احترف Git

Scott Chacon

Ben Straub

Version 2.1.421, 2024-08-16

توطئة

تمهيد المترجم

كتاب احترف جت من «أمهات الكتب» في صناعة البرمجيات. فهو يعلّم نظامًا من أشهر أنظمة إدارة النسخ على الإطلاق، هو جت، ويعلّم فن التطوير باستخدام نظام إدارة نسخ موزع. فليس يقتصر نفعه على من يريد استخدام Git و GitHub، بل يمتد إلى من أراد استخدام GitLab أو حتى نظام آخر مثل Mercurial.

دراسة هذا الكتاب ضرورية لأي مبرمج.

وقد حرصت على تقليل الإنجليزية في الكتاب، إلا مما يلزم (مثل الأوامر ونواتجها) فترجمت ما نحتاجه منها مع إبقاء أصله كله، حتى يفهم الكتاب العامة والخاصة فلا نجعل إتقان الإنجليزية شرطًا لفهم كتاب عربي. على أن معرفة الإنجليزية ضرورة للمبرمجين ولمستخدمي جت لا محالة.

ودراسة العلم بلغة الإنسان الأولى تساعده على الفهم العميق لما يتعلم. ولا سبيل للنهضة إلا بنقل العلوم واستيعابها بلغتنا.

ولا يعني هذا إهمال اللغات الأجنبية؛ فتعلمها ضرورة.

أما عن الألفاظ: فلغة أي فن من الفنون هي دائما غريبة على مسامع من لم يعتادوها. حتى الإنجليزية؛ اسأل شخصا وُلد وعاش وشاخ في بلد يتحدث هذه اللغة، ولم يسمع أو يقرأ في حياته بأي لغة أخرى، عن عبارات مثل "fork a process" أو "clone a repository"...

والأمثلة على ذلك كثيرة، منها: «لا يدخل الزحاف في شيء من الأوتاد، وإنما يدخل في الأسباب خاصةً.» وهذا من كتاب في علم العَروض. والذي أتى بمصطلحاته، مثل الأسباب والأوتاد والزحاف، هو الخليل ابن أحمد، واضع أسس علم النحو، وجامع معجم العين، أول معجم عربي.

ومن درس علم التجويد سيجد أن كل لفظ يُعرَّف أولا بمعناه في اللغة ثم بمعناه في علم التجويد، فيقولون مثلا «الإخفاء لغةً هو...، واصطلاحًا هو...».

فلكل علم لغته الخاصة داخل اللغة العامة. ومثال آخر على ذلك لغة الحساب الحديث، ففيها الاسم «زائد» يُستعمل حرف عطف، والفعل المنصرف «يساوي» يُستعمل جامدًا ولا ينصرف أبدا، فنقول «اثنين زائد ثلاثة يساوي خمسة». فهل هذه الجملة ليست عربية؟

ولن يجد العربي غير المتخصص عبارة مثل «استنسخ مستودعًا» أو «أودِع التعديلات المؤهلة» أغرب مما يجد الإنجليزي غير المتخصص عبارة مثل "clone a repository" أو "commit the staged changes". فكل فن له لغته الخاصة داخل اللغة العامة.

ولسنا اليوم بصدد تغيير قواعد اللغة هنا كما فعل أهل الحساب، إنما نحتاج فقط إلى توسيع معاني بعض الكلمات كما فعل أهل التجويد. ولن نأتِ بمصطلحات كثيرة كما فعل الخليل رحمه الله في علمَي العَروض والقافية.

وأرجو من القارئ الكريم التماس العذر لي إن بدا مني خطأ، وأن ينبهني إليه على صفحة مسائل ترجمة الكتاب على جت‌هب.

تمهيد Scott Chacon

مرحبا بكم في الإصدارة الثانية من كتاب احترف جت. نُشرت الإصدارة الأولى منذ ما يزيد الآن على أربعة أعوام. وتغيّرت أمور كثيرة منذ ذلك الوقت، إلا أن الكثير من الأشياء المهمة لم تتغير. ومع أن أكثر الأوامر والمفاهيم الأساسية لا تزال سارية اليوم — لأن الفريق الأساسي القائم على جت يقوم بمجهود خيالي للحفاظ على التوافقية مع الإصدارات السابقة (backward compatibility) — فقد ظهرت بعض الإضافات والتغييرات البارزة في المجتمع الذي حول جت. المراد من الإصدارة الثانية من هذا الكتاب هو تناول تلك التغييرات، وتحديثه ليكون أكثر إفادةً للمستخدم الجديد.

عندما كتبت الإصدارة الأولى، كان جت صعب الاستخدام نسبيا، وبالكاد يستخدمه الخارقون (hackers) الأشداء. كان بدأ ينتشر في بعض المجتمعات، لكن لم تقترب حاله مما صار عليه اليوم من الوجود المطلق في كل مكان. ثم بدأت تستخدمه أكثر مجتمعات المصادر المفتوحة. ولقد تقدم جت تقدما مذهلا: في عمله على ويندوز، وفي انفجار أعداد الواجهات الرسومية له على جميع المنصات، وفي دعم بيئات التطوير له، وفي الاستخدام التجاري. وكتاب احترف جت الذي جاء منذ أربعة أعوام لم يكن يعلم شيئًا من هذا كله. فكان ذكر جميع تلك الآفاق الجديدة في مجتمع جت من أكبر همومنا في هذه الإصدارة الجديدة.

وأيضا تضاعف بسرعة مجتمع المصادر المفتوحة الذي يستخدم جت. فعندما جلست أول مرة أكتب هذا الكتاب منذ قرابة خمسة أعوام (فقد استغرق الأمر مني وقتا لإطلاق الإصدارة الأولى)، كنت وقتئذٍ قد بدأت العمل في شركة مغمورة جدا تطوّر موقع استضافة جت يسمى جِتهَب (GitHub). وعند نشره، لم يكن للموقع سوى نحو بضعة آلاف مستخدم، ولم نكن إلا أربعة موظفين نعمل عليه. وبينما أنا أكتب هذه المقدمة الآن، إذ أعلن جتهب استضافتنا للمشروع رقم عشرة ملايين، مع قرابة خمسة ملايين حساب مطوِّر مسجَّل، وأكثر من ٢٣٠ موظفًا. سواءً أحببت أم كرهت، لقد غيّر جتهب كثيرًا في جماعات ضخمة من مجتمع المصادر المفتوحة بطريقة لم يكد يتخيلها أحد عند كتابتي الإصدارة الأولى.

كتبت فصلا قصيرا في النسخة الأصلية من كتاب احترف جت عن جتهب ليكون مثالا لاستضافة جت، ولم أشعر قط بالارتياح الكامل إلى هذا الفصل؛ لم يعجبني أنني أكتب ما أشعر أنه في الأصل ذُخرا للمجتمع، ثم أتحدث فيه عن شركتي. ومع أني ما زلت لا أحب تضارب المصالح، فإن أهمية جتهب في مجتمع جت لا يمكن التغاضي عنها. وبدلا من كون هذا الجزء من الكتاب مثالا على استضافة جت، فقد قررت تحويله إلى شرح عميق لماهية جتهب وكيفية استخدامه بشكل فعّال. فإذا كنت تنوي تعلم استخدام جت، فستساعدك معرفة استخدام جتهب في المشاركة في مجتمع مترامي الأطراف، فهي قيّمة بغض النظر عن استضافة جت التي ستقرر استخدامها لمشاريعك البرمجية.

أما التغيير الكبير الآخر منذ النشر السابق فكان تطوير وبروغ بروتوكول HTTP لمعاملات جت الشبكية. فغيّرنا معظم أمثلة الكتاب من SSH إلى HTTP لأنه أسهل كثيرا.

لقد كان مذهلا مشاهدة جت ينمو عبر الأعوام من نظام إدارة نسخ مغمور نسبيا إلى نظام إدارة النسخ المهيمن في المشروعات التجارية والمفتوحة. أنا سعيد بكون كتاب احترف جت قد أبلى بلاءً حسنًا وكان قادرًا أن يكون من الكتب التقنية، القليلة في السوق، الناجحة والمفتوحة بالكامل معًا.

أرجو أن تنتفع بهذه الإصدارة المحدثة من احترف جت.

تمهيد Ben Straub

الإصدارة الأولى من هذا الكتاب هي ما جعلني مولعا بجت. كانت هي ما عرّفني أسلوب صناعة برمجيات وجدته طبيعيا أكثر من كل ما رأيت سابقًا. لقد كنت بالفعل مطوِّرا منذ عدة أعوام وقتئذٍ، لكن هذا كان المنعطف الصحيح الذي ساقني إلى طريق أمتع كثيرا من الطريق الذي كنت عليه.

واليوم وبعد أعوام، أنا مساهم في تطبيق جت رائد، وعملت في أكبر شركة استضافة جت، وطُفت العالم معلِّمًا الناس جت. وعندما سألني Scott إذا ما كنت أرغب في العمل على الإصدارة الثانية من الكتاب، لم أحتجْ حتى للتفكير.

لقد كان العمل على هذا الكتاب شرف عظيم وسعادة كبيرة لي. وأرجو أن يساعدك بقدر ما ساعدني.

إهداء

إلى زوجتي Becky، التي بغير وجودها لم تكن هذه المغامرة لتبدأ قَط. — Ben

أهدي هذه الإصدارة إلى فتياتي: إلى زوجتي Jessica التي ساندتني طوال السنين، وإلى ابنتي Josephine التي ستساندني عندما أكون شيخًا مسنًا لا يدرك ما يدور حوله. — Scott

الرخصة

مساهمات

لأن هذا الكتاب مفتوح المصدر، فقد تبرع لنا الناس بالعديد من التصحيحات وتعديلات المحتوى. إليكم جميع من ساهم في النسخة الإنجليزية من المشروع مفتوح المصدر Pro Git. شكرا لكم جميعا على مساعدتنا في جعل هذا الكتاب أفضل للناس جميعا.

المساهمون حتى الإيداع `be823c3a`:

4wk-	HairyFotr	pedrorijo91
Adam Laflamme	Hamid Nazari	Pessimist
Adrien Ollier	Hamidreza Mahdavipanah	Peter Kokot
ajax333221	haripetrov	peterwwillis
Akrom K	Haruo Nakayama	Petr Bodnar
Alan D. Salewski	Helmut K. C. Tessarek	Petr Janeček
Alba Mendez	Hemant Kumar Meena	Petr Kajzar
Aleh Suprunovich	Hidde de Vries	petsuter
Alexander Bezzubov	HonkingGoose	Philippe Blain
Alexandre Garnier	Howard	Philippe Miossec
alex-koziell	i-give-up	Phil Mitchell
Alex Povel	Ignacy	Pratik Nadagouda
Alfred Myers	Igor	Rafi
allen joslin	Ilker Cat	rahrah
Amanda Dillon	iprok	Raphael R
andreas	Jan Groenewald	Ray Chen
Andreas Bjørnestad	Jaswinder Singh	Rex Kerr
Andrei Dascalu	Jean-Noël Avila	Reza Ahmadi
Andrei Korshikov	Jeroen Oortwijn	Richard Hoyle
Andrew Layman	Jim Hill	Ricky Senft
Andrew MacFie	jingsam	Rintze M. Zelle
Andrew Metcalf	jliljekrantz	rmzelle

Andrew Murphy
AndyGee
AnneTheAgile
Anthony Loiseau
Antonello Piemonte
Antonino Ingargiola
Anton Trunov
applecuckoo
Ardavast Dayleryan
Artem Leshchev
atalakam
Atul Varma
axmbo
Bagas Sanjaya
Benjamin Dopplinger
Ben Sima
Billy Griffin
Bob Kline
Bohdan Pylypenko
Borek Bernard
Brett Cannon
bripmccann
brotherben
Buzut
Cadel Watson
Carlos Martín Nieto
Carlos Tafur
Chaitanya Gurrapu
Changwoo Park
Christian Decker
Christoph Bachhuber
Christopher Wilson
Christoph Prokop
C Nguyen
CodingSpiderFox
Cory Donnelly
Cullen Rhodes
Cyril
Damien Tournoud
Daniele Tricoli
Joel Davies
Johannes Dewender
Johannes Schindelin
johnhar
John Lin
Jonathan
Jon Forrest
Jon Freed
Jordan Hayashi
Joris Valette
Josh Byster
Joshua Webb
Junjie Yuan
Junyeong Yim
Justin Clift
Kaartic Sivaraam
KatDwo
Katrin Leinweber
Kausar Mehmood
Keith Hill
Kenneth Kin Lum
Kenneth Lum
Klaus Frank
Kristijan "Fremen" Velkovski
Krzysztof Szumny
Kyrylo Yatsenko
Lars Vogel
Laxman
Lazar95
leerg
Leonard Laszlo
Linus Heckemann
Logan Hasson
Louise Corrigan
Luc Morin
Lukas Röllin
maks
Marat Radchenko
Marcin Sędłak-Jakubowski
Marie-Helene Burle
Rob Blanco
Robert P. Goldman
Robert P. J. Day
Robert Theis
Rohan D'Souza
Roman Kosenko
Ronald Wampler
root
Rory
Rüdiger Herrmann
Sam Ford
Sam Joseph
Sanders Kleinfeld
sanders@oreilly.com
Sarah Schneider
SATO Yusuke
Saurav Sachidanand
Scott Bronson
Scott Jones
Sean Head
Sean Jacobs
Sebastian Krause
Sergey Kuznetsov
Severino Lorilla Jr
sharpiro
Shengbin Meng
Sherry Hietala
Shi Yan
Siarhei Bobryk
Siarhei Krukau
Skyper
slavos1
Smaug123
Snehal Shekatkar
Solt Budavári
Song Li
spacewander
Stephan van Maris
Steven Roddis
Stuart P. Bentley

Daniel Hollas
Daniel Knittl-Frank
Daniel Shahaf
Daniel Sturm
Daniil Larionov
Danny Lin
Dan Schmidt
Davide Angelocola
David Rogers
delta4d
Denis Savitskiy
devwebcl
Dexter
Dexter Morganov
DiamondeX
Dieter Ziller
Dino Karic
Dmitri Tikhonov
Dmitriy Smirnov
Doug Richardson
dualsky
Duncan Dean
Dustin Frank
Eden Hochbaum
Ed Flanagan
Eduard Bardají Puig
Eric Henziger
evanderiel
Explorare
eyherabh
Ezra Buehler
Fabien-jrt
Fady Nagh
Felix Nehrke
Filip Kucharczyk
flip111
flyingzumwalt
Fornost461
franjozen
Frank
Marius Žilėnas
Markus KARG
Marti Bolivar
Mashrur Mia (Sa'ad)
Masood Fallahpoor
Mathieu Dubreuilh
Matt Cooper
Matthew Miner
Matthieu Moy
Matt Trzcinski
Mavaddat Javid
Max Coplan
Máximo Cuadros
Michael MacAskill
Michael Sheaver
Michael Welch
Michiel van der Wulp
Miguel Bernabeu
Mike Charles
Mike Pennisi
Mike Thibodeau
Mikhail Menshikov
Mitsuru Kariya
mmikeww
mosdalsvsocld
nicktime
Niels Widger
Niko Stotz
Nils Reuße
Noelle Leigh
noureddin
OliverSieweke
Olleg Samoylov
Osman Khwaja
Otto Kekäläinen
Owen
Pablo Schläpfer
Pascal Berger
Pascal Borreli
Patrice Krakow
SudarsanGP
Suhaib Mujahid
Susan Stevens
Sven Selberg
td2014
Thanix
Thomas Ackermann
Thomas Hartmann
Tiffany
Tomas Fiers
Tomoki Aonuma
Tom Schady
Tvirus
twekberg
Tyler Cipriani
Ud Yzr
uerdogan
UgmaDevelopment
ugultopu
un1versal
Vadim Markovtsev
Vangelis Katsikaros
Vegar Vikan
Victor Ma
Vipul Kumar
Vitaly Kuznetsov
Volker-Weissmann
Volker Weißmann
Wesley Gonçalves
William Gathoye
William Turrell
Wlodek Bzyl
Xavier Bonaventura
xJom
xtreak
yakirwin
Yann Soubeyrand
Y. E
Your Name
Yue Lin Ho

Frederico Mazzone	patrick96	Yunhai Luo
Frej Drejhammar	Patrick Steinhardt	Yusuke SATO
goekboet	paveljanik	zwPapEr
grgbnc	Pavel Janík	ᐯᕮᒪᑎᑕᕮᖶ
Guthrie McAfee Armstrong	Paweł Krupiński	狄卢

مقدمة

أنت على وشك قضاء عدة ساعات من عمرك في القراءة عن جِت (Git)، فلنأخذ منها دقيقة لنشرح ما سنقدمه لك؛ هذا ملخص سريع لأبواب الكتاب العشرة وملاحقه الثلاثة.

في **الباب الأول**، نتناول أنظمة إدارة النسخ وأسس جت — لا شيء تقني، وإنما نعرف ما هو جت ولماذا أتى في أرض ممتلئة بأنظمة إدارة النسخ، وما يجعله مختلفا، ولماذا يستخدمه الكثيرون. بعدئذٍ نشرح كيفية تنزيل جت وإعداده للمرة الأولى إن لم يكن لديك بالفعل على نظامك.

في **الباب الثاني**، نمر على مبادئ استخدام جت: كيف تستخدمه في ٨٠٪ من الحالات التي ستقابلها معظم الوقت. بعد قراءة هذا الفصل، ستكون قادرا على استنساخ مستودع، ورؤية ما قد حدث في تاريخ المشروع، وتعديل ملفات، والمساهمة بتعديلات. لو اشتعل الكتاب ذاتيًّا وقتئذٍ، فسيكون جت طيِّعًا في متناولك وتنتفع به، إلى أن تحصل على نسخة أخرى من الكتاب.

أما **الباب الثالث** فعن نموذج التفريع في جت، الذي غالبا ما يوصف بأنه ميزته القاتلة للمنافسة. تتعلم هنا ما الذي يميز جت حقًّا عن الآخرين. وعندما تنهيه، سترغب في التوقف لحظات للتفكر كيف عشت حياتك سابقًا بغير أن يكون تفريع جت جزءًا منها.

يتناول **الباب الرابع** جت على الخادوم. وهو لمن يريدون إعداد جت داخل منظمتهم أو على خادومهم الخاص للتعاون. ونستكشف أيضا الخيارات المستضافة إذا كنت تفضّل أن يديره لك شخصٌ آخر.

يشرح **الباب الخامس** بالتفصيل الممل أساليب سير العمل الموزَّع المختلفة وكيفية تحقيقها مع جت. عندما تفرغ من هذا الباب، ستكون قادرا على العمل ببراعة مع العديد من المستودعات البعيدة، واستخدام جت عبر البريد الإلكتروني، واللعب برشاقة بالكثير من الفروع البعيدة والرُقَع المساهَم بها.

يتناول **الباب السادس** بعمق خدمة استضافة جتهَب (GitHub) وأدواتها. فنورد إنشاء حساب

وإدارته، وإنشاء مستودعات جت واستعمالها، وأساليب سير العمل الشائعة للمساهمة في مشروعات الآخرين وقَبول المساهمات في مشروعاتك، وواجهة جتهب البرمجية، وبحرًا من النصائح الصغيرة لجعل حياتك أسهل عموما.

الباب السابع عن أوامر جت المتقدمة. ستتعلم هنا عن أمورٍ مثل إتقان أمر الإرجاع المرعب (`reset`)، واستعمال طريقة البحث الثنائي لتحديد العلل، وتحرير التاريخ، واختيار المراجعات بالتفصيل، والمزيد المزيد. يسعى هذا الباب لإتمام معرفتك بجت حتى تكون أستاذًا بحق.

يتناول **الباب الثامن** تهيئة بيئة جت الخاصة بك. هذا يشمل إعداد بُرمِجات الخطاطيف (hook scripts) لفرض أو تشجيع السياسات المفضلة واستعمال إعدادات تهيئة البيئة لكي تعمل بالطريقة التي تريدها، وكذلك بناء مجموعتك الخاصة من البُرمِجات (scripts) لفرض سياسة إيداع مخصصة.

يناقش **الباب التاسع** جت والأنظمة الأخرى لإدارة النسخ. هذا يشمل استخدام جت داخل عالم Subversion (SVN)، وتحويل المشروعات من الأنظمة الأخرى إلى جت. فلا تزال منظمات كثيرة تستخدم SVN ولا تنوي التغيير، لكنك في هذه المرحلة ستكون قد تعلمت قوة جت الخيالية — فيريك هذا الفصل كيف تدبر أمورك إن كنت ما زلت مضطرا إلى استخدام خادوم SVN. نذكر أيضا كيفية استيراد مشروعات من الأنظمة الأخرى، إذا أقنعت الجميع بالغطس في جت.

يغوص **الباب العاشر** في أعماق جت المظلمة لكن الجميلة. لأنك عندئذٍ تعلم كل شيء عن جت وتستطيع استخدامه ببراعة ورشاقة، يمكننا الانتقال إلى مناقشة كيف يخزن جت كائناته، وما هو نموذج الكائنات، وما تفاصيل الملفات المُعلبة (packfiles) وموافيق (بروتوكولات) الخواديم، والمزيد. سنشير خلال هذا الكتاب إلى فصول هذا الباب، إن رغبت في الغوص عميقا في تلك المرحلة، لكن إذا كنت مثلنا وتريد الغوص في التفاصيل التقنية، فقد تحب قراءة الباب العاشر أولا. نترك هذا الخيار لك.

في **الملحق الأول**، نرى عددا من أمثلة استخدام جت في بيئات معينة مختلفة. نذكر عددا من الواجهات الرسومية وبيئات التطوير المختلفة التي ربما تريد استخدام جت فيها وما المتاح لك. إذا كنت مهتما بنظرة عامة على استخدام جت في الطرفية أو بيئة التطوير أو محرر النصوص، ألقِ نظرة هنا.

في **الملحق الثاني**، نستكشف برمجة جت وتوسيعه عبر أدوات مثل libgit2 وJGit. إذا كنت مهتما بكتابة أدوات مخصصة سريعة ومعقدة وتحتاج وصولا إلى المستويات الدنيا من جت، هنا مكانك لمعرفة كيف يبدو المنظر العام لهذه المنطقة.

وأخيرا، في **الملحق الثالث**، نمر على جميع أوامر جت المهمة واحدا تلو الآخر ونراجع أين شرحناه في الكتاب وماذا فعلنا به. إذا أردت أن تعرف أين في الكتاب استخدمنا أمر جت معين، يمكنك البحث عنه هنا.

هيا بنا نبدأ.

الباب الأول: البدء

هذا الباب عن البدء مع جِت (Git). نبدأ بشرح خلفية عن أدوات إدارة النسخ، ثم ننتقل إلى كيفية تشغيل جت على نظامك، وأخيرا كيفية إعداده لبدء العمل معه. في نهاية الفصل ستكون قد فهمت سبب وجود جت، ولماذا عليك استخدامه، وأن تكون قد أعددته للاستخدام.

عن إدارة النسخ

ما هي «إدارة النسخ»، ولماذا عليك أن تهتم؟ إدارة النسخ هي نظام يسجل التعديلات على ملف أو مجموعة من الملفات عبر الزمان، حتى يمكنك استدعاء نسخ معينة منها فيما بعد. نستخدم في أمثلة هذا الكتاب ملفات مصادر برمجية لإدارة نُسخها، ولو أن في الحقيقة يمكنك فعل ذلك مع أكثر أنواع الملفات الحاسوبية.

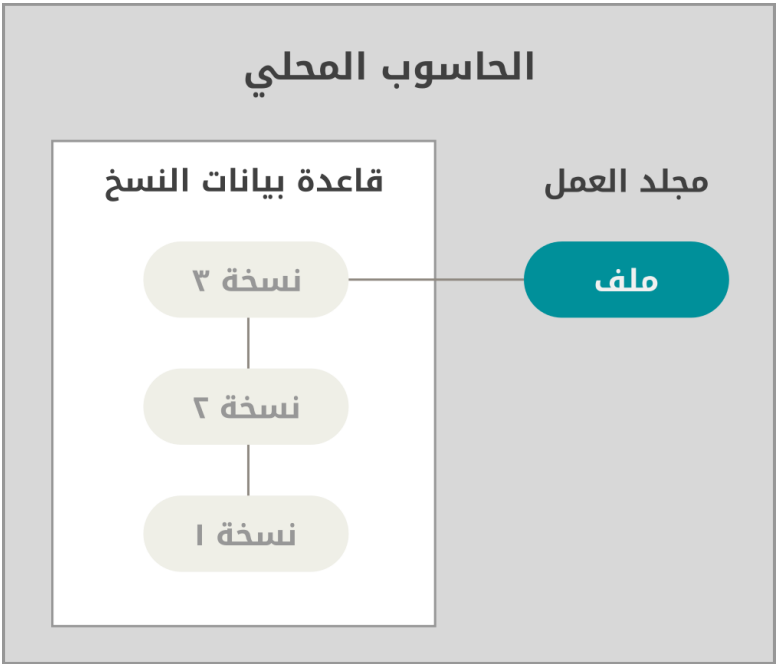
إذا كنت مصمم رسوميات أو وب وتريد الإبقاء على جميع نسخ صورة أو تخطيط (وهذا شيء يُفترض أنك تريد فعله بالتأكيد)، فإن استخدام نظام إدارة نسخ (VCS) لهو قرار حكيم جدا. فهو يسمح لك بإرجاع ملف محدد إلى حالة سابقة، أو إرجاع المشروع كله إلى حالة سابقة، أو مقارنة التعديلات عبر الزمان، أو معرفة مَن آخر من عدّل شيئا قد يكون سبّب مشكلة، أو مَن تسبب في إحداث علّة، وغير ذلك. استخدام نظام إدارة نسخ أيضا يعني في العموم أنك إذا دمّرت أشياء أو فقدت ملفات، فإنك تستطيع استرداد الأمور بسَهولة. وكل هذا تحصل عليه مقابل عبء ضليل جدا.

الأنظمة المحلية لإدارة النسخ

طريقة إدارة النسخ عند الكثيرين هي نسخ الملفات إلى مجلد آخر (ربما بختم زمني، إذا كان المستخدم بارعا). هذه الطريقة شائعة جدا لأنها سهلة جدا، لكنها أيضا خطّاءة جدا جدا. فسهل جدا أن تنسى أي مجلد أنت فيه وتغيّر في الملف الخاطئ أو تنسخ على ملفات لم ترد إبدالها.

لحل هذه القضية، طوّر المبرمجون منذ زمن بعيد «أنظمة محلية لإدارة النسخ» ذات قاعدة بيانات بسيطة تحفظ

جميع التعديلات على الملفات المراد التحكم في نُسخها.

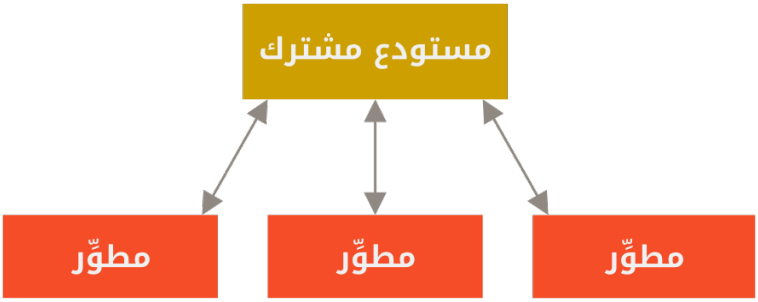


شكل ١. رسم توضيحي للإدارة المحلية للنسخ

كان من أشهر أنظمة إدارة النسخ نظام يسمى RCS، وهو ما زال موزعا مع حواسيب كثيرة اليوم. يعمل RCS (https://www.gnu.org/software/rcs/) بالاحتفاظ بمجموعات الرقع (أي الفروقات بين الملفات) بصيغة مخصوصة على القرص؛ فيمكنه إذًا إحياء أي ملف من أي حقبة زمنية بمجرد جمع الرقع.

الأنظمة المركزية لإدارة النسخ

المشكلة الكبرى الأخرى التي واجهت الناس هي أنهم يحتاجون إلى التعاون مع مطوِّرين على أنظمة أخرى. ولحلها، أُنشئت «الأنظمة المركزية لإدارة النسخ» (CVCS). لهذه الأنظمة (مثل CVS و Subversion و Perforce) خادوم وحيد به جميع الملفات المراقَبة، وعدد من العملاء الذين يستنسخون الملفات من ذلك المركز الوحيد. كان هذا هو المعيار المتبع لإدارة النسخ لأعوام عديدة.



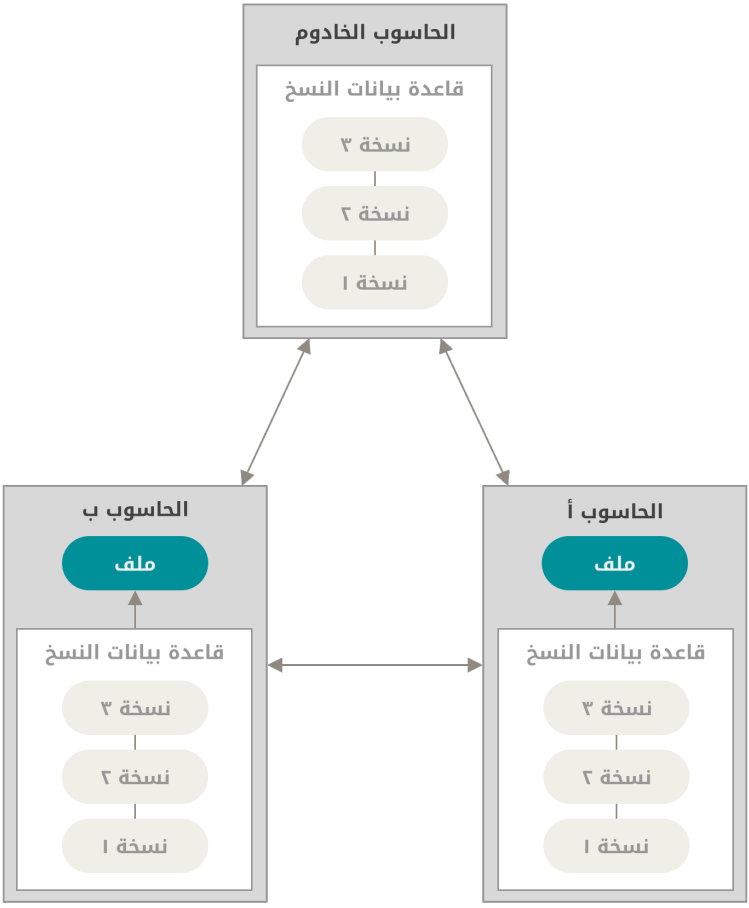
شكل ٢. رسم توضيحي للإدارة المركزية للنسخ

لهذا الترتيب مزايا كثيرة، لا سيما على الإدارة المحلية للنسخ. مثلا، الجميع يعلم، إلى حدٍ ما، كل ما يفعله الآخرون في المشروع نفسه. ولدى المديرين تحكم مفصّل في تحديد مَن يستطيع فعل ماذا، وإنه لأسهل كثيرا إدارة نظام مركزي مقارنةً بالتعامل مع قواعد بيانات محلية عند كل عميل.

ولكن لهذا الترتيب بعض العيوب الخطيرة. أكثرها وضوحا هو نقطة الانهيار الحاسمة الذي يمثلها الخادوم المركزي. فإذا تعطل الخادوم ساعةً، ففي تلك الساعة لن يستطيع أحد مطلقا التعاون أو حتى حفظ تعديلاتهم على ما يعملون عليه. وإذا تلف قرص قاعدة البيانات المركزية، ولم تحفظ أي نسخ احتياطية، ستفقد كل شيء تماما: تاريخ المشروع برُمّته، ما عدا أي لقطات فردية تصادف أن يُبقيها الناس على أجهزتهم المحلية. تعاني الأنظمة المحلية لإدارة النسخ أيضا من هذه العلة نفسها، وقتما جعلت التاريخ الكامل للمشروع في مكان وحيد، فإنك تعرض نفسك لفقد كل شيء.

الأنظمة الموزعة لإدارة النسخ

الآن تتدخل الأنظمة الموزعة لإدارة النسخ (DVCS). في نظام موزع (مثل جت و Mercurial و Darcs)، لا يستنسخ العملاء اللقطة الأخيرة فقط من الملفات، ولكنهم يستنسخون المستودع برُمّته، بما في ذلك تاريخه بالكامل. لذا فإن انهار أحد الخوادم فجأة، وكانت هذه الأنظمة تتعاون عبر هذا الخادوم، فيمكن نسخ مستودع أي عميل إلى الخادوم مجددا لإعادته للعمل. كل نسخة هي في الحقيقة نسخة احتياطية كاملة لجميع البيانات.



شكل ٣. رسم توضيحي للإدارة الموزعة للنسخ

علاوة على ذلك، الكثير من هذه الأنظمة تتعامل جيدا مع وجود العديد من المستودعات البعيدة التي يمكنها العمل معها، لذا يمكنك التعاون مع مجموعات مختلفة من الناس بأساليب متعددة في الوقت نفسه داخل المشروع الواحد. هذا يسمح لك بتكوين أساليب سير عمل متعددة لم تكن ممكنة في الأنظمة المركزية، كالنماذج الشجرية.

تاريخ جت بإيجاز

مثل الكثير من الأشياء العظيمة في الحياة، بدأ جت بشيء من التدمير الإبداعي والخلافات المُتَّقدة.

نواة لينكس هي مشروع برمجي مفتوح المصدر ذو امتداد شاسع إلى حد ما. في السنوات الأولى من تطوير نواة لينكس (١٩٩١-٢٠٠٢)، كانت التعديلات البرمجية تتناقل في صورة رقع وملفات مضغوطة. وفي عام ٢٠٠٢، بدأ مشروع نواة لينكس باستخدام نظام إدارة نسخ موزع احتكاري يسمى BitKeeper.

ولكن في عام ٢٠٠٥، تدهورت العلاقة بين المجتمع الذي يطور نواة لينكس والشركة التجارية التي تتطور BitKeeper، وأسقطت صفة المجانية عن الأداة. دفع هذا مجتمع تطوير لينكس (وبالأخص Linus Torvalds، مؤسس لينكس) إلى تطوير أداتهم الخاصة بناءً على بعض ما تعلموه في أثناء استخدامهم BitKeeper. وكانت من أهداف النظام الجديد ما يلي:

- السرعة
- التصميم البسيط
- دعم وثيق للتطوير اللاخطي (آلاف الفروع المتوازية)
- موزع بالكامل
- التعامل مع مشروعات ضخمة كنواة لينكس بكفاءة (من ناحية السرعة وحجم البيانات)

منذ ولادة جت في عام ٢٠٠٥، وقد نما ونضج حتى صار سهل الاستخدام، ومع هذا فقد احتفظ بهذه الصفات الأولية. إنه سريع لدرجة مذهلة. إنه كفء جدا مع المشروعات الضخمة. إن له نظام تفريع خيالي للتطوير اللاخطي (انظر التفريع في جت).

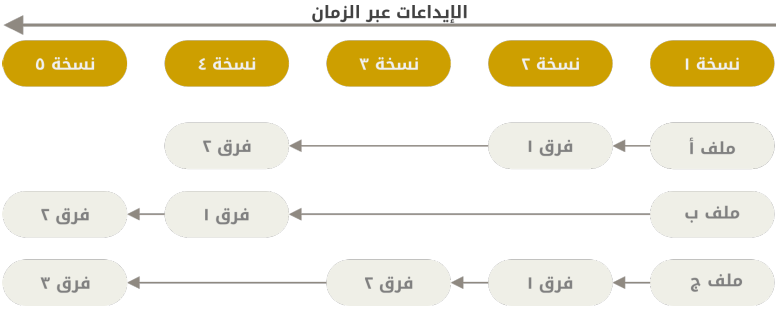
ما هو جت؟

إذًا، ما هو جت باختصار؟ هذا فصل مهم ويجب استيعابه جيدا، لأنك إذا فهمت ماهية جت وأصول طريقة عمله، فسيكون سهل جدا عليك استخدام جت بفعالية. وخلال تعلمك جت، عليك تصفية ذهنك من

كل ما تعمله عن أنظمة إدارة النسخ الأخرى، مثل CVS أو Subversion أو Perforce — يساعدك هذا على تجنب أي التباسات خفية عندما تستخدمه. ومع أن واجهة جت قريبة الشبه بالأنظمة الأخرى، إلا أن جت يخزن المعلومات بطريقة وينظر إليها نظرةً مختلفة أشد الاختلاف، ويساعدك فهم هذه الفروق على تجنب الالتباس عند استخدامه.

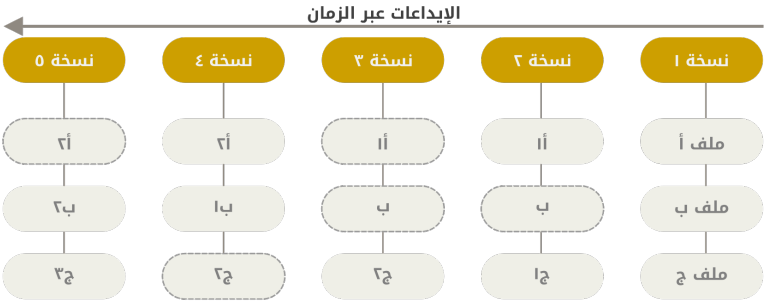
لقطات، وليس فروقات

الفرق الأكبر بين جت وأي نظام آخر (بما في ذلك Subversion وأشباهه)، هو نظرة جت إلى بياناته. من حيث المفهوم، تخزن معظم الأنظمة الأخرى المعلومات في صورة سلسلة تعديلات على الملفات. هذه الأنظمة الأخرى (CVS و Subversion و Perforce إلخ) تنظر إلى المعلومات التي تخزنها على أنها مجموعة من الملفات والتعديلات التي تتم على كل ملف عبر الزمان (يوصف هذا غالبا بأنه إدارة نسخ بناءً على الفروقات).



