، يمكنك القيام بشيء من هذا القبيل:

$ git commit -m 'initial commit'

$ git add forgotten\_file

$ git commit --amend

تنتهي كل هذه الأوامر الثلاثة بالتزام واحد - حيث يحل الالتزام الثاني محل نتائج الأول.

### [unstaging ملف مرحلي](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Undoing-Things#Unstaging-a-Staged-File)

يوضح القسمان التاليان كيفية تغيير منطقة التدريج وتغييرات دليل العمل. الجزء الجميل هو أن الأمر الذي تستخدمه لتحديد حالة هاتين المنطقتين يذكرك أيضًا بكيفية التراجع عن التغييرات التي أجريتها. على سبيل المثال ، لنفترض أنك غيرت ملفين وتريد الالتزام بهما كتغييرين منفصلين ، لكنك قمت بطريق الخطأ git add \*وكتابتهما على حد سواء. كيف يمكنك unstage واحد من الاثنين؟ و git statusالقيادة يذكرك:

$ git add .

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: README.txt

# modified: benchmarks.rb

#

مباشرة أسفل النص "التغييرات المطلوب الالتزام بها" ، يشير ذلك git reset HEAD <file>...إلى استخدام unstage. لذلك ، دعونا نستخدم هذه النصيحة لإلغاء تثبيت ملف benchmarks.rb:

$ git reset HEAD benchmarks.rb

benchmarks.rb: locally modified

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: README.txt

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: benchmarks.rb

#

الأمر غريب بعض الشيء ، لكنه يعمل. يتم تعديل ملف benchmarks.rb لكن مرة أخرى بدون تغيير.

### [إلغاء تعديل ملف معدّل](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Undoing-Things#Unmodifying-a-Modified-File)

ماذا لو كنت تدرك أنك لا تريد الاحتفاظ بالتغييرات التي قمت بإجرائها على ملف benchmarks.rb؟ كيف يمكنك بسهولة تعديله - إعادته إلى ما بدا عليه عندما ارتكبت آخر مرة (أو استنسخته في البداية ، أو حصلت عليه في دليل العمل الخاص بك)؟ لحسن الحظ ، git statusيخبرك كيف تفعل ذلك أيضًا. في إخراج المثال الأخير ، تبدو المنطقة غير المُدرجة كما يلي:

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: benchmarks.rb

#

إنه يخبرك بوضوح تام كيف تتجاهل التغييرات التي أجريتها (على الأقل ، الإصدارات الأحدث من Git ، 1.6.1 والإصدارات الأحدث ، قم بذلك - إذا كان لديك إصدار أقدم ، فنحن نوصي بشدة بالترقية لتحديث بعض هذه التغييرات ميزات سهولة الاستخدام أجمل). دعونا نفعل ما تقول:

$ git checkout -- benchmarks.rb

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: README.txt

#

يمكنك أن ترى أن التغييرات قد عادت. يجب أن تدرك أيضًا أن هذا أمر خطير: أي تغييرات قمت بإجرائها على هذا الملف قد ولت - لقد قمت فقط بنسخ ملف آخر فوقه. لا تستخدم هذا الأمر مطلقًا إلا إذا كنت تعلم تمامًا أنك لا تريد الملف. إذا كنت بحاجة فقط لإخراجها من الطريق ، فسوف نذهب إلى الأمام والخفق في الفصل التالي ؛ هذه هي عموما أفضل الطرق للذهاب.

تذكر أنه يمكن دائمًا استرداد أي شيء يتم ارتكابه في Git. حتى الالتزامات التي كانت في الفروع التي تم حذفها أو الالتزامات التي تم الكتابة عليها مع --amendالالتزام يمكن استردادها (انظر الفصل 9 لاستعادة البيانات). ومع ذلك ، فإن أي شيء تخسره ولم يلتزم به أبدًا لن يُرجح رؤيته أبدًا.