شكل ٤. تخزين البيانات في صورة تعديلات على نسخة أساسية من كل ملف

لا ينظر جت إلى بياناته ولا يخزنها بهذه الطريقة. بل يعتبرها أشبه بلقطات من نظام ملفات مصغر. في جت، كل مرة تصنع إيداعا (`commit`)، أو تحفظ حالة مشروعك، يلتقط جت صورة لما تبدو عليه ملفاتك جميعًا في هذه اللحظة، ويخزن إشارة لهذه اللقطة. وحتى يُحسن استغلال الموارد، فإن الملفات التي لم تتغير لا يخزنها جت مجددا، بل يخزن فقط إشارةً إلى الملف السابق المطابق الذي خزّنه سابقا. فإن جت يعتبر أن بياناته **سيل من اللقطات**.



شكل ٥. تخزين البيانات في صورة لقطات من المشروع على مر الزمان

هذا تمييز مهم بين جت وأكثر الأنظمة الأخرى. إنه يجعل جت يعيد التفكير في أغلب جوانب إدارة النسخ التي نسختها معظم الأنظمة الأخرى من الأجيال السابقة. إنه يجعل جت أشبه بنظام ملفات مصغر ذي أدوات خارقة مبنية عليه، بدلا من مجرد نظام إدارة نسخ. عندما نتناول التفريع في جت في التفريع في جت، سنستكشف بعضًا من المنافع التي تحصل عليها عندما تنظر إلى بياناتك هذه النظرة.

أغلب العمليات محلية

أكثر العمليات في جت لا تحتاج إلا إلى ملفات وموارد محلية لكي تعمل؛ فعموما لا حاجة إلى أي معلومات من حواسيب أخرى على الشبكة. إذا كنت معتادًا على نظام إدارة نسخ مركزي، حيث معظم العمليات مثقلة بعبء زمن الانتقال في الشبكة، فإن هذا الجانب من جت سيجعلك تظن أن الله قد منّ عليه بقدرة لدنيّة لتكون له هذه السرعة. فلأن لديك التاريخ الكامل للمشروع بين يديك على قرصك المحلي، فإن معظم العمليات تبدو آنية.

مثلا، لتصفح تاريخ المشروع، لا يحتاج جت إلى السفر إلى الخادوم ليعود إليك حاملا التاريخ ليعرضه لك — إنما يقرؤه من قاعدة بياناتك المحلية. هذا يعني أنك ترى تاريخ المشروع أسرع من طرفة العين. وإذا أردت أن ترى التعديلات التي حدثت على ملف بين نسخته الآن ومنذ شهر، فيستطيع جت أن يأتي بهذا الملف منذ شهر ويحسب الفرق على حاسوبك، بدلا من الاضطرار إلى طلب هذا الفرق من خادوم بعيد أو طلب النسخة القديمة منه وحساب الفرق محليا.

هذا أيضا يعني أنك ما زلت تستطيع فعل كل شيء، إلا القليل النادر، إذا كنت بغير اتصال بالإنترنت أو بشبكتك الوهمية الخاصة (VPN). فإذا كنت في طائرة أو قطار، وتريد العمل قليلا، تستطيع الإيداع بكل سعادة (إلى نسختك المحلية، أتتذكر؟) حتى تجد اتصالا شبكيا للرفع. وإذا عدت إلى المنزل ولم تجد عميل شبكتك الوهمية يستطيع العمل، فإنك ما زلت تستطيع العمل. أما في الكثير من الأنظمة الأخرى، فالعمل من غير اتصال إما أليم جدا وإما مستحيل أصلا. في Perforce مثلا، لا يمكنك فعل الكثير إن لم تكن متصلا بالخادوم. في Subversion و CVS تستطيع تعديل الملفات، لكن لا تستطيع إيداع أي تعديلات في قاعدة بياناتك (لأن قاعدة بياناتك غير متصلة). ربما تظن أن هذا ليس بالأمر العظيم، لكنك إذًا ستتفاجأ بضخامة الفرق الذي يصنعه.

في جت السلامة

يضمن جت سلامة البيانات دائمًا، فهو يحسب قيم البصمات لكل شيء قبل أن يخزنه، وبعدئذٍ يشير إلى الأشياء ببصماتها. هذا يعني أن تعديل محتويات أي ملف أو مجلد بغير علم جت مستحيل. هذا مبني في أساس جت ومن أركان فلسفته. فمستحيل فقد معلومات أثناء النقل أو حتى فساد ملفات من غير أن يكتشف جت ذلك.

الآلية التي يستخدمها جت لحساب البصمة معروفة باسم بصمة SHA-1. وهي تنتج سلسلة نصية من أربعين (٤٠) رقما ستعشريا (0–9 و a–f) محسوبين من محتوى الملف أو بنية المجلد في جت. تشبه بصمة SHA-1 هذا:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

ترى قيم البصمات هذه في كل مكان في جت لأنه يستخدمها كثيرا. في الحقيقة، يخزن جت كل شيء في قاعدة بياناته، ليس بأسماء الملفات، بل بقيم بصمة محتواها.

يضيف جت بيانات فقط في العموم

أكثر الإجراءات في جت لا تفعل شيئا سوى أن **تضيف** بيانات إلى قاعدة بيانات جت. ومن الصعب أن تجعله يفعل شيئا لا يمكن التراجع عنه أو أن تجعله يمسح بيانات بأي طريقة. مثلما الحال مع أي نظام إدارة نسخ، يمكن أن تفقد أو تدمر التعديلات التي لم تودعها بعد، لكن ما إن تودعها في جت، فمن العسير جدا أن تفقدها، خصوصا إذا كنت تدفع (`push`) قاعدة بياناتك بانتظام إلى مستودع آخر.

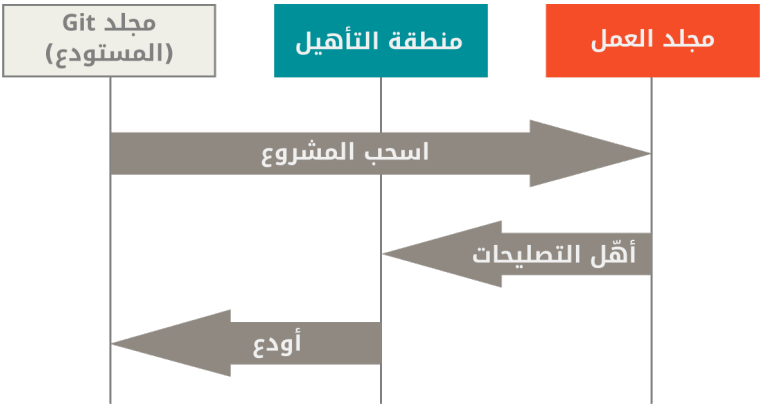
هذا يجعل استخدام جت مبهجًا لأننا نعلم أننا نستطيع التجريب بغير خطر التخريب. لنظرة أعمق على كيفية تخزين جت لبياناته وكيفية استعادة البيانات التي تبدو لك قد فُقدت، انظر التراجع عن الأفعال.

المراحل الثلاثة

انتبه الآن وركّز؛ هذا هو أهم شيء عليك أن تتذكره دوما عن جت إذا أردت أن تمضي رحلة تعلمك بسلاسة. في جت ثلاث مراحل يمكن أن تكون ملفاتك فيها: معدّل، ومؤهل، ومُودَع.

- معدل يعني أنك عدلت الملف لكنك لم تُودِع التعديلات بعد في قاعدة بياناتك.
- مؤهل يعني أنك حددت ملفًا معدلًا في نسخته الحالية ليكون ضمن لقطة الإيداع التالية.
- مُودَع يعني أن البيانات صارت مخزنة بأمان في قاعدة بياناتك.

يقودنا هذا إلى الأقسام الرئيسية الثلاثة في أي مشروع جت: شجرة العمل، ومنطقة التأهيل، ومجلد جت.



شكل ٦. شجرة العمل، ومنطقة التأهيل، ومجلد جت

شجرة العمل (أو مجلد العمل) هي نسخة واحدة مسحوبة من المشروع. تُجلب لك هذه الملفات من قاعدة البيانات المضغوطة في مجلد جت وتُوضع لك على القرص لتستخدمها أو تعدلها.

منطقة التأهيل هي ملف يخزن معلومات عما سيكون في إيداعك التالي، وعادةً يكون ملف منطقة التأهيل في مجلد جت لمشروعك. المصطلح التقني في لغة جت هو «الفهرس» (`index`)، لكن العبارة «منطقة التأهيل» (`staging area`) مناسبة ومستخدمة أيضا.

مجلد جت هو المكان الذي يخزن فيه جت البيانات الوصفية وقاعدة بيانات الكائنات لمشروعك. هذا هو أهم جزء في جت، وهو الذي يُنسخ عندما تستنسخ (`clone`) مستودعا من حاسوب آخر.

يبدو أسلوب سير العمل الأساسي في جت مثل هذا:

١. تعدّل ملفات في شجرة عملك.

٢. تنتقي من تلك التعديلات ما تؤهله ليكون جزءًا من إيداعك التالي، وهذا لا يضيف إلا هذه التعديلات إلى منطقة التأهيل.

٣. تصنع إيداعا، وهذا يلتقط صورة للملفات كما هي من منطقة التأهيل ويخزن هذه اللقطة في مجلد جت لمشروعك إلى الأبد.

إذا كانت نسخة معينة من أحد الملفات موجودة داخل مجلد جت، فإنها تعتبر مُودَعة. وإذا كانت معدّلة وقد أضيفت إلى منطقة التأهيل، فإنها مؤهَّلة. وإذا كانت معدّلة بعد آخر مرة سُحبت فيها لكنها لم تؤهل بعد، فإنها معدَّلة. ستتعلم المزيد في أسس جت عن هذه الحالات وكيف يمكنك استغلالها أو تخطي مرحلة التأهيل برمتها.

سطر الأوامر

لاستخدام جت طرائق عديدة مختلفة. فلدينا أدوات سطر الأوامر الأصلية، وأيضا الكثير من الواجهات الرسومية ذات القدرات المتفاوتة. نستخدم في هذا الكتاب جت من سطر الأوامر. فسطر الأوامر هو المكان الوحيد الذي يمكنك فيه تنفيذ جميع أوامر جت، فمعظم الواجهات الرسومية لا تتيح إلا جزءًا من وظائف جت للتسهيل. وإذا كنت تعرف كيف تستخدم نسخة سطر الأوامر، ففي الغالب أنك أيضا ستعرف كيف تستخدم النسخة الرسومية، لكن العكس ليس بالضرورة صحيح. وأيضا اختيارك لعميل رسومي يخضع لذوقك الشخصي، لكن جميع المستخدمين لديهم أدوات سطر الأوامر مثبتة ومتاحة.

لذلك فإننا نتوقع منك معرفة كيف تفتح الطرفية (Terminal) في الأنظمة اليونكسية أو موجه سطر الأوامر (Command Prompt) أو PowerShell في ويندوز. إن لم تكن تعلم عما نتحدث، فعليك التوقف الآن وبحث هذا سريعا حتى تستطيع السير مع الأمثلة والأوصاف التي في الكتاب.

تثبيت جت

قبل الشروع في استخدام جت، عليك جعله متاحًا على حاسوبك. حتى لو كان مثبتًا بالفعل، فغالبا من الأفضل تحديثه إلى آخر نسخة. يمكنك إما تثبيته من حزمة أو عبر مثبت آخر وإما تنزيل المصدر البرمجي وبناءه بنفسك.

كُتب هذا الكتاب عن جت نسخة ٢. لكن لأن جت متميز في الحفاظ على التوافقية مع الإصدارات السابقة، فأي نسخة حديثة ينبغي أن تعمل جيدا. ومع أن معظم الأوامر ينبغي أن تعمل حتى في نسخ جت الأثرية، فقد لا يعمل بعضها أو يعمل باختلاف طفيف.

التثبيت على لينكس

إذا أردت تثبيت أدوات جت الأساسية على لينكس عبر مثبت برمجيات مبنية، فيمكنك غالبا فعل ذلك عبر أداة إدارة الحزم التي في توزيعتك. فإذا كنت على فيدورا (أو أي توزيعة قريبة منها تستخدم حزم RPM، مثل ردهات (RHEL) أو CentOS)، فيمكنك استخدام `dnf`:

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

وإذا كنت على توزيعة ديبانية مثل أوبنتو، جرب `apt`:

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

لخيارات أخرى، توجد على موقع جت تعليمات لتثبيته على توزيعات لينكسية ويونكسية عديدة، في https://git-scm.com/download/linux.

التثبيت على ماك أو إس

توجد طرائق عديدة لتثبيت جت على ماك. ربما أسهلها هو تثبيت أدوات سطر أوامر إكس كود (Xcode Command Line Tools). وعلى ماك مافريكس (Mavericks, 10.9) أو أحدث، يمكنك فعل هذا بمجرد محاولة تنفيذ `git` في الطرفية لأول مرة مطلقا.

```
CONSOLE
$ git --version
```

فإذا لم يكن مثبتاً لديك بالفعل، فسيحثك على تثبيته.

أما إذا أردت نسخة أحدث، فيمكنك أيضا تثبيته عبر مثبت برمجيات مبنية. يوجد مثبت جت لماك على موقع جت، في https://git-scm.com/download/mac.

شكل ٧. مثبت جت على ماك أو إس

التثبيت على ويندوز

لتثبيت جت على ويندوز عدة طرائق أيضا. النسخة المبنية الأكثر رسميةً متاحة على موقع جت. عليك فقط الذهاب إلى https://git-scm.com/download/win وسيبدأ التنزيل تلقائيا. لاحظ أن هذا مشروع يسمى «جت لويندوز» (Git for Windows)، وهو منفصل عن جت نفسه؛ للمزيد من المعلومات عنه، اذهب إلى https://gitforwindows.org.

أما إذا أردت مثبتا آليا فيمكنك استخدام حزمة Git على Chocolatey (https://community.chocolatey.org/packages/git) . لاحظ أن المجتمع هو من يرعى حزمة Chocolatey.

التثبيت من المصدر البرمجي

ربما يفيد بعض الناس تثبيت جت من المصدر البرمجي بدلا من ذلك، لأنك عندئذٍ ستحصل على أحدث نسخة إطلاقا. مثبتات البرمجيات المبنية تميل إلى التأخر قليلا، لكن لأن جت قد نضج في الأعوام الأخيرة، فلم يعد هذا يشكّل فارقا كما كان.

إذا أردت تثبيت جت من المصدر البرمجي، فستحتاج إلى المكتبات التالية التي يعتمد عليها جت: `autotools` و `curl` و `zlib` و `openssl` و `expat` و `libiconv`. مثلا، إذا كنت على نظام يستخدم `dnf` (مثل فيدورا) أو `apt-get` (مثل الأنظمة الديبانية)، فيمكنك استخدام أحد هذين الأمرين لتثبيت أقل اعتماديات مطلوبة لبناء جت وتثبيته:

CONSOLE

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

وتحتاج هذه الاعتماديات حتى تضيف التوثيق بصيغه المختلفة (`doc` و `html` و `info`):

CONSOLE

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

يحتاج مستخدمو ردهات والتوزيعات الردهاتية مثل CentOS وScientific Linux إلى تفعيل مستودع EPEL (الشرح بالإنجليزية) (https://docs.fedoraproject.org/en-US/epel/#how_can_i_use_these_extra_packages) حتى يستطيعوا تثبيت حزمة `docbook2X`.

إذا كنت تستخدم توزيعة ديبانية (ديبان أو أوبنتو أو إحدى مشتقاتهما)، فتحتاج أيضا حزمة `install-`

:info

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

إذا كنت تستخدم توزيعة تستخدم RPM (فيدورا أو ردهات أو إحدى مشتقاتهما)، فتحتاج أيضا حزمة `getopt` (المثبتة مبدئيا في التوزيعات الديبانية):

```
CONSOLE
$ sudo dnf install getopt
```

إضافة إلى ذلك، إذا كنت تستخدم فيدورا أو ردهات أو إحدى مشتقاتهما، فتحتاج أن تفعل هذا أيضا:

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

بسبب اختلافات في أسماء الأوامر.

عندما يكون لديك جميع الاعتماديات المطلوبة، يمكنك تنزيل أحدث ملف مضغوط موسوم برقم إصدار، من عدة أماكن. يمكنك تنزيله من موقع نواة لينكس، من https://www.kernel.org/pub/software/scm/git، أو من النسخة المقابلة على موقع جت‌هب، من https://github.com/git/git/tags. غالبا يكون أوضح قليلا على جت‌هب ما هي النسخة الأحدث، ولكن في صفحة موقع نواة لينكس ستجد بصمات الإصدارات، إذا أحببت تفقّد صحة الملفات التي نزّلتها.

بعدئذٍ قم بالبناء والتثبيت:

```
CONSOLE
$ tar -zxf git-2.8.0.tar.gz
$ cd git-2.8.0
$ make configure
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

بعد إتمام هذا، يمكنك الحصول على جت عبر جت نفسه للتحديثات:

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

إعداد جت لأول مرة

الآن وقد صار جت على نظامك، ستودّ عمل بعض الأمور لتخصيص بيئته لك. تحتاج عملها مرة واحدة فقط على أي حاسوب؛ فإنها تبقى عندما تحدّث جت. يمكنك أيضا تعديلها في أي وقت بالمرور على الأوامر مرة أخرى.

في جت أداة «تهيئة»، `git config`، لتعرض أو تضبط متغيرات التهيئة التي تتحكم في جميع مناحي مظهر وسلوك جت. وتُخزَّن هذه المتغيرات في ثلاثة أماكن مختلفة:

١. ملف `[path]/etc/gitconfig`: يحتوي القيم التي تُطبّق على جميع المستخدمين ومستودعاتهم. إذا أعطيت الخيار `--system` («نظام») إلى أمر التهيئة `git config`، فإنه يقرأ ويكتب في هذا الملف تحديدًا. طبعًا تحتاج صلاحيات إدارية لتعديل هذا الملف لأنه ملف إعدادات خاص بالنظام.

٢. ملف `~/.gitconfig` أو `~/.config/git/config`: القيم الخاصة بك أنت تحديدًا. يمكنك جعل جت يقرأ ويكتب في هذا الملف تحديدًا بالخيار `--global` («عام»)، والذي يؤثر في جميع مستودعاتك على هذا النظام.

٣. ملف `config` في مجلد جت (أي `.git/config`) في أي مستودع أنت فيه الآن: القيم الخاصة بهذا المستودع وحده. يمكنك إجبار جت على القراءة والكتابة في هذا الملف بالخيار `--local` («محلي»)، ولكن في الحقيقة هذا هو المفترض. بالطبع تحتاج إلى التواجد في مكان ما في مستودع جت حتى يمكنك استخدام هذا الخيار.

قيم كل مستوى تطغى على قيم المستوى السابق، لذا فقيم `.git/config` تتفوق على التي في `[path]/etc/gitconfig`.

في أنظمة ويندوز، يبحث جت عن ملف `.gitconfig` في مجلد المنزل، `$HOME` (والذي غالبا يكون `C:\Users\$USER`). ويبحث كذلك عن `[path]/etc/gitconfig` ، ولكن بالنسبة إلى جذر MSys، وهو أينما قررت تثبيت جت فيه على نظامك عندما شغّلت المثبت. وإذا كنت تستخدم Git for Windows النسخة `2.x` أو أحدث، فستجد أيضا ملف إعداد على مستوى النظام، في `C:\Documents and Settings\All Users\Application Data\Git\config` على ويندوز إكس بي، وفي `C:\ProgramData\Git\config` على ويندوز فيستا والأحدث. لا يمكن تغيير هذا الملف إلا بتنفيذ الأمر `git config -f <ملف>` بحساب المدير.

يمكنك رؤية جميع إعداداتك ومن أين أتت باستخدام:

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

هويتك

أول شيء تحتاج فعله عند تثبيت جت هو ضبط اسمك وعنوان بريدك الإلكتروني. هذا مهم لأن كل إيداع جت يستخدم هاتين المعلومتين، ويصيران جزءًا ثابتًا في الإيداعات التي ستبدأ في صنعها.

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

مجددًا، لن تحتاج إلى فعل هذا إلا مرة واحدة إذا استخدمت الخيار `--global` («عام»)، لأن جت عندئذٍ يستخدم هاتين المعلومتين لكل ما تفعله على هذا النظام. وإن احتجت إلى تجاوز إحدى هاتين القيمتين في مشروعات محددة، يمكنك تنفيذ الأمر في ذلك المشروع بغير خيار `--global` .

تساعدك الكثير من الواجهات الرسومية في فعل هذا عند تشغيلها لأول مرة.

محررك

الآن وقد أعددنا هُويّتك، يمكنك ضبط محررك المبدئي للنصوص، والذي يستخدمه جت عندما يريد منك أن تكتب رسالة. إذا لم يكن مضبوطا، فيستخدم جت المحرر المبدئي لنظامك.

إذا أردت استخدام محررا آخر، مثل Emacs، فيمكنك فعل الآتي:

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

على ويندوز، إذا أردت ضبط محرر آخر، فعليك تحديد المسار الكامل لملفه التنفيذي. والذي يختلف باختلاف طريقة تخزين محررك.

في حالة ++Notepad، وهو محرر برمجيات مشهور، ستريد غالبا أن تستخدم نسخة ٣٢-بت منه، لأن حتى وقت كتابة هذا، لا تدعم نسخة ٦٤-بت جميع الإضافات. إذا كنت على ويندوز ٣٢-بت أو تستخدم محرر ٦٤-بت على ويندوز ٦٤-بت، فإنك ستكتب شيئا مثل هذا:

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

Vim و Emacs و ++Notepad هي محررات نصوص شهيرة يستخدمها المبرمجون على ويندوز والأنظمة اليونكسية مثل لينكس وماك. إذا كنت تستخدم محررًا آخر، أو نسخة ٣٢-بت، فرجاءً انظر التعليمات الخاصة بإعداد محررك المفضل مع جت في git config core.editor commands.

إذا لم تضبط محررك مثل هذا، فإنك قد تجد نفسك في حالة محيرة جدا، عندما يحاول جت فتحه. مثال ذلك على ويندوز أن يحاول جت فتح المحرر فلا يستطيع فيغلق مبكرًا.

اسم الفرع المبدئي

عندما تنشئ مستودعًا جديدًا بالأمر `git init`، فإن جت سينُشئ فيه فرعًا، والذي يسميه مبدئيا `master`. يمكنك ضبط اسم آخر للفرع الأوليّ ابتداءً من النسخة 2.28 من جت.

لجعل اسم الفرع المبدئي هو `main`، نفّذ:

```
CONSOLE
$ git config --global init.defaultBranch main
```

تفقّد إعداداتك

إذا أردت تفقّد إعدادات تهيئتك، فيمكنك استخدام خيار السرد مع أمر التهيئة — `git config --list` — والذي يسرد لك جميع الإعدادات التي يراها جت وقتئذٍ:

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

ربما ترى بعض الأسماء مكررة، هذا لأن جت قد وجد الاسم نفسه في أكثر من ملف (`[path]/etc/gitconfig` و `~/.gitconfig` مثلا). يستخدم جت في مثل هذه الحالة القيمة الأخيرة لكل اسم يراه.

يمكنك أيضا سؤال جت عن القيمة التي يظنها لاسم معين، بالأمر `git config <اسم>`:

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

قد يقرأ جت متغير تهيئة معين من أكثر من ملف، فمن الممكن أن تجد بعض القيم مثيرة للدهشة ولا تعرف من أين أتت. يمكنك في مثل هذه الحالة سؤال جت: من أين لك هذا؟ — أي باستخدام خيار إظهار الأصل `--show-origin`، الذي سيخبرك بالملف الذي غلب على أمرهم في قيمة هذا المتغير:

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

الحصول على المساعدة

إذا احتجت يوما إلى المساعدة في جت، فعندك ثلاث طرائق متكافئة للحصول على صفحة الدليل الشامل (manpage) لأي أمر من أوامر جت:

```
CONSOLE
$ git help <أمر>
$ git <أمر> --help
$ man git-<أمر>
```

مثلا، للحصول على صفحة مساعدة الأمر `git config`، نفّذ هذا:

```
CONSOLE
$ git help config
```

هذه الأوامر جميلة لأنك تستطيع استخدامها في أي مكان، حتى عندما تكون غير متصل بالإنترنت. إن لم تكن صفحات المساعدة وهذا الكتاب كافيين واحتجت مساعدة شخصية، يمكنك تجربة إحدى قنوات IRC مثل `#git` أو `#github` أو `#gitlab` على خادوم Libera Chat، والذي تجده على https://libera.chat. هذه القنوات مليئة باستمرار بمئات الخبراء في جت والذين أغلب الأوقات يودّون المساعدة.

وإذا كنت غير محتاج إلى صفحة الدليل الكبيرة الكاملة، ولكن تحتاج فقط إلى تجديد معرفتك بالخيارات

المتاحة لأحد أوامر جت، فيمكنك طلب المساعدة الموجزة بالخيار `-h` ، مثل:

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

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

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

الخلاصة

أنت الآن مُلم بماهية جت وكيف هو مختلف عن أي نظام إدارة نسخ مركزي ربما تكون استخدمه سابقا. وكذلك الآن لديك جت مثبتا على نظامك ومضبوطا بهويتك الشخصية. حان الآن موعد تعلم بعض أسس جت.

الباب الثاني: أسس جت

لو لم تكن ستقرأ إلا بابًا واحدًا لتنطلق مع جت، فهذا هو. يشرح هذا الباب جميع الأوامر الأساسية التي تحتاجها لعمل الغالبية العظمى من الأمور التي ستقضي أغلب وقتك مع جت تفعلها بعد ذلك. على نهاية هذا الباب ستكون قادرًا على تهيئة مستودع وابتدائه، وبدء تعقب ملفات وإيقافه، وتأهيل تعديلات وإيداعها. سنريك أيضا كيف تضبط جت ليتجاهل ملفات معينة أو أنماطًا معينة من الملفات، وكيف تتراجع عن الأخطاء بسرعة وسَهولة، وكيف تتصفح تاريخ مشروعك، وكيف ترى التعديلات بين الإيداعات، وكيف تدفع إلى المستودعات البعيدة وتجذب منها.

الحصول على مستودع جت

في المعتاد تحصل على مستودع جت بإحدى طريقتين:

١. تأتي مجلدًا محليًّا ليس تحت إدارة نُسخ، وتحوله إلى مستودع جت،

٢. أو تستنسخ مستودع جت موجودا بالفعل.

في كلتا الحالتين، سيصير معك مستودع جت على حاسوبك المحلي وجاهز للعمل.

ابتداء مستودع في مجلد موجود

إذا كان لديك مجلد مشروع ليس تحت إدارة نسخ الآن، وتريد أن تبدأ في إدارته باستخدام جت، تحتاج أولا إلى الذهاب إلى ذلك المجلد. إن لم تفعل هذا من قبل، فهذا قد يختلف قليلا حسب نظامك:

للينكس:

CONSOLE

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

لماك أو إس:

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

لويندوز:

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

ثم اكتب:

```
CONSOLE
$ git init
```

هذا ينشئ لك مجلد فرعيا جديدا يُسمى `.git` (يبدأ اسمه بنقطة، فتجعله مجلدا مخفيا) ويحتوى كل الملفات الضرورية لمستودعك — أيْ هيكل مستودع جت. حتى الآن، لا شيء في مشروعك متعقَّب بعد. انظر ~~داخل جت~~ للمزيد من المعلومات عن تفاصيل الملفات التي في مجلد `.git` الذي أنشأته للتو.

إذا أردت أن تبدأ في إدارة نسخ ملفات موجودة (وليس مجلدا فارغا)، فعليك بدء تعقب هذه الملفات وصنع إيداع مبدئي. يمكنك تحقيق هذا ببعض أوامر الإضافة، `git add`، والتي تحدد الملفات التي تريد تعقبها، ثم أمر الإيداع، `git commit`:

```
CONSOLE
$ git add *.c
$ git add LICENSE
$ git commit -m 'Initial project version'  # إيداع «النسخة المبدئية من المشروع»
```

سنعرف ماذا تفعل هذه الأوامر خلال لحظات. لكن الآن، لديك مستودع جت به ملفات متعقبة وإيداع مبدئي.

استنساخ مستودع موجود

إذا كنت تريد الحصول على نسخة من مستودع جت موجود — مثلا مشروع تحب المشاركة فيه — فإن الأمر الذي تريده هو أمر الاستنساخ، `git clone`. إذا كنت تعرف أنظمة أخرى لإدارة النسخ مثل Subversion، ستلاحظ أن الأمر هو `clone` (استنساخ) وليس `checkout` (سحب). هذا فرق مهم؛ فبدلا من جلب مجرد نسخة للعمل عليها، يحضر لك جت تقريبا كل شيء لدى الخادوم؛ كل نسخة من كل ملف عبر تاريخ المشروع، يجذبها جت إليك عندما تكتب `git clone`. في الحقيقة، إذا تلف قرص الخادوم، يمكنك في الغالب استخدام ربما أي استنساخ من أي عميل لإرجاع الخادوم إلى حالته عندما أُستنسخ (قد تفقد بعض الخطاطيف الخاصة بالخادوم وأشياء من هذا القبيل، لكن جميع البيانات التي تحت إدارة النسخ ستكون موجودة — انظر تثبيت جت على خادوم للمزيد من التفاصيل).

استنسخ مستودعًا بالأمر `git clone <رابط>`. مثلا إذا أردت استنساخ مكتبة جت القابلة للربط المسماة `libgit2`، يمكنك فعل ذلك هكذا:

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

هذا ينشئ مجلدًا اسمه `libgit2`، ويبتدئ مجلد `.git` فيه، ويجذب جميع بيانات هذا المستودع، ويسحب نسخة عمل من النسخة الأخيرة منه. فإذا ذهبت إلى داخل مجلد `libgit2` الجديد الذي أُنشئ آنفا، فستجد فيه ملفات المشروع تنتظرك للعمل عليها أو استخدامها.

إذا أردت استنساخ المستودع إلى مجلد باسم غير `libgit2`، يمكنك تعيين هذا الاسم الجديد بإضافته إلى معاملات الأمر:

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

هذا الأمر يفعل الشيء نفسه الذي يفعله الأمر السابق، لكن اختلف المجلد الهدف فصار `mylibgit`.

يستطيع جت التعامل مع عدد من موافيق (بروتوكولات) النقل المختلفة. استخدم المثال السابق ميفاق `https://`، ولكنك قد ترى أيضا `git://`، أو `user@server:path/to/repo.git` الذي يستخدم ميفاق SSH. يخبرك تثبيت جت على خادوم بجميع الخيارات التي يستطيع الخادوم إعدادها حتى يمكنك الوصول إلى مستودع جت الخاص بك، ومزايا وعيوب كلٍ منها.

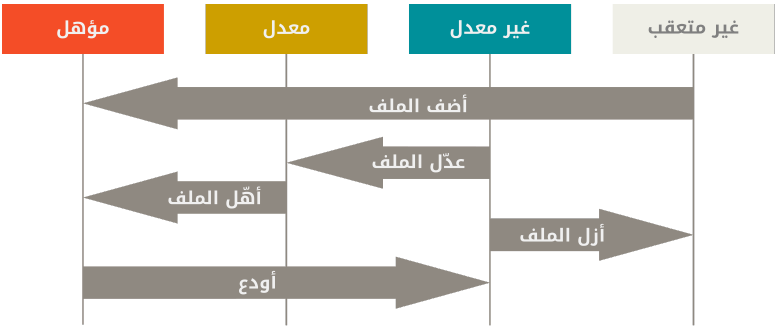
تسجيل التعديلات في المستودع

الآن لديك مستودع جت أصيل على حاسوبك، وأمامك نسخة مسحوبة من جميع ملفاته، أي «نسخة عمل». غالبا ستود البدء بعمل تعديلات وإيداع لقطات من هذه التعديلات في مستودعك كل مرة يصل مشروعك إلى مرحلة تريد تسجيلها.

تذكر أن كل ملف في مجلد العمل لديك يمكن أن يكون في حالة من اثنتين: متعقَّب أو غير متعقَّب. الملفات المتعقبة هي الملفات التي كانت في اللقطة الأخيرة أو أي ملف أُهِّل حديثًا. ويمكن أن تكون غير معدَّلة، أو معدلة، أو مؤهلة. باختصار، الملفات المتعقبة هي الملفات التي يعرفها جت.

الملفات غير المتعقبة هي كل شيء آخر: أيّ ملفات في مجلد عملك لم تكن ضمن لقطتك الأخيرة وليست في منطقة التأهيل. عندما تستنسخ مستودعا أول مرة، تكون جميع ملفاتك متعقبة وغير معدلة، لأن جت سَحَبها لك للتو ولم تعدّل فيها شيئا بعد.

وعندما تبدأ في تعديل الملفات، سيراها جت معدلة، لأنك غيّرتها عما كانت عليه في إيداعك الأخير. وعندما تشرع في العمل، ستنتقي من هذه الملفات ما تؤهله ثم تودِع هذه التعديلات المؤهلة، ثم تعيد الكَرَّة.



شكل ٨. دورة حياة حالة ملفاتك

فحص حالة ملفاتك

الأداة الرئيسية التي تستعملها لتحديد أي الملفات في أي حالة هي أمر الحالة، `git status`. إذا نفّذت هذا الأمر مباشرةً بعد استنساخ، سترى شيئا مثل هذا:

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

هذا يعني أن لديك مجلد عمل نظيف؛ أي أن لا ملف من ملفاتك المتعقبة معدل. وأيضا لا يرى جت أي ملفات غير متعقبة، وإلا لَسَرَدها هنا. وأخيرا، يخبرك هذا الأمر أي فرع أنت فيه، ويعلِمك أنه لم يختلف عن أخيه الفرع الذي في المستودع البعيد. ذلك الفرع حتى الآن هو دائما `master`، وهو المبدئي؛ لا تحتاج أن تقلق بشأنه هنا. سيناقش التفريع في جت الفروع والإشارات بالتفصيل.

غيّرت شركة جت‌هب (GitHub) اسم المستودع المبدئي من `master` إلى `main` في منتصف عام ٢٠٢٠، ثم تبعتها خدمات استضافة جت الأخرى. لذلك قد تجد أن اسم الفرع المبدئي هو `main` في المستودعات التي أُنشئت حديثًا، وليس `master`. وأيضا يمكنك تغيير اسم الفرع المبدئي (كما رأيت في اسم الفرع المبدئي)، فربما ترى اسمًا آخر للفرع المبدئي.

ولكن ما زال جت نفسه يستعمل `master` اسمًا للفرع المبدئي، لذلك فهذا ما سنستعمل خلال الكتاب.

لنقُل أنك أضفت ملفًا جديدًا إلى مشروعك، مثلا ملف `README` («اقرأني») صغير. إذا لم يكن هذا الملف موجودًا من قبل، ونفّذت أمر الحالة `git status`، فسترى ملفك غير المتعقب هكذا:

CONSOLE

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

    README

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

نرى أن ملفك الجديد غير متعقب، لأنه تحت عنوان "Untracked files" («ملفات غير متعقبة») في ناتج الحالة. «غير متعقب» لا يعني إلا أن جت يرى ملفًا لم يكن في اللقطة السابقة (الإيداع الأخير)، ولم تؤهله بعد. ولن يبدأ جت في ضمه إلى لقطات الإيداعات إلا بعد أن تخبره بذلك بأمر صريح. إنه لا يفعل ذلك لكيلا تضم بالخطأ ملفات رقمية مولدة أو ملفات أخرى لم تشأ ضمها أصلا. ولكنك تريد ضم `README`، فهيا بنا نبدأ تعقب هذا الملف.

تعقب ملفات جديدة

لبدء تعقب ملف جديد، استخدم أمر الإضافة `git add`. مثلا لبدء تعقب ملف `README`، نفّذ هذا:

```
CONSOLE
$ git add README
```

إذا نفذت أمر الحالة مجددا، سترى ملف `README` قد صار متعقبا ومؤهلا للإيداع:

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

    new file:   README
```

نعرف أنه مؤهلٌ لأنه تحت عنوان "Changes to be committed" («تعديلات ستُودَع»). إذا أودعت الآن، فإن نسخة الملف وقت تنفيذ أمر الإضافة `git add` هي التي ستكون في اللقطة التاريخية التالية. تذكر أنك عندما نفذت أمر الابتداء `git init` سابقا، أتبعته بأمر الإضافة `git add <ملفات>` والذي بدأ تعقب الملفات التي في مجلدك. أمر الإضافة `git add` يأخذ مسار ملف أو مجلد. فإن كان مجلدًا فإنه يضيف جميع الملفات التي فيه وفي أي مجلد فرعي فيه.