[السابق](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-%D9%85%D8%B1%D8%A7%D8%AC%D8%B9%D8%A9-%D8%AA%D8%A3%D8%B1%D9%8A%D8%AE-%D8%B9%D9%85%D9%84%D9%8A%D8%A7%D8%AA-commit) | [التالى](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Working-with-Remotes)

[حول هذا الموقع](https://git-scm.com/site)  
بقع ، اقتراحات ، وتعليقات هي موضع ترحيب.

جيت عضو في [برنامج الحفاظ على حرية البرمجيات](https://git-scm.com/sfc)

# .5 مبادئ بوابة - العمل مع أجهزة التحكم عن بعد

## العمل مع أجهزة التحكم عن بعد

لتتمكن من التعاون في أي مشروع Git ، تحتاج إلى معرفة كيفية إدارة مستودعاتك عن بُعد. المستودعات عن بعد هي إصدارات من مشروعك يتم استضافتها على الإنترنت أو الشبكة في مكان ما. يمكنك الحصول على العديد منها ، يكون كل منها إما للقراءة فقط أو للقراءة / الكتابة لك. يتضمن التعاون مع الآخرين إدارة هذه المستودعات عن بُعد ودفع البيانات وسحبها من وإلى عندما تريد مشاركة العمل. تتضمن إدارة المستودعات عن بعد معرفة كيفية إضافة مستودعات عن بعد وإزالة أجهزة التحكم عن بعد التي لم تعد صالحة وإدارة مختلف الفروع عن بعد وتعريفها على أنها متتبعة أم لا ، وأكثر من ذلك. في هذا القسم ، سنغطي مهارات الإدارة عن بُعد.

### [عرض أجهزة التحكم عن بعد الخاصة بك](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Working-with-Remotes#Showing-Your-Remotes)

لمعرفة الخوادم البعيدة التي قمت بتكوينها ، يمكنك تشغيل الأمر git remote. يسرد الأسماء المختصرة لكل مقبض بعيد قمت بتحديده. إذا قمت باستنساخ مستودعك ، فيجب أن ترى على الأقل الأصل - هذا هو الاسم الافتراضي الذي يعطيه Git للخادم الذي استنسخت منه:

$ git clone git://github.com/schacon/ticgit.git

Initialized empty Git repository in /private/tmp/ticgit/.git/

remote: Counting objects: 595, done.

remote: Compressing objects: 100% (269/269), done.

remote: Total 595 (delta 255), reused 589 (delta 253)

Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.

Resolving deltas: 100% (255/255), done.

$ cd ticgit

$ git remote

origin

يمكنك أيضًا تحديد -v، والذي يوضح لك عنوان URL الذي قام Git بتخزينه لتوسيع نطاق الاسم المختصر إلى:

$ git remote -v

origin git://github.com/schacon/ticgit.git

إذا كان لديك أكثر من جهاز تحكم عن بعد ، فإن الأمر يسردهم جميعًا. على سبيل المثال ، يبدو مستودع Grit الخاص بي شيئًا كهذا.

$ cd grit

$ git remote -v

bakkdoor git://github.com/bakkdoor/grit.git

cho45 git://github.com/cho45/grit.git

defunkt git://github.com/defunkt/grit.git

koke git://github.com/koke/grit.git

origin git@github.com:mojombo/grit.git

هذا يعني أنه يمكننا سحب المساهمات من أي من هؤلاء المستخدمين بسهولة تامة. لكن لاحظ أن جهاز التحكم عن بُعد هو عنوان URL لـ SSH فقط ، لذلك فهو الوحيد الذي يمكنني الدفع إليه (سنقوم بتغطية سبب ذلك في الفصل 4).

### [إضافة مستودعات عن بعد](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Working-with-Remotes#Adding-Remote-Repositories)

لقد ذكرت وأعطيت بعض العروض التوضيحية لإضافة مستودعات عن بعد في الأقسام السابقة ، ولكن إليك كيفية القيام بذلك بشكل صريح. لإضافة مستودع Git بعيد جديد كاسم قصير يمكنك الرجوع إليه بسهولة ، قم بتشغيل git remote add [shortname] [url]:

$ git remote

origin

$ git remote add pb git://github.com/paulboone/ticgit.git

$ git remote -v

origin git://github.com/schacon/ticgit.git

pb git://github.com/paulboone/ticgit.git

يمكنك الآن استخدام السلسلة pb في سطر الأوامر بدلاً من عنوان URL بالكامل. على سبيل المثال ، إذا كنت ترغب في جلب جميع المعلومات التي يمتلكها بولس ولكنك لا تملكها حتى الآن في مستودعك ، يمكنك تشغيل git fetch pb:

$ git fetch pb

remote: Counting objects: 58, done.

remote: Compressing objects: 100% (41/41), done.

remote: Total 44 (delta 24), reused 1 (delta 0)

Unpacking objects: 100% (44/44), done.

From git://github.com/paulboone/ticgit

\* [new branch] master -> pb/master

\* [new branch] ticgit -> pb/ticgit

يمكن الوصول إلى فرع Paul الرئيسي محليًا pb/master- يمكنك دمجه في أحد فروعك ، أو يمكنك التحقق من فرع محلي في تلك المرحلة إذا كنت ترغب في فحصه.

### [جلب وسحب من أجهزة التحكم عن بعد](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Working-with-Remotes#Fetching-and-Pulling-from-Your-Remotes)

كما رأيت للتو ، للحصول على البيانات من مشاريعك البعيدة ، يمكنك تشغيل:

$ git fetch [remote-name]

يخرج الأمر إلى هذا المشروع البعيد ويقوم بسحب جميع البيانات من هذا المشروع البعيد الذي لم يكن لديك بعد. بعد القيام بذلك ، يجب أن يكون لديك مراجع لجميع الفروع من جهاز التحكم عن بعد ، والتي يمكنك دمجها أو تفتيشها في أي وقت. (سوف نتعرف على الفروع وكيفية استخدامها بمزيد من التفصيل في الفصل 3.)

إذا قمت بنسخ مستودع ، فإن الأمر يضيف تلقائيًا مستودع التخزين البعيد هذا تحت اسم الأصل. لذا ، يمكنك git fetch originإحضار أي عمل جديد تم دفعه إلى هذا الخادم منذ استنساخه (أو آخر عملية جلب منه). من المهم ملاحظة أن أمر الجلب يسحب البيانات إلى مستودعك المحلي - لا يدمجها تلقائيًا مع أي من عملك أو يعدل ما تعمل عليه حاليًا. يجب عليك دمجها يدويًا في عملك عندما تكون جاهزًا.

إذا كان لديك فرع تم إعداده لتتبع فرع بعيد (انظر القسم التالي والفصل 3 للحصول على مزيد من المعلومات) ، يمكنك استخدام git pullالأمر لجلب فرع ثم دمجه تلقائيًا في فرعك الحالي. قد يكون هذا سير عمل أسهل أو أكثر راحة لك ؛ افتراضيًا ، يقوم git cloneالأمر تلقائيًا بإعداد الفرع الرئيسي المحلي لتتبع الفرع الرئيسي البعيد على الخادم الذي استنسخت منه (على افتراض أن جهاز التحكم عن بعد له فرع رئيسي). يعمل التشغيل git pullبشكل عام على جلب البيانات من الخادم الذي استنسخت منه أصلاً ، ويحاول تلقائيًا دمجها في الشفرة التي تعمل عليها حاليًا.