تأهيل ملفات معدلة

لنعدّل ملفًا جعلناه متعقبًا بالفعل. إذا عدّلت الملف المتعقب `CONTRIBUTING.md` ونفذت أمر الحالة مجددًا، فترى ما يشبه هذا:

```
CONSOLE
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
```

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

    new file:   README

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

    modified:   CONTRIBUTING.md
```

يظهر اسم الملف `CONTRIBUTING.md` تحت عنوان "Changes not staged for commit" («تعديلات غير مؤهلة للإيداع») — والذي يعني أن ملفًا متعقبًا قد تغيّر في مجلد العمل، ولكنه لم يؤهل بعد. لتأهيله، نفذ أمر الإضافة `git add`. يُستخدم أمر الإضافة لأغراض عديدة: لبدء تعقب ملفات جديدة، ولتأهيل الملفات، ولأفعال أخرى مثل إعلان حل الملفات في نزاعات الدمج. ربما من المفيد أن تعتبرها بمعنى «أضف تحديدا هذا المحتوى إلى الإيداع التالي» بدلا من «أضف هذا الملف إلى المشروع». لننفذ `git add` الآن لتأهيل ملف `CONTRIBUTING.md` ثم ننفذ `git status` مجددا:

CONSOLE

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

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

كلا الملفين مؤهلان وسيكونان في إيداعك التالي. لنقُل أنك تذكرت الآن تعديلًا طفيفًا أردته في ملف `CONTRIBUTING.md` قبل إيداعه. ستفتح الملف مجددا، وتصنع تعديلك، وتحفظه وتغلقه. الآن أنت جاهز للإيداع. ولكن، لننفذ `git status` مرة أخرى:

CONSOLE

```
$ vim CONTRIBUTING.md
```

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

    new file:   README
    modified:   CONTRIBUTING.md

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

    modified:   CONTRIBUTING.md
```

يا للهول! لقد صار `CONTRIBUTING.md` مسرودًا أنه مؤهلا وكذلك غير مؤهل. كيف يُعقل هذا؟ يتضح أن جت يؤهل الملف تماما كما هو عندما تنفذ `git add`. فإذا أودعت الآن، فإن ما سيودع هو نسخة `CONTRIBUTING.md` التي كانت موجودة عندما نفذت أمر الإضافة `git add` آخر مرة، وليس نسخة الملف الظاهرة لديك في مجلد العمل عندما تنفذ أمر الإيداع `git commit`. فإن عدّلت ملفًا بعد تنفيذ أمر الإضافة، فتحتاج إلى تنفيذه مرة أخرى لتأهيل النسخة الأخيرة من الملف:

CONSOLE

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

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

الحالة الموجزة

مع كون ناتج أمر الحالة `git status` شاملًا، إلا أنه كثير الكلام. يتيح جت أيضا خيارًا للحالة الموجزة،

لترى تعديلاتك بإيجاز: إذا نفذت `git status -s` أو `git status --short`، فسيعطيك الأمر ناتجًا أقصر كثيرا:

CONSOLE

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

أمام الملفات الجديدة التي لم تُتعقب علامتا `??`. والملفات الجديدة المؤهلة أمامها `A` (اختصار «أضيف»). والملفات المعدّلة أمامها `M` (اختصار «معدّل»). وهكذا. ويوجد عمودان في الناتج أمام أسماء الملفات: العمود الأيسر يوضّح حالته في منطقة التأهيل، والعمود الأيمن يوضّح حالته في شجرة العمل. لذا ففي ناتج مثالنا هذا، ملف `README` معدّل في مجلد العمل ولكنه ليس مؤهلا بعد، ولكن ملف `lib/simplegit.rb` معدّل ومؤهل. وملف `Rakefile` معدّل ومؤهل ثم معدّل مرة أخرى، ففيه تعديلات مؤهلة وتعديلات غير مؤهلة.

تجاهل ملفات

سيكون لديك غالبا فئة من الملفات التي لا تريد من جت أن يضيفها آليًا ولا حتى أن يخبرك أنها غير متعقبة. هذه غالبا ملفات مولّدة آليًا مثل ملفات السجلات أو ملفات مبنية. يمكنك في مثل هذه الحالات إنشاء ملف يسمى `.gitignore` (يبدأ بنقطة، لجعله مخفيا) والذي يسرد أنماط أسماء هذه الملفات ليتجاهلها. هذا مثال على ملف `.gitignore` :

CONSOLE

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

يطلب السطر الأول من جت أن يتجاهل أي ملفات ينتهي اسمها بـ ".o" أو ".a" — ملفات الكائنات

وملفات المكتبات المضغوطة التي قد تُنتج أثناء بناء مصدرك البرمجي. ويطلب السطر الثاني من جت أن يتجاهل جميع الملفات التي ينتهي اسمها بعلامة التلدة (`~`)، التي تستعملها محررات نصوص عديدة مثل Emacs لتمييز الملفات المؤقتة. يمكنك أيضا إضافة مجلد `log` أو `tmp` أو `pid`، أو الوثائق المولدة آليًا، إلخ. إعداد ملف التجاهل `gitignore`. لمستودعك الجديد قبل الانطلاق في المشروع هو تفكير حسن عموما، لكيلا تودع بالخطأ ملفات يقينا لا تريدها في مستودعك.

إليك قواعد الأنماط التي تستطيع استعمالها في ملف التجاهل:

- تُهمل الأسطر الفارغة أو الأسطر البادئة بعلامة `#`.
- يمكن استعمال أنماط توسيع المسارات (glob) المعتادة (ستُوّضح بالتفصيل)، وستُطبق في جميع مجلدات شجرة العمل.
- يمكنك بدء الأنماط بفاصلة مائلة (`/`) لمطابقة الملفات أو المجلدات في المجلد الحالي فقط، وليس أي مجلد فرعي.
- يمكنك إنهاء الأنماط بفاصلة مائلة (`/`) لتحديد مجلد.
- يمكنك نفي نمط ببدئه بعلامة تعجب (`!`).

تشبه أنماط glob نسخة مُيسَّرة من «التعابير النمطية»، وتستعملها الصدفات. فتُطابق النجمة (`*`) صفر أو أكثر من المحارف؛ ويُطابق `[abc]` أي حرف داخل القوسين المربعين (أي `a` أو `b` أو `c` في هذه الحالة)؛ وتُطابق علامة الاستفهام الغربية (`?`) محرَفًا واحدًا؛ ولمطابقة مدًى من المحارف، نكتب أول محرف وآخر محرف (بترتيبهما في Unicode) داخل قوسين مربعين وبينهما شرطة، فمثلا لمطابقة رقمًا من الأرقام المغربية (من `0` إلى `9`) نكتب `[0-9]`. يمكنك أيضا استخدام نجمتين لمطابقة أي عدد من المجلدات الفرعية، فمثلا يطابق النمط `a/**/z` كلًا من `a/z` و `a/b/z` و `a/b/c/z` وهكذا.

إليك مثال آخر على ملف `gitignore` :

```
# a تجاهل كل الملفات ذات الامتداد ⇐#
*.a

# a بالأعلى ⇐# لكن تعقب lib.a، حتى لو كنت تتجاهل جميع ملفات
!lib.a

# مثلا subdir/TODO في المجلد الحالي، وليس TODO تجاهل فقط ⇐#
/TODO

# build اسمه وكل شيء داخله تجاهل أي مجلد ⇐#
build/

# مثلا doc/server/arch.txt ولكن ليس doc/notes.txt تجاهل ⇐#
doc/*.txt

# pdf في مجلد doc أو أي مجلد فرعي فيه تجاهل جميع ⇐#
doc/**/*.pdf
```

إذا أردت نقطة بداية لمشروعك، فإن جت‌هب يرعى قائمةً شاملة نسبيًا من أمثلة ملفات التجاهل الحسنة لعشرات المشروعات واللغات في https://github.com/github/gitignore.

قد يكون لدى المستودع في الحالات اليسيرة ملف تجاهل واحد في مجلد الجذر، والذي يطبق على المستودع بجميع مجلداته الفرعية. ولكن ممكن كذلك وجود ملفات تجاهل أخرى في مجلدات فرعية. وملفات التجاهل الداخلية هذه لا تطبق قواعدها إلا على الملفات التي في مجلداتها. ومثلا لدى مستودع نواة لينكس ٢٠٦ ملف تجاهل.

يخرج عن نطاق الكتاب الغوص في تفاصيل ملفات التجاهل المتعددة؛ انظر `man gitignore` للتفاصيل.

رؤية تعديلاتك المؤهلة وغير المؤهلة

إذا كنت تجد ناتج أمر الحالة `git status` شديد الغموض — تريد معرفة ما الذي عدّلته على وجه التحديد، وليس مجرد أسماء الملفات التي تعدّلت — فيمكنك استخدام أمر الفرق `git diff`. نتناوله بالتفصيل فيما بعد، لكنك في الغالب تستخدمه لإجابة أحد التساؤلين: ما الذي عدّلته ولم تؤهله بعد؟ وما الذي أهّلته وعلى وشك إيداعه؟ وبالرغم من أن أمر الحالة `git status` يجيبهما إجابةً عامة جدا بسرد أسماء الملفات، إلا أن أمر الفرق `git diff` يُظهر لك بالتحديد السطور المضافة والمزالة: الرُقعة، إن جاز التعبير.

لنقُل أنك عدّلت ملف `README` مجددا وأهّلته، ثم عدّلت ملف `CONTRIBUTING.md` من غير تأهيله. إذا نفذت أمر الحالة، سترى من جديد شيئا مثل هذا:

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

    modified:   README

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

    modified:   CONTRIBUTING.md
```

لرؤية ما الذي عدّلته ولم تؤهله بعد، اكتب `git diff` من غير أي معاملات أخرى:

```
CONSOLE
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
```

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

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

يقارن هذا الأمر بين محتويات مجلد العمل ومنطقة التأهيل، ويُخبرك الناتج بما عدّلته ولم تؤهله بعد.

إذا أردت رؤية ما الذي أهّلته ليكون في الإيداع التالي، يمكنك استخدام `git diff --staged`. يقارن هذا الأمر بين تعديلاتك المؤهلة وإيداعك الأخير:

CONSOLE

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

مهمٌ ملاحظة أن `git diff` وحده لا يُظهر جميع التعديلات التي تمت بعد الإيداع الأخير—إنما التعديلات غير المؤهلة فقط. فإذا أهّلت جميع تعديلاتك، فلن يعطيك `git diff` أي ناتج.

مثالٌ آخر: إذا أهّلت ملف `CONTRIBUTING.md` ثم عدّلته، يمكنك استخدام أمر الفرق لمعرفة التعديلات على الملف التي أُهِّلت والتعديلات التي لم تؤهل. فإذا كانت بيئتنا تبدو هكذا:

CONSOLE

```
$ git add CONTRIBUTING.md
$ echo '# test line' >> CONTRIBUTING.md
$ git status
```

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

    modified:   CONTRIBUTING.md

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

    modified:   CONTRIBUTING.md
```

يمكننا إذًا استخدام `git diff` لرؤية ما الذي لم يؤهل بعد:

CONSOLE

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

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

واستخدام `git diff --cached` لرؤية ما الذي أهّلته حتى الآن (الخياران `--staged` و `--cached` مترادفان):

CONSOLE

```
$ git diff --cached
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
```

```
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

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

فروقات جت باستخدام أداة خارجية

سنستمر في استخدام أمر الفرق `git diff` بطرائق متنوعة خلال الكتاب. ولكن توجد طريقة أخرى لرؤية هذه الفروقات إذا كنت تفضل برنامج عرض فروقات خارجي أو رسومي. يمكنك باستخدام `git difftool` بدلا من `diff` أن ترى هذه الفروقات في برنامج مثل emerge أو vimdiff أو برامج كثيرة أخرى (بما فيها البرامج التجارية). نفذ `git difftool --tool-help` لترى ما المتاح على نظامك.

إيداع تعديلاتك

الآن وقد هيّأت منطقة تأهيلك كما تحب، يمكنك أن تودع تعديلاتك. تذكر أنه لن يُحفظ في هذا الإيداع أي شي ما زال غير مؤهل — أيْ أيّ ملفات أنشأتها أو عدّلتها ولم تنفذ `git add` عليها بعدما عدلتها؛ بل ستبقى ملفات معدلة على القرص. لنقُل أنك عندما نفذت أمر `git status` رأيت أن كل شيء مؤهل، لذا فأنت الآن مستعد لإيداع تعديلاتك. أسهل طريقة للإيداع هي كتابة `git commit`:

CONSOLE

```
$ git commit
```

فعل هذا يفتح محررك المختار.

يعيّن «محررَك المختار» متغيرُ بيئة المحرر، `EDITOR`، في صدفتك، والذي غالبا يكون `vim` أو `emacs`. مع أنك تستطيع جعله أي شيء تريده بالأمر `git config --global core.editor` كما رأيت في البدء

يُظهر المحرر النصَ التالي (المثال من 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
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#   new file:   README
#   modified:   CONTRIBUTING.md
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

والتي يترجم أولها إلى: «أدخل رسالة الإيداع لتعديلاتك. الأسطر البادئة بعلامة `#` ستُهمل، ورسالة فارغة ستلغي الإيداع.»

نرى أن رسالة الإيداع المبدئية تشمل ناتج أمر الحالة الأحدث في صورة تعليق، وأن بها سطر فارغ في أولها. يمكنك إزالة هذه التعليقات وكتابة رسالة إيداعك، أو تركها في مكانها لتتذكر ماذا تودع.

إذا احتجت تذكيرًا أشد تفصيلًا بما عدّلت، يمكنك إمرار الخيار `-v` لأمر الإيداع، `git commit`. يضع هذا فروقات تعديلاتك في المحرر، كي ترى بالتحديد ما التعديلات الذاهبة للإيداع.

عندما تحفظ وتغلق المحرر، سيصنع جت إيداعك بالرسالة التي كتبتها (باستثناء الفروقات والتعليقات، أي الأسطر البادئة بعلامة `#`).

يمكنك عوضًا عن ذلك كتابة رسالة إيداعك في أمر الإيداع نفسه، بالخيار `-m`، مثل هذا:

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

الآن قد صنعت إيداعك الأول! نرى أن الإيداع أعطاك بعض المعلومات عن نفسه، مثل الفرع الذي أودعت فيه (`master`)، وبصمة الإيداع (`463dc4f`)، وعدد الملفات المعدّلة (`2 files changed`)، وإحصاءات عن السطور المضافة والمزالة في هذا الإيداع (`2 insertion(+)`).

تذكر أن هذا الإيداع يسجل اللقطة التي أعددتها في منطقة تأهيلك. أيّ شيء عدّلته ولم تؤهله سيظل جالسًا في مجلد العمل وهو معدَّل؛ يمكنك صنع إيداع آخر لإضافته إلى تاريخ مشروعك. في كل مرة تصنع إيداعًا، تسجل من مشروعك لقطة يمكنك إرجاع مشروعك إليها أو المقارنة معها فيما بعد.

تخطي منطقة التأهيل

مع أن منطقة التأهيل مفيدة لدرجة مدهشة في صياغة الإيداعات كما تشاء بالضبط، إلا أنها أحيانا أعقد مما تحتاج في سير عملك. يوفر لك جت اختصارا سهلًا متى أردت تخطي منطقة التأهيل: إضافة الخيار `-a` إلى أمر `git commit` تجعل جت يؤهل من نفسه كل ملف متعقب قبل هذا الإيداع، لتتخطى مرحلة الإضافة:

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

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

    modified:   CONTRIBUTING.md

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

لاحظ أنك لم تحتجْ إلى تنفيذ `git add` على ملف `CONTRIBUTING.md` في هذه الحالة قبل الإيداع، لأن خيار `-a` يضم جميع الملفات المعدلة. هذا مريح، لكن احذر: قد يضم هذا الخيار تعديلات غير مرغوب فيها.

إزالة ملفات

لإزالة ملف من جت، عليك أن تزيله من ملفاتك المتعقبة (أو بتعبير أدق، من منطقة تأهيلك)، ثم تودع. يفعل أمر الإزالة `git rm` هذا، وأيضا يزيل الملف من مجلد عملك حتى لا تراه ملفًا غير متعقب في المرة القادمة.

إذا أزلت الملف من مجلد عملك فقط، سيظهر تحت عنوان "Changes not staged for commit" («تعديلات غير مؤهلة للإيداع») في ناتج أمر الحالة:

CONSOLE

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

        deleted:    PROJECTS.md

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

عندئذٍ تنفيذك أمر `git rm` يؤهل إزالة الملف:

CONSOLE

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

    deleted:    PROJECTS.md
```

عندما تودع في المرة القادمة ستجد أن الملف قد ذهب ولم يعد متعقبًا. وإذا كنت قد عدّلت الملف أو كنت قد أضفته بالفعل إلى منطقة التأهيل، فعليك فرض الإزالة بالخيار `-f`. هذه ميزة أمان لكيلا تزيل بالخطأ بيانات لم تسجلها بعد في لقطة ولا يمكن استردادها من جت.

أمرٌ آخر مفيد قد تود فعله هو إبقاء الملف في شجرة عملك لكن إزالته من منطقة تأهيلك. بلفظ آخر، تريد أن ينسى جت وجوده ولا يتعقبه ولكن يبقيه لك على قرصك. هذا مفيد خصوصا إن نسيت إضافة شيء إلى ملف التجاهل `.gitignore` ثم أهّلته بالخطأ، مثل ملف سجل كبير أو مجموعة من الملفات المبنية. استعمل الخيار `--cached` لهذا:

CONSOLE

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

يمكنك إعطاء الأمر أسماء ملفات أو مجلدات أو أنماط توسيع المسارات (glob). يعني هذا أن بإمكانك فعل أشياء مثل:

CONSOLE

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

لاحظ الشرطة المائلة الخلفية (`\`) قبل النجمة `*`؛ هذا ضروري، لأن جت يقوم بنفسه بتوسيع أسماء

الملفات بعد أن تقوم صدفتك بتوسيعها. فبغير الشرطة المائلة الخلفية، ستوسع الصدفة أسماء الملفات قبل أن يراها جت. يحذف هذا الأمر جميع الملفات ذات الامتداد `.log` في مجلد `log/`. أو يمكنك فعل شيء مثل هذا:

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

يحذف هذا الأمر جميع الملفات المنتهي اسمها بعلامة التلدة (`~`).

نقل ملفات

لا يتعقب جت حركة الملفات تعقبًا صريحًا، خلافا لكثير من أنظمة إدارة النسخ الأخرى. فإذا غيّرت اسم ملف في جت، لا يخزن جت بيانات وصفية تخبره أنك غيرته. لكن جت ذكي جدا في تخمين ذلك وهو أمام الأمر الواقع — سنتعامل مع اكتشاف نقل الملفات بعد قليل.

لذا فقد تجد أنه من المحيّر وجود أمر «نقل» (`mv`) في جت. فإذا أردت تغيير اسم ملف في جت، يمكنك طلبه هكذا:

```
CONSOLE
$ git mv file_from file_to
```

وسيعمل كما ينبغي. وفي الحقيقة، إذا نفذت أمرا مثل هذا، ونظرت إلى الحالة، سترى أن جت يعتبره تغيير اسم ملف:

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

    renamed:    README.md -> README
```

ولكن هذا مكافئ لتنفيذ شيء مثل هذا:

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

يخمن جت في سِرّه أن الاسم قد تغير، لذا فلا يهم إن غيّرت اسمه بهذه الطريقة أو عبر نظام التشغيل أو مدير الملفات (مثلا بأمر النظام `mv`). الفرق الحقيقي الوحيد هو أن `git mv` أمر واحد وليس ثلاثة؛ إنه وسيلة راحة. والأهم أنك تستطيع استخدام أي أداة تريدها لتغيير أسماء الملفات، ثم تتعامل مع الإضافة والإزالة في جت فيما بعد، قبل الإيداع.

رؤية تاريخ الإيداعات

بعدما صنعت عدداً من الإيداعات، أو استنسخت مستودعًا ذا تاريخ من الإيداعات بالفعل، قد تود الالتفات إلى الماضي ورؤية ماذا حدث. أسهل وأقوى أداة لفعل هذا هي أمر السجل، `git log`:

تستعمل هذه الأمثلة مشروعًا صغيرا جدا يسمى "simplegit". للحصول على المشروع، نفذ:

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

عندما تنفذ `git log` داخل هذا المشروع، ترى شيئا مثل هذا:

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

    Change version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
```

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

    Remove unnecessary test

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

    Initial commit
```

عندما تنادي أمر السجل بلا مُعامِلات، أيْ `git log` فقط، فإنه افتراضيًا يسرد لك الإيداعات التي في هذا المستودع بترتيب زمني عكسي؛ أيْ أن الإيداع الأحدث يظهر أولًا. يسرد هذا الأمر كما ترى كل إيداع مع بصمته واسم مؤلفه وبريده وتاريخ الإيداع ورسالته.

يتيح أمر السجل عددًا عظيمًا متنوعًا من الخيارات لتُظهر بالضبط ما تريد. سنعرض لك هنا بعضًا من أشهرها.

واحد من أكثر الخيارات إفادةً هو `-p` أو `--patch` («رُقعة»)، والذي يظهر لك الفرق (أي **الرقعة**) الذي أتى به كل إيداع. يمكنك أيضا تقييد عدد السجلات المعروضة، مثلا بالخيار `-2` لإظهار آخر بيانَيْن فقط.

CONSOLE

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

    Change version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
```

```
      s.platform  =   Gem::Platform::RUBY
      s.name      =   "simplegit"
-     s.version   =   "0.1.0"
+     s.version   =   "0.1.1"
      s.author    =   "Scott Chacon"
      s.email     =   "schacon@gee-mail.com"
      s.summary   =   "A simple gem for using Git in Ruby code."

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

    Remove unnecessary test

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

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

يعرض هذا الخيار المعلومات نفسها أيضا ولكن مع إتباع كل بيان بالفروقات. هذا مفيد جدا لمراجعة الأكواد (code review) أو للنظر السريع فيما حدث في سلسلة من الإيداعات التي أضافها زميل. ولدى أمر السجل كذلك عددًا من خيارات التلخيص. فمثلا إذا أردت رؤية إحصاءات مختصرة عن كل إيداع، جرّب خيار الإحصاء `--stat`:

CONSOLE

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

```
    Change version number

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

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

    Remove unnecessary test

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

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

    Initial commit

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

خيار `--stat` كما ترى يطبع لك تحت بيان كل إيداع قائمة بالملفات المعدلة وعددها وعدد السطور المضافة والمزالة في هذه الملفات. ثم يضع تلخيصًا لهذه المعلومات في النهاية.

وخيار آخر مفيد جدًا هو `--pretty` («جميل»). والذي يغيّر ناتج السجل إلى صيغ أخرى غير الصيغة المبدئية. تأتي مع جت بعض القيم التي يمكن استعمالها مع هذا الخيار. قيمة `oneline` («سطر واحد») تطبع كل إيداع على سطر وحيد، والذي يفيد عندما تكون ناظرًا إلى إيداعات كثيرة. وكذلك، القيم `short` («قصير») و `full` («كامل») و `fuller` («أكمل») تُظهر لك ناتجًا مثل المبدئي مع زيادة أو نقصان في بعض المعلومات.

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

القيمة الأكثر إمتاعًا هي `format` («صياغة»)، والتي تتيح لك تحديد صيغة ناتج السجل التي تفضلها. هذا مفيد خصوصًا عندما تقوم بتوليد ناتج لكي يقرؤه ويحلله برنامج أو بُريمج (script) — فلأنك تحدد الصيغة بصراحة ووضوح، فإنك تطمئن أنها لن تتغيّر مع تحديث جت.

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

يسرد متغيرات مفيدة لصياغة السجلات باستخدام `git log --pretty=format` بعض المتغيرات المفيدة التي تفهمها `format`:

جدول ا. متغيرات مفيدة لصياغة السجلات باستخدام `git log --pretty=format`

المتغير	وصف الناتج
%H	بصمة الإيداع
%h	بصمة الإيداع المختصرة
%T	بصمة الشجرة
%t	بصمة الشجرة المختصرة
%P	بصمات الآباء

المتغير	وصف الناتج
%p	بصمات الآباء المختصرة
%an	اسم المؤلف
%ae	بريد المؤلف
%ad	تاريخ التأليف (الصيغة تتبع `--date=option`)
%ar	تاريخ التأليف، نسبي
%cn	اسم المودِع
%ce	بريد المودِع
%cd	تاريخ الإيداع
%cr	تاريخ الإيداع، نسبي
%s	الموضوع

ربما تتساءل عن الفرق بين **المؤلف والمودِع**. المؤلف هو من كتب العمل في الأصل، بينما المودِع هو من طبّق العمل في النهاية. فمثلا إذا أرسلت رقعة إلى مشروع، وطبّقها أحد الأعضاء الأساسيين، فيجب الاعتراف بالفضل لكليكما — أنت مؤلفًا، والعضو الأساسي مودِعًا. سنتناول هذا التمييز بالتفصيل في ~~جت~~ المونج.

القيمتان `oneline` و `format` مفيدتان خصوصًا مع خيار آخر لأمر السجل يسمى `--graph` («رسم»). يضيف هذا الخيار رسمًا لطيفًا بالمحارف لإظهار تاريخ التفريع والدمج.

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

سيصير هذا النوع من الناتج ممتعًا أكثر أثناء تناولنا التفريع والدمج في الباب التالي.

هذه فقط بعض خيارات تنسيق الناتج اليسيرة المتاحة في `git log` — متاح عدد أكبر من ذلك كثيرا. يسرد خيارات شائعة لأمر السجل الخيارات التي تناولناها حتى الآن، وكذلك بعض خيارات التنسيق الشائعة الأخرى التي قد تفيد، إضافةً إلى كيفية تعديل ناتج أمر السجل.

جدول ٢. خيارات شائعة لأمر السجل

الخيار	الوصف
`-p`	أظهر الرقعة التي أتى بها كل إيداع.
`--stat`	أظهر إحصاءات الملفات المعدّلة في كل إيداع.
`--shortstat`	اعرض فقط سطر التعديلات/الإضافات/الإزالات من أمر `--stat`.
`--name-only`	اسرد أسماء الملفات المعدلة بعد كل إيداع.
`--name-status`	اسرد أسماء الملفات مرفقة بحالتها: معدّل/مضاف/مزال.

الخيار	الوصف
`--abbrev-commit`	أظهر فقط الحروف القليلة الأولى من البصمة، بدلًا من الأربعين جميعًا.
`--relative-date`	اعرض التاريخ بصيغة نسبية (مثلا "2 weeks ago") بدلا من صيغة التاريخ الكاملة.
`--graph`	اعرض رسمًا بالمحارف لتاريخ التفريع والدمج بجانب ناتج السجل.
`--pretty`	اعرض الإيداعات بصيغة أخرى. قيم الخيار المتاحة تشمل `oneline` و `short` و `full` و `fuller` و `format` (والتي تتيح لك تحديد صياغتك المخصوصة).
`--oneline`	اختصار لاستخدام `--pretty=oneline --abbrev-commit` معًا.

تقييد ناتج السجل

إضافةً إلى خيارات صياغة الناتج، يتيح أمر السجل عددًا من خيارات تقييد الناتج؛ أيْ خيارات تتيح لك إظهار جزء من الإيداعات فقط. لقد رأيت أحد هذه الخيارات بالفعل — خيار `-2` الذي يُظهر آخر إيداعين فقط. الحقيقة أن استخدام `-<ن>`، حيث `ن` هو أي عدد صحيح موجب، يُظهر لك آخر `ن` إيداعًا. لن تستخدم هذا كثيرا في الواقع، لأن جت بطبيعته يمرر الناتج كله إلى برنامج عرض ("pager" مثل `less`) حتى ترى ناتج السجل صفحةً صفحة.

لكن خيارات التقييد بالزمن مثل `--since` («منذ») و `--until` («حتى») مفيدة جدا. مثلا، هذا الأمر يسرد الإيداعات التي تمت خلال الأسبوعين السابقين:

CONSOLE

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

يعمل هذا الأمر مع العديد من الصيغ — يمكنك تحديد تاريخ محدد مثل `"2008-01-15"` أو تاريخ نسبي مثل `"2 years 1 day 3 minutes ago"`.

يمكنك أيضا سرد الإيداعات المطابقة لمعايير بحث معينة. مثلا خيار `--author` يتيح لك سرد إيداعات مؤلف معين فقط، و `--grep` يتيح لك البحث عن كلمات معينة في رسائل الإيداعات.

يمكنك استخدام `--author` أو `--grep` أكثر من مرة في المرة، والذي يسرد الإيداعات التي توافق أي نمط `--author` معطى وتوافق أي نمط `--grep` معطى؛ ولكن إضافة خيار `--all-match` يقيّد الناتج إلى الإيداعات الموافقة لجميع أنماط `--grep`.

مصفاة مفيدة جدا أخرى هي خيار `-S` (والمعروف بالاسم الدارج: خيار «فأس» جت)، والذي يأخذ سلسلة نصية ولا يظهر إلا الإيداعات التي عدّلت عدد تواجداتها. مثلا، إذا أردت إظهار آخر إيداع أضاف أو أزال إشارة إلى دالة معينة، يمكنك تنفيذ:

CONSOLE

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

آخر خيار تصفية مفيد جدا لأمر السجل هو إعطاؤه مسار. فإذا أعطيته مجلدًا أو ملفًا، فإنه يقيّد ناتج السجل إلى الإيداعات التي عدّلت هذه الملفات. يكون هذا دائما آخر خيار وفي الغالب يُسبق بشرطتين (`--`) لفصل المسارات عن الخيارات:

CONSOLE

```
$ git log -- path/to/file
```

نسرد في خيارات تقييد ناتج أمر السجل هذه الخيارات وبعض الخيارات الأخرى حتى تكون مرجعًا لك.

جدول ٣. خيارات تقييد ناتج أمر السجل

الخيار	الوصف
-<ن>	أظهر فقط آخر ن إيداعًا.
--since أو --after	قيّد الناتج إلى الإيداعات التي تمت بعد التاريخ المعطى.
--until أو --before	قيّد الناتج إلى الإيداعات التي تمت قبل التاريخ المعطى.
--author	لا تظهر إلا الإيداعات التي يطابق اسم مؤلفها السلسلة النصية المعطاة.
--committer	لا تظهر إلا الإيداعات التي يطابق اسم مودِعها السلسلة النصية المعطاة.
--grep	لا تظهر إلا الإيداعات التي تشتمل رسالتها على السلسلة النصية المعطاة.
-S	لا تظهر إلا الإيداعات التي أضافت أو أزالت سطورًا برمجية فيها السلسلة النصية المعطاة.

مثلا، إذا أردت رؤية أيّ الإيداعات عدّلت ملفات الاختبارات في مصدر جت والتي أودعها Junio Hamano في شهر أكتوبر عام ٢٠٠٨، وليست إيداعات دمج، يمكنك فعل شيء مثل هذا:

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

```
51a94af - Fix "checkout --track -b newbranch" on detached HEAD
b0ad11e - pull: allow "git pull origin $something:$current_branch" into an unborn branch
```

حوالي أربعين ألف إيداع في تاريخ مصدر جت، وهذا الأمر لا يُظهر منها إلا الإيداعات الستة المطابقة لتلك المعايير.

منع عرض إيداعات الدمج

حسب أسلوب سير العمل في مستودعك، قد يكون عدد ضخم من الإيداعات في تاريخ سجلك مجرد إيداعات دمج، وهي لا تفيد كثيرًا. لمنع عرضها وإزحامها تاريخ سجلك، أضف إلى أمر السجل خيار `--no-merges` («لا دمج»).

التراجع عن الأفعال

قد تحتاج في أي مرحلة إلى التراجع عن فعلٍ ما. سنرى هنا بعض الأدوات الأساسية للتراجع عن تعديلاتك. كن حذرًا، لأن بعض هذه التراجعات لا يمكن التراجع عنها فيما بعد. هذه من المناطق القليلة في جت التي يمكنك أن تفقد فيها شيئا من عملك إذا فعلت شيئا خطأ.

واحد من أشهر التراجعات هو عندما تودع قبل الأوان وتنسى إضافة ملفات أو تخطئ في رسالة إيداعك. إذا أردت إعادة هذا الإيداع، فقم بالتعديلات التي نسيتها، وأهّلها، ثم أودع مجددا مع خيار التصحيح `--amend`:

CONSOLE

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

يأخذ هذا الأمر منطقة تأهيلك ويستخدمها للإيداع. وإذا لم تقم بأي تعديلات منذ إيداعك الأخير (مثلا نفذت هذا الأمر مباشرةً بعد إيداعك السابق)، فإن لقطتك ستتطابق تمامًا بلا اختلاف، ولن تتغيّر سوى رسالة الإيداع.

سيظهر لك محرر رسالة الإيداع، ولكنك ستجد فيه رسالة الإيداع السابقة في انتظارك لتعدّلها إن شئت أو تغيّرها تمامًا.

مثلا، إذا أودعت ثم أدركت أنك نسيت تأهيل تعديلات على ملف تريدها في هذا الإيداع، يمكنك فعل شيء مثل هذا:

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

ستجد في النهاية إيداعًا واحدًا، فالإيداع الثاني يحل محل الأول.

مهمٌ فهم أنك عندما تصحح إيداعك الأخير، فإنك لا تصلحه ولكن تستبدله برُمّتِه وتضع مكانه إيداعًا جديدًا محسَّنًا وتزيح القديم عن الطريق. في الحقيقة، هذا كأن الإيداع السابق لم يحدث أصلًا، ولن يظهر في تاريخ مستودعك.

الفائدة الواضحة لتصحيح الإيداعات هو التحسينات الطفيفة للإيداع الأخير، بغير إزحام تاريخ مستودعك برسائل إيداعات من نوعية «عذرا، نسيت إضافة ملف» أو «سحقا، خطأ مطبعي في الإيداع السابق، أصلحته».

لا تصحح إلا الإيدعات التي لا تزال محلية ولم تُدفع بعد إلى أي مكان آخر. فتصحيح إيداع قد دُفع بالفعل ثم فرض الدفع (`git push --force`) سيسبب مشاكل للمتعاونين معك. لمعرفة ما سيحدث إن فعلت هذا وكيف تتعافى إذا كنت الطرف المتلقي، اقرأ مخذورات إعادة التأسيس.

إلغاء تأهيل ملف مؤهل

سيوضح الفصلان التاليان كيف تتعامل مع التعديلات في منطقة تأهيلك ومجلد عملك. الجميل أن الأمر الذي تستخدمه لمعرفة حالة إحدى هاتين المنطقتين يذكّرك أيضا بكيفية التراجع عن تعديلاتهما. لنقُل مثلا أنك عدّلت ملفين وأردت إيداع كلٍ منهما في إيداع منفصل، ولكنك كتبت خطأً `git add *` فأهّلت كليهما. كيف يمكنك إلغاء تأهيل أحدهما؟ أمر الحالة يذكّرك:

CONSOLE

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

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

مباشرةً تحت "Changes to be committed" («تعديلات ستُودَع») تجده يقول استخدم `git reset` لإلغاء التأهيل. فلنعمل بهذه النصيحة إذًا، لإلغاء تأهيل ملف `CONTRIBUTING.md`:

CONSOLE

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

    renamed:    README.md -> README

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

    modified:   CONTRIBUTING.md
```

هذا الأمر غريب قليلًا، لكنه يعمل. ملف `CONTRIBUTING.md` معدّل لكنه عاد من جديد غير مؤهل.

صدقًا إن `git reset` أمر خطير، خصوصا مع الخيار `--hard`. مع ذلك، فإن الملف الذي في مجلد عملك لم يُمس في الموقف الموضح بالأعلى، لذا فهذا الأمر آمن نسبيا في مثل هذا الموقف.

هذا الأمر السحري هو كل ما تحتاج معرفته الآن عن أمر الإرجاع `git reset`. سنغوص في ~~Reset~~ ~~Demystified~~ في تفاصيل أعمق كثيرا عن أمر الإرجاع وماذا يفعل وكيف تتقنه لتفعل أفعالا شيقة وممتعة جدا.

إعادة ملف معدل إلى حالته قبل التعديل

ماذا لو أدركت أنك لم تعد تريد تعديل ملف `CONTRIBUTING.md` من الأساس؟ كيف يمكنك إرجاعه إلى حالته عند الإيداع الأخير (أو الاستنساخ الأول، أو كيفما حصلت عليه في مجلد عملك)؟ لحسن الحظ، يخبرك أمر الحالة بهذا أيضا. في ناتج المثال الأخير، كان جزء التعديلات غير المؤهلة هكذا:

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

    modified:   CONTRIBUTING.md
```

فيخبرك أن تستخدم الأمر `git checkout -- <ملفات>` لتجاهل التعديلات التي في مجلد عملك. لنفعل ما يخبرنا به:

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

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

كما ترى، أُلغيت التعديلات.

من المهم جدا فهم أن `git checkout -- <ملفات>` أمر خطير؛ أي تعديلات محلية قمت بها على هذا الملف قد ضاعت، فقد أزال جت للتو هذا الملف ووضع مكانه آخر نسخة مؤهلة أو مودعة منه. إياك أبدا أن تستعمل هذا الأمر، إلا أن تكون واعيا أشد الوعي أنك لا تريد هذه التعديلات المحلية غير المحفوظة.

إذا أردت الإبقاء على تعديلاتك على هذا الملف لكنك لا تزال تريد إزاحته جانبًا الآن، فسنشرح التخبئة والتفريع في التفريع في جت؛ هاتان الطريقتان في العموم أفضل.

تذكر أن أي شيء تودعه في جت يمكن شِبه دائمًا استعادته. حتى الإيداعات في الفروع المحذوفة أو الإيداعات المبدلة بخيار التصحيح (`--amend`) يمكن استعادتها (انظر استرجاع البيانات لاستعادة البيانات). مع ذلك، أي شيء تفقده لم يكن مودعًا، صعب أن تراه مرة أخرى.

التراجع بأمر الاستعادة git restore

أضافت النسخة 2.23.0 من جت أمرًا جديدًا: `git restore`. هذا في الأصل بديل لأمر الإرجاع `git reset` الذي ناقشناه للتو. ابتداءً من النسخة 2.23.0 من جت، سيستخدم جت أمر الاستعادة `git restore` بدلا من أمر الإرجاع `git reset` في الكثير من عمليات التراجع.

لنرتد على آثارنا قَصصًا ونعيد الكرّة ونتراجع بأمر الاستعادة `git restore` بدلًا من أمر الإرجاع `git reset`.