### [دفع إلى أجهزة التحكم عن بعد الخاصة بك](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Working-with-Remotes#Pushing-to-Your-Remotes)

عندما يكون لديك مشروعك في نقطة تريد مشاركتها ، يجب عليك دفعها إلى الأعلى. الأمر في ذلك بسيط: git push [remote-name] [branch-name]. إذا كنت ترغب في دفع فرعك الرئيسي إلى originالخادم الخاص بك (مرة أخرى ، يقوم الاستنساخ عمومًا بإعداد كلا الاسمين لك تلقائيًا) ، ثم يمكنك تشغيل هذا لدفع عملك إلى الخادم مرة أخرى:

$ git push origin master

يعمل هذا الأمر فقط إذا كنت مستنسخة من خادم لديك حق الوصول للكتابة وإذا لم يدفعه أحد في هذه الأثناء. إذا استنسخت أنت وشخص آخر في نفس الوقت ودفعوا إلى أعلى ثم دفعت إلى أعلى ، سيتم رفض دفعتك عن حق. يجب عليك هدم عملهم أولاً وإدماجه في عملك قبل أن يُسمح لك بالدفع. انظر الفصل 3 للحصول على مزيد من المعلومات التفصيلية حول كيفية الدفع إلى الخوادم البعيدة.

### [فحص جهاز التحكم عن بعد](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Working-with-Remotes#Inspecting-a-Remote)

إذا كنت ترغب في رؤية المزيد من المعلومات حول جهاز تحكم عن بعد معين ، يمكنك استخدام git remote show [remote-name]الأمر. إذا قمت بتشغيل هذا الأمر باستخدام اسم قصير معين ، على سبيل المثال origin، ستحصل على شيء مثل هذا:

$ git remote show origin

\* remote origin

URL: git://github.com/schacon/ticgit.git

Remote branch merged with 'git pull' while on branch master

master

Tracked remote branches

master

ticgit

يسرد عنوان URL الخاص بالمستودع عن بعد بالإضافة إلى معلومات فرع التتبع. يخبرك الأمر بشكل مفيد أنه إذا كنت في الفرع الرئيسي وقمت بتشغيله git pull، فسيتم دمجه تلقائيًا في الفرع الرئيسي على جهاز التحكم عن بعد بعد أن يجلب جميع المراجع عن بُعد. كما يسرد جميع المراجع عن بعد التي هدمها.

هذا مثال بسيط من المحتمل أن تصادفه. عند استخدام Git بشكل أكثر كثافة ، قد ترى المزيد من المعلومات من git remote show:

$ git remote show origin

\* remote origin

URL: git@github.com:defunkt/github.git

Remote branch merged with 'git pull' while on branch issues

issues

Remote branch merged with 'git pull' while on branch master

master

New remote branches (next fetch will store in remotes/origin)

caching

Stale tracking branches (use 'git remote prune')

libwalker

walker2

Tracked remote branches

acl

apiv2

dashboard2

issues

master

postgres

Local branch pushed with 'git push'

master:master

يعرض هذا الأمر الفرع الذي يتم دفعه تلقائيًا عند تشغيله git pushعلى فروع معينة. كما يوضح لك الفروع البعيدة الموجودة على الخادم التي ليس لديك بعد ، والفروع البعيدة التي تمت إزالتها من الخادم ، والفروع المتعددة التي يتم دمجها تلقائيًا عند التشغيل git pull.

### [إزالة وإعادة تسمية أجهزة التحكم عن بعد](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Working-with-Remotes#Removing-and-Renaming-Remotes)

إذا كنت ترغب في إعادة تسمية مرجع ، في الإصدارات الأحدث من Git يمكنك تشغيل git remote renameلتغيير الاسم المختصر لجهاز التحكم عن بعد. على سبيل المثال، إذا كنت تريد إعادة تسميته pbل paul، يمكنك أن تفعل ذلك مع git remote rename:

$ git remote rename pb paul

$ git remote

origin

paul

تجدر الإشارة إلى أن هذا يغير أسماء فروعك البعيدة ، أيضًا. ما كان يتم الرجوع إليه في pb/masterالآن في paul/master.