إلغاء تأهيل ملف مؤهل بأمر الاستعادة

سيوضح الفصلان التاليان كيف تتعامل مع التعديلات في منطقة تأهيلك ومجلد عملك بأمر الاستعادة `git restore`. الجميل أن الأمر الذي تستخدمه لمعرفة حالة إحدى هاتين المنطقتين يذكّرك أيضا بكيفية التراجع عن تعديلاتهما. لنقُل مثلا أنك عدّلت ملفين وأردت إيداع كلٍ منهما في إيداع منفصل، ولكنك كتبت خطأً `git add *` فأهّلت كليهما. كيف يمكنك إلغاء تأهيل أحدهما؟ أمر الحالة يذكّرك:

```
CONSOLE
$ git add *
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   CONTRIBUTING.md
    renamed:    README.md -> README
```

مباشرةً تحت "Changes to be committed" («تعديلات ستُودَع») تجده يقول استخدم `git restore --staged <ملفات>` لإلغاء التأهيل. فلنعمل بهذه النصيحة إذًا، لإلغاء تأهيل ملف `CONTRIBUTING.md`:

```
CONSOLE
$ git restore --staged CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    renamed:    README.md -> README

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

ملف `CONTRIBUTING.md` معدّل لكنه عاد من جديد غير مؤهل.

إعادة ملف معدل إلى حالته قبل التعديل بأمر الاستعادة

ماذا لو أدركت أنك لم تعد تريد تعديل ملف `CONTRIBUTING.md` من الأساس؟ كيف يمكنك إرجاعه إلى حالته عند الإيداع الأخير (أو الاستنساخ الأول، أو كيفما حصلت عليه في مجلد عملك)؟ لحسن الحظ، يخبرك أمر الحالة بهذا أيضا. في ناتج المثال الأخير، كان جزء التعديلات غير المؤهلة هكذا:

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

فيخبرك أن تستخدم الأمر `git restore <ملفات>` لتجاهل التعديلات التي في مجلد عملك. لنفعل ما يخبرنا به:

```
CONSOLE
$ git restore CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    renamed:    README.md -> README
```

من المهم جدا فهم أن `git restore <ملفات>` أمر خطير، أي تعديلات محلية قمت بها على هذا الملف قد ضاعت، فقد أزال جت للتو هذا الملف ووضع مكانه آخر نسخة مؤهلة أو مودعة منه. إياك أبدا أن تستعمل هذا الأمر، إلا أن تكون واعيا أشد الوعي أنك لا تريد هذه التعديلات المحلية غير المحفوظة.

التعامل مع المستودعات البعيدة

حتى تستطيع التعاون في أي مشروع يستخدم جت، تحتاج إلى معرفة كيف تدير مستودعاتك البعيدة.

المستودعات البعيدة هي نُسخ من مشروعك، وهذه النسخ مستضافة على الإنترنت أو على شبكة داخلية. يمكن أن يكون لديك عددًا منها، وكل واحد منها غالبًا يسمح لك إما بالقراءة فحسب («القراءة فقط»)، وإما بالتحرير كذلك («القراءة والكتابة»). والتعاون مع الآخرين يشمل إدارة هذه المستودعات البعيدة ودفع البيانات إليها وجذبها منها عندما تحتاج إلى مشاركة العمل. وإدارة المستودعات البعيدة تشمل معرفة كيف تضيفها في مستودعك وكيف تزيلها إن لم تعد صالحة وكيف تدير العديد من الفروع البعيدة وكيف تجعل الفروع البعيدة متعقَّبة أو غير متعقَّبة، وغير ذلك. سنتناول في هذا الفصل بعضًا من مهارات الإدارة هذه.

المستودعات البعيدة قد تكون على جهازك المحلي

من الممكن جدا أن تعمل مع مستودع «بعيد» (”remote“)، ولكنه في الحقيقة على الجهاز الذي تستخدمه نفسه. كلمة «بعيد» لا تعني بالضرورة أن المستودع في مكانٍ ما آخر على الشبكة أو الإنترنت، ولكنها تعني فقط أنه في مكان آخر. فالعمل مع مستودع بعيد مثل هذا ما زال يحتاج جميع عمليات الدفع والجذب والاستحضار المعتادة مثل أي مستودع بعيد آخر.

سرد مستودعاتك البعيدة

لسرد المستودعات البعيدة التي هيأتها، استخدم أمر البعيد `git remote`. فإنه يُظهر لك الاسم المختصر لكل بعيد في مستودعك. وإذا استنسخت مستودعا، فإنك على الأقل سترى `origin` («الأصل»)، وهو الاسم الذي يعطيه جت للمستودع الذي استنسخت منه:

CONSOLE

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

```
origin
```

يمكنك أيضا استخدام الخيار `-v` («إطناب»)، والذي يُظهر لك الروابط التي خزنها جت للأسماء المختصرة للمستودعات البعيدة ليستعملها لقراءة ذلك المستودع البعيد ولتحريره:

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

إذا كان لديك أكثر من مستودع بعيد واحد، فإن هذا الأمر سيسردهم جميعًا. مثلا، قد يبدو مستودع مرتبط بعدد من المستودعات البعيدة للعمل مع رهط من المتعاونين هكذا:

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

هذا يعني أننا نستطيع جذب المساهمات من أيٍّ من هؤلاء المستخدمين بسهولة. وقد يكون لدينا إذن الدفع إلى واحد أو أكثر منهم، ولكن لا نستطيع معرفة هذا من هنا.

لاحظ أن هذه المستودعات البعيدة تستخدم موافيق (بروتوكولات) متنوعة؛ سنتحدث عن هذا في تثبيت جت على خادوم.

إضافة مستودعات بعيدة

ذكرنا أن أمر الاستنساخ `git clone` يضيف لك من تلقاء نفسه الأصل البعيد `origin`، ورأيت مثالين على ذلك. إليك الآن معرفة كيف تضيف مستودعا بعيدا بأمر صريح. لإضافة مستودع جت بعيد جديد وإعطائه اسمًا مختصرًا للإشارة إليه به بسهولة فيما بعد، نفذ `git remote add <shortname> <url>`، أي الاسم المختصر ثم الرابط:

CONSOLE

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

يمكنك الآن استخدام الاسم `pb` في سطر الأوامر، بدلا من الرابط بكامله. مثلا إذا أردت استحضار جميع المعلومات التي لدي پول ولكن ليست لديك في مستودعك بعد، يمكنك استخدام أمر الاستحضار معه، أي `git fetch pb`:

CONSOLE

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

الآن صار فرع `master` من مستودع پول متاحا محليًا بالاسم `pb/master`؛ يمكنك دمجه في أحد فروعك، أو سحب إيداعه الأخير إلى فرع محلي إذا أردت تفقّده. سنتناول ما هي الفروع وكيف نستعملها بتفصيل عميق

في التفريع في جت.

الاستحضار والجذب من مستودعاتك البعيدة

كما رأيت للتو، للحصول على بيانات من مستودعاتك البعيدة، يمكنك تنفيذ:

CONSOLE

```
$ git fetch <البعيد>
```

يذهب هذا الأمر إلى المستودع البعيد ويجذب منه كل البيانات التي لديه وليست لديك بعد. بعد أن تفعل هذا، ستجد لديك إشارات لجميع الفروع التي لدى هذا البعيد، فيمكنك دمجها أو فحصها في أي وقت.

إذا استنسخت مستودعًا، فإن أمر الاستنساخ يضيف آليًّا هذا المستودع البعيد بالاسم "origin". لذا فإن `git fetch origin` يستحضر أي عمل قد دُفع إلى هذا المستودع بعدما استنسخته (أو استحضرت منه) آخر مرة. مهمٌ ملاحظة أن أمر الاستحضار ينزّل فقط البيانات إلى مستودعك المحلي، ولكنه لا يدمجها مع عملك ولا يعدّل أي شيء تعمل عليه. عليك دمجها يدويًا مع عملك عندما تكون مستعدًا لذلك.

إذا كان الفرع الحالي مضبوطًا ليتعقب فرعًا بعيدًا (انظر الفصل التالي و الباب الثالث: التفريع في جت للمزيد من المعلومات)، فيمكنك استخدام أمر الجذب `git pull` ليستحضر آليًّا هذا الفرع البعيد ثم يدمجه في الفرع الحالي. هذا قد يكون أسهل أو أريح لك. وأمر الاستنساخ بطبيعة الحال يضبط لك آليًّا الفرع المبدئي المحلي ليتعقب الفرع المبدئي البعيد (`master` أو `main` أو أيًّا كان اسمه) في المستودع الذي استنسخت منه. يستحضر أمر الجذب البيانات من المستودع الذي استنسخت منه في الأصل عادةً ثم يحاول دمجها آليًّا في الفرع الذي تعمل فيه حاليا.

ابتداءً من النسخة 2.27 من جت، سيحذرك أمر الجذب `git pull` إن لم يكن متغير `pull.rebase` مضبوطًا. وسيبقى يحذرك حتى تعيّن له قيمة.

إن أردت سلوك جت المبدئي (التسريع متى أمكن، وإلا فإنشاء إيداع دمج)، فنفذ:

```
git config --global pull.rebase "false"
```

وإذا أردت إعادة التأسيس عند الجذب، فنفذ:

```
git config --global pull.rebase "true"
```

الدفع إلى مستودعاتك البعيدة

عندما يكون مشروعك في مرحلة تود مشاركتها، عليك دفعه إلى المنبع. الأمر الذي يفعل هذا يسير: `git push <remote> <branch>`، أي اسم المستودع البعيد ثم الفرع: فإذا أردت دفع فرع `master` الخاص بك إلى المستودع الأصل `origin` (غالبا يضبط لك الاستنساخ هذين الاسمين آليًّا)، فتنفيذ هذا الأمر يدفع أي إيداعات صنعتها إلى الخادوم:

CONSOLE

```
$ git push origin master
```

يعمل هذا الأمر فقط إذا استنسخت من مستودع لديك إذن تحريره، ولم يدفع أي شخص آخر إليه في هذه الأثناء. أما إذا استنسخت أنت وشخص آخر في وقت واحد، ودفع هو إلى المنبع، ثم أردت أنت الدفع إلى المنبع، فإن دفعك سيُرفض عن حق. وسيتوجب عليك عندئذٍ استحضار عمله أولا وضمه إلى عملك قبل أن يُسمح لك بالدفع. انظر التفريع في جت لمعلومات أشد تفصيلا عن الدفع إلى مستودعات بعيدة.

فحص مستودع بعيد

إذا أردت رؤية معلومات أكثر عن بعيد معين، فاستخدم الأمر `git remote show <البعيد>`. إذا

نفذت هذا الأمر مع اسم مختصر معين، مثل `origin` ، فسترى شيئا مثل هذا:

CONSOLE

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

إنه يسرد لك رابط المستودع البعيد إضافةً إلى معلومات تعقب الفروع. وللإفادة يخبرك كذلك بأنك إذا كنت في فرع `master` واستخدمت أمر الجذب `git pull` فإنه تلقائيا سيدمج فرع `master` البعيد في الفرع المحلي بعد استحضاره. ويسرد لك أيضا جميع الإشارت البعيدة التي جذبها إليك.

هذا مثال يسير غالبا ستقابله. ولكن عندما تستخدم جت بكثرة، فسيعطيك `git remote show` معلومات أكثر كثيرًا:

CONSOLE

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

```
  Local branches configured for 'git pull':
    dev-branch merges with remote dev-branch
    master     merges with remote master
  Local refs configured for 'git push':
    dev-branch                     pushes to dev-branch                     (up to
date)
    markdown-strip                 pushes to markdown-strip                 (up to
date)
    master                         pushes to master                         (up to
date)
```

يُظهر لك هذا الأمر ما الفرع الذي يجذب جت إليه تلقائيا عندما تنفذ `git push` وأنت في فرع معين. ويُظهر لك كذلك ما فروع المستودع البعيد التي ليست لديك بعد، وما الفروع البعيدة التي لديك وحُذفت من البعيد، وما الفروع المحلية التي يمكن الدمج تلقائيا في فروعها المتعقِبة للبعيد عندما تنفذ `git pull`.

تغيير اسم بعيد أو حذفه

يمكنك استعمال `git remote rename` لتغيير الاسم المختصر لمستودع بعيد. مثلا، لتغيير اسم `pb` إلى `paul`، نفّذ:

CONSOLE

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

مهم ملاحظة أن هذا يغيّر أسماء فروعك المتعقِبة للبعيد أيضا. فالذي كان يسمى `pb/master` صار `paul/master`.

وإذا أردت حذف بعيد لسبب ما — نقلت المستودع، أو لم تعد تستخدم خادوم مرآة معين، أو ربما مساهم لم يعد يساهم — يمكنك استخدام إما `git remote remove` وإما `git remote rm`:

CONSOLE

```
$ git remote remove paul
```

```
$ git remote
origin
```

وما إن تحذف الإشارة إلى بعيدٍ هكذا، فإن جميع الفروع المتعقِبة له وجميع إعدادات التهيئة المرتبطة به ستُحذف كذلك.

الوسوم

مثل معظم أنظمة إدارة النسخ، يستطيع جت وسم المراحل المهمة في تاريخ المشروع. يستعمل الناس هذه الآلية في الغالب لتمييز الإصدارات (`v1.0` و `v2.0` وهكذا). سنتعلم في هذا الفصل كيف نسرد الوسوم الموجودة وكيف ننشئ وسومًا ونحذفها وما أنواع الوسوم المختلفة.

سرد وسومك

سرد الوسوم الموجودة في مستودع جت سهل جدا؛ فقط اكتب `git tag` (اختياريًا مع `-l` أو `--list`):

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

يسرد لك هذا الأمر الوسوم بترتيب أبجدي؛ أي أن ترتيب عرضها ليس له أهمية حقيقية.

يمكنك أيضا البحث عن الوسوم التي تطابق نمطًا معينًا. يحتوي مستودع مصدر جت مثلا على أكثر من خمسمئة وسم. فإذا كنت مهتمًا برؤية سلسلة 1.8.5 فقط، فنفذ هذا:

CONSOLE
```
$ git tag -l "v1.8.5*"
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
```

```
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

سرد الوسوم بأنماط يحتاج الخيار `-l` أو `--list`

إذا لم تُرِد إلا قائمة الوسوم بكاملها، فتنفيذ `git tag` يفترض أنك تريد سرد الوسوم فيعطيك إياه؛ استخدام `-l` أو `--list` في هذه الحالة اختياري.

لكن إذا أعطيته نمطًا لمطابقة وسوم عديدة، فيجب عليك استخدام خيار السرد: `-l` أو `--list`.

إنشاء وسوم

يدعم جت نوعين من الوسوم: خفيفة، ومعنونة.

الوسم الخفيف كأنه فرع لا يتغيّر: مجرد إشارة إلى إيداع معين.

لكن على النقيض، الوسوم المعنونة هي كائنات كاملة في قاعدة بيانات جت؛ يحسب جت بصمتها، ويسجل معها اسم الواسم، وبريده، وتاريخ الوسم، ورسالته، ويمكن توقيعها وتوثيقها باستعمال GNU Privacy Guard (GPG). من الأفضل في العموم إنشاء وسوم معنونة حتى تتمتع بكل هذه المعلومات، لكن إذا أردت وسمًا مؤقتًا أو لسبب ما لم تشأ الاحتفاظ بكل هذه المعلومات، فلا تزال الوسوم الخفيفة متاحة.

الوسوم المعنونة

إنشاء الوسوم المعنونة في جت يسير. الطريقة الأسهل هي إضافة `-a` إلى أمر الوسم:

CONSOLE

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

```
$ git tag
v0.1
v1.3
v1.4
```

والخيار `-m` يعيّن رسالة الوسم، التي تخزّن معه. وإذا لم تعيّن رسالة للوسم المعنون، فإن جت سيفتح لك محررك حتى تكتبها فيه.

يمكنك أيضا رؤية تاريخ الوسم مع الإيداع الموسوم بأمر الإظهار `git show`:

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

my version 1.4

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

    Change version number
```

يُظهر لك هذا معلومات الواسم وتاريخ وسم الإيداع ورسالة الوسم، ثم معلومات الإيداع نفسه.

الوسوم الخفيفة

طريقة أخرى لوسم الإيداعات هي باستعمال الوسوم الخفيفة. هذا يعني تخزين بصمة الإيداع في ملف؛ لا معلومات أخرى تُخزّن. لإنشاء وسم خفيف، لا تعطِ أمر الوسم أيًّا من الخيارات `-a` أو `-s` أو `-m`؛ أعطه فقط اسم الوسم:

CONSOLE
```
$ git tag v1.4-lw
$ git tag
v0.1
```

```
v1.3
v1.4
v1.4-lw
v1.5
```

تنفيذ `git show` على مثل هذا الوسم لن يعطيك معلومات الوسم الإضافية، بل يُظهر فقط معلومات الإيداع:

CONSOLE

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

    Change version number
```

الوسم لاحقًا

يمكنك أيضا وسم إيداعات قديمة تخطيتها. لنفترض مثلا أن تاريخ إيداعاتك يبدو هكذا:

CONSOLE

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

الآن لنفترض أنك نسيت وسم المشروع عند `v1.2`، والتي كانت عند إيداع "Update rakefile". يمكنك فعل هذا بعدما حدث ما حدث. لوسم ذلك الإيداع، اكتب في نهاية أمر الوسم بصمةَ الإيداع (أو جزءًا من

أولها):

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

والآن ستجد أنك قد وسمت الإيداع:

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

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

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

    Update rakefile
...
```

مشاركة الوسوم

لا ينقل أمر الدفع، بطبيعته، الوسومَ إلى المستودعات البعيدة. فعليك دفعها بأمر صريح بعد إنشائها. تشبه هذه العملية كثيرًا عملية مشاركة الفروع البعيدة — نفّذ `git push origin <اسم-الوسم>`.

```
CONSOLE
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
```

```
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.5 -> v1.5
```

وإذا كانت لديك العديد من الوسوم التي تريد دفعها جملةً واحدة، فيمكنك إضافة خيار الوسوم `--tags` إلى أمر الدفع `git push`، لينقل إلى المستودع البعيد جميع وسومك التي ليست هناك بالفعل.

CONSOLE

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

والآن، عندما يستنسخ أحدهم مستودعك أو يجذب منه، فسيحصل على جميع وسومك أيضا.

يدفع أمر الدفع كلا النوعين من الوسوم

سيدفع `git push <البعيد> --tags` الوسوم الخفيفة والوسوم المعنونة. لا يوجد حاليا خيار لدفع الوسوم الخفيفة فقط، لكن الأمر `git push <البعيد>` `--follow-tags` سيدفع الوسوم المعنونة فقط إلى الخادوم البعيد.

حذف الوسوم

لحذف وسم من مستودعك المحلي، نفذ `git tag -d <اسم‌الوسم>`. مثلا، يمكننا حذف الوسم الخفيف الذي أنشأناه سابقًا كالتالي:

CONSOLE

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

لاحظ أن هذا لا يحذف الوسم من أي مستودع بعيد. توجد طريقتان شائعتان لحذف وسم ما من مستودع بعيد:

الطريقة الأولى هي `git push <البعيد> :refs/tags/<اسم‌الوسم>`:

CONSOLE

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

لاستيعاب ما تفعله هذه الطريقة يمكن ترى أنها تدفع القيمة الفارغة التي قبل النقطتين الرأسيتين إلى اسم الوسم على المستودع البعيد، فعمليًّا تحذفه.

الطريقة الأخرى (والبديهية أكثر) لحذف وسم من مستودع بعيد، هي:

CONSOLE

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

سحب الوسوم

إذا أردت رؤية نُسخ الملفات التي يشير إليها وسمٌ ما، يمكنك سحب هذا الوسم بأمر `git checkout`، مع إن هذا يضع مستودعك في حالة "detached HEAD"، والتي لها بعض الآثار الجانبية السيئة:

CONSOLE

```
$ git checkout v2.0.0
Note: switching to 'v2.0.0'.

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

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

  git switch -c <new-branch-name>
```

```
Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

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

$ git checkout v2.0-beta-0.1
Previous HEAD position was 99ada87... Merge pull request #89 from schacon/appendix-final
HEAD is now at df3f601... Add atlas.json and cover image
```

في حالة "detached HEAD"، إذا أجريت تعديلات وصنعت إيداعًا، فإن الوسم سيبقى كما هو، وإيداعك الجديد لن ينتمي إلى أي فرع ولن يمكن الوصول إليه أبدا، إلا ببصمته. لذا، فإن احتجت إجراء تعديلات — مثلا لإصلاح علة في نسخة قديمة — فغالبا ستحتاج إلى إنشاء فرع:

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

إذا فعلت هذا ثم صنعت إيداعًا، فإن فرع `version2` سيكون مختلفًا عن وسم `v2.0.0` لأنه سيكون متقدمًا عنه بتعديلاتك، لذا كن حذرًا.

كُنيات جت

قبل أن نتقدم إلى الباب التالي، نود أن نعرّفك ميزة في جت ستجعل استعمالك أسهل وأريح وأكثر ألفة: الكُنيات. لن نستعملها في أي موضع آخر في هذا الكتاب للوضوح، لكنك إذا كنت تنوي استعمال جت باستمرار، فيجب أن تعرف الكُنيات.

لا يُخّمن جت الأمر الذي تريده إذا كتبت جزءًا منه. فإذا لم تشأ كتابة كل أمر بكامله، فيمكنك ضبط كُنية لكل أمر تريده بسهولة بأمر التهيئة `git config`. هذه أمثلة ربما تحب إعدادها:

CONSOLE

```
$ 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 أن يمكنك كتابة `git ci` مثلا بدلا من أن تكتب `git commit`. وبالاستمرار مع جت، ستجد أوامر أخرى تستعملها كثيرا؛ لا تتردد في إنشاء كُنيات لها.

هذه الطريقة تصلح كذلك لإنشاء الأوامر التي تظن أنها يجب أن توجد. مثلا، لتصحيح صعوبة الاستخدام التي واجهتها عند إلغاء تأهيل ملف، يمكنك إضافة كُنية خاصة بك لإلغاء التأهيل `unstage` إلى جت:

CONSOLE

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

يجعل هذا الأمرين التاليين متكافئين:

CONSOLE

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

هذا أسهل وأوضح. وكذلك من الشائع إضافة أمر `last` «الأخير»، مثل هذا:

CONSOLE

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

فيمكنك عندئذٍ رؤية إيداعك الأخير بسهولة:

CONSOLE

```
$ 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>
```

وكما يمكنك أن تخمن، إنما يترجم جت الأمر الجديد إلى ما جعلته كُنيةً له. ولكنك أحيانا قد تريد تنفيذ أمر خارجي، بدلا من أمر فرعي في جت. في هذه الحالة تبدأ الأمر بعلامة تعجب: `!`. يفيد هذا عندما تكتب أدواتك الخاصة التي تعمل مع مستودع جت. نوضّح ذلك بعمل الكُنية `git visual` لتشغيل `gitk`:

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

الخلاصة

الآن تستطيع فعل جميع عمليات جت المحلية الأساسية: إنشاء مستودع أو استنساخه، وعمل تعديلات، وتأهيلها وإيداعها، وعرض تاريخ جميع التعديلات التي مر بها المستودع. التالي: سنشرح ميزة جت القاتلة للمنافسة: نموذج التفريع.

الباب الثالث: التفريع في جت

معظم أنظمة إدارة النسخ بها نوع ما من دعم التفريع. التفريع يعني أنك تنشق عن مسار التطوير الرئيسي، وتستمر بالعمل من غير أن تؤثر في ذلك المسار الرئيسي. هذه عملية مكلفة نوعا ما في أدوات إدارة نسخ كثيرة، وغالبا تحتاج منك إلى إنشاء نسخة جديدة من مجلد مشروعك، الذي قد يحتاج وقتًا طويلًا في المستودعات الكبيرة.

يسمي البعض نموذج التفريع في جت بأنه «ميزته القاتلة للمنافسة»، وهي بكل تأكيد تميّزه في مجتمع إدارة النسخ. لماذا هي مميزة هكذا؟ لأن طريقة التفريع في جت خفيفة خفة مستحيلة، فتجعل إنشاء فرع جديد عملية شِبه آنيّة، والانتقال بين الفروع ذهابًا وإيابًا له تلك السرعة نفسها تقريبا. وخلافا للكثير من الأنظمة الأخرى، يشجع جت على أساليب سير العمل التي تعتمد على التفريع والدمج كثيرا، عدة مرات في اليوم حتى. وفهم هذه الميزة وإتقانها يعطيانك أداة قوية وفريدة، وقد يغيّران تمامًا الطريقة التي تطوّر بها.

الفروع بإيجاز

لنفهم حقًا طريقة التفريع في جت، علينا أن نتراجع خطوة إلى الخلف ونتدبر طريقته في تخزين البيانات.

كما قد تتذكر من ما هو جت؟، لا يخزن جت بياناته في صورة فروقات، بل في صورة لقطات.

وعندما تُودع، يخزن جت كائنَ إيداع فيه إشارة إلى لقطة المحتوى الذي أهّلته. وفيه كذلك اسم المؤلف وعنوان بريده ورسالة الإيداع والإشارات إلى الإيداعات السابقة له مباشرةً (الإيداعات الآباء): لا أب للإيداع المبدئي، وأب واحد للإيداعات العادية، وأبوين أو أكثر لإيداعات الدمج، وهي الإيداعات الناتجة من دمج فرعين أو أكثر.

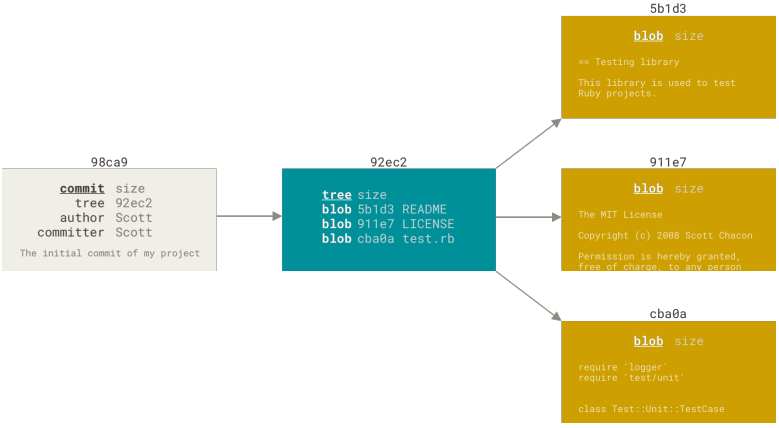
حتى نستطيع تصور هذا، لنفترض أن لديك مجلدًا به ثلاثة ملفات، وأنك أهّلتها جميعها ثم أودعتها. يحسب تأهيل الملفات بصمة كل ملف (بصمة SHA-1 التي ذكرناها في ما هو جت؟)، ويخزن نسخة الملف هذه في

المستودع (وهي ما يسميها جت «كتلة رقمية» ("blob")، ونسميها «كتلة» اختصارا)، ويضيف تلك البصمة إلى منطقة التأهيل:

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

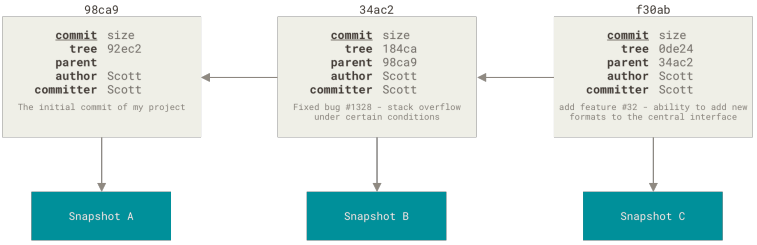
عندما تودع بأمر `git commit`، فإن جت يحسب أيضا بصمات كل مجلد ومجلد فرعي (في هذه الحالة، مجلد جذر المشروع فقط)، ويخزنها في صورة كائنات أشجار في المستودع. ثم ينشئ جت كائنَ إيداعٍ فيه بيانات وصفية وإشارة إلى شجرة جذر المشروع، حتى يستطيع إعادة إنشاء تلك اللقطة عند الحاجة.

صار في مستودعك الآن خمسة كائنات: ثلاث **كتل** (كلٌّ منها يمثل محتويات ملف من الثلاثة)، و**شجرة** واحدة (تسرد محتويات المجلد وما الكتل التي تشير إليها أسماء الملفات)، و**إيداع** واحد (فيه إشارة إلى شجرة الجذر تلك وكذلك البيانات الوصفية للإيداع).



شكل ٩. إيداعٌ وشجرته

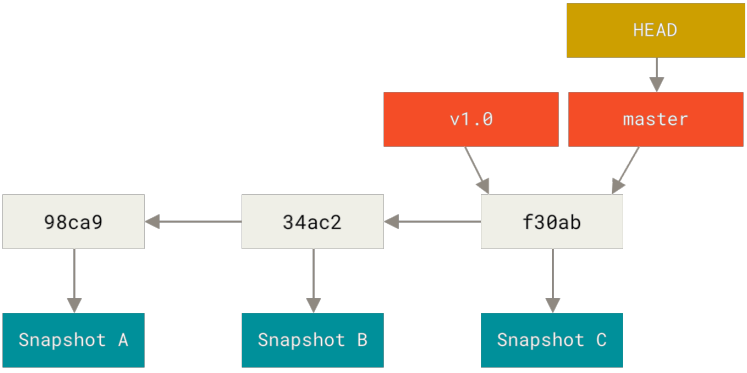
إذا أجريت تعديلات وأودعتها، فإن إيداعك التالي سيخزن إشارةً إلى الإيداع السابق له مباشرةً.



شكل ١٠. إيداعات وآباؤها

فإنما الفرع في جت هو إشارة متحركة تشير إلى أحد هذه الإيداعات. والفرع المبدئي في جت يُسمى `master`. فعندما تشرع في صنع الإيداعات، فإن جت يعطيك فرعا رئيسا يسمى `master` ويشير إلى آخر إيداع صنعته. ويتقدم فرع `master` تلقائيًا مع كل إيداعٍ تودِعه.

فرع `master` في جت ليس مميزًا. فهو تمامًا مثل أي فرع آخر. والسبب الوحيد لوجوده في أغلب المستودعات أن أمر `git init` ينشئه بهذا الاسم المبدئي وأكثر الناس لا يبالون بتغييره.



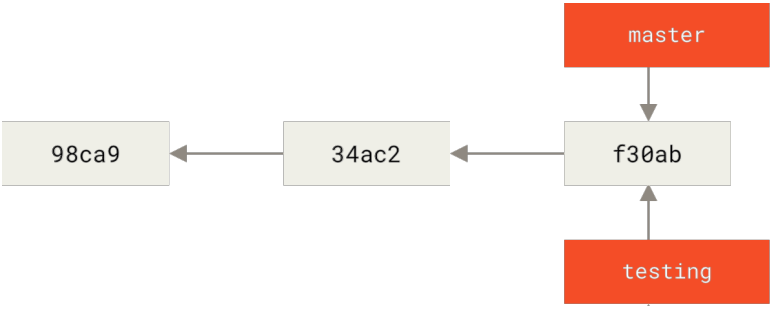
شكل ١١. فرع وتاريخ إيداعاته

إنشاء فرع جديد

ماذا يحدث عندما تنشئ فرعًا جديدًا؟ الإجابة: ينشئ جت إشارة جديدة لك لتحركها كما تشاء. لنقُل إنك أردت إنشاء فرع جديد اسمه `testing`. تفعل هذا بأمر التفريع، `git branch`:

```
CONSOLE
$ git branch testing
```

ينشئ هذا إشارةً إلى الإيداع الذي تقف عنده الآن.



شكل ١٢. فرعان يشيران إلى سلسلة الإيداعات نفسها

كيف يعرف جت في أيّ فرع أنت الآن؟ إنه يحتفظ بإشارة مخصوصة تسمى «إشارة الرأس» (`HEAD`). لاحظ أن هذه مختلفة كثيرًا عن مفهوم `HEAD` في الأنظمة الأخرى مثل Subversion وCVS. في جت، هذه إشارة إلى الفرع المحلي الذي تقف فيه الآن. في حالتنا هذه، ما زلتَ واقفًا في فرع `master`. فما على أمر `git branch` إلا إنشاء فرع جديد؛ ليس عليه الانتقال إليه.

شكل ١٣. إشارة الرأس `HEAD` تشير إلى فرع

يمكنك رؤية هذا بسهولة بأمر السجل، والذي يُظهر لك ما تشير إليه إشارات الفروع، وذلك بالخيار `--decorate`.

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

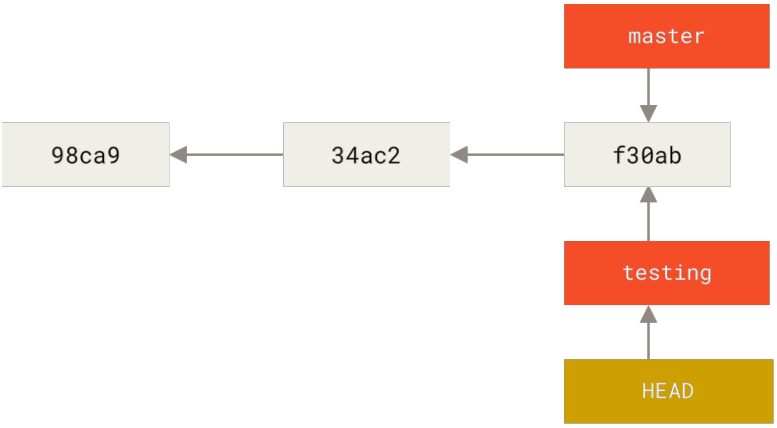
يمكنك رؤية فرعَي `master` و `testing` عند إيداع `f30ab`.

الانتقال بين الفروع

للانتقال إلى فرع موجود، استخدم أمر السحب `git checkout`. هيا بنا نتنقل إلى فرعنا الجديد `testing`:

```
CONSOLE
$ git checkout testing
```

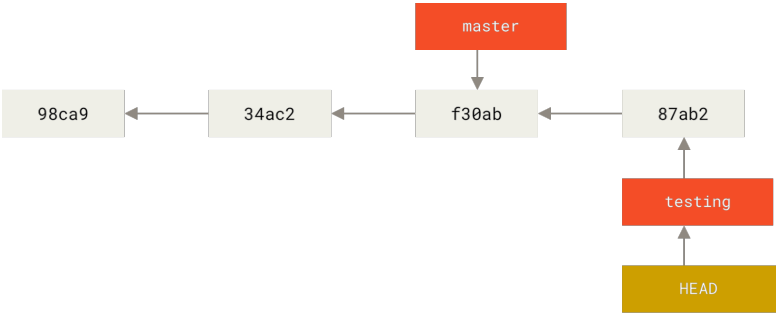
يحرك هذا الأمر إشارة الرأس لتشير إلى فرع `testing`.



شكل ١٤. إشارة الرأس تشير إلى الفرع الحالي

ما دلالة هذا؟ لنصنع إيداعًا آخر إذًا.

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



شكل ١٥. فرع الرأس يتقدم عند صنع إيداع

هذا يدعو للتفكر، لأن الآن فرع `testing` قد تقدم، بينما قعد فرع `master` في مكانه مشيرًا إلى الإيداع القديم نفسه عندما انتقلنا إلى الفرع الجديد بأمر السحب. لنعد إلى فرع `master`:

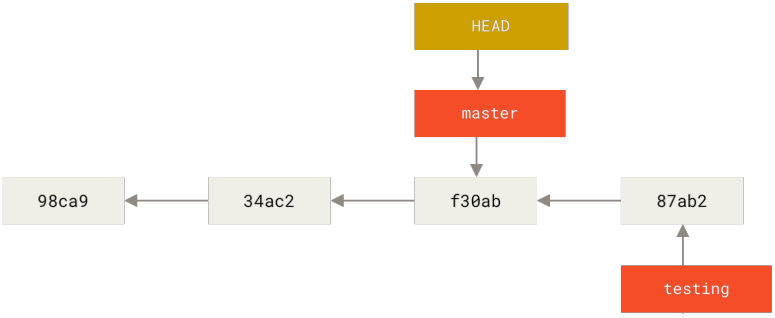
```
CONSOLE
$ git checkout master
```

لا يُظهر أمر السجل جميع الفروع طوال الوقت

إذا نفذت `git log` الآن، فستتساءل أين ذهب فرع `testing` الذي أنشأته، لأنه لن يظهر في ناتجه.

لم يتبخر الفرع، ولكن جت لا يعلم أنك مهتمٌ به الآن، ولا يُظهر لك جت إلا ما يظن أنك مهتم به. بلفظ آخر، لا يُظهر لك أمر السجل بطبيعته إلا تاريخ الفرع الذي تقف فيه حاليًا.

لإظهار تاريخ فرع آخر، عليك طلب ذلك صراحةً، مثل `git log testing`. ولإظهار جميع الفروع، اطلب ذلك من `git log` بالخيار `--all`.



شكل ١٦. تتحرك إشارة الرأس عندما تنتقل إلى فرع آخر

فَعَل هذا الأمر فعلين: أعاد إشارة الرأس لتشير إلى فرع `master`، وأرجع الملفات في مجلد العمل إلى حالها كما كانت في اللقطة التي يشير إليها `master`. هذا يعني أيضا أن التغييرات التي ستصنعها الآن ستُبنى على نسخة قديمة من المشروع. أي أنه عمليًّا يتراجع عما فعلت في فرع `testing` لكي تستطيع السير في اتجاه آخر.

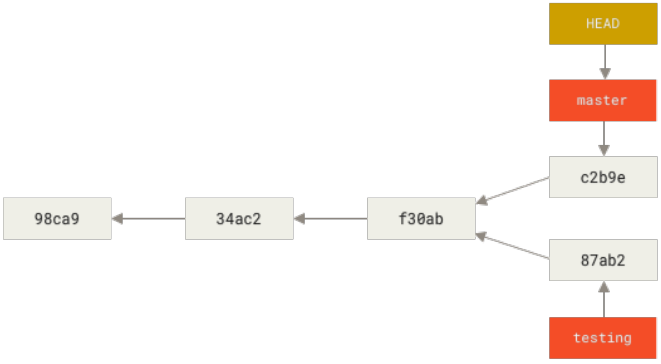
الانتقال بين الفروع يغيّر الملفات التي في مجلد عملك

مهمٌ ملاحظة أنك عندما تنتقل إلى فرع آخر في جت، فإن الملفات التي في مجلد عملك ستتغير. فإذا انتقلت إلى فرع قديم، سيعود مجلد عملك إلى ما كان عليه عند آخر إيداع في هذا الفرع. وإن لم يستطع جت تغيير الملفات تغييرا نظيفا، فلن يسمح لك بالتبديل أصلا.

لنُجري بعض التعديلات ونودع مجددًا:

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

الآن افترق تاريخ مشروعك (انظر تاريخ مفترِق). فلقد أنشأتَ فرعًا وانتقلت إليه وعملت فيه قليلا، ثم عدت إلى الفرع الرئيس وعملت فيه عملا آخر. كلا هذين التغييرين منعزلان في فرعين منفصلين: يمكنك التنقل بينهما ذهابًا وإيابًا ثم دمجهما معًا عندما تكون مستعدًا. وكل هذا فعلتَه بسَهولة بأوامر التفريع `branch` والسحب `checkout` والإيداع `commit`.



شكل ١٧. تاريخ مفترِق

يمكنك أيضا رؤية هذا بسهولة بأمر السجل، فإذا نفّذت `git log --oneline --decorate --graph --all` فسيُظهر لك تاريخ إيداعاتك ومواضع إشارات فروعك وكيف افترق تاريخك.

CONSOLE

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

ولأن الفرع في جت ليس إلا ملفًا هيّنًا فيه ٤٠ حرفًا تمثّل بصمة الإيداع الذي يشير إليه الفرع، فإن إنشاء الفروع وإزالتها عمليتان رخيصتان سريعتان. فعملية إنشاء فرع جديد تماثل في سرعتها ويسرها عملية كتابة ٤١ بايتًا إلى ملف (وهم ٤٠ محرفًا للبصمة ثم محرف نهاية السطر).

هذا اختلاف عظيم عن طريقة التفريع في معظم الأنظمة القديمة لإدارة النسخ، والتي يُنسخ فيها جميع ملفات المشروع إلى مجلد آخر. قد يحتاج هذا عدة ثوانٍ أو حتى دقائق، حسب حجم المشروع. ولكن تلك العملية في جت دائمًا عملية آنيّة. وأيضا لأننا نسجّل آباء الإيداعات عندما نودع، فإن إيجاد قاعدة مناسبة للدمج هي عملية يفعلها جت من أجلنا آليًّا، وهي سهلة جدا عموما. تشجّع هذه الميزات المطورين على إنشاء فروع واستعمالها بكثرة.

لنرَ لماذا عليك فعل هذا.

إنشاء فرع جديد والانتقال إليه في خطوة واحدة

من المعتاد أن ترغب في الانتقال إلى فرع جديد فور إنشائه — يمكنك إنشاء فرع والانتقال إليه بأمر واحد: `git checkout -b <اسم_الفرع_الجديد>`.

ابتداءً من النسخة 2.23 من جت يمكنك استعمال أمر التبديل بدلًا من أمر السحب من أجل:

- الانتقال إلى فرع موجود: `git switch testing-branch`.
- إنشاء فرع جديد والانتقال إليه: `git switch -c new-branch`. الخيار `-c` للإنشاء، ويمكنك استخدام الخيار الكامل: `--create`.
- العودة إلى الفرع المسحوب سابقًا: `git switch -`.

أسس التفريع والدمج

لننظر مثالا سهلا عن التفريع والدمج بأسلوب سير عمل قد تستعمله في الحقيقة. ستتبع هذه الخطوات:

١. تقوم ببعض الأعمال على موقع وب.

٢. تنشئ فرعا لـ«قصة المستخدم» الجديدة التي تعمل عليها.

٣. تقوم ببعض الأعمال في هذا الفرع.

وبينما أنت هنا، تأتيك مكالمة بأن علة أخرى خطيرة تحتاج منك إصلاحا عاجلا. فستفعل الآتي:

١. تنتقل إلى فرعك الإنتاجي ("production").

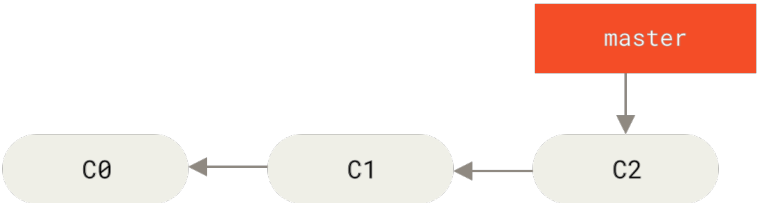
٢. تنشئ فرعا لإضافة الإصلاح العاجل.

٣. وبعد اختباره، تدمج فرع الإصلاح العاجل، وتدفعه إلى فرع الإنتاج.

٤. تعود إلى قصة المستخدم الأصلية وتكمل عملك عليها.

أسس التفريع

أولا، لنقُل أنك تعمل على مشروعك، ولديك بضعة إيداعات بالفعل في فرع `master`.



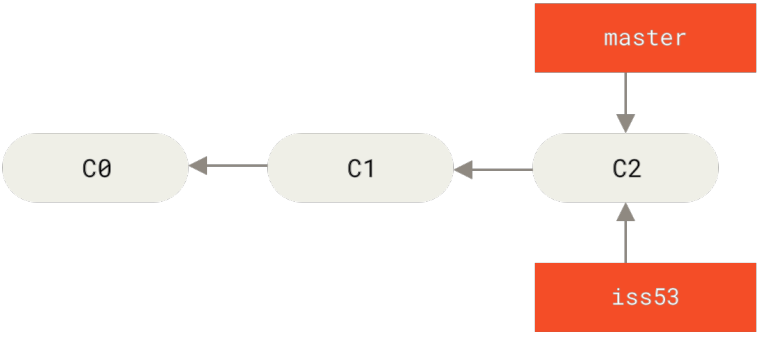
شكل ١٨. تاريخ إيداعات بسيط

ثم قررت أنك ستعمل على المسألة رقم ٥٣ في نظام متابعة المسائل الذي تستخدمه شركتك. فتنفّذ أمر `git checkout` مع الخيار `-b`، لإنشاء فرعٍ جديدٍ والانتقال إليه في الوقت نفسه:

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

وهذا اختصار للأمرين:

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

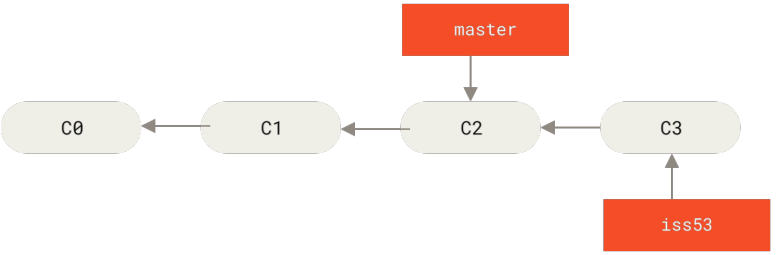


شكل ١٩. إنشاء إشارة إلى فرع جديد

يمكنك العمل على موقعك وصنع بعض الإيداعات. فعل هذا يحرك فرع `iss53` إلى الأمام، لأنه الفرع المسحوب (أي أنه الفرع الذي تشير إليه إشارة الرأس `HEAD` لديك):

CONSOLE

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



شكل ٢٠. تقدَّم فرع `iss53` بعملك عليه

ثم تأتيك مكالمة الآن بأن الموقع به علة، وعليك حلها فورا. لا تحتاج مع جت أن تنشر إصلاحك لهذه العلة مع تعديلات `iss53` التي صنعتها، ولا تحتاج أيضا إلى بذل المجهود للتراجع عن هذه التعديلات حتى تستطيع العمل على إصلاح علة الموقع. ليس عليك إلا أن تنتقل عائدا إلى فرعك الرئيس `master`.

ولكن، قبل هذا، عليك ملاحظة أن إن كان مجلد عملك أو منطقة تأهيلك فيهما تعديلات غير مودعة وتختلف عما في الفرع الذي تريد الانتقال إليه، فلن يسمح لك جت بالانتقال. من الأفضل دوما أن تجعل حالة العمل نظيفة قبل الانتقال بين الفروع. يمكن التحايل على هذا بطريقة أو بأخرى (تحديدا، التخبئة وتصحيح الإيداعات) والتي سنتطرق إليها فيما بعد في ~~Stashing and Cleaning~~. ولكن لنفترض الآن أنك أودعت كل تعديلاتك، حتى يتسنى لك الانتقال عائدا إلى فرعك الرئيس:

CONSOLE

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

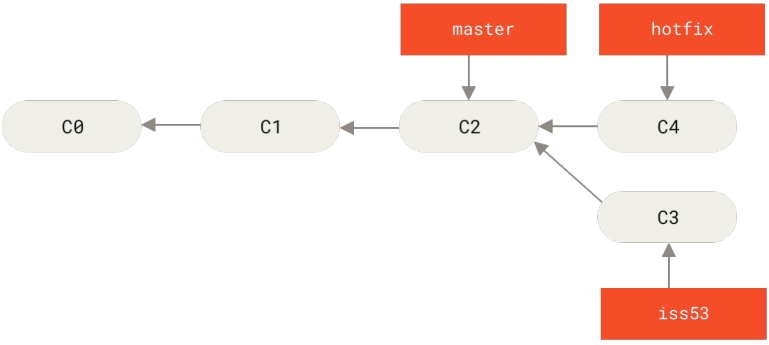
ستجد الآن أن مجلد عملك مطابق تماما لما كان عليه قبل أن تبدأ العمل على المسألة رقم ٥٣، ويمكنك الآن

التركيز على إصلاح العلة الجديدة. هذه النقطة مهمة ويجب تذكرها: عندما تنتقل من فرع إلى آخر، يعيد جت مجلد عملك إلى ما كان عليه آخر مرة أودعت فيها في هذا الفرع، فيضيف ويحذف ويعدّل المِلفات آليا، حتى يجعل نسخة العمل مشابهة تماما لما كان عليه الفرع عند آخر إيداع تم فيه.

عليك الآن العمل على الإصلاح العاجل. لننشئ فرع `hotfix` لتعمل عليه حتى تنتهي:

CONSOLE

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



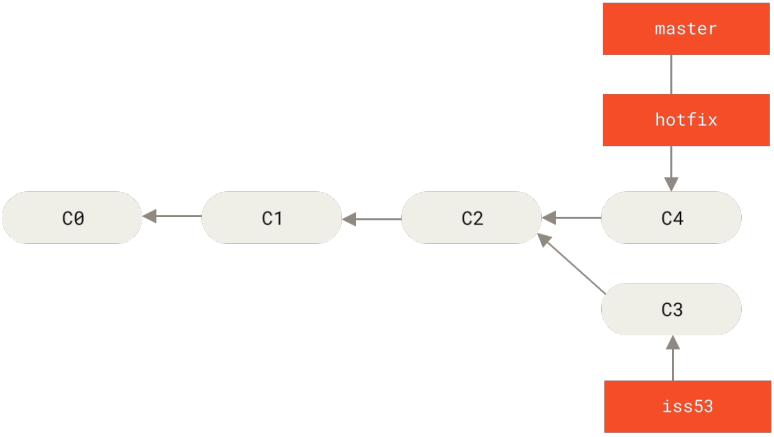
شكل ٢١. فرع الإصلاح العاجل (`hotfix`) مبني على الفرع الرئيس (`master`)

يمكنك الآن إجراء الاختبارات والتأكد من أن الإصلاح الذي صنعته هو المراد. ثم دمج فرع الإصلاح العاجل في الفرع الرئيس حتى تدفعه إلى الإنتاج. يمكنك فعل هذا بأمر الدمج `git merge` :

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

ستلاحظ عبارة "fast-forward" («تسريع») في ناتج الدمج. هذا لأن الإيداع `C4` الذي يشير إليه فرع الإصلاح العاجل كان مباشرةً أمام الإيداع `C2` الذي تقف فيه، فلم يفعل جت سوى تحريك الإشارة إلى الأمام. بلفظ آخر: عندما تريد دمج إيداع في إيداع آخر يمكن الوصول إليه بتتبع تاريخه، فإن جت لا يعقد الأمور بل يحرك الإشارة إلى الأمام، فلا أعمال مفترقة ليحاول دمجها — يسمى هذا «تسريعًا» ("fast-forward").

تعديلاتك الآن موجودة في لقطة الإيداع التي يشير إليها الفرع الرئيس، فيمكنك الآن نشرها.



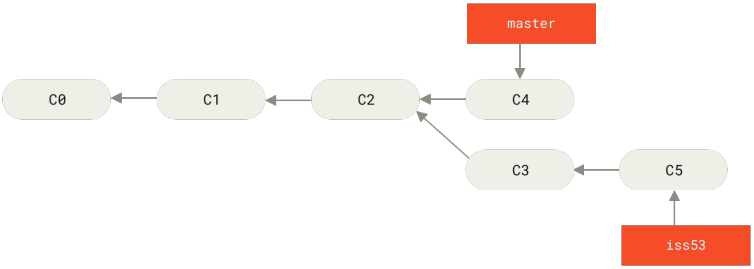
شكل ٢٢. تسريع `master` إلى `hotfix`

بعد نشر إصلاحك شديد الأهمية، تكون مستعدا للعودة إلى عملك الذي كنت تفعله قبل هذه المقاطعة. ولكن عليك أولا حذف فرع `hotfix` لأنك لم تعد تحتاج إليه؛ فالفرع الرئيس يشير إلى الشيء نفسه. يمكنك حذفه بالخيار `-d` مع أمر التفريع `git branch`:

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

يمكنك الآن العودة إلى فرع العمل الحالي الخاص بالمسألة رقم ٥٣، وإكمال العمل عليها.

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



شكل ٢٣. استكمال العمل على `iss53`

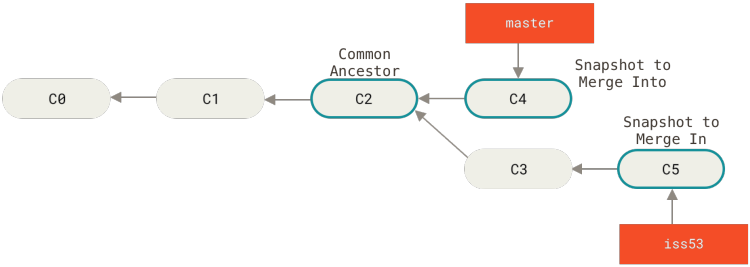
من الواجب ملاحظة أن مِلفات فرع `iss53` لا تحتوى على عملك في فرع `hotfix`. فإذا احتجت إلى جذبه إليها، ادمج فرع `master` في فرع `iss53` بالأمر `git merge master`، أو أجّله حتى تقرر جذب فرع `iss53` إلى `master` فيما بعد.

أسس الدمج

إذا رأيت أن عملك على المسألة رقم ٥٣ قد اكتمل وصار جاهزا لدمجه في الفرع الرئيس، فستدمج فرع `iss53` في فرع `master`، تماما مثلما دمجت فرع `hotfix` سابقا: ليس عليك سوى سحب الفرع الذي تريد الدمج فيه ثم تنفيذ أمر الدمج:

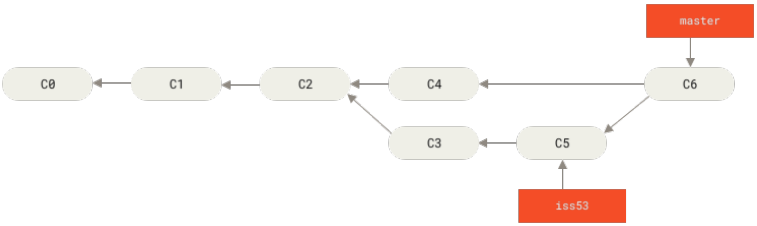
```
CONSOLE
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

يبدو هذا مختلفا قليلا عن دمج `hotfix` الذي أجريتَه سابقا. ففي حالتنا هذه قد افترق تاريخ التطوير منذ نقطة سابقة. ولأن الإيداع الأخير في الفرع الذي تقف فيه ليس سَلَفًا مباشرا (أب أو جد أو جد أعلى) للفرع الذي تريد دمجه، فإن على جت القيام ببعض العمل. في هذه الحالة، يعمل جت دمجًا ثلاثيًا نموذجيًّا، للقطتين اللتين يشيران إليهما رأسَا الفرعين، مع السلف المشترك للاثنتين.



شكل ٢٤. اللقطات الثلاثة المستعملة في دمج نموذجي معتاد

فبدلا من مجرد تحريك الإشارة إلى الأمام، ينشئ جت لقطة جديدة ناتجة عن هذا الدمج الثلاثي، وينشئ آليًّا إيداعًا جديدًا يشير إليها. يسمى هذا «إيداع دمج»، ويتميز بأن له أكثر من أب.



شكل ٢٥. إيداع دمج

الآن قد دُمِج عملك، ولم تعد في حاجة إلى فرع `iss53`. فيمكنك غلق هذه المسألة في نظام متابعة المسائل، وحذف الفرع:

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

أسس نزاعات الدمج

لا تسير هذه العملية بسلاسة في بعض الأحيان. فإذا عدّلت الجزء نفسه في الملف نفسه تعديلا مختلفا في الفرعين اللذين تنوي دمجهما، فلن يستطيع جت أن يدمجهما دمجا نظيفا. فإن كان إصلاحك للمسألة رقم ٥٣ عدّل الجزء نفسه من الملف الذي عدّلته في فرع الإصلاح العاجل، فستواجه نزاع دمج يشبه هذا:

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

لم ينشئ جت آليًّا إيداعَ دمجٍ جديدًا، بل أوقف العملية حتى تحل النزاع. فإذا أردت رؤية الملفات غير المدموجة في أي وقت بعد نزاع الدمج، نفّذ أمر الحالة `git status`:

```
CONSOLE
$ git status
On branch master
You have unmerged paths.
```

```
  (fix conflicts and run "git commit")

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

    both modified:      index.html

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

ستظهر الملفات المتنازع عليها ولم تُدمج بعد أنها غير مدموجة "unmerged". ويضيف جت علامات معيارية لحل النزاعات إلى الملفات المتنازع عليها، حتى تتمكن من تحريرها يدويا وحل تلك النزاعات. فستجد أن في ملفك جزءًا يشبه هذا:

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

تجد نسخة `HEAD` (فرع `master`، لأنه الفرع الذي سحبته قبل أمر الدمج) في النصف الأعلى من هذه الكتلة (كل ما هو فوق سطر `=======`)، ونسخة `iss53` في النصف الأسفل منها. وحتى تحل هذا النزاع، عليك اختيار أحد الجزأين أو دمج محتواهما بنفسك. فمثلا قد تحله بتغيير الكتلة كلها إلى:

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

يحمل هذا الحل شيئا من كلا الجزأين. أما الأسطر `<<<<<<<` و `=======` و `>>>>>>>` فقد أزلناها بالكامل. وبعد حل كل نزاع مثل هذا في كل ملف متنازع عليه، نفّذ أمر الإضافة `git add` على كل ملف لإعلام جت أنه قد حُلّ. فتأهيل الملف في جت يعلن نزاعه محلولا.

وإذا أردت استعمال أداة رسومية لحل هذه المشاكل، فيمكنك تنفيذ `git mergetool`، والذي يشغّل أداة دمج رسومية مناسبة ويسير معك خلال النزاعات:

CONSOLE

```
$ git mergetool

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

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

إذا أردت استعمال أداة دمج أخرى غير الأداة المبدئية (اختار جت في هذه الحالة أداة `opendiff` لأننا نفّذاه على نظام ماك)، فيمكنك رؤية قائمة بجميع الأدوات المدعومة في الأعلى بعد جملة "one of the following tools". ليس عليك سوى كتابة اسم الأداة التي تريدها.

إذا احتجت أدوات متقدمة أكثر لحل النزاعات العويصة، فسنتحدث أكثر عن الدمج في ~~Advanced Merging~~.

بعد إغلاق أداة الدمج، فسيسألك جت عما إذا كان الدمج ناجحا. إذا أجبت بنعم، فسيؤهل الملف لك لإعلان أنه قد حُل. ويمكنك عندئذٍ استعراض الحالة مجددا للتحقق أن جميع النزاعات قد حُلت:

CONSOLE

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

```
Changes to be committed:

    modified:   index.html
```

فإذا كنت راضيا عن هذا، وتأكدت من أن كل شيء كان عليه نزاع قد أُهِل، نفّذ `git commit` لاختتام إيداع الدمج. ورسالة الإيداع المبدئية تشبه هذا:

```
CONSOLE
Merge branch 'iss53'

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


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

فيها يمكنك شرح حلك للنزاع وتعليل تعديلاتك إن لم تكن واضحة، إذا ظننت أن هذا يفيد من يرى هذا الإيداع فيما بعد.

إدارة الفروع

الآن وقد أنشأت فروعا ودمجتها وحذفتها، لنرَ أدوات إدارة فروع ستفيدك عندما تشرع في استعمال الفروع

طوال الوقت.

ليس أمر التفريع `git branch` لإنشاء فروع وحذفها فحسب. فإذا نفّذته بلا مُعامِلات، سيسرد لك فروعك الحالية:

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

لاحظ مِحرف النجمة `*` أمام فرع `master`؛ إنه يعني أن هذا الفرع هو الفرع المسحوب حاليا (أي أنه الفرع الذي تشير إليه إشارة الرأس `HEAD`). يعني هذا أنك إذا أودعت الآن، فإن فرع `master` سيتقدم إلى الأمام بعملك الجديد. لرؤية آخر إيداع في كل فرع، نفّذ `git branch -v`:

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

والخياران المفيدان `--merged` («مدموج») و `--no-merged` («غير مدموج») يصفّيان هذه القائمة فلا ترى إلا الفروع التي دمجتها أو التي لم تدمجها في الفرع الذي تقف فيه. فللفروع المدموجة في الفرع الحالي، نفّذ `git branch --merged`:

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

ترى `iss53` في القائمة لأنك دمجته سابقًا. والفروع التي في هذه القائمة وليس أمامها نجمة (`*`)، يمكنك في العموم حذفها بأمان بالأمر `git branch -d`؛ لن تفقد شيئًا بحذفها لأنك بالفعل ضممت ما فيها من عمل

إلى فرع آخر.

لرؤية جميع الفروع التي بها عمل غير مدموج بعد، نفّذ `git branch --no-merged`:

CONSOLE

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

يُظهر لك هذا الأمر فرعك الآخر. ستفشل محاولة حذفه بالأمر `git branch -d` لأن به عملًا غير مدمج بعد:

CONSOLE

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

إن رغبت حقًا ويقينًا في حذف الفرع وفقد ما فيه من عمل، فأجبر جت على حذفه بالخيار `-D`، كما تخبرك الرسالة.

إذا لم تعطِ إيداعًا أو اسمَ فرعٍ إلى الخيارين `--merged` و `--no-merged`، فإنهما، على الترتيب، سيسردان ما الذي دُمج أو لم يُدمج في الفرع الحالي.

يمكنك دائمًا إعطاؤهما اسم فرع للسؤال عن حالة دمجه من غير أن تحتاج إلى الانتقال أولا إلى هذا الفرع بأمر السحب، مثلا: ما الذي لم يُدمج في فرع `master`؟

CONSOLE

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

تغيير اسم فرع

لا تغيّر اسم فرع ما زال الآخرون يستعملهُ. ولا تغيّر اسم فرع مثل `master` أو `main` أو `mainline` قبل أن تقرأ فصل تغيير اسم الفرع الرئيس.

هَبْ فرعًا لديك اسمه `bad-branch-name` وتريد جعله `corrected-branch-name` مع الإبقاء على تاريخه بالكامل. وتريد أيضا تغيير اسمه على الخادوم البعيد (جت‌هب GitHub أو جت‌لاب GitLab أو غيرهما). كيف تفعل هذا؟

غيّر اسم الفرع محليًّا بالأمر `git branch --move` :

CONSOLE

```
$ git branch --move bad-branch-name corrected-branch-name
```

هذا يغيّر `bad-branch-name` إلى `corrected-branch-name`، ولكن هذا التغيير محليٌّ فقط حتى الآن. ولجعل الآخرين يرون الفرع الصحيح في المستودع البعيد، عليك دفعه:

CONSOLE

```
$ git push --set-upstream origin corrected-branch-name
```

لنلقِ نظرةً على حالنا الآن:

CONSOLE

```
$ git branch --all
* corrected-branch-name
  main
  remotes/origin/bad-branch-name
  remotes/origin/corrected-branch-name
  remotes/origin/main
```

لاحظ أنك في فرع `corrected-branch-name` وأنه متاح في المستودع البعيد. ولكنّ الفرع ذا الاسم الخاطئ متاح كذلك هناك، ولكن يمكنك حذفه بالأمر التالي:

CONSOLE

```
$ git push origin --delete bad-branch-name
```

الآن قد حلّ اسم الفرع الصحيح محل اسم الفرع الخاطئ في كل مكان.

تغيير اسم الفرع الرئيس

تغيير اسم فرع مثل `master` أو `main` أو `mainline` أو `default` سيُعطّل التكاملات والخدمات والأدوات المساعدة وبُرمِجات البناء والإصدار التي يستخدمها مستودعك. لذا عليك التشاور مع زملائك في المشروع قبل الإقدام على هذا الأمر. وعليك كذلك أن تبحث بحثًا وافيًا في مستودعك وتحدّث أيّ إشارة إلى الاسم القديم للفرع في الكود والبُرمِجات.

غيّر اسم فرع `master` المحلي إلى `main` بالأمر:

CONSOLE

```
$ git branch --move master main
```

لم يعد لدينا أي فرع محلي `master`، لأننا غيّرنا اسمه إلى `main`.

ولجعل الآخرين يرون فرع `main` الجديد، عليك دفعه إلى المستودع البعيد. هذا يجعل الفرع الجديد متاحًا هناك:

CONSOLE

```
$ git push --set-upstream origin main
```

نجد الآن أنفسنا في الحالة التالية:

CONSOLE

```
$ git branch --all
* main
  remotes/origin/HEAD -> origin/master
  remotes/origin/main
```

```
  remotes/origin/master
```

اختفى فرعك المحلي `master`، وحلّ محله الفرع `main`. وصار `main` في المستودع البعيد. ولكن فرع `master` القديم بقي موجودًا في المستودع البعيد. فسيظل المشاركون الآخرون يتّخِذون فرع `master` أساسًا لأعمالهم، حتى تتّخذَ إجراءً آخر.

بين يديك الآن عددٌ من المهام لاجتياز تلك المرحلة الانتقالية:

- على جميع المشروعات المعتمِدة على هذا المشروع تحديث كودها و/أو إعداداتها.
- عليك تحديث أي مِلفات إعدادات خاصة بالاختبارات.
- عليك مواءمة بُرمِجات البناء والإصدار.
- عليك مواءمة إعدادات خادوم مستودعك، مثل الفرع المبدئي وقواعد الدمج والأمور الأخرى التي تعتمد على أسماء الفروع.
- عليك تحديث الإشارات إلى الفرع القديم في التوثيق.
- عليك إغلاق أو دمج كل طلبات الجذب الموجهة إلى الفرع القديم.

بعد فعل جميع هذه المهام، والتيقن أن فرع `main` يقوم بعمله تمامًا مثل فرع `master`، يمكنك حذف فرع `master`:

CONSOLE
```
$ git push origin --delete master
```

أساليب العمل التفريعية

بما أنك الآن تعلم أسس التفريع والدمج، ماذا يمكنك أو يجدر بك فعله بهما؟ سنتناول في هذا الفصل بعض أشهر أساليب العمل التي يجعلها هذا التفريع الخفيف ممكنة، لكي تقرر إذا ما كنت تود أن تجعلها جزءًا من

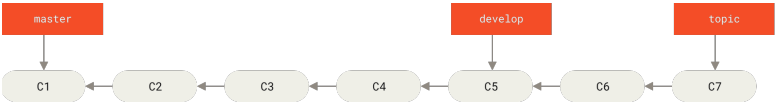
دورة التطوير التي تتبعها.

الفروع طويلة العمر

يستعمل جت دمجا ثلاثيا غير معقد، فيسهّل الدمج بين فرعين مرات عديدة عبر مدة زمنية طويلة. يتيح لك هذا وجود عدد من الفروع المفتوحة دائما لتستعملها لمراحل مختلفة من دورة التطوير، لأنك تستطيع أن تدمج باستمرار فيما بينها.

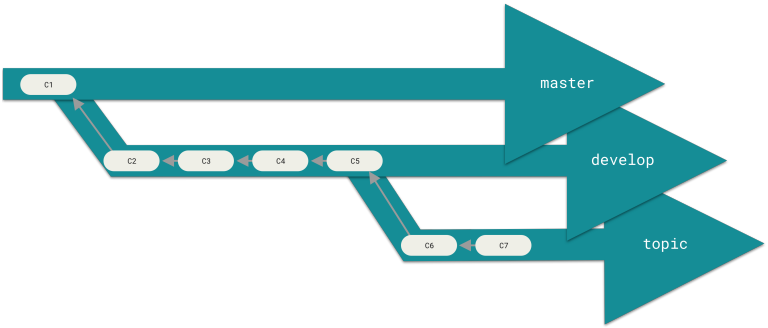
الكثيرون من المطورين مستخدمي جت يعتمدون هذا النهج في أسلوب سير العمل، فيخصصون مثلا الفرع الرئيس للمصدر المستقر تماما وحسب، أو للذي أُصدر فعلا، أو للذي سيُصدر. ويكون لديهم فرعا موازيا اسمه `develop` أو `next` مثلا، ليعملوا منه أو ليستعملوه لاختبار الاستقرار، فليس بالضرورة أن يكون مستقرا دوما، ولكن عند استقراره، يمكن دمجه في الفرع الرئيس. ويستعملون هذا الفرع ليجذبوا فيه فروع المواضيع (الفروع قصيرة العمر، مثل فرع `iss53` المذكور سابقا) عندما تكون جاهزة، لضمان اجتيازها جميع الاختبارات وأنها لا تُحدِث عللًا.

نحن فعليا نتحدث عن إشارات ترتقي في سلّم ايداعاتك. فالفروع المستقرة في أسفله، أما طليعة التطوير ففي أعلاه.



شكل ٢٦. منظور خطي لتفريع الاستقرار المتزايد

لعل الأسهل تصور أنها صومعات عمل منعزلة، فتتخرج دفعة من الإيداعات إلى صومعة أخرى أكثر استقرارا عندما تجتاز جميع الاختبارات.



شكل ٢٧. منظور «صومعي» لتفريع الاستقرار المتزايد

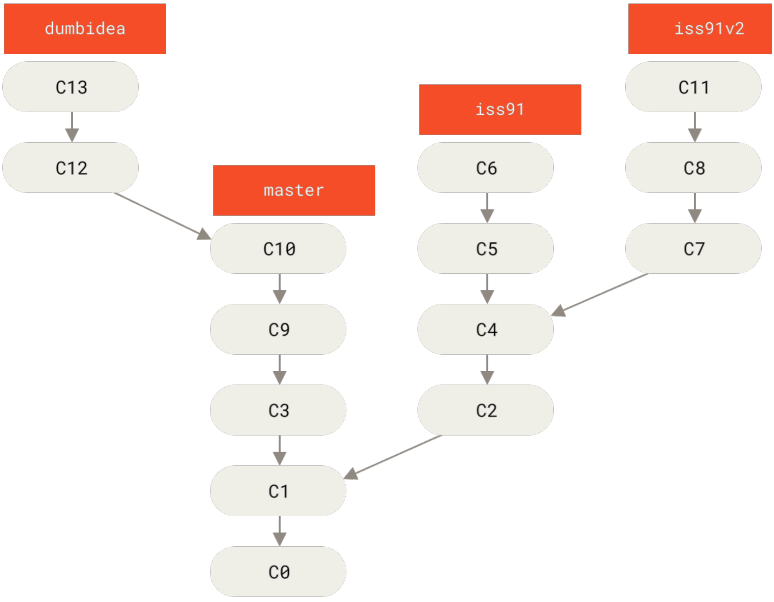
يمكنك فعل هذا بعدة مستويات من الاستقرار. فلدى بعض المشروعات الكبيرة فرع `proposed` أو `pu` («تحديثات مقترحة») ويدمجوا فيه فروعا قد لا تكون جاهزة لأن تكون في فرع `next` أو `master`. فالأمر أن فروعك في مستويات مختلفة من الاستقرار، فعندما يصل أحدها إلى مستوى استقرار أعلى، فإنه يُدمج في الفرع الأعلى. ونكرر: ليس ضروريا استعمال عدد من الفروع طويلة العمر، ولكنه كثيرا ما يفيد، خصوصا عندما تتعامل مع مشروعات معقدة أو كبيرة جدا.

فروع المواضيع

لكن فروع المواضيع تفيد جميع المشروعات بغض النظر عن حجمها. فرع الموضوع هو فرع قصير العمر تنشئه وتستعمله لميزة واحدة أو ما يخصها من عمل. لعلك لم تفعل هذا قط مع نظام إدارة نسخ آخر، لأن التفريع والدمج غالبا ما يكونا بطيئين جدا في الأنظمة الأخرى. ولكن الشائع مع جت هو إنشاء فروع والعمل عليها ودمجها وحذفها عدة مرات في اليوم الواحد.

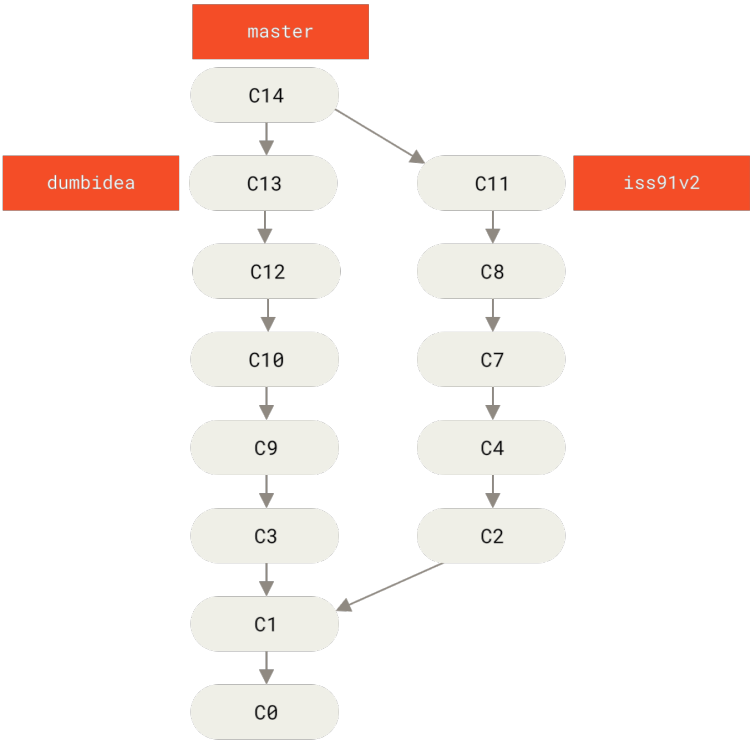
وقد رأيت هذا في الفصل السابق في فرعَي `iss53` و `hotfix` اللذين أنشأتهما، فقد صنعت بضعة إيداعات فيهما ثم حذفتهما فور دمجهما في فرعك الرئيس. يسمح لك هذا الأسلوب بـ«تبديل السياق» سريعا وبالكامل، لأنك قسّمت عملك إلى صومعات، وكل صومعة (فرع) ليس فيها إلا التعديلات التي تخص موضوعا واحدا،فيسّهل ذلك رؤيتها عند المراجعة (code review) وغير ذلك. ويمكنك إبقاء التعديلات هناك دقائق أو أياما أو شهورا، ثم دمجها عندما تكون جاهزة، بغض النظر عن ترتيب إنشائها أو العمل عليها.

لنقُل مثلا إنك عملت (في `master`)، ثم تفرّعت لإصلاح علة (`iss91`)، وعملت عليها قليلا، ثم تفرّعت مجددا (من الفرع الثاني) لتجرب طريقة أخرى لإصلاح العلة نفسها (`iss91v2`)، ثم عدت إلى فرعك الرئيس (`master`) وعملت فيه قليلا، ثم تفرّعت منه لتجربة شيء لست واثقا أنه جيد (فرع `dumbidea`). سيبدو تاريخ إيداعك الآن مثل هذا:



شكل ٢٨. فروع مواضيع متعددة

لنقُل إنك الآن وجدت إصلاحك الثاني للعلة (`iss91v2`) أفضل، وأنك أريت زملاءك فرع `dumbidea` فأخبروك أنه عبقري. فيمكنك إذًا إلقاء فرع `iss91` الأصلي (وفقد الإيداعين `C5` و `C6`)، ودمج الفرعين الآخرين في الفرع الرئيس. سيبدو تاريخك الآن كهذا:



شكل ٢٩. التاريخ بعد دمج `dumbidea` و `iss91v2`

سنتحدث بتفصيل أكبر عن مختلف أساليب سير العمل الممكنة في مشروعات جت في ~~جت الموزع~~، فعليك قراءة هذا الفصل قبل أن تقرر أي أسلوب تفريع سيتبعه مشروعك التالي.

من المهم تذكر أنك عندما تفعل أيًّا من هذا فإن هذه الفروع تبقى محلية بالكامل. فعندما تتفرّع وتدمج، يحدث كل شيء داخل مستودع جت الخاص بك وحسب، لا يحدث أي تواصل مع الخادوم.