إذا كنت تريد إزالة مرجع لسبب ما - لقد قمت بنقل الخادم أو لم تعد تستخدم مرآة معينة ، أو ربما لم يعد مساهمًا يساهم - يمكنك استخدام git remote rm:

$ git remote rm paul

$ git remote

origin

# 6 مبادئ Git - Tagging

## علامات

مثل معظم VCSs ، لدى Git القدرة على تمييز نقاط محددة في التاريخ باعتبارها مهمة. بشكل عام ، يستخدم الأشخاص هذه الوظيفة لتمييز نقاط الإصدار (الإصدار 1.0 وما إلى ذلك). في هذا القسم ، ستتعلم كيفية سرد العلامات المتاحة ، وكيفية إنشاء علامات جديدة ، وأنواع العلامات المختلفة.

### [قائمة العلامات الخاصة بك](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Tagging#Listing-Your-Tags)

قائمة العلامات المتاحة في بوابة واضحة. فقط اكتب git tag:

$ git tag

v0.1

v1.3

يسرد هذا الأمر العلامات بالترتيب الأبجدي ؛ الترتيب الذي تظهر به ليس له أهمية حقيقية.

يمكنك أيضًا البحث عن علامات بنمط معين. على سبيل المثال ، يحتوي Git source repo على أكثر من 240 علامة. إذا كنت مهتمًا فقط بالنظر إلى السلسلة 1.4.2 ، يمكنك تشغيل هذا:

$ git tag -l 'v1.4.2.\*'

v1.4.2.1

v1.4.2.2

v1.4.2.3

v1.4.2.4

### [إنشاء العلامات](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Tagging#Creating-Tags)

يستخدم Git نوعين رئيسيين من العلامات: خفيف الوزن ومشروح. تشبه العلامة خفيفة الوزن إلى حد كبير الفرع الذي لا يتغير - إنه مجرد مؤشر لالتزام محدد. ومع ذلك ، يتم تخزين العلامات المشروحة ككائنات كاملة في قاعدة بيانات Git. انهم الاختباري. يحتوي على اسم tagger والبريد الإلكتروني والتاريخ ؛ لديك رسالة علامات. ويمكن توقيعه والتحقق منه مع GNU Privacy Guard (GPG). يوصى عمومًا بإنشاء علامات توضيحية حتى تتمكن من الحصول على كل هذه المعلومات ؛ ولكن إذا كنت تريد علامة مؤقتة أو لسبب ما لا ترغب في الاحتفاظ بالمعلومات الأخرى ، تتوفر علامات خفيفة الوزن أيضًا.

### [العلامات المشروحة](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Tagging#Annotated-Tags)

إنشاء علامة توضيحية في Git أمر بسيط. أسهل طريقة هي تحديد -aوقت تشغيل tagالأمر:

$ git tag -a v1.4 -m 'my version 1.4'

$ git tag

v0.1

v1.3

v1.4

و -mيحدد رسالة علامات، والتي يتم تخزينها مع العلامة. إذا لم تحدد رسالة لعلامة توضيحية ، فسيقوم Git بتشغيل المحرر الخاص بك حتى تتمكن من كتابتها.

يمكنك مشاهدة بيانات العلامة مع الالتزام الذي تم وضع علامة عليه باستخدام git showالأمر:

$ git show v1.4

tag v1.4

Tagger: Scott Chacon <schacon@gee-mail.com>

Date: Mon Feb 9 14:45:11 2009 -0800

my version 1.4

commit 15027957951b64cf874c3557a0f3547bd83b3ff6

Merge: 4a447f7... a6b4c97...

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'

يعرض ذلك معلومات الوسم والتاريخ الذي تم فيه تمييز الالتزام ورسالة التعليق التوضيحي قبل إظهار معلومات الالتزام.