الفروع البعيدة

الإشارات البعيدة هي تلك الإشارات الموجودة في مستودعاتك البعيدة، كالفروع والوسوم. يمكنك سرد جميع الإشارات البعيدة بالأمر `git ls-remote <البعيد>`، أو سرد الفروع البعيدة ومعلوماتها بالأمر `git`

`git remote show <البعيد>` (حيث `<البعيد>` هو الاسم المختصر المستودع البعيد). ولكن الشائع هو الانتفاع بـ«الفروع المتعقِّبة للبعيد».

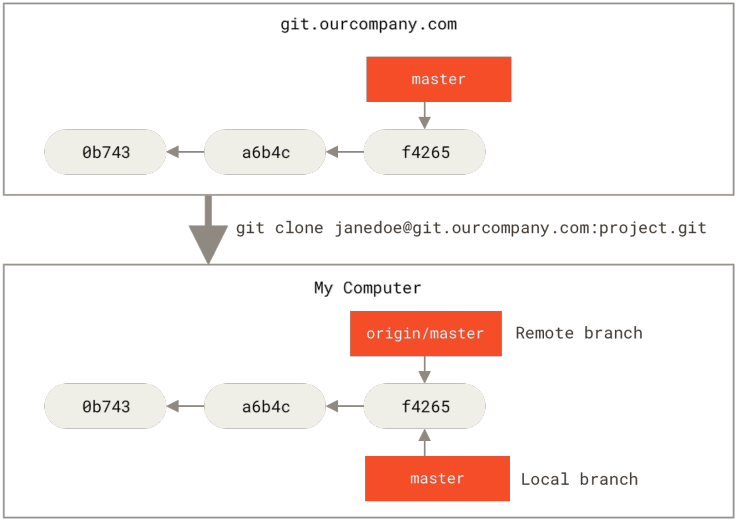
الفرع المتعقِّب لبعيد هو إشارة إلى حالة فرع بعيد. أي أنه إشارة محلية (أي في المستودع الذي على حاسوبك) لكن لا يمكنك تحريكها؛ إن جت يحركها لك عند الاتصال مع الخادوم، حتى يضمن أنها دائما تمثل حالة المستودع البعيد. اعتبرها إشارات مرجعية مثل علامات المتصفح، لتذكرك أين كانت فروع مستودعك البعيد عندما تواصلت معها آخر مرة.

يكون شكل أسماء الفروع المتعقِّبة للبعيد `<remote>/<branch>` (أيْ اسم المستودع البعيد ثم شرطة مائلة ثم اسم الفرع). فمثلا إذا أردت رؤية كيف بدا فرع `master` في مستودعك البعيد `origin` عندما اتصلت به آخر مرة، فانتقل إلى فرع `origin/master`. وإذا كنت تعمل مع زميل على مسألةٍ ودفَعَ فرع `iss53` إلى المستودع البعيد، فقد يكون لديك فرع محلي بالاسم نفسه، ولكن الفرع الذي على الخادوم سيمثله عندك الفرع المتعقِّب للبعيد الذي اسمه `origin/iss53`.

لعل الكلام غامض، فدعنا ننظر إلى مثال. لنقُل إن لديك خادوم جت على شبكتك عنوانه `git.ourcompany.com`. إذا استنسخته، فإن أمر الاستنساخ سيسميه `origin` لك، ويجذب كل ما فيه من بيانات، وينشئ إشارة إلى ما يشير إليه فرع `master` عليه ويسميه `origin/master` محليا. وسيعطيك جت أيضا فرع `master` محلي خاص بك بادئا من المكان نفسه الذي فيه فرع `master` الخاص بالأصل، حتى يتسنّى لك البدء بالعمل.

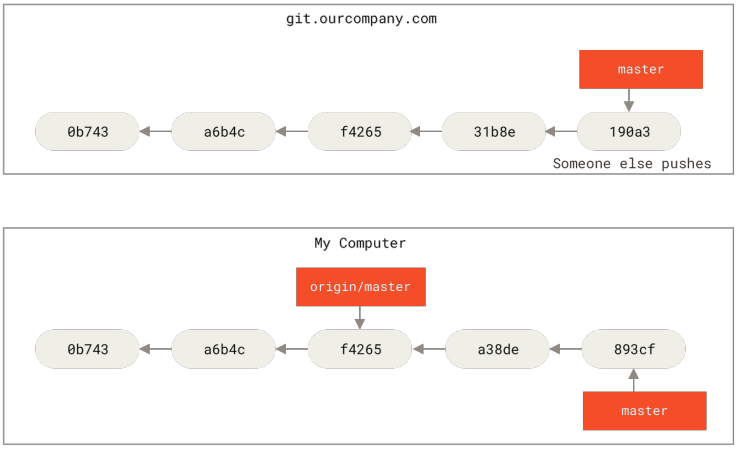
الاسم "origin" ليس مميزا

تماما مثلما أن اسم الفرع الرئيس "master" لا يحمل أي معنى خاص في جت، فكذلك اسم المستودع البعيد الأصل "origin". فإن "master" هو الاسم المبدئي لأول فرع ينشئه جت عندما تستخدم `git init` (وهو السبب الوحيد لشيوعه)، وكذلك "origin" هو الاسم المبدئي للمستودع البعيد عندما تستخدم `git clone`. فإذا استخدمت `git clone -o yalla` مثلا، فإنك ستجد أن `yalla/master` هو اسم الفرع البعيد المبدئي.



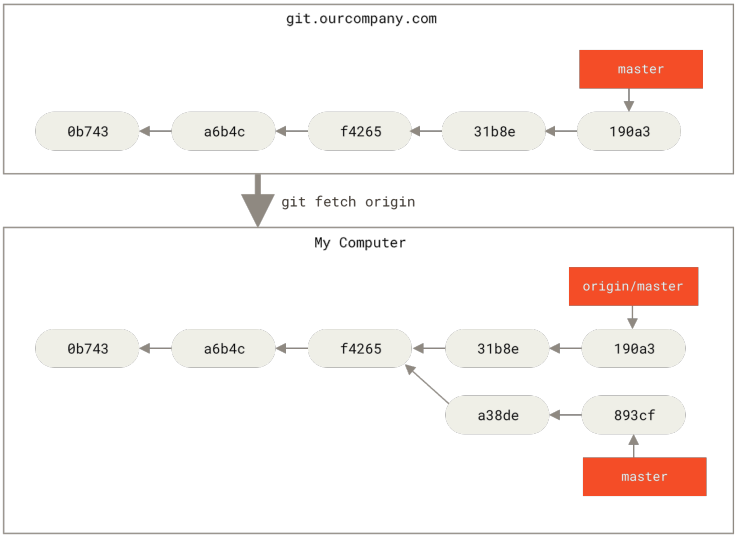
شكل ٣٠. المستودعان البعيد والمحلي بعد الاستنساخ

إذا عملت في فرعك الرئيس المحلي، ودفع أحد إلى الفرع الرئيس في المستودع البعيد، فإن تاريخَي الفرعين سيتقدمان مفترقين. وإن تجنبت الاتصال مع مستودعك البعيد على الخادوم الأصل، فلن تتحرك إشارة `origin/master` التي لديك.



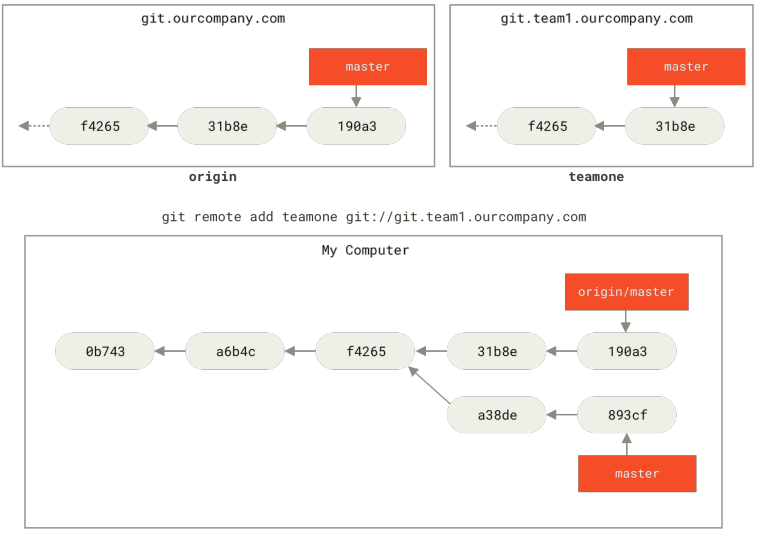
شكل ٣١. قد يفترق العمل المحلي والبعيد

لمزامنة عملك مع مستودع بعيد، نفّذ الأمر `git fetch <البعيد>` (في حالتنا `git fetch origin`). فهذا الأمر يبحث عن المستودع المسمى "origin" (في حالتنا `git.ourcompany.com`)، ويستحضر البيانات التي عليه وليست عندك بعد، ويحدّث قاعدة بياناتك المحلية، ويحرك إشارة `origin/master` الخاص بك لتشير إلى موقعها الجديد المحدَّث.



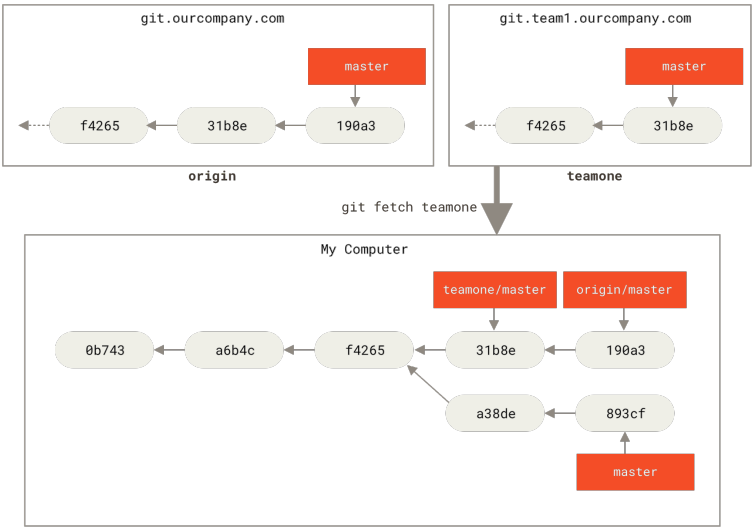
شكل ٣٢. يحدّث أمر الاستحضار `git fetch` فروعك المتعقِّبة للبعيد

لتمثيل وجود خواديم بعيدة عديدة ولإيضاح منظر الفروع المتعقِّبة لهذه المستودعات البعيدة، لنقُل إن لديك خادوم جت داخلي آخر، والذي لا يستخدمه إلا فريق واحد من أجل التطوير، وإن عنوانه هو `git.team1.ourcompany.com`. يمكنك إضافته إشارةً بعيدة جديدة في مشروعك، بأمر `git remote add` كما رأينا في أسس جت، وتسميته `teamone`، والذي يُعتبر اسما مختصرا لرابطه الكامل.



شكل ٣٣. إضافة إشارة إلى خادوم بعيد آخر

والآن، نفّذ أمر `git fetch teamone` لاستحضار كل ما لدى خادوم `teamone` البعيد وليس لديك بعد. ولأن ليس لديه من البيانات إلا جزءًا من التي لدى خادومك الأصلي (`origin`) الآن، فلن يستحضر جت شيئا، ولكنه سيضبط فرعا متعقبا للبعيد يسمى `teamone/master` ليشير إلى الإيداع الذي يشير إليه فرع `master` في مستودع `teamone`.



شكل ٣٤. فرع متعقب للبعيد للفرع `teamone/master`

الدفع

عندما تريد أن تشارك فرعا مع العالَم، فعليك دفعه إلى مستودع بعيد لديك إذن تحريره. ففروعك المحلية لا تُزامَن آليًّا إلى المستودعات البعيدة، حتى التي دفعت إليها؛ عليك دفع الفروع التي تريد مشاركتها بأمر صريح. يسمح لك هذا أن تستعمل فروعا خصوصية للأعمال التي لا تريد مشاركتها، وألا تدفع إلا فروع المواضيع التي تريد التعاون عليها.

مثلا إذا كان لديك فرعا اسمه `serverfix` وتريد العمل عليه مع الآخرين، يمكنك دفعه بالطريقة نفسها التي دفعت بها فرعك الأول؛ نفّذ `git push <remote> <branch>` (أي اسم المستودع البعيد ثم الفرع):

CONSOLE

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

```
 * [new branch]      serverfix -> serverfix
```

هذا اختصار، لأن جت يفك اسم الفرع `serverfix` إلى `refs/heads/serverfix:refs/heads/` ~~داخل جت~~، ولكن عامةً يمكنك تركه. `serverfix`، الذي يعني «ادفع فرعي المحلي `serverfix` إلى المستودع البعيد `origin` لتحدّث فرع `serverfix` عليه». سنفصّل شرح جزء `refs/heads/` في كذلك يمكنك تنفيذ `git push origin serverfix:serverfix` الذي يفعل الشيء نفسه؛ إنه يقول: «خذ فرعي المسمى `serverfix` واجعله فرع `serverfix` في المستودع البعيد». هذه الصياغة مفيدة لدفع فرع محلي إلى فرع بعيد باسم مختلف. فمثلا إن لم تُرِده أن يسمى `serverfix` في المستودع البعيد، فنفّذ `git push origin serverfix:awesomebranch`، فهذا يدفع فرعك المحلي `serverfix` إلى فرع `awesomebranch` في المستودع البعيد.

لا تكتب كلمة مرورك كل مرة

إذا كنت تستعمل رابط HTTPS للدفع، فإن خادوم جت سيسألك عن اسم مستخدمك وكلمة مرورك للاستيثاق. المعتاد أن عميل جت سيسألك عن هذا في الطرفية حتى يعرف الخادوم إذا ما كان مسموحا لك بالدفع.

إذا لم تشأ أن تكتب كلمة مرورك في كل مرة تدفع فيها، فعليك إعداد «تذكُّر مؤقت للاستيثاق» ("credential cache"). أسهل خيار هو جعله في ذاكرة الحاسوب لعدة دقائق، والذي يمكنك إعداده بالأمر `git config --global credential.helper cache`.

لمعلومات أكثر عن خيارات تذكر الاستيثاق المتاحة، انظر ~~Credential~~ ~~Storage~~.

وفي المرة التالية التي يستحضر أحد زملائك من الخادوم، سيحصل على إشارة إلى حيث يشير فرع `serverfix` على الخادوم؛ سيجدها عنده في الفرع البعيد `origin/serverfix`:

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

من المهم ملاحظة أنك عندما تستحضر، يجلب لك هذا فروعًا جديدة متعقبة للبعيد، أي أنك لا تحصل تلقائيًّا على نسخ محلية منها يمكنك تعديلها. بلفظ آخر، لا تحصل آليًّا على فرع `serverfix` جديد في هذه الحالة: لم تُعطَ إلا إشارة `origin/serverfix` التي لا يمكنك التعديل فيها.

لدمج هذا العمل في فرعك الحالي، يمكنك تنفيذ `git merge origin/serverfix`. وإذا أردت فرع `serverfix` خاصًا بك تستطيع العمل فيه، يمكنك تفريعه من الفرع المتعقب للبعيد:

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

يعطيك هذا فرعًا محليًا يمكنك العمل فيه، والذي يبدأ من حيث يقف `origin/serverfix`.

تعقب الفروع

إن سحب فرع محلي من فرع متعقب للبعيد ينشئ آليًّا ما يسمى «فرع متعقِّب» (والفرع الذي يتعقبه يسمى «الفرع المنبع»). الفروع المتعقبة هي فروع محلية ذات علاقة مباشرة بفرع بعيد. فإذا كنت في فرع متعقب وكتبت `git pull`، فسيعرف جت تلقائيا أي مستودع بعيد يستحضر منه وأي فرع يدمج فيه.

عندما تستنسخ مستودعًا، ينشئ جت فرعًا باسم الفرع المبدئي فيه (مثل `master`) ويجعله يتعقب الفرع المبدئي في المستودع الأصل (`origin/master`). ولكن يمكنك إعداد فروع متعقبة أخرى إذا أردت، لتعقب مستودعات بعيدة أخرى، أو لتعقب فرع غير الرئيس. أيسر حالة مثلما رأيتَ آنفًا، عند اتحاد اسم الفرع

المحلي والبعيد، أيْ `git checkout -b <branch> <remote>/<branch>`. وهذه العملية شائعة بما يكفي أن جت يتيح اختصارها بالخيار `--track`:

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

وفي الحقيقة أن هذا شائع جدا حتى إن جت يتيح اختصارا لهذا الاختصار. فإذا كان اسم الفرع الذي تريد سحبه، أولا غير موجود محليًّا بالفعل، وثانيا يطابق تماما اسما في مستودع بعيد واحد، فسينشئ لك جت فرعا متعقِّبا:

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

ولإعداد فرع محلي باسم مختلف عن الفرع البعيد، فسهلٌ استعمال الصيغة الأولى مع اسم فرع محلي مختلف:

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

الآن، فرعك المحلي `sf` سيجذب آليًّا من `origin/serverfix`.

إذا كان لديك بالفعل فرعا محليا وتريد ضبطه ليجذب من فرع بعيد جذبته للتو، أو تريد تغيير الفرع المنبع الذي تتعقبه، استعمل الخيار `-u` أو `--set-upstream-to` مع أمر التفريع `git branch` لضبطه بأمر صريح في أي وقت.

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

الاسم المختصر للمنبع

عندما يكون لديك فرع متعقب مضبوط، يمكنك الإشارة إلى فرعه المنبع بالاختصار `@{upstream}` أو `@{u}`. فإذا كنت في `master` وكان يتعقب `origin/master`، يمكنك تنفيذ أمر مثل `git merge @{u}` بدلا من `git merge origin/master` إن أردت.

لرؤية الفروع المتعقِّبة التي ضبطتها، استعمل الخيار `-vv` مع أمر التفريع `git branch`، ليسرد لك فروعك المحلية مع معلومات مزيدة فيها ما يتعقبه كل فرع وإذا كان فرعك متقدما عنه (`ahead`) أو متأخرا (`behind`) أو كليهما.

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

فنرى هنا أن فرع `iss53` يتعقب `origin/iss53` وأنه «متقدم» (`ahead`) باثنين، أي أن لدينا إيداعين محليين ولم ندفعهما إلى الخادوم بعد. ونرى أيضا أن فرع `master` يتعقب `origin/master` وأنه محدَّث. ثم نرى بعدهما أن فرع `serverfix` يتعقب فرع `server-fix-good` على خادوم `teamone` وأنه متقدم عنه بثلاثة ومتأخر بواحد، أي أن لدى الخادوم إيداعا لم ندمجه في فرعنا بعد، وأن لدينا ثلاثة إيداعات محلية لم ندفعها إليه. ثم نرى في النهاية أن فرع `testing` لا يتعقب أي فرع بعيد.

مهم ملاحظة أن هذه الأعداد ليست إلا منذ آخر استحضار (`fetch`) من كل خادوم نتعقبه. فلا يحاول هذا الأمر الاتصال بالخواديم؛ إنما يخبرك بما يحفظه على حاسوبك عما فيها. فإذا أردت من أعداد التقدم والتأخر أن تكون أحدث ما يكون، فعليك الاستحضار من جميع خواديمك البعيدة قبل تنفيذ هذا الأمر مباشرةً. ويمكنك فعل ذلك هكذا:

CONSOLE

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

الجذب

نعلم أن أمر الاستحضار `git fetch` يستحضر التعديلات التي في المستودع البعيد وليست لديك بعد، ولكنه لا يعدّل مجلد عملك إطلاقا؛ إنما يجلب البيانات لك ويتركك تدمجها بنفسك. ولكن لدى جت أمر يسمى أمر الجذب `git pull`، وهذا الأمر عمليًّا يكافئ استحضارًا `git fetch` متبوعًا مباشرةً بدمج `git merge`، في معظم الحالات. فإذا كان لديك فرع متعقِّب مضبوط كما في الفصل السابق، إما بضبطه صراحةً وإما بأن يضبطه لك أمر الاستنساخ `git clone` أو أمر السحب `git checkout`، فإن أمر الجذب `git pull` سينظر أيّ مستودع وأيّ فرع يتعقبهما فرعك الحالي، ويستحضر ما في المستودع البعيد ويحاول دمجه في فرعك.

الأفضل عمومًا هو الاستخدام الصريح لأمرَي الاستحضار `fetch` والدمج `merge`، فالسحر الذي يقوم به أمر الجذب كثيرًا ما يكون مُلغِزًا.

حذف فروع بعيدة

لنقُل إنك قضيت ما تريد من فرع بعيد، مثلا انتهيت أنت وزملاؤك من إضافة ميزة جديدة ودمجتموها في الفرع الرئيس. يمكنك حذف فرع بعيد بالخيار `--delete` مع أمر الدفع `git push`. فإذا أردت حذف فرع `serverfix` من الخادوم، يمكنك تنفيذ الأمر التالي:

CONSOLE

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

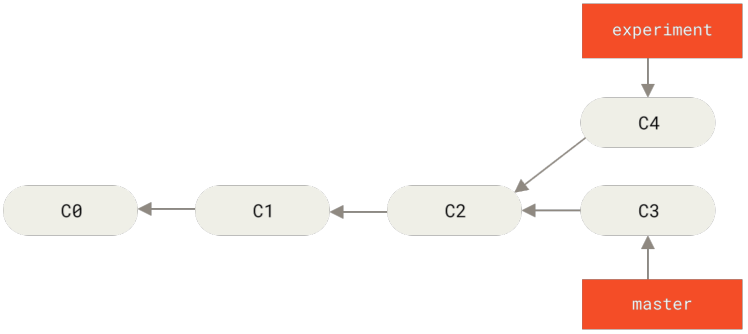
لا يفعل هذا الأمر سوى أنه يحذف الإشارة من على الخادوم. ولكن خواديم جت عمومًا تبقى البيانات موجودة وقتًا، إلى أن يعمل جامع المهملات، فغالبًا ستستطيع استعادته بسَهولة إن حذفته بالخطأ.

إعادة التأسيس

توجد طريقتان في جت لضم التعديلات من فرع إلى آخر: الدمج `merge` وإعادة التأسيس `rebase`. سنتعلم في هذا الفصل ما هي إعادة التأسيس، وكيف نفعلها، ولماذا هي أداة مذهلة فعلا، ومتى لن تود استخدامها.

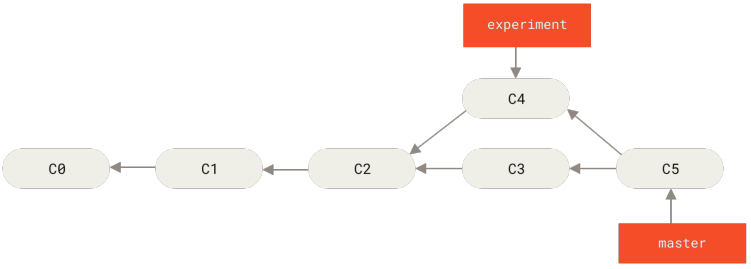
أسس إعادة التأسيس

إذا عدت إلى مثال سابق في أسس الدمج، ستجد أن عملك افترق إلى إيداعات في فرعين مختلفين.



شكل ٣٥. تاريخ بسيط مفترق

أسهل طريقة لضم الفرعين، كما ناقشنا بالفعل، هي أمر الدمج `merge`، والذي يقوم بدمج ثلاثي بين آخر لقطتين في الفرعين (`C3` و `C4`) وآخر سلف مشترك لهما (`C2`)، وينشئ لقطة جديدة (وإيداعًا).



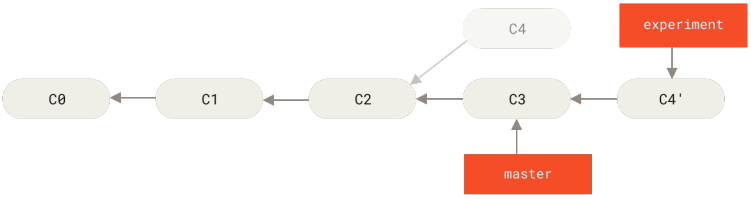
شكل ٣٦. الدمج لضم تاريخ العمل المفترق

لكن توجد طريقة أخرى: يمكنك أخذ رُقعة التعديلات ("patch") التي صنعتها في هذا الإيداع (`C4`) وإعادة تطبيقها على الإيداع الآخر (`C3`). هذه ما نسميها «إعادة التأسيس» ("rebasing") في جت. فبأمر إعادة التأسيس `rebase` يمكنك أخذ جميع التعديلات التي أودعتها في فرعٍ ما، وإعادة صنعها في فرع آخر.

في هذا المثال سنسحب فرع `experiment`، ثم نعيد تأسيسه على الفرع الرئيس `master`.

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

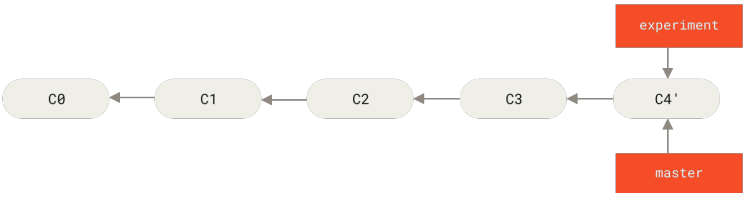
تتم هذه العملية بالذهاب إلى السلف المشترك للفرعين (الفرع الحالي الذي تقف فيه وتريد إعادة تأسيسه، والفرع الذي تريد إعادة التأسيس عليه)، وحساب التعديلات التي تمت في كل إيداع في الفرع الحالي وحفظها في ملفات مؤقتة، ثم ضبط الفرع الحالي إلى الإيداع الذي عنده الفرع الذي تريد إعادة التأسيس عليه، وأخيرا تطبيق كل تعديل عليه بالترتيب.



شكل ٣٧. إعادة تأسيس تعديلات `C4` على `C3`

يمكنك الآن العودة إلى الفرع الرئيس وعمل دمج تسريع ("fast-forward").

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



شكل ٣٨. تسريع فرع `master`

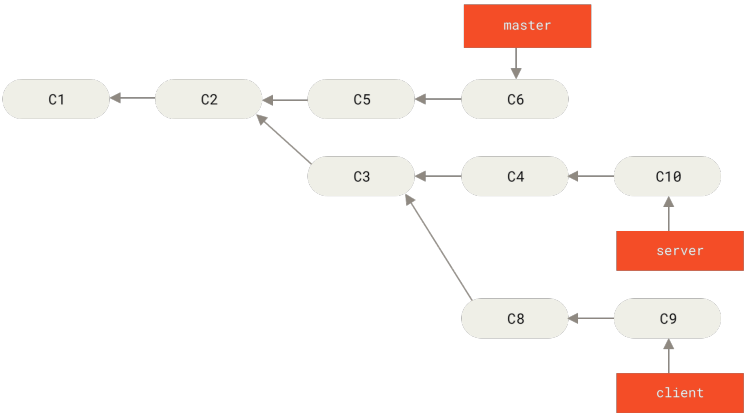
اللقطة التي يشير إليها إيداع `C4'` الآن مطابقة تماما لتلك التي كان يشير إليها `C5` في مثال الدمج. لا فرق في الناتج النهائي، ولكن تعطينا إعادة التأسيس تاريخًا أنظف. فإذا نظرت إلى سجل فرع مُعاد تأسيسه، ستجده تاريخًا خطيًّا: يبدو أن كل العمل تم على التوالي، ولو أنه في الأصل قد تم على التوازي.

غالبا ستفعل هذا لضمان أن إيداعاتك ستُطبّق بنظافة على فرعٍ بعيد — مثلا في مشروع تريد المشاركة فيه لكنك لست مطورًا فيه. فستعمل في هذه الحالة في فرع، ثم تعيد تأسيسه على `origin/master` عندما تكون جاهزًا لتسليم رُقعتك إليهم. فهكذا لن يُجهِد المطورين ضمُ عملك، فما الأمر إلا تسريعًا، أو تطبيقًا نظيفًا للرقعة.

لاحظ أن اللقطة التي يشير إليها الإيداع النهائي، سواء كان آخر الإيداعات المعاد تأسيسها أو كان الإيداع النهائي لدمج، هي اللقطة نفسها؛ ليس الاختلاف إلا في التاريخ. فإعادة التأسيس تعيد صنع تعديلات تاريخ عمل في تاريخ عمل آخر بترتيبها نفسه، لكن الدمج يدمج آخر نقطتين معًا.

إعادات تأسيس شيقة أكثر

يمكنك أيضا جعل إعادة التأسيس تطبّق التعديلات على فرع غير «الفرع المستهدف». لنرَ مثلا تاريخا مثل «تاريخ فيه فرع موضوع متفرع من فرع موضوع آخر». لقد أنشأت فرع موضوع (`server`) لإضافة ميزات في جزء الخادوم في مشروعك، وصنعت إيداعا. بعدئذٍ أنشأت فرعا من هذا الفرع لعمل تعديلات في جزء العميل (`client`) وصنعت بضعة إيداعات. ثم في آخر الأمر عدت إلى فرع `server` وصنعت بضعة إيداعات أخرى.

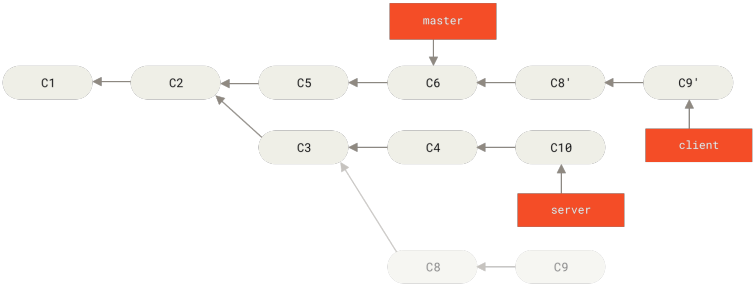


شكل ٣٩. تاريخ فيه فرع موضوع متفرع من فرع موضوع آخر

لنقُل إنك قررت أنك تريد دمج تعديلاتك الخاصة بجزء العميل في المسار الرئيس لكي تصدرها، ولكنك تريد الإبقاء على تعديلات جزء الخادوم حتى تختبرها أكثر. إن التعديلات التي في `client` وليست في `server` (وهي `C8` و `C9`) تستطيع إعادة تطبيقها على فرع `master` بالخيار `--onto` مع أمر `git rebase`:

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

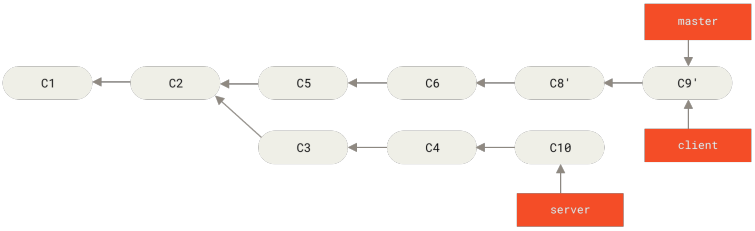
إنما يقول هذا: «احسب فروقات فرع `client` (التعديلات التي سُجّلت فيه) منذ أن افترق عن فرع `server`، ثم أعد تطبيقها في فرع `client` كأنه قد تفرّع من فرع `master` وليس من `server`.» صعبة قليلا، لكن النتيجة عظيمة.



شكل ٤٠. إعادة تأسيس فرع موضوع على فرع موضوع آخر

عندئذٍ يمكنك تسريع الفرع الرئيس `master` (انظر «تسريع الفرع الرئيس لضم تعديلات فرع `client`»):

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

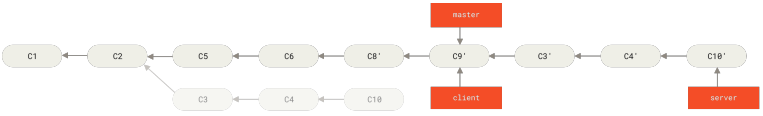


شكل ٤١. تسريع الفرع الرئيس لضم تعديلات فرع `client`

لنقُل إنك قررت جذب فرع `server` كذلك. يمكنك إعادة تأسيس فرع `server` على الفرع الرئيس بلا حاجة إلى سحبه أولا، بالأمر `git rebase <basebranch> <topicbranch>` (أي الفرع الأساس ثم فرع الموضوع)، وهذا يسحب لك فرع الموضوع (`server` في حالتنا) ويعيد تطبيق ما فيه من عمل على الفرع الأساس (`master`):

```
CONSOLE
$ git rebase master server
```

هذا يُعيد تطبيق العمل الذي في فرع الخادوم `server` على العمل الذي في الفرع الرئيس `master`، كما يظهر في «إعادة تأسيس فرع `server` على الفرع الرئيس».



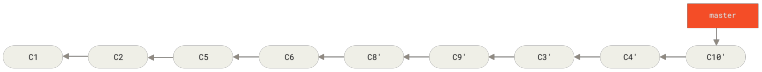
شكل ٤٢. إعادة تأسيس فرع `server` على الفرع الرئيس

عندئذٍ يمكنك تسريع الفرع الأساس (`master`):

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

ثم تستطيع حذف الفرعين `client` و `server` لأن كل ما فيهما قد ضُمَّ بالفعل ولم تعد بحاجة إليهما، فيصير تاريخك في نهاية هذه العملية كما في «تاريخ الإيداعات في النهاية»:

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



شكل ٤٣. تاريخ الإيداعات في النهاية

محذورات إعادة التأسيس

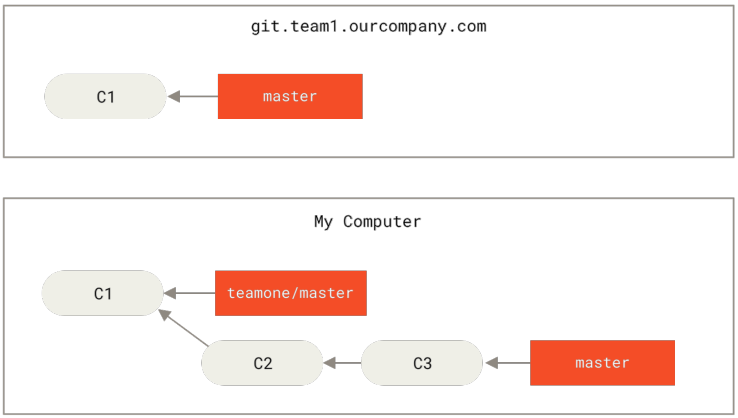
ولكن... نعيم إعادة التأسيس ليس بغير عيوب، والتي يمكن اختصارها في سطر واحد:

لا تعد تأسيس إيداعات لها وجود خارج مستودعك فربما قد بنى الناس عليها عملا.

إذا اتبعت هذه النصيحة الإرشادية، فستكون بخير. وإن لم تفعل، فسيكرهك الناس ويحتقرك الأهل والأصحاب.

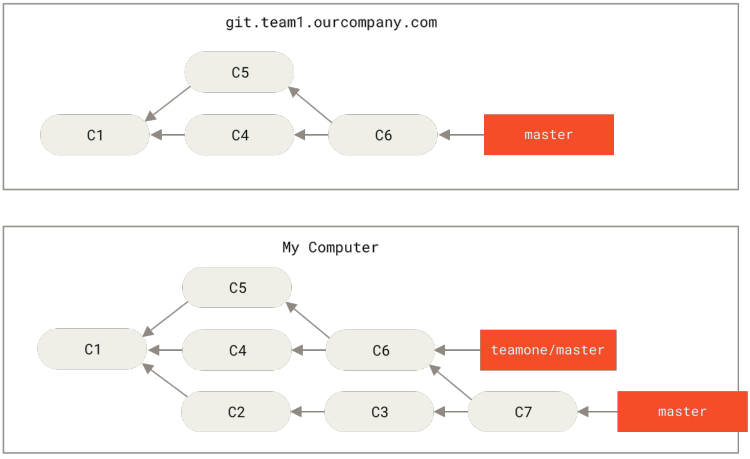
فعندما تعيد التأسيس، فإنك تهجر الإيداعات الموجودة وتصنع إيداعات جديدة شبيهة بالقديمة لكن مختلفة عنها. وإذا دفعت هذه الإيداعات إلى مستودعٍ ما وجذبها الآخرون وبنوا عليها أعمالا، ثم جئت فأعدت كتابة هذه الإيداعات بأمر `git rebase` ثم دفعتها من جديد، فسيضطر زملاؤك إلى إعادة دمج أعمالهم، وستؤول الأمور إلى فوضى عندما تحاول جذب أعمالهم إلى عملك.

لنرَ كيف يمكن لإعادة تأسيس عملٍ منشور أن تسبب مشاكل. لنقُل إنك استنسخت من خادوم مركزي، ثم بنيت عليه عملا. سيبدو تاريخ إيداعك مثل هذا:



شكل ٤٤. استنسخ مستودعا، وابنِ عملا عليه

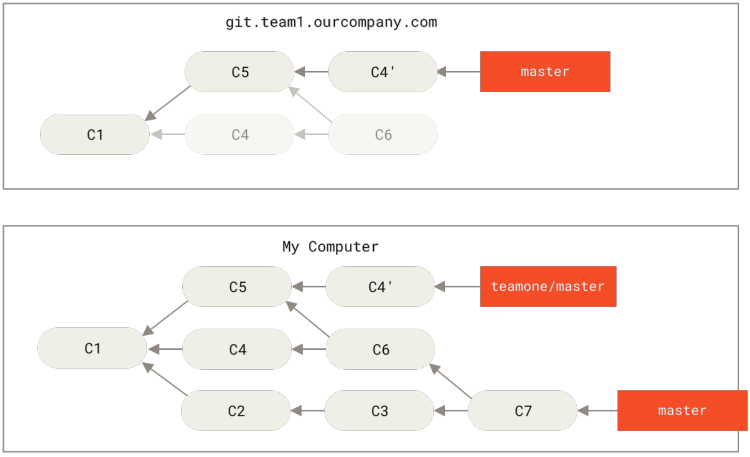
ثم جاء شخصٌ آخر وصنع المزيد من الإيداعات، والتي شملت دمجًا، ثم دفعها إلى الخادوم المركزي. فقمت باستحضار (`fetch`) الفرع البعيد الجديد ودمجه في عملك، فصار تاريخك مثل الآتي:



شكل ٤٥. استحضر المزيد من الإيداعات، وادمجها في عملك

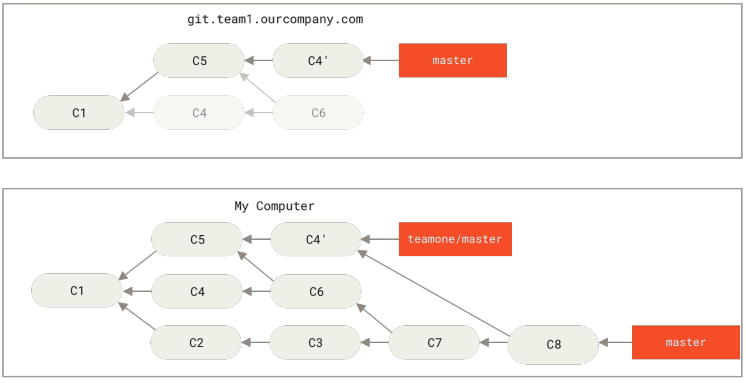
بعدئذ، قرر الذي دفع العمل المدموج أن يتراجع ويعيد تأسيس عمله بدل دمجه، فدفع بالقوة (`git push --force`) لإعادة كتابة التاريخ على الخادوم. ثم استحضرت (`fetch`) من هذا الخادوم، جالبًا

الإيداعات الجديدة.