### [توقيع العلامات](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Tagging#Signed-Tags)

يمكنك أيضًا توقيع علاماتك باستخدام GPG ، على افتراض أن لديك مفتاحًا خاصًا. كل ما عليك فعله هو الاستخدام -sبدلاً من -a:

$ git tag -s v1.5 -m 'my signed 1.5 tag'

You need a passphrase to unlock the secret key for

user: "Scott Chacon <schacon@gee-mail.com>"

1024-bit DSA key, ID F721C45A, created 2009-02-09

إذا قمت بتشغيل git showهذه العلامة ، يمكنك أن ترى توقيع GPG الخاص بك مرفقًا بها:

$ git show v1.5

tag v1.5

Tagger: Scott Chacon <schacon@gee-mail.com>

Date: Mon Feb 9 15:22:20 2009 -0800

my signed 1.5 tag

-----BEGIN PGP SIGNATURE-----

Version: GnuPG v1.4.8 (Darwin)

iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN

Ki0An2JeAVUCAiJ7Ox6ZEtK+NvZAj82/

=WryJ

-----END PGP SIGNATURE-----

commit 15027957951b64cf874c3557a0f3547bd83b3ff6

Merge: 4a447f7... a6b4c97...

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'

بعد ذلك بقليل ، سوف تتعلم كيفية التحقق من العلامات الموقعة.

### [العلامات خفيفة الوزن](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Tagging#Lightweight-Tags)

هناك طريقة أخرى لوضع العلامات على ارتكابها وهي باستخدام علامة خفيفة الوزن. هذا هو الأساس المجموع الاختباري الالتزام المخزن في ملف - لا يتم الاحتفاظ بأي معلومات أخرى. لإنشاء علامة خفيفة الوزن، لا توريد -a، -sأو -mالخيار:

$ git tag v1.4-lw

$ git tag

v0.1

v1.3

v1.4

v1.4-lw

v1.5

هذه المرة ، إذا كنت تعمل git showعلى العلامة ، فلن ترى معلومات العلامة الإضافية. الأمر يظهر فقط الالتزام:

$ git show v1.4-lw

commit 15027957951b64cf874c3557a0f3547bd83b3ff6

Merge: 4a447f7... a6b4c97...

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'

### [التحقق من العلامات](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Tagging#Verifying-Tags)

للتحقق من علامة موقعة ، تستخدمها git tag -v [tag-name]. يستخدم هذا الأمر GPG للتحقق من التوقيع. تحتاج إلى المفتاح العمومي للموقِّع في حلقة المفاتيح لكي يعمل هذا بشكل صحيح:

$ git tag -v v1.4.2.1

object 883653babd8ee7ea23e6a5c392bb739348b1eb61

type commit

tag v1.4.2.1

tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.

gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A

gpg: Good signature from "Junio C Hamano <junkio@cox.net>"

gpg: aka "[jpeg image of size 1513]"

Primary key fingerprint: 3565 2A26 2040 E066 C9A7 4A7D C0C6 D9A4 F311 9B9A

إذا لم يكن لديك المفتاح العام للموقع ، فستحصل على شيء مثل هذا بدلاً من ذلك:

gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A

gpg: Can't check signature: public key not found

error: could not verify the tag 'v1.4.2.1'

### [العلامات في وقت لاحق](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Tagging#Tagging-Later)

يمكنك أيضًا تمييز التعهدات بعد تجاوزها. لنفترض أن سجل الالتزام يشبه هذا:

$ git log --pretty=oneline

15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'

a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support

0d52aaab4479697da7686c15f77a3d64d9165190 one more thing

6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'

0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function

4682c3261057305bdd616e23b64b0857d832627b added a todo file

166ae0c4d3f420721acbb115cc33848dfcc2121a started write support

9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile

964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo

8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme

الآن ، لنفترض أنك نسيت وضع علامة على المشروع في الإصدار 2.1 ، والذي كان على التزام "rakefile المحدث". يمكنك إضافته بعد الحقيقة. لوضع علامة على ذلك الالتزام ، يمكنك تحديد مجموع الالتزام (أو جزء منه) في نهاية الأمر:

$ git tag -a v1.2 9fceb02

يمكنك أن ترى أنك وضعت علامة على الالتزام:

$ git tag

v0.1

v1.2

v1.3

v1.4

v1.4-lw

v1.5

$ git show v1.2

tag v1.2

Tagger: Scott Chacon <schacon@gee-mail.com>

Date: Mon Feb 9 15:32:16 2009 -0800

version 1.2

commit 9fceb02d0ae598e95dc970b74767f19372d61af8

Author: Magnus Chacon <mchacon@gee-mail.com>

Date: Sun Apr 27 20:43:35 2008 -0700

updated rakefile

...

### [تبادل العلامات](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Tagging#Sharing-Tags)

بشكل افتراضي ، git pushلا ينقل الأمر العلامات إلى خوادم بعيدة. يجب عليك دفع العلامات صراحةً إلى خادم مشترك بعد إنشائها. هذه العملية تشبه مشاركة الفروع البعيدة - يمكنك تشغيلها git push origin [tagname].

$ git push origin v1.5

Counting objects: 50, done.

Compressing objects: 100% (38/38), done.

Writing objects: 100% (44/44), 4.56 KiB, done.

Total 44 (delta 18), reused 8 (delta 1)

To git@github.com:schacon/simplegit.git

\* [new tag] v1.5 -> v1.5

إذا كان لديك الكثير من العلامات التي تريد رفعها دفعة واحدة ، فيمكنك أيضًا استخدام --tagsالخيار git pushللأمر. سيؤدي هذا إلى نقل جميع علاماتك إلى الخادم البعيد غير الموجود بالفعل.

$ git push origin --tags

Counting objects: 50, done.

Compressing objects: 100% (38/38), done.

Writing objects: 100% (44/44), 4.56 KiB, done.

Total 44 (delta 18), reused 8 (delta 1)

To git@github.com:schacon/simplegit.git

\* [new tag] v0.1 -> v0.1

\* [new tag] v1.2 -> v1.2

\* [new tag] v1.4 -> v1.4

\* [new tag] v1.4-lw -> v1.4-lw

\* [new tag] v1.5 -> v1.5

الآن ، عندما يقوم شخص آخر باستنساخ أو سحب من مستودعك ، سيحصل على جميع علاماتك أيضًا.

# 7 مبادئ بوابة - نصائح والخدع

## النصائح والحيل

قبل أن ننتهي من هذا الفصل حول Git الأساسي ، قد تجعل بعض النصائح والحيل القليلة من تجربة Git أكثر بساطة أو أسهل أو مألوفة. يستخدم العديد من الأشخاص Git دون استخدام أي من هذه النصائح ، ولن نشير إليها أو نفترض أنك استخدمتها لاحقًا في الكتاب ؛ ولكن ربما يجب أن تعرف كيفية القيام بها.

### [لصناعة السيارات في استكمال](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Tips-and-Tricks#Auto-Completion)

إذا كنت تستخدم Bash shell ، فإن Git يأتي مع برنامج نصي للإكمال التلقائي يمكنك تمكينه. قم بتنزيل كود مصدر Git ، وانظر في contrib/completionالدليل ؛ يجب أن يكون هناك ملف يسمى git-completion.bash. انسخ هذا الملف إلى الدليل الرئيسي الخاص بك ، وأضفه إلى ملفك .bashrc:

source ~/.git-completion.bash

إذا كنت ترغب في إعداد Git لإكمال Bash shell تلقائيًا لجميع المستخدمين ، انسخ هذا البرنامج النصي إلى /opt/local/etc/bash\_completion.dالدليل على أنظمة Mac أو إلى /etc/bash\_completion.d/الدليل على أنظمة Linux. هذا هو دليل البرامج النصية التي سيقوم Bash بتحميلها تلقائيًا لتوفير عمليات إكمال shell.

إذا كنت تستخدم Windows مع Git Bash ، وهو الإعداد الافتراضي عند تثبيت Git على Windows مع msysGit ، فيجب أن يكون الإكمال التلقائي مسبقًا.