شكل ٤٦. شخصٌ يدفع إيداعات معاد تأسيسها، هاجرًا بذلك الإيداعات التي بنيت عليها عملك

كلاكما الآن في مأزق. فإذا جذبت، ستصنع إيداع دمج يضم كلا التاريخين، وسيبدو مستودعك مثل هذا:



شكل ٤٧. عندما تدمج العمل نفسه مجددا في إيداع دمج جديد

فإذا نظرت في السجل `git log` عندما يصير تاريخك كهذا، فسترى إيداعين متطابقين في اسم المؤلف

وتاريخ الإيداع ورسالته، فيسبب اللَبس. وأضف إلى ذلك أنك إذا دفعت هذا التاريخ إلى الخادوم، فستعيد تقديم كل هذه الإيداعات المعاد تأسيسها إلى الخادوم المركزي من جديد، والذي سيسبّب لبسًا لأكثر للناس. يمكننا الافتراض أن المطور الآخر لا يريد الإيداعين `C4` و `C6` في التاريخ، ولذا أعاد تأسيسهما.

أعد التأسيس عندما تعيد التأسيس

إذا **وجدت** نفسك في مثل هذا الموقف، فلدى جت وسائل سحرية أخرى قد تساعدك. فلو أن زميلك دفع بالقوة تعديلاتٍ تعيد كتابة عمل بنيتَ عليه، فالتحدي هو أن تعرف ما هو عملك وما الذي أعاد كتابته ذاك الشخص.

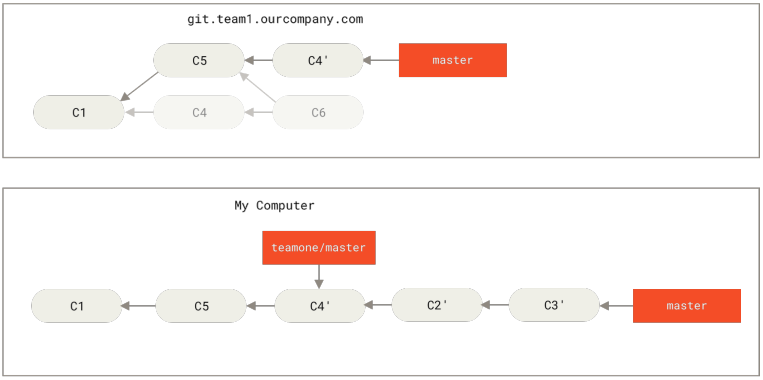
الخبر الجميل أن جت لا يحسب للإيداع وحده بصمة SHA-1، ولكن يحسبها أيضا للرُقعة (الفروقات) التي صنعها ذلك الإيداع. وهذه البصمة تسمى «معرِّف الرقعة» ("patch-id").

فإذا جذبت عملا معاد كتابته وأعدت تأسيسه على الإيداعات الجديدة من زميلك، فغالبا سينجح جت في تمييز ما هو عملك الفريد ويطبّقه على الفرع الجديد.

فمثلا في الموقف الافتراضي السابق، لو أننا أعدنا التأسيس (بالأمر `git rebase teamone/master`)، بدل الدمج عندما كنا في خطوة «شخصٌ يدفع إيداعات معاد تأسيسها، هاجرًا بذلك الإيداعات التي بنيت عليها عملك»، فإن جت سوف:

- يحدد العمل الذي تفرّد به فرعنا (`C2`، `C3`، `C4`، `C6`، `C7`)
- يحدد ما الذي ليس بإيداعات دمج (`C2`، `C3`، `C4`)
- يحدد ما الذي لم تُعد كتابته في الفرع المستهدف (فقط `C2` و `C3`، لأن `C4` له رقعة `C4'` نفسها)
- يطبّق هذه الإيداعات على فرع `teamone/master`

فبدلا مما رأينا في «عندما تدمج العمل نفسه مجددا في إيداع دمج جديد»، سنحصل على نتيجة مثل «أعد التأسيس على عمل معاد تأسيسه ومدفوع بالقوة».



شكل ٤٨. أعد التأسيس على عمل معاد تأسيسه ومدفوع بالقوة

لن ينجح هذا إلا إذا كان الإيداعان `C4` و `C4'` اللذين صنعهما زميلك لهما الرقعة نفسها تقريبا، وإلا فلن يعرف جت عندما يعيد التأسيس أنهما متطابقين وسيضيف رقعة أخرى شبيهة بالإيداع `C4` (والتي غالبا ستفشل في أن تُطبَّق بنظافة، لأن تعديلاتها ستكون مطبقة بالفعل ولو جزئيا).

ويمكنك أن تختصر هذا باستعمال خيار إعادة التأسيس `--rebase` مع أمر الجذب، أي تنفيذ `git pull --rebase` بدلا من `git pull` المجرد. أو يمكنك فعل ذلك يدويا بالاستحضار `git fetch` ثم إعادة التأسيس، أي تنفيذ `git rebase teamone/master` في هذه الحالة.

وإذا كنت تستخدم `git pull` وتريد جعل خيار `--rebase` خيارًا مفترضًا دومًا، يمكنك تفعيل قيمة التهيئة `pull.rebase` بأمر مثل `git config --global pull.rebase true`.

إذا كنت أبدا لا تعيد تأسيس إلا الإيداعات التي لم تغادر حاسوبك، فستكون بخير. وإن كنت تعيد تأسيس إيداعات قد دفعتها، ولكن لم يبنِ عليها أحدٌ آخر إيداعاتٍ، فستكون بخير أيضا. أما إن كنت تعيد تأسيس إيداعات قد دفعتها إلى العالَم وربما بنى عليها الناس أعمالا، فقد تجد نفسك في ورطة مُغيظة منهِكة، ثم ازدراء زملائك لك.

إن وجدت أنت أو زميلٌ لك أن ذلك ضروري يومًا ما، تأكد أن الجميع يعرفون استخدام `git pull --`

`rebase`، لكي تحاول جعل معاناة ما بعد الحادثة أقل سوءًا ولو قليلا.

بين إعادة التأسيس والدمج

الآن وقد رأيت إعادة التأسيس والدمج عمليًّا، قد تتساءل أيهما أفضل. قبل أن نستطيع الإجابة عن هذا السؤال، لنرجع إلى الوراء قليلا ونتحدث عن معنى التاريخ.

إحدى وجهتَي النظر أن تاريخ إيداعات مستودعك هي **سجل لما حدث فعلا**. أي أنها وثيقة تاريخية، ولها قيمة في ذاتها، ويجب ألا يُعبث بها. فمن هذا المنظور، يكاد يُعدّ تغيير تاريخ الإيداعات استهزاءً ومسبّة، لأنك تكذب بشأن ما حدث فعلا. إذًا ماذا لو كانت لدينا فوضى من إيداعات الدمج؟ هكذا جرت الأمور، ويجب أن يحتفظ بها المستودع من أجل الأجيال القادمة.

وجهة النظر المقابلة هي أن تاريخ الإيداعات هو **قصة صناعة مشروعك**. ولأنك لا تنشر المسودة الأولى من كتاب، فلم إذًا تنشر عملًا أشعث أغبر؟ ففي أثناء عملك على مشروع، قد تحتاج سجلا لجميع عثراتك وطرقك المسدودة. ولكن عندما يحين وقت إظهار عملك إلى العالم، فقد تود أن تحكي قصةً متماسكة عن كيفية الوصول من «أ» إلى «ب». ولذلك يستخدم أصحاب هذا المذهب أدوات مثل إعادة التأسيس وتصفية الفروع لإعادة كتابة إيداعاتهم قبل دمجها في الفرع الرئيس، يستخدمون `rebase` و `filter-branch` ليحكون القصة بالطريقة الأنسب للقراء في المستقبل.

لنعد الآن إلى السؤال عن التفضيل بين الدمج وإعادة التأسيس: لعلك وجدت أنه أعقد من أن تكون له إجابة يسيرة. فإن جت أداةٌ قوية، ويتيح لك فعل الكثير بتاريخ مستودعك، ولكن كل فريق وكل مشروع هو حالة خاصة. الآن وقد علمت الأسلوبين وطريقة عملهما، عليك أن تقرر بنفسك ما الأنسب لحالتك الخاصة.

ويمكنك الجمع بين ميزات كليهما: أعد تأسيس التعديلات المحلية قبل دفعها حتى تنظف عملك، ولكن لا تعد أبدًا تأسيس أي شيء دفعته إلى مستودعٍ ما.

الخلاصة

تناولنا أسس التفريع والدمج في جت. ينبغي أن يَسهُل عليك الآن إنشاء فروع جديدة والانتقال إليها والانتقال بين الفروع ودمج فروع محلية معًا. ستقدر أيضا على مشاركة فروعك بدفعها إلى مستودع مشترك، وعلى العمل مع آخرين على فروع مشتركة، وعلى إعادة تأسيس فروعك قبل مشاركتها. التالي: سنتحدث عما تحتاج لتشغيل خادومك الخاص بك لاستضافة مستودعات جت.

الباب الرابع: جت على الخادوم

تستطيع الآن فعل معظم مهامك اليومية التي تستخدم فيها جت. ولكن عليك استعمال مستودع بعيد لأي عمل تعاوني به. ومع أنك نظريا تستطيع دفع التعديلات إلى مستودعات الآخرين الشخصية المحلية والجذب منها، إلا أن فعل هذا متجنَّب لأن من السهل التسبب في خلط الأمور ومَن يعمل على ماذا، إن لم تكن حذرا. وكذلك إذا أردت أن يصل زملاؤك إلى مستودعك حتى عندما يكون حاسوبك مغلقًا أو غير متصل بالشبكة؛ فكثيرا ما يفيد وجود مستودع مشترك مضمون. لذا فالمستحب للتعاون مع غيرك أن تعدّ مستودعًا وسيطًا يستطيع كلاكما الوصول إليه، وتجذبا منه وتدفعا إليه.

تشغيل خادوم جت عملية يسيرة. أولًا تحدد الموافيق (البروتوكولات) التي تريد منه دعمها. وسيتناول الفصل الأول من هذا الباب الموافيق المتاحة ومزايا وعيوب كل منها. ثم تشرح الفصول التالية بعض الترتيبات المعتادة العاملة بهذه الموافيق وكيف تشغّل خادومك بها. وأخيرا سنرى بعض الخيارات المستضافة، إذا لم تمانع استضافة مشاريعك البرمجية على خواديم الآخرين وكنت لا تريد المعاناة بإعداد خادومك الخاص ورعايته.

إن لم تكن مهتمًا بتشغيل خادومك الخاص، فيمكنك تخطي هذه الفصول والانتقال إلى الفصل الأخير من هذا الباب، الذي يريك بعض الخيارات لإعداد حساب مستضاف، ثم انتقل إلى الباب التالي، الذي نناقش فيه التفاصيل الدقيقة المختلفة للعمل في بيئة إدارة موزعة للنسخ.

المستودع البعيد هو عموما مستودع **مجرد**، أيْ مستودع جت ليس له مجلد عمل. ولأن المستودع لا يُستعمل إلا ملتقًى للتعاون، فلا داعي إلى سحب لقطة منه على الحاسوب؛ إنما هو لبيانات جت وحدها. أي بأيسر الكلمات: المستودع المجرد هو محتويات مجلد `.git` الخاص بمشروعك، ولا شيء غير ذلك.

الموافيق (البروتوكولات)

يتيح جت أربعة موافيق مختلفة لنقل البيانات: المحلي، وHTTP، وSSH (أي Secure Shell)، وGit. سنناقش ما هم وما الظروف التي فيها ستود (أو لا تود) استعمالهم.

الميفاق المحلي

الأكثر بدائية هو **الميفاق المحلي** ("Local protocol")، الذي يكون المستودع البعيد فيه هو مجلد آخر على الحاسوب نفسه. ويُستعمل غالبا إذا كان كل مَن في فريقك لديه وصول إلى نظام ملفات مشترك مثل NFS (https://en.wikipedia.org/wiki/Network_File_System) مضموم (mounted)، أو في الحالة الأندر أن يكون كل شخص مستخدما لحاسوب واحد. ولكن تلك الأخيرة ليست حالة مثالية لأن وقتئذٍ ستبيت كل مشاريعك البرمجية على جهاز واحد، وهذا يُزيد احتمال فقد البيانات فقدا كارثيا.

إذا كان لديك نظام ملفات مشترك مضموم، فيمكنك إذًا استنساخ مستودع محلي مكوَّن من ملفات عادية، والدفع إليه، والجذب منه. فلاستنساخ مستودع مثل هذا، أو لإضافته مستودعًا بعيدًا في مشروع موجود بالفعل، فاستعمل مسار المستودع كأنه رابط URL. مثلا، لاستنساخ مستودع محلي، يمكنك فعل شيء مثل هذا:

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

أو مثل هذا:

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

ولكن تصرف جت يختلف قليلا إذا بدأت العنوان بالميفاق `file://` صريحا. فإذا كتبت المسار فقط، فإن جت يحاول صنع روابط صلبة (hardlinks) أو نسخ الملفات التي يحتاجها مباشرةً. أما إذا كتبت `file://`، فإن جت ينفّذ العمليات التي يستعملها في المعتاد لنقل الملفات عبر الشبكة، ويكون هذا في

الغالب أقل كثيرا في الكفاءة. السبب الرئيسي لكتابة البادئة `file://` هي إذا كنت تريد نسخة نظيفة من المستودع بغير الإشارات أو الكائنات الإضافية — وغالبا ما يكون ذلك بعد الاستيراد من نظام إدارة نسخ آخر أو أمر شبيه (انظر دواخل جت لعمليات الصيانة). سنستعمل المسار العادي هنا لأنه أسرع في أغلب الأحيان.

لإضافة مستودع محلي إلى مشروع جت موجود، نفّذ أمرًا مثل هذا:

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

عندئذٍ يمكنك الدفع إلى ذلك البعيد والجذب منه بالاسم المختصر الجديد `local_proj` كما كنت تفعل تمامًا عبر الشبكة.

المزايا

مزايا المستودعات «الملفاتية» أنها سهلة وتستعمل تصاريح الملفات واتصال الشبكة الموجودين فعلا. وإذا كان لديك نظام ملفات مشترك يصل إليه جميع فريقك، فإعداد مستودع عملية سهلة جدا: تضع نسخة المستودع المجرد في مكانٍ يستطيع الجميع الوصول إليه، وتضبط أذونات القراءة والتحرير كما تفعل لأي مجلد مشترك آخر. سنناقش كيفية تصدير نسخة مستودع مجردة لهذا الهدف في تثبيت جت على خادوم.

هذا أيضا خيار ظريف وسريع لجلب عمل شخص آخر من مستودعه. فإذا كنت وزميلك تعملان على مشروع واحد ويريدك أن تسحب شيئا ما، فتنفيذ أمر مثل `git pull /home/badr/project` أسهل كثيرا في الغالب من أن يدفع عمله إلى خادوم بعيد ثم تستحضره أنت بعد ذلك.

العيوب

عيوب هذه الطريقة هي أن الوصول المشترك غالبا ما يكون أصعب كثيرا من الوصول الشبكي العادي، في إعداده وفي الوصول إليه من أماكن مختلفة. فإذا أردت أن تدفع من حاسوبك المحمول عندما تكون في المنزل، فستحتاج إلى ضم القرص البعيد، وهذا قد يكون صعبًا وبطيئًا مقارنةً بوصول عبر الشبكة.

من المهم ذكر أن هذا ليس بالضرورة الخيار الأسرع إذا كنت تستعمل نوعًا من الضم (mount) المشترك. فالمستودع المحلي لا يكون سريعا إلا إذا كان لديك وصول سريع إلى البيانات. فإن مستودع على NFS يكون في الغالب أبطأ من مستودع عبر SSH على الخادوم نفسه، بفرض أن جت يعمل من الأقراص المحلية في كل نظام.

وأخيرا، لا يحمي هذا الميفاق المستودعَ من الإتلاف غير المقصود. فكل مستخدم لديه وصول صَدفي كامل للمجلد «البعيد»، فلا شيء يمنعه من تعديل ملفات جت الداخلية أو إزالتها وتخريب المستودع.

ميفاقا HTTP

يستطيع جت التواصل عبر HTTP بطريقتين مختلفتين. لم يعرف جت قبل النسخة 1.6.6 منه إلا واحدة منهما، وكانت ساذجة جدا وعموما للقراءة فقط. ولكن في نسخة 1.6.6، جاء ميفاق جديد جعل جت يتفاوض بذكاء في شأن نقل البيانات، بطريقة تشبه ما كان يفعل عبر SSH. وصار ميفاق HTTP الجديد هذا في الأعوام الأخيرة أشهر كثيرا لأنه أيسر للمستخدم وأذكي في تواصله. فانتشرت تسمية النسخة الجديدة «ميفاق HTTP الذكي» (Smart HTTP)، والقديمة «ميفاق HTTP البليد» (Dumb HTTP). وسنتناول الميثاق الذكي أولا.

ميفاق HTTP الذكي

يعمل الميفاق الذكي بطريقة كبيرة الشبه بميفاقَي SSH وGit، لكنه يعمل عبر منافذ HTTPS (الآمن) المعيارية ويستعمل آليّات استيثاق HTTP المتنوعة، ما يعني أنه غالبا أسهل للمستخدم من شيء مثل SSH، لأنك مثلا تستطيع استعمال اسم المستخدم وكلمة المرور بدلا من الاضطرار إلى إعداد مفاتيح SSH.

لعله أشهر طريقة لاستعمال جت اليوم، لأن من الممكن إعداده ليتيح المستودع بغير هُوية مثل ميفاق Git، وكذلك ليتيح الدفع إليه بهُوية وتعمية مثل ميفاق SSH. فبدلا من الاضطرار إلى إعداد رابطَين (URL) مختلفين للعمليتين، يمكنك الآن استعمال رابط واحد لكليهما. وإذا حاولت الدفع فطلبَ المستودعُ الاستيثاقَ منك (وهو ما يجب أن يحدث في المعتاد)، فسيسألك الخادوم عن اسم المستخدم وكلمة المرور. والأمر نفسه

للقراءة وحدها.

وفي الواقع، في خدمات مثل جت‌هب، الرابط الذي تستعمله لتصفح المستودع عبر المتصفح (مثلا https://github.com/schacon/simplegit) يمكنك استعماله هو نفسه للاستنساخ، وكذلك للدفع إليه إذا كان لديك الإذن.

ميفاق HTTP البليد

إن لم يستجب الخادوم لطلب ميفاق HTTP الذكي من جت، فسيحاول عميل جت استعمال ميفاق HTTP **البليد** الأيسر. يتوقع الميفاق البليد أن يقدَّم له مستودع جت المجرد كما تُقدَّم الملفات العادية من خادوم الوب. جمال الميفاق البليد هو سهولة إعداده. فليس عليك إلا وضع مستودع جت مجرد في مكانٍ ما في جذر مستندات HTTP، وإعداد خطاف `post-update` ، وتكون قد أتممت المهمة (انظر ~~خطاطيف جت~~). عندئذٍ يستطيع استنساخ المستودع كل مَن يستطيع الوصول إلى خادوم الوب الذي وضعته عليه. فللسماح بقراءة مستودعك عبر HTTP، افعل شيئًا مثل هذا:

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

هذا كل ما في الأمر. وخطاف `post-update` الذي يأتي مبدئيا مع جت ينفذ الأمر المناسب (`git update-server-info`) لجعل الاستحضار والاستنساخ عبر HTTP يعمل بطريقة صحيحة. ويعمل هذا الأمر عندما تدفع إلى المستودع (عبر SSH مثلا). عندئذٍ يستطيع الآخرون الاستنساخ بمثل هذا الأمر:

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

نستعمل في حالتنا هذه مسار `/var/www/htdocs` وهو الشائع مع خواديم Apache. لكن يمكنك

استعمال أي خادوم وب سكونيّ (استاتيكي)؛ ليس عليك سوى وضع المستودع المجرد في مساره، فبيانات جت تقدَّم لطالبها مثل أي ملفات ساكنة عادية (انظر باب ~~دواخل جت~~ للتفاصيل عن كيف تُقدَّم بالتحديد).

وفي العموم ستختار إما تشغيل خادوم HTTP ذكي للقراءة والتحرير، وإما جعل الملفات متاحة للقراءة فقط بالطريقة البليدة. فمن النادر تشغيل نوعَي الخدمتين معا.

المزايا

سنركز على مزايا النسخة الذكية من ميفاق HTTP.

بساطة الرابط الواحد للوصول بنوعيه تسّهل الأمور كثيرا على المستخدم النهائي، وكذلك عدم الاستيثاق إلا عند الحاجة. والاستيثاق باسم مستخدم وكلمة مرور لهو أيضا مزية كبيرة فيه على SSH، فلا يحتاج المستخدمون أن يولّدوا مفاتيح SSH محليًّا ثم يرفعوا مفاتيحهم العمومية إلى الخادوم ليسمح لهم بالتواصل. وإن هذه لمزية عظيمة في قابلية الاستخدام، للمستخدمين الأقل حنكة، وللمستخدمين على أنظمة عليها SSH غير موجود أو غير شائع. وهو أيضا كفؤ وسريع جدا، مثل SSH.

ويمكنك كذلك إتاحة مستودعاتك للقراءة فقط عبر HTTPS الآمن، ما يعني أن بإمكانك تعمية المحتوى خلال نقله، أو حتى جعل العملاء يستعملون شهادات SSL موقّعة مخصوصة.

شيء جميل آخر هو أن HTTP و HTTPS مستعملان بكثرة حتى إن الجدران النارية (firewalls) الخاصة بالمؤسسات تُضبط في الغالب لتتيح مرورهما عبر منافذها.

العيوب

قد يكون إعداد جت عبر HTTPS أصعب قليلا من SSH على بعض الخواديم. ولكن غير ذلك، فلا تكاد توجد مزية لأحد الموافيق الأخرى على HTTP الذكي في تقديم محتوى جت.

فإذا كنت تستعمل HTTP للدفع المستوثَق، فإعطاء اسم المستخدم وكلمة المرور قد يكون في بعض الأحيان

أعقد قليلا من استعمال المفاتيح عبر SSH. ولكن توجد عدة أدوات للاحتفاظ بهما يمكنك استعمالها لجعل العملية أخف كثيرا، مثل Keychain Acess على نظام ماك أو إس و Credential Manager على نظام ويندوز. اقرأ ~~Credential Storage~~ لترى كيف تضبط نظامك للاحتفاظ بكلمة مرور HTTP آمن على نظامك.

ميفاق SSH

النقل عبر SSH شائع عند استضافة جت ذاتيا. هذا لأن معظم الخواديم مُعدَّة فعلا لقَبول الوصول عبره، وإن لم تكن معدة فإعدادها سهل. أيضا SSH هو ميفاق شبكي استيثاقي، ويوجد في كل مكان، وسهل عموما إعداده واستعماله.

لاستنساخ مستودع جت عبر SSH، استعمل رابط `ssh://` مثل:

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

أو استعمل الصيغة المختصرة شبيهة `scp` لميفاق SSH:

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

وفي كلتا الحالتين، إن لم تحدد اسم المستخدم الاختياري، فسيعدّه جت مطابقًا لاسم مستخدم النظام الحالي على حاسوبك.

المزايا

مزايا SSH عديدة. أولا، سهل الإعداد نسبيا، فعفاريته (daemons) منتشرة، ولأكثر مديري الشبكات خبرة فيها، وأغلب أنظمة التشغيل تأتي بها معدَّة أو بأدوات لإدراتها. ثانيا، التواصل عبره آمن، فكل البيانات تُنقل بعد التعمية والاستيثاق. وأخيرا، إنه كفؤ، فيجعل البيانات ذات أقل حجم ممكن قبل نقلها، مثل موافيق HTTPS و Git والمحلي.

العيوب

نقيصة SSH أنه لا يدعم وصول المجهولين إلى مستودعك. فإذا كنت تستعمل SSH، **فيجب** على الناس الحصول على وصول SSH لجهازك، حتى لرؤيتها فحسب، وهذا لا يجعل SSH مستحسَنًا للمشروعات المفتوحة، التي يود الناس أن يستنسخوها للنظر فيها فقط. أما إذا كنت لا تستعمله إلا داخل شبكة مؤسستك، فقد يكون SSH هو الميفاق الوحيد الذي تحتاج إلى التعامل معه. وإذا وددت أن تأذن للمجهولين بالاطلاع فقط على مشروعاتك ولكن تريد SSH أيضا، فعليك إعداد SSH للدفع ثم إعداد ميفاق آخر للآخرين حتى يستحضروا منه.

ميفاق Git

أخيرا، لدينا ميفاق Git. هذا عفريت (daemon) مخصوص مرفق مع جت، ويستمع على منفذ مخصص (9418) ليتيح خدمة شبيهة بميفاق SSH، لكن بغير استيثاق أو تعمية. وعليك إنشاء ملف اسمه `git-daemon-export-ok` في مستودعك لتجعل جت يقدّمه عبر ميفاق Git؛ فالعفريت لن يقدّم مستودعًا ليس فيه هذا الملف، ولكن غير ذلك لا يوجد أمان. إما أن يكون المستودع متاحا للجميع لاستنساخه، وإما لا يكون. هذا يعني أنه عموما لا يمكن الدفع عبر هذا الميفاق. يمكنك تفعيل إذن الدفع، ولكن لانعدام الاستيثاق، فأي شخص على الإنترنت يجد رابط مشروعك يستطيع الدفع إليه. يكفي أن نقول أن هذا نادر.

المزايا

ميفاق Git غالبا ما يكون أسرع ميفاق نقل شبكي متاح. فإن كنت تخدم كمًّا كبيرًا من النقل الشبكي لمشروع عمومي، أو تقدّم مشروعًا كبيرًا لا يحتاج استيثاق المستخدمين للاطلاع عليه، فغالبا ستودّ إعداد عفريت جت ليقدّمه. فهو يستعمل الآلية نفسها التي يستعملها SSH لنقل البيانات، لكن بغير عبء التعمية والاستيثاق.

العيوب

بسبب عدم وجود TLS أو أي نوع من التعمية، فإن الاستنساخ عبر `git://` قد يسبب ثغرة تنفيذ تعليمات برمجية عشوائية (arbitrary code execution)، لذا ينبغي تجنبه إلا إن كنت تعي جيدا ماذا تفعل.

- عندما تنفذ `git clone git://example.com/project.git`، فإن مخترقا يتحكم في جهاز التوجيه (router) الخاص بك سيستطيع تعديل المستودع الذي استنسخته للتو، مثلا يضيف تعليمات برمجية خبيثة فيه. وعندما تصرّف (compile) أو تشغل ما في هذا المستودع الذي استنسخته للتو، فستنفذ تلك التعليمات البرمجية الخبيثة. ابتعد عن تنفيذ `git clone http://example.com/project.git` للسبب نفسه (أي ميفاق HTTP غير الآمن).
- تنفيذ `git clone https://example.com/project.git` (الآمن) لا يعني من تلك العلة (إلا إن استطاع المخترق أن يحضر شهادة TLS للنطاق الذي تتصل به، `example.com` في هذا المثال). تنفيذ `git clone git@example.com:project.git` (بميفاق SSH) لا يعني من تلك العلة أيضا، إلا إن كنت قبلت بصمة مفتاح SSH خاطئة.

وكذلك لا يدعم الاستيثاق، فأي أحد سيستطيع استنساخ المستودع (ولكن غالبا هذا ما تريده بالتحديد). ولعله أيضا من أصعب الموافيق في الإعداد. فيجب أن يشغّل عفريته الخاص، الذي يحتاج تهيئة `xinetd` أو `systemd` أو ما يشبههما. وليس هذا يسيرًا دائما. ويحتاج أيضا وصولًا عبر الجدار الناري إلى منفذ 9418، وهذا ليس منفذًا معتادًا تسمح به دائما الجدران النارية الخاصة بالمؤسسات؛ فخلف تلك الجدران الكبيرة، هذا المنفذ الغامض غالبا ما يُحظر.

تثبيت جت على خادوم

سنتناول الآن إعداد خدمة جت لتشغيل هذه الموافيق على خادومك.

سنعرض هنا الأوامر والخطوات اللازمة لتثبيت أساسي مبسط على خادوم لينكسي، لكن ممكن أيضا تشغيل هذه الخدمات على ماك أو إس أو ويندوز. وفي الحقيقة إعداد خادوم إنتاجي ضمن بنيتك التحتية بالتأكيد سيشمل اختلافات في التدابير الأمنية أو أدوات نظام التشغيل، ولكننا نرجو أن يعطيك هذا فكرة عامة عما ينطوي عليه الأمر.

حتى تعد أي خادوم جت، عليك أولا تصدير مستودع موجود إلى مستودع جديد مجرد، والمستودع المجرد هو مستودع ليس فيه مجلد عمل. هذا في المعتاد عملية سهلة. فلاستنساخ مستودع لإنشاء مستودع جديد مجرد، نفّذ أمر الاستنساخ بالخيار `--bare` («مجرد»): والعُرف أن أسماء مجلدات المستودعات المجردة تنتهي باللاحقة `.git` ، مثل هذا:

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

سيكون لديك الآن نسخة من بيانات مجلد جت في مجلد `my_project.git` .

هذا يكافئ تقريبا شيئا مثل هذا:

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

توجد بعض الاختلافات الطفيفة في ملف التهيئة (”config“)، ولكن لغرضنا اليوم فيكادا يتطابقان. فهذا الأخير يأخذ مستودع جت وحده بغير مجلد عمله وينشئ مجلدا مخصصا له وحده.

وضع المستودع المجرد على خادوم

الآن وقد صار لديك نسخة مجردة من مستودعك، فلا تحتاج سوى وضعه على الخادوم وضبط الموافيق (البروتوكولات). لنقُل إنك أعددت خادوما يسمى `git.example.com` ، ولديك وصول SSH له، وتريد تخزين كل مستودعات جت الخاصة بك في مجلد `/srv/git` فيه. إذا كان مجلد `/srv/git` موجودا على هذا الخادوم، يمكنك إعداد مستودعك الجديد بنسخ مستودعك المجرد إليه:

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

يستطيع الآن المستخدمين الذين لديهم إذن قراءة مجلد `/srv/git` عبر SSH أن يستنسخوا مستودعك بالأمر:

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

وإذا كان لمستخدمٍ إذن تحرير هذا المجلد، ودخل عبر SSH إلى الخادوم، فسيستطيع الدفع إليه متى شاء.

وسيضيف جت إذن التحرير للمجموعة إلى المستودع بطريقة صحيحة إذا استخدمت خيار `--shared` («مشترك») مع أمر الابتداء `git init`. لاحظ أن هذا لن يُتلف أي إيداعات أو إشارات أو أي شيء آخر.

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

قد رأيت كم هو سهل إنشاء نسخة مجردة من مستودع جت ووضعها على خادوم عندكم وصول SSH إليه. فيمكنكم الآن التعاون في مشروع واحد.

مهمٌ ملاحظة أنك حقا لا تحتاج غير هذا لتشغيل خادوم جت مفيد يصل إليه العديدون: تنشئ حسابات SSH، وتضع مستودعًا مجردًا في مكانٍ للجميع إذن قراءته وتحريره. وتمت العملية بنجاح؛ لا شيء آخر مطلوب.

سنرى في الفصول القليلة التالية كيف يمكنك التوسع لإعدادات أكثر تعقيدا. وسيشمل هذا عدم الاضطرار إلى إنشاء حساب مستخدم لكل مستخدم، وإضافة إذن القراءة للجميع للمستودعات، وإعداد واجهات الوب، والمزيد. لكن تذكر أن للتعاون مع بعض الناس على مشروع خصوصي، كل ما تحتاجه هو خادوم SSH ومستودع مجرد.

الترتيبات الصغيرة

إذا كانت شركتك صغيرة أو كنت تجرب جت في مؤسسة ولستم إلا عددا قليلا من المطورين، فقد تكون الأمور يسيرة عليك. فأحد أعقد مناحي إعداد خادوم جت هو إدارة المستخدمين. فإن أردت لبعض المستودعات ألا يقرأها إلا مستخدمون معينون وألا يحررها إلا جماعة أخرى، فقد تجد ترتيب الأذونات

والوصول أصعب.

الوصول عبر SSH

إذا كان لديك خادومٌ يستطيع جميع المطورين الوصول إليه عبر SSH، فمن الأسهل عموما إعداد مستودعك الأول عليه، لأنك بهذا لن تحتاج إلى عمل شيء تقريبا (كما شرحنا في الفصل السابق). وإذا احتجت إلى أذونات وصول أعقد لمستودعاتك، فيمكنك تحقيقها بأذونات نظام الملفات العادية الخاص بنظام تشغيل خادومك.

وإذا أردت وضع مستودعاتك على خادوم ليس فيه حساب لكل مطور يحتاج إذن التحرير في فريقك، فعليك إعداد وصول SSH لكلٍ منهم. إذا كنت تريد فعل هذا على خادوم لديك، فسنفترض أن لديك بالفعل خادوم SSH مثبت عليه، وأنه وسيلتك للتواصل مع الخادوم الجهاز.

توجد أكثر من طريقة لإعطاء إذن الوصول لكل واحد في فريقك. الأولى هي إعداد حساب لكل واحد، وهي عملية سهلة لكن قد تكون مرهقة. فربما لا تريد تنفيذ `adduser` (أو بديله المحتمل `useradd`) والاضطرار إلى ضبط كلمة مرور مؤقتة لكل مستخدم جديد.

الطريقة الثانية هي إنشاء حساب مستخدم `git` واحد على الجهاز، وطلب مفتاح SSH العمومي من كل مستخدم تريد إعطاءه إذن التحرير، ثم إضافة هذه المفاتيح إلى ملف `~/.ssh/authorized_keys` في حساب `git` الجديد هذا. وعندئذٍ سيستطيع كل شخص الوصول إلى ذلك الجهاز عبر حساب `git`. وهذا لا يؤثر على بيانات الإيداعات بأي شكل، فحساب مستخدم SSH الذي تتصل عبره لا يؤثر على الإيداعات التي تسجلها.

طريقة أخرى هي أن يستوثق خادوم SSH الخاص بك من خادوم LDAP أو مصدر استيثاق مركزي آخر أعددته. وما دام لكل مستخدم وصول صَدَفي إلى الجهاز، فأي آلية استيثاق SSH تخطر على بالك ستعمل.

توليد مفتاح SSH عمومي لك

العديد من خواديم جت تستوثق باستعمال مفاتيح SSH العمومية. فعلى كل مستخدم توليد مفتاح إن لم يكن لديه مفتاح فعلا. تتشابه هذه العملية عبر أنظمة التشغيل المختلفة. أولا عليك التحقق أنك لا تملك مفتاحًا فعلا. المكان المبدئي لتخزين مفاتيح SSH الخاصة بالمستخدم هو مجلد `~/.ssh`. فانظر فيه لتعرف إذا كان لديك مفتاحٌ فعلا:

CONSOLE

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

عليك البحث عن ملفين اسم أحدهما يشبه `id_dsa` أو `id_rsa`، والآخر هو الاسم نفسه لكن بالامتداد `.pub`.. فملف `.pub` هو مفتاحك العمومي، والملف الآخر هو نظيره الخصوصي. وإذا لم يكن لديك هذين الملفين (أو لم يكن لديك مجلد `.ssh` من الأساس)، فيمكنك إنشاءهما بتشغيل برنامج يسمى `ssh-keygen` («توليد مفتاح SSH»)، والذي يأتي مع حزمة SSH على أنظمة لينكس وماك أو إس ومع Git for Windows على ويندوز:

CONSOLE

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

هو أولا يتحقق المكان الذي تريد حفظ المفتاح فيه (`.ssh/id_rsa`)، ثم يطلب مرتين إدخال عبارة المرور

(passphrase)، ويمكنك تركها فارغة إذا لم تشأ إدخال عبارة مرور عند استعمال المفتاح. أما إذا أردتها، فعليك إضافة الخيار `-o`؛ فهو يحفظ المفتاح الخصوصي بصيغة أفضل من الصيغة المبدئية في مقاومة كسر عبارات المرور بالقوة العمياء. ويمكنك أيضا استخدام أداة `ssh-agent` («عميل SSH») لمنع الحاجة إلى إدخال عبارة المرور في كل مرة.