اضغط على المفتاح Tab عند كتابة أمر Git ، ويجب أن يعرض مجموعة من الاقتراحات لتختار منها:

$ git co<tab><tab>

commit config

في هذه الحالة ، تشير كتابة git ثم الضغط على المفتاح Tab مرتين إلى الالتزام والتهيئة. مضيفا m<tab>اكتمال git commitتلقائيا.

هذا يعمل أيضا مع الخيارات ، والتي ربما تكون أكثر فائدة. على سبيل المثال ، إذا كنت تقوم بتشغيل git logأمر ولا يمكنك تذكر أحد الخيارات ، فيمكنك البدء في كتابته والضغط على Tab لمعرفة المطابقات:

$ git log --s<tab>

--shortstat --since= --src-prefix= --stat --summary

هذه خدعة جميلة وقد توفر لك بعض الوقت وقراءة الوثائق.

### [بوابة الأسماء المستعارة](https://git-scm.com/book/ar/v1/%D9%85%D8%A8%D8%A7%D8%AF%D8%A6-Git-Tips-and-Tricks#Git-Aliases)

بوابة لا تستنتج الأمر الخاص بك إذا قمت بكتابته جزئيًا. إذا كنت لا ترغب في كتابة النص الكامل لكل أمر من أوامر Git ، فيمكنك إعداد اسم مستعار لكل أمر باستخدام بسهولة git config. فيما يلي بعض الأمثلة التي قد ترغب في إعدادها:

$ git config --global alias.co checkout

$ git config --global alias.br branch

$ git config --global alias.ci commit

$ git config --global alias.st status

هذا يعني أنه ، على سبيل المثال ، بدلاً من الكتابة git commit، تحتاج فقط إلى الكتابة git ci. أثناء الاستمرار في استخدام Git ، ستستخدم على الأرجح أوامر أخرى بشكل متكرر أيضًا ؛ في هذه الحالة ، لا تتردد في إنشاء أسماء مستعارة جديدة.

يمكن أن تكون هذه التقنية مفيدة للغاية في إنشاء أوامر تعتقد أنها يجب أن تكون موجودة. على سبيل المثال ، لتصحيح مشكلة قابلية الاستخدام التي واجهتها أثناء إلغاء تثبيت ملف ، يمكنك إضافة اسمك المستعار المستعار إلى Git:

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

هذا يجعل الأمرين التاليين مكافئين:

$ git unstage fileA

$ git reset HEAD fileA

هذا يبدو أكثر وضوحا بعض الشيء. من الشائع أيضًا إضافة lastأمر ، مثل هذا:

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

بهذه الطريقة ، يمكنك رؤية الالتزام الأخير بسهولة:

$ git last

commit 66938dae3329c7aebe598c2246a8e6af90d04646

Author: Josh Goebel <dreamer3@example.com>

Date: Tue Aug 26 19:48:51 2008 +0800

test for current head

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

كما يمكنك أن تقول ، فإن Git يستبدل الأمر الجديد بكل ما تسميه. ومع ذلك ، ربما تريد تشغيل أمر خارجي ، بدلاً من أمر فرعي Git. في هذه الحالة ، تبدأ الأمر !بحرف. يعد هذا مفيدًا إذا كتبت أدواتك الخاصة التي تعمل مع مستودع Git. يمكننا أن نظهر من خلال الاسم المستعار git visualلتشغيل gitk:

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

# 8 مبادئ بوابة - ملخص

## ملخص

في هذه المرحلة ، يمكنك القيام بجميع عمليات Git الأساسية المحلية - إنشاء مستودع أو استنساخه ، وإجراء تغييرات ، وتنظيم هذه التغييرات والالتزام بها ، وعرض محفوظات جميع التغييرات التي مر بها المستودع. بعد ذلك ، سنغطي ميزة Git's killer: نموذجها المتفرّع.