الآن، على كل مستخدم فَعَلَ هذا أن يرسل مفتاحه العمومي إليك أو إلى من يدير خادوم جت (بفرض أنك أعدت خادوم SSH ليستعمل المفاتيح العمومية). وليس عليهم سوى نسخ محتويات ملف `.pub` وإرسالها إليك بالبريد الإلكتروني. تبدو المفاتيح العمومية مثل هذا:

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

لشرح أعمق لإنشاء مفتاح SSH على أنظمة التشغيل المختلفة، انظر دليل جت‌هب لمفاتيح SSH (بالإنجليزية): https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent.

إعداد الخادوم

لنعدّ معا وصولَ SSH على الخادوم. سنستعمل في هذا المثال طريقة `authorized_keys` («المفاتيح المستوثَقة») لاستيثاق مستخدميك. سنفترض أيضا أنك تستعمل توزيعة لينكس معتادة مثل أوبنتو.

معظم المشروح هنا يمكن عمله آليًّا بأمر `ssh-copy-id`، بدلا من نسخ المفاتيح العمومية وتثبيتها يدويا.

أولا، أنشئ حساب مستخدم باسم `git` وأنشئ مجلد `.ssh` له.

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

سنحتاج الآن إلى إضافة بعض مفاتيح SSH العمومية للمطورين إلى ملف المفاتيح المستوثَقة الخاص بالمستخدم `git`. لنفرض أن لديك بعض المفاتيح الموثوقة وأنك حفظتها في ملفات مؤقتة. للتذكير، تبدو المفاتيح العمومية هكذا:

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

ليس عليك إلا إضافتها إلى ملف `authorized_keys` الخاص بالمستخدم `git` الموجود في مجلد `.ssh` الخاص به:

```
CONSOLE
$ cat /tmp/id_rsa.badr.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.shams.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.wafaa.pub >> ~/.ssh/authorized_keys
```

أعدّ الآن مستودع مجرد لهم بأمر الابتداء `git init` مع الخيار `--bare`، لتنشئ المستودع بلا مجلد عمل:

```
CONSOLE
$ cd /srv/git
$ mkdir project.git
$ cd project.git
```

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

عندئذٍ يستطيع بدر أو شمس أو وفاء دفع النسخة الأولى من مشروعهم إلى المستودع بإضافته مستودعًا بعيدًا في نسختهم المحلية، ودفع الفرع الذي لديهم إليه. لاحظ أن في كل مرة تريد فيها إضافة مشروع، على شخصٍ ما الوصول إلى الجهاز عبر الصدفة وإنشاء مستودع مجرد. لنجعل `gitserver` اسم المضيف ("hostname") للخادوم الذي أعددت عليه المستودع ومستخدم `git`. إذا كنت تشغّل الخادوم داخليا وأعددت DNS ليشير الاسم `gitserver` إلى هذا الخادوم، فيمكنك استعمال الأوامر كما هي تقريبا (بفرض أن `myproject` هو مشروع موجود وفيه مِلفات):

CONSOLE
```
# على حاسوب بدر
$ cd myproject
$ git init
$ git add .
$ git commit -m 'Initial commit'
$ git remote add origin git@gitserver:/srv/git/project.git
$ git push origin master
```

الآن سيستطيع الآخرون استنساخه إلى أجهزتهم ودفع تعديلاتهم إليه بالسَهولة نفسها:

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

بهذه الطريقة ستحصل سريعا على خادوم جت يتيح إذنَي القراءة والتحرير لبضعة مطورين.

عليك أيضا ملاحظة أن حتى الآن، أولئك المستخدمين جميعهم يمكنهم أيضا الولوج إلى الخادوم والحصول على صدفة المستخدم `git`. إذا أردت تقييد هذا، فعليك تغيير الصدفة إلى شيء آخر في ملف `/etc/passwd`.

يمكنك بسَهولة تقييد حساب المستخدم `git` إلى الأنشطة المرتبطة بجت باستعمال أداة صدفة مقيَّدة اسمها `git-shell` («صدفة جت») وهي مرفقة مع جت. فإذا ضبطتها لتكون صدفة ولوج لحساب المستخدم `git` فإن هذا الحساب لن يكون له وصول صدفة طبيعي إلى خادومك. لاستعمالها، حدد `git-shell` لتكون صدفة ولوج لهذا الحساب بدلا من `bash` أو `csh`. ولفعل هذا، عليك أولا إضافة المسار الكامل لأمر `git-shell` إلى مِلف `/etc/shells` إذا لم يكن موجودا فيه بالفعل:

```
CONSOLE
$ cat /etc/shells       # انظر إن كانت صدفة جيت هنا، وإلا…
$ which git-shell       # فتحقق أن صدفة جيت مثبتة على نظامك
$ sudo -e /etc/shells  # ثم أضف مسارها من الأمر السابق إلى ملف الصدفات
```

يمكنك الآن تغيير صدفة المستخدم بالأمر `chsh <username> -s <shell>`:

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

عندئذٍ يستطيع المستخدم `git` استعمال SSH للدفع والجذب من مستودعات جت، بغير أن يكون له وصولًا صدفيًّا إلى خادومك. وإن حاول، فسيرى رسالة رفض ولوج مثل هذه:

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

ولكن حتى الآن لدى المستخدمين القدرة على توجيه منفذ SSH (أي "port forwarding") للوصول إلى أي خادوم آخر يستطيع ذلك الخادوم الاتصال به. فإذا أردت منع هذا، فأضف الخيارات التالية في أول كل مفتاح تريد تقييده في ملف المفاتيح المستوثقة (`authorized_keys`):

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

ستبدو نتيجة التعديل مثل هذا:

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

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

عندئذٍ ستظل تعمل أوامر جت الشبكية، ولكن المستخدمين لن يعودوا قادرين على الوصول إلى صدفة. وكما ترى في رسالة الرفض، يمكنك أيضا إعداد مجلد في مجلد منزل المستخدم `git` لتخصيص أمر `git-shell` قليلا. فيمكنك مثلا تقييد أوامر جت التي يقبلها الخادوم، أو تخصيص الرسالة التي يراها المستخدمين عندما يحاولون الوصول عبر SSH. نفّذ الأمر `git help shell` للحصول على معلومات مزيدة عن تخصيص الصدفة.

عفريت جت

(من المترجم) كلمة "daemon" ظهرت في MIT اسمًا للعمليات الخدمية التي تعمل في خلفية نظام التشغيل ولا يلاحظها المستخدم. جاءت التسمية نسبة إلى «عفريت ماكسويل»(https://ar.wikipedia.org/wiki/عفريت_ماكسويل) في الفيزياء، الذي يستعمل المعنى القديم للكلمة (في اليونانية والعربية)، وهو الكائن الغَيْبي الذي يعمل في الخفاء، وليس بالضرورة شيطانًا. فأترجمها «عفريت»، وأترجم فعل الصيرورة منها ("daemonize") إلى «عفرتة». الاسم المناظر على ويندوز هو «خدمة» ("service")، وقد بدأ استخدامه حديثًا في لينكس كذلك.

سنعدّ الآن عفريتًا ليقدِّم المستودعات بميفاق "Git". هذا هو الخيار الشائع لإتاحة وصول سريع بغير استيثاق لبيانات جت. تذكر أنه غير مستوثَق، فأي شيء تتيحه عبر هذا الميفاق سيكون عموميا للجميع داخل الشبكة.

فإذا كنت تستخدمه على خادوم خارج جدارك الناري، فعليك ألا تستخدمه إلا للمشروعات التي يمكن أن يراها العالم. وإذا كان خادومك داخل جدارك الناري، فيمكنك استخدامه للمشروعات التي يحتاج الكثير من الناس أو الأجهزة الوصولَ إليه وصول قراءة فقط (مثل خواديم البناء أو التكامل المستمر)، إن لم تكن تريد إضافة مفتاح SSH لكلٍ منهم.

وفي جميع الأحوال، إن ميفاق Git سهل الإعداد نسبيا. فلستَ تحتاج إلا إلى عفرتة هذا الأمر:

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

خيار `--reuseaddr` («أعد استخدام العنوان») يسمح بإعادة تشغيل الخادوم بغير انتظار انتهاء وقت (timeout) الاتصالات القديمة. وخيار `--base-path` («أساس المسار») يتيح للناس استنساخ المشروعات بغير تحديد المسار بكامله. أما المسار الذي في آخر الأمر يخبر عفريت جت مكان المستودعات التي سيُصدِّرها. وإذا كنت تستخدم جدارا ناريا، فستحتاج أيضا إلى فتح منفذ 9418 فيه على الجهاز الذي تعدّه عليه.

طريقة عفرتة هذه العملية تختلف حسب نظام تشغيلك.

لأن `systemd` هو نظام الابتداء (init) الأكثر شيوعا على توزيعات لينكس الحديثة، فيمكنك استخدامه لهذا الغرض. ليس عليك سوى إنشاء الملف `/etc/systemd/system/git-daemon.service` بهذه المحتويات:

```
CONSOLE
[Unit]
Description=Start Git Daemon

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

Restart=always
RestartSec=500ms

StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=git-daemon

User=git
Group=git

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

ربما لاحظت أن اسم المستخدم واسم المجموعة اللذين يشتغل تحتهما عفريت جت هما `git`. يمكنك تغييرهما إلى ما يناسبك، ولكن تأكد من وجود اسم المستخدم المختار على نظامك. وتأكد أيضا من أن الملف التنفيذي لبرنامج جت موجود فعلا في المسار `/usr/bin/git` وإلا فغيّره إلى ما يناسب.

وأخيرا، نفّذ `systemctl enable git-daemon` لبَدء تشغيل الخدمة (العفريت) آليًّا مع بدء تشغيل النظام، ويمكنك تشغيل الخدمة بالأمر `systemctl start git-daemon` وإيقافها بالأمر `systemctl stop git-daemon`.

على الأنظمة الأخرى، قد يناسبك `xinetd` أو بُرَيمج في نظام `sysvinit` أو شيئًا آخر — طالما أنك جعلت هذا الأمر مُعَفرَت ومُراقَب بطريقةٍ ما.

ثم تحتاج إلى إخبار جت بأي المستودعات التي يسمح بالوصول إليها بغير استيثاق عبر خادوم جت. يمكنك فعل هذا بإنشاء ملف اسمه `git-daemon-export-ok` في كل مستودع.

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

وجود هذا الملف يخبر جت بقَبول إتاحة هذا المشروع بغير استيثاق.

ميفاق HTTP الذكي

لدينا الآن وصولا مستوثَقا عبر SSH ووصولا غير مستوثَق عبر `git://`، ولكن يوجد أيضا ميفاق يمكنه عمل كِلا الأمرين في وقت واحد. إعداد HTTP الذكي هو ببساطة مجرد تفعيل بُرَيمج CGI الآتي مع جت المسمى `git-http-backend` على الخادوم. يقرأ هذا البريمج المسار والترويسات (headers) التي يرسلها أمر الاستحضار `git fetch` أو الدفع `git push` إلى رابط HTTP ويحدد إذا كان العميل يستطيع التواصل عبر HTTP (وهذا صحيح لأي عميل جت منذ الإصدارة 1.6.6). وإذا رأي البريمج أن العميل ذكي، فسيتواصل معه بذكاء، وإلا فسيتواصل معه بالميفاق البليد (ولذا فهو متوافق مع الإصدارات القديمة التي تريد القراءة فحسب).

لنرَ إعدادا بسيطا جدا. سنستخدم فيه أباتشي (Apache) كخادوم CGI. إن لم يكن لديك أباتشي مُعدًّا، فيمكنك إعداده على حاسوب لينكسي بفعل شيءٍ كهذا:

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

هذا أيضا يفعّل وِحدات `mod_cgi` و `mod_alias` و `mod_env`، والتي تحتاجها جميعها حتى يعمل هذا

الإعداد بشكل صحيح.

سنحتاج أيضا إلى ضبط مجموعة المستخدمين اليونكسية الخاصة بمجلدات `/srv/git` إلى `www-data`، حتى يتسنّى لخادوم الوب قراءة المستودعات وتحريرها، لأن عملية أباتشي التي تشغّل بُريمج CGI الخاص بنا تعمل (مبدئيا) تحت هذا المستخدم:

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

وكذلك سنحتاج إلى إضافة بعض الأشياء إلى تهيئة أباتشي لتشغيل `git-http-backend` معالجاً ("handler") لأي شيء يأتي من المسار `/git` على خادومك.

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

إن أهملت متغير البيئة `GIT_HTTP_EXPORT_ALL`، فلن يتيح جت للعملاء غير المستوثَقين إلا تلك المستودعات التي فيها الملف `git-daemon-export-ok`، تماما مثلما فعل عفريت جت.

وأخيرا سنحتاج إلى إخبار أباتشي أن يسمح بالطلبات إلى `git-http-backend` وأن يستوثق عمليات التحرير بطريقةٍ ما، مثلا بكتلة `Auth` كهذه:

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

يتطلب هذا إنشاء ملف `.htpasswd` فيه كلمات المرور لجميع المستخدمين المقبولين. هذا مثال على إضافة المستخدم "schacon" إليه:

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

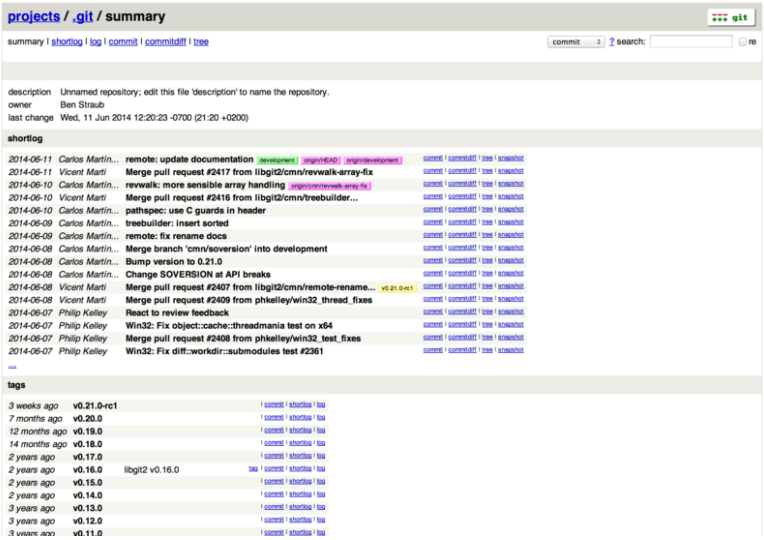
لدى أباتشي طرائق عديدة لاستيثاق المستخدمين، عليك اختيار إحداها وتطبيقها. ليس هذا إلا أيسر مثال استطعنا الإتيان به. ومن شبه المؤكد أنك أيضا ستحتاج إلى إعداد هذا عبر SSH حتى تكون هذه البيانات كلها معمّاة.

لا نود الخوض عميقا في دوامة دقائق تهيئة أباتشي، لأنك قد تستخدم خادوما آخر أو أن لديك احتياجات استيثاق مختلفة. وإنما الأمر أن مع جت برنيج CGI اسمه `git-http-backend`، والذي عند ندائه يفعل كل المفاوضات لإرسال واستقبال البيانات عبر HTTP. ولكنه لا ينفذ أي استيثاق بنفسه. لكن هذا سهل التحكم فيه في مرحلة خادوم الوب الذي يناديه. يمكنك فعل هذا مع ربما أي خادوم وب يدعم CGI، لذا فانطلق مع الخادوم الذي تعرفه حق المعرفة.

لمزيد من المعلومات عن تهيئة الاستيثاق في أباتشي، انظر وثائق أباتشي هنا: https://httpd.apache.org/docs/current/howto/auth.html (بالإنجليزية).

GitWeb

تستطيع الآن الوصول إلى مشروعك مع إذن التحرير أو مع إذن القراءة فقط. وقد تود الآن إعداد واجهة وب رسومية يسيرة له. يأتي جت مع بُرنْج CGI يسمى جتوِب والذي يُستخدم أحيانا لهذا الغرض.



شكل ٤٩. واجهة وب جت‌وب

فإذا أردت رؤية كيف يبدو جت‌وب لمشروعك، فمع جت أمر يشغّل نسخة مؤقتة منه إذا كان لديك خادوم وب خفيف على نظامك مثل `lighttpd` أو `webrick`. على الأنظمة اللينكسية غالبا يكون `lighttpd` مثبتاً، فقد تستطيع تشغيله بتنفيذ `git instaweb` في مجلد مشروعك. وإن كنت على ماك، فإن نسخة Leopard تأتي بلغة Ruby مثبتة مبدئيا، فيكون `webrick` هو الظن الأقرب. لبَدء `instaweb` بشيء غير `lighttpd`، فعليك تشغيله بخيار `--httpd`.

```
CONSOLE
$ git instaweb --httpd=webrick
[2009-02-21 10:02:21] INFO  WEBrick 1.3.1
[2009-02-21 10:02:21] INFO  ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```

فهذا يشغّل خادوم HTTPD على منفذ 1234 ويفتح متصفح الوب على تلك الصفحة آليًّا. فهو يسهّل عليك. وعندما تقضي ما تريد وتود إيقاف الخادوم، نفّذ الأمر نفسه لكن بالخيار `--stop`:

```
CONSOLE
$ git instaweb --httpd=webrick --stop
```

وإذا كنت تود تشغيل واجهة الوب على خادوم طوال الوقت لفريقك أو لمشروع مفتوح تستضيفه، فستحتاج إلى إعداد بريمج الـCGI ليقدّمه خادوم الوب العادي الذي تستخدمه. بعض توزيعات لينكس تأتي بحزمة `gitweb` والتي قد تستطيع تثبيتها بأمر مثل `apt` أو `dnf`، لذا فقد تود تجربة هذا أولا. سنمر سريعا جدا على تثبيت جت‌وب يدويا. عليك أولا الحصول على مصدر جت، الذي فيه جت‌وب، ثم توليد بُرَيمج CGI المخصص:

```
CONSOLE
$ git clone https://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/srv/git" prefix=/usr gitweb
    SUBDIR gitweb
    SUBDIR ../
make[2]: `GIT-VERSION-FILE' is up to date.
    GEN gitweb.cgi
    GEN static/gitweb.js
$ sudo cp -Rf gitweb /var/www/
```

لاحظ أنك تحتاج إلى إخبار هذا الأمر بمكان مستودعاتك في المتغير `GITWEB_PROJECTROOT`. والآن، تحتاج إلى جعل أباتشي يستخدم CGI لهذا البريمج، ويمكنك فعل ذلك بإضافة مستضيف وهمي (VirtualHost) له:

```
CONSOLE
<VirtualHost *:80>
    ServerName gitserver
    DocumentRoot /var/www/gitweb
    <Directory /var/www/gitweb>
        Options +ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
        AllowOverride All
        order allow,deny
        Allow from all
        AddHandler cgi-script cgi
        DirectoryIndex gitweb.cgi
    </Directory>
</VirtualHost>
```

مجددا، لتقديم جت‌وب تستطيع استخدام أي خادوم وب يدعم CGI أو Perl. وإذا أردت شيئا آخر فليس إعداده بالصعب. يمكنك الآن زيارة `http://gitserver/` لرؤية مستودعاتك على الشابكة.

GitLab

لعلك وجدت أن جت‌وب GitWeb ساذجا قليلا أو كثيرا. فإذا كنت تبحث عن خادوم جت حديث ومكتمل الخصائص، فيوجد عدد من الحلول مفتوحة المصدر والتي يمكنك تثبيتها بدلا منه. ولأن جت‌لاب من أشهرها، فإننا سنتناول تثبيته واستخدامه مثالا. هذا الخيار أصعب من جت‌وب وسيحتاج منك رعاية أكثر، لكنه مكتمل الخصائص.

التثبيت

جت‌لاب هو تطبيق وب مبني على قاعدة بيانات، لذا فتثبيته أعقد من بعض خواديم جت الأخرى. لكن لحسن الحظ هذه العملية موثقة بالكامل ومدعومة جيدا. يوصي جت‌لاب بقوة بتثبيته على خادومك عبر الحزمة الرسمية الحافلة، المسماة حزمة "Omnibus GitLab".

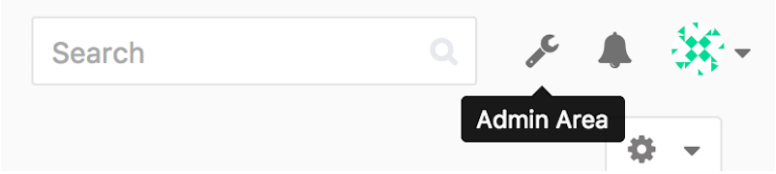
خيارات التثبيت الأخرى هي:

- GitLab Helm chart، لاستخدامه مع Kubernetes.
- حزم Docker لـ GitLab، لاستخدامها مع Docker.
- من مِلفات المصدر.
- موفِّرو الخدمات السحابية، مثل AWS و Google Cloud Platform و Azure و OpenShift و Digital Ocean.

للمزيد من المعلومات (بالإنجليزية) انظر GitLab Community Edition (CE) readme (https://gitlab.com/gitlab-org/gitlab-foss/-/blob/master/README.md).

الإدارة

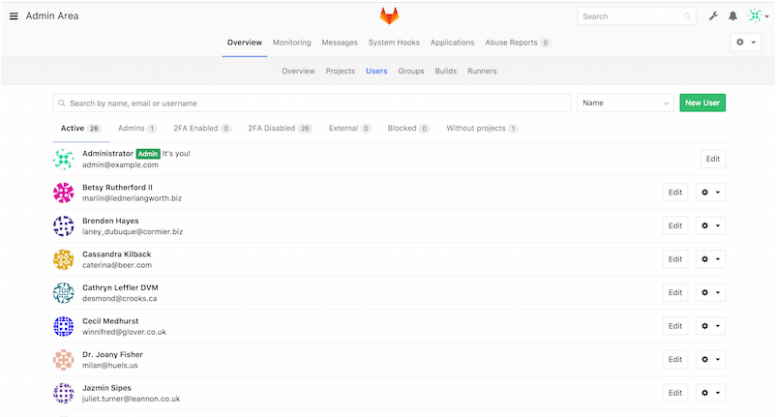
واجهة إدارة جت‌لاب هي واجهة وب. ليس عليك إلا توجيه متصفحك إلى اسم المُضيف (”hostname“) أو عنوان IP الذي ثبتَّ عليه جت‌لاب، ثم لُج بحساب المدير. اسم المستخدم المبدئي هو `admin@local.host`، وكلمة المرور المبدئية هي `5iveL!fe` (والتي عليك تغييرها فورا). بعد الولوج، اضغط على رمز ”Admin area“ (منطقة الإدارة) في القائمة التي على اليمين بالأعلى.



شكل ٥٠. زر ”Admin area“ (منطقة الإدارة) في قائمة جت‌لاب

المستخدمون

على كل من يريد استخدام خادوم جت‌لاب الخاص بك الحصول على حساب مستخدم. حسابات المستخدمين أمر يسير؛ هي في الأساس معلومات شخصية مرتبطة ببيانات الولوج. كل حساب مستخدم لديه **مساحة أسماء** (”namespace“)، وهي تجميع منطقي للمشروعات الخاصة به. فمثلا إذا كان لدى المستخدم `shams` مشروع اسمه `project`، فإن رابط ذلك المشروع سيكون `http://server/shams/project`.



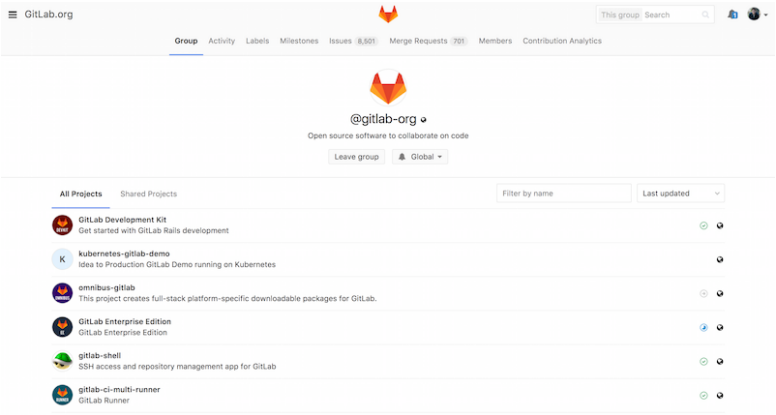
شكل ٥١. شاشة إدارة المستخدمين على جت‌لاب

يمكنك إزالة حساب مستخدم بطريقتين: «حظر» (”block“) حساب مستخدم يمنعه من الولوج إلى خادومك، ولكن كل بياناته التي في مساحة أسمائه ستبقى، والإيداعات الموقَّعة ببريده ستظل تشير إلى صفحته الشخصية على خادومك.

أما «محو» (”destroy“) مستخدمٍ فيزيله تماما من قاعدة البيانات ومن نظام الملفات؛ ستُزال كل المشروعات والبيانات التي في مساحة أسمائه، وكذلك كل المجموعات التي يملكها. طبعا هذا الفعل مستديم الأثر وأشد إتلافا، ونادرا ما ستحتاجه.

المجموعات

المجموعة في جت‌لاب هي تجميعة من المشروعات، مع معلومات عن كيفية وصول المستخدمين إلى هذه المشروعات. كل مجموعة لديها «مساحة أسماء مشروعات»، تماما مثلما للمستخدمين، لذا فإن كان في المجموعة `training` مشروع اسمه `materials`، فسيكون رابطه `http://server/training/materials`.



شكل ٥٢. شاشة إدارة المجموعات على جت‌لاب

ترتبط كل مجموعة بعدد من المستخدمين، ولكل منهم مستوى من الصلاحيات لمشروعات المجموعة وكذلك المجموعة نفسها. وتتراوح هذه المستويات من "Guest" («زائر»: للمسائل "issues" والمحادثات "chat" فقط)، إلى "Owner" («مالك»: للتحكم الكامل في المجموعة وأعضائها ومشروعاتها). وهذه المستويات كثيرة يصعب سردها هنا، لكنّ شاشة الإدارة في جت‌لاب فيها رابطا مفيدا.

المشروعات

المشروع في جت‌لاب هو تقريبا مستودع جت واحد. ينتمي كل مشروع إلى مساحة أسماء واحدة، إما لمستخدم وإما لمجموعة. وإذا كان المشروع ينتمي إلى مستخدم، فلدى مالك المشروع تحكم مباشر في من لديه حق الوصول إلى المشروع. وإذا كان ينتمي إلى مجموعة، فصلاحيات أعضاء المجموعة هي التي تحدد.

كل مشروع له مستوى ظهور، ليحدد من يستطيع رؤية صفحات هذا المشروع ومستودعه. فإذا كان المشروع «خصوصيا» ("Private")، فلا يظهر إلا لمن يحددهم مالك المشروع بالاسم. وإذا كان «داخليا» ("Internal")، فإنه لا يظهر إلا للمستخدمين الوالِجين. وإذا كان «عموميا» ("Public") فإنه يظهر للجميع. لاحظ أن هذا يتحكم في الوصول عبر `git fetch` وكذلك عند الوصول عبر واجهة الوب لهذا المشروع.

الخطاطيف

يدعم جت‌لاب الخطاطيف، على مستوى المشروع وعلى مستوى النظام ككل. سيرسل خادوم جت‌لاب طلب HTTP بطريقة POST مع JSON وصفي كلما حدثَ حدثٌ مناسب. وهذه طريقة عظيمة لربط مستودعات جت وخادوم جت‌لاب بباقي الأدوات الآلية التي تستخدمها ضمن التطوير، مثل خواديم التكامل المستمر CI، وغرف المحادثة، وأدوات النشر (”deployment“).

الاستخدام الأساسي

أول ما قد تود عمله على جت‌لاب هو إنشاء مشروع، وذلك بالضغط على رمز ”+“ على شريط الأدوات. ستُسأل عن اسم المشروع، ومساحة الأسماء التي ينتمي إليها، ومستوى ظهوره. معظم ما تحدده هنا ليس مستديمًا، فتستطيع تغييره فيما بعد من واجهة الإعدادات. اضغط على ”Create Project“ («إنشاء المشروع») لإتمام العملية.

وما إن يكون المشروع حيًّا، قد تود ربطه بمستودع محلي. يمكن التواصل مع أي مشروع عبر HTTPS أو SSH. ويمكنك استعمال أي منهما لإضافة مستودع جت بعيد في مستودعك المحلي. ستجد الروابط في أعلى الصفحة الرئيسية للمشروع. فمثلا تنفيذ هذا الأمر في مستودع محلي سينشئ بعيدا بالاسم `gitlab` مربوطا بالمشروع المستضاف:

CONSOLE

```
$ git remote add gitlab https://server/namespace/project.git
```

أما إذا لم يكن لديك نسخة محلية من المستودع، فيمكنك استنساخه بسهولة هكذا:

CONSOLE

```
$ git clone https://server/namespace/project.git
```

أيضا تتيح واجهة الوب طرائق مختلفة مفيدة للنظر إلى المستودع نفسه. فتُظهر الصفحة الرئيسية لكل مشروع آخر الأنشطة، والروابط التي في أعلى الصفحة تُريك ملفات المشروع وتاريخ إيداعاته.

التعاون

أسهل طريقة للتعاون على مشروع على جت‌لاب هي إعطاء كل مستخدم إذن الدفع مباشرةً إلى المستودع. يمكنك إضافة المستخدمين إلى المشروع بالذهاب إلى قسم الأعضاء ("Members") في إعدادات هذا المشروع، وربط المستخدمين الجدد بمستويات الوصول المناسبة. (مررنا على مستويات الوصول في المجموعات). إذا كان للمستخدم مستوى وصول «مطور» ("Developer") فيستطيع دفع الإيداعات والفروع مباشرةً إلى المستودع.

لكن طريقة أخرى للتعاون بغير ضم المطورين إلى المستودع هي طلبات الدمج ("merge request"). فتتيح هذه الميزة لأي مستخدم يستطيع رؤية المشروع أن يساهم فيه بطريقة محكومة. فالمستخدمون ذوو الوصول المباشر يمكنهم إنشاء فرع ودفع إيداعاتهم إليه ثم إنشاء طلب لدمج فرعهم في الفرع الرئيس أو أي فرع آخر. أما المستخدمون الذين ليس لديهم إذن الدفع للمستودع، فيمكنهم «اشتقاق» ("fork") المستودع لإنشاء نسختهم الخاصة منه، ودفع إيداعاتهم إلى نسختهم الخاصة بهم، ثم إنشاء طلب لدمج اشتقاقهم في المشروع الأصلي. يسمح هذا النموذج للمالك بالتحكم الكامل في كل ما يدخل المستودع ومتى يدخل، وفي الوقت نفسه يسمح بمساهمات المستخدمين غير الموثوق فيهم.

طلبات الدمج والمسائل هما أهم وحدات النقاشات الطويلة في جت‌لاب. فكل طلب دمج يسمح بنقاش على كل سطر من سطور التعديل المقترح (والذي يدعم نوعا خفيفا من مراجعة الكود)، إضافةً إلى نقاش عام. يمكن تكليف مستخدم بإتمام طلب دمج أو مسألة، أو جعلهما ضمن مرحلة ("milestone").

ركّز هذا الفصل في معظمه على خصائص جت‌لاب المرتبطة بجت. ولكنه مشروع ناضج ومكتمل الخصائص، ويتيح ميزات أخرى ليساعد فريقك في العمل معا، مثل موسوعات المشروعات وأدوات رعاية الأنظمة. من محاسن جت‌لاب أنك ما إن تتم إعداد الخادوم وتشغيله، فيندر أن تحتاج إلى تعديل ملف تهيئة أو الوصول إلى الخادوم عبر SSH؛ فواجهة الوب تتيح معظم أفعال الإدارة والاستخدام العام.

خيارات الاستضافة الخارجية

إن لم تشأ خوض غمار العمل المطلوب لإعداد خادوم جت الخاص بك، فلديك عدة خيارات لاستضافة مشروعاتك على مواقع استضافة خارجية متخصصة. لهذا منافع عديدة: أن مواقع الاستضافة عموما أسرع في الإعداد وأسهل في بدء المشروعات عليها، وليس عليك رعاية الخادوم ولا صيانته ولا مراقبته. حتى إن أعددت وشغّلت خادومك الخاص داخليا، فقد تود استخدام موقع استضافة عمومي لمشاريعك البرمجية المفتوحة؛ فهذا يسهّل على المجتمع أن يجدها ويساعدك فيها.

لدينا اليوم عدد مهول من الاستضافات، كلٌ له مزايا وعيوب مختلفة. تجد قائمة محدَّثة في صفحة الاستضافات في موسوعة جت الرسمية: https://archive.kernel.org/oldwiki/git.wiki.kernel.org/index.php/GitHosting.html.

سنتناول جتهب بالتفصيل في ~~GitHub~~، لأنه أكبر استضافة جت مطلقا، وقد تحتاج إلى التعامل مع مشروعات مستضافة عليه على أيّ حال، ولكن توجد عشرات الخيارات الأخرى إذا لم تشأ إعداد خادوم جت الخاص بك.

الخلاصة

لديك عدة خيارات لإعداد وتشغيل مستودع جت بعيد حتى يمكنك التعاون من الآخرين أو مشاركة عملك.

يتيح لك تشغيل خادومك الخاص تحكمًا كبيرًا ويسمح لك بتشغيله داخل جدارك الناري (firewall) الخاص، لكن مثل هذا الخادوم يحتاج قدرًا لا بأس به من الوقت لإعداده ورعايته. أما إذا وضعت مشاريعك البرمجية على خادوم مستضاف، فستجده أسهل إعدادًا ورعايةً. لكن يجب أن يكون مسموحًا لك بذلك، فإن بعض المؤسسات لا تسمح.

نتوقع أن من السهل نسبيا تحديد الحل أو توليفة الحلول الأنسب لك ولمؤسستك.

الملحق الرابع: دليل المصطلحات

access	وصول، إذن
access control level	مستوى الوصول (؟)
advantage	مزيّة (ج: مزايا) (وليس ~~ميزة/ميزات~~؛ هذه feature؛ انظر الفرق في الملحق الآخر)
alias	كُنية (ج: كُنيات)
amend	تصحيح
annotated tag	وسم معنون (انظر أيضا lightweight tag)
Apache	أباتشي
archive, archived file	ملف مضغوط
argument	مُعامِل (ج: مُعامِلات)
assign, assignment	إسناد
attribute	خاصية
authentication	استيثاق، يستوثق
autocompletion	إكمال آلي

automatically	تلقائيا، آليا
avatar	صورة شخصية، صورة تشخيصية
backward compatibility	التوافقية مع الإصدارات السابقة
bare	مجرد
blob	كتلة (ج: كتل)
branch, branching	فرع، تفريع
bug	عِلة (ج: علل)
change	تعديل
chapter (in the book)	باب
character	مَحْرِف (ج: مَحارف)
check	تحقّق [الشيء] (بغير «من»)، تفقّد [الشيء]
check out	سحب
checksum	[قيمة] بصمة
clone	استنساخ
command	أمر
command line	سطر أوامر

commit, commit, committed, do/make a commit	يودع، إيداع، مودع، يصنع إيداعا
compress, compression, compressed	يضغط، ضغط، مضغوط
computer	حاسوب (ج: حواسيب)
configure, configuration	ضبط، تهيئة
continuous integration	تكامل مستمر
contribute, contribution	يساهم، مساهمة
custom	مخصص
customized	مخصص، مفضل
daemon	عفريت (ج: عفاريت)
daemonize	عفرتة
dashboard	صفحة رئيسية
default	مبدئي، مفترض
delta	فرق (ج: فروقات)
detached HEAD [state]	¯_(ツ)_/¯
develop, development	تطوير

developer	مطوِّر
directory	مجلد
disk	قرص
distributed	موزع
download	تنزيل
downstream	مصب (انظر أيضا upstream)
entry	بيان
environment	بيئة
environment variable	متغير بيئة (ج: متغيرات بيئة)
error-prone	خطّاء، عُرضة للخطأ
exit non-zero (verb, intransitive)	إنهاء التنفيذ بقيمة خروج غير الصفر
experience (previous knowledge)	خبرة
experience (usage, like user ∿ (UX), developer ∿ (DX))	تجربة
fast-forward	تسريع
fast-forward merge	دمج تسريع

feature	خصيصة (ج: خصائص)، ميزة (ج: ميزات) (وليس مزية/مزايا؛ هذه advantage؛ انظر الفرق في الملحق الآخر) (انظر أيضا killer feature)
folder	مجلد
fork	اشتق، يشتق، اشتقاق
fully featured	مكتمل الخصائص (انظر أيضا feature)
Git	جت
GitHub	جت‌هب
GitLab	جت‌لاب
GitWeb	جت‌وب
glob	أنماط توسيع [المسارات]
graphical user interface, GUI	واجهة رسومية
hacker (1. expert or eager learner: meanings 1–7 in the Jargon File http://catb.org/esr/jargon/html/H/) (hacker.html)	خارق

مخترق	hacker (2. cracker: meaning 8 in the the Jargon File or the standalone entry http://catb.org/esr/jargon/html/C/) (cracker.html)
معالج	handler (Apache)
قرص [صلب]	hard disk
رابط صلب	hardlink
بصمة	hash
الفرع الرأس(؟)	HEAD branch
ترويسة	header
إشارة الرأس	HEAD [pointer]
شجري	hierarchical
تاريخ	history
خطاف (ج: خطاطيف)	hook
برنيج خطاف	hook script
يستضيف، استضافة	host, hosting

hostname	اسم المضيف(؟)
include	ضم
index (in Git)	الفهرس
inode	مؤشر فهرسة
install, installation (1. software)	تثبيت
install, installation (2. non-software)	تركيب
integrated development environment, IDE	بيئة تطوير [متكاملة]
integrity	سلامة
issue	مشكلة، مسألة
killer feature	ميزة قاتلة للمنافسة (انظر أيضا feature)
landing page	صفحة استقبال
license	رخصة
lightweight tag	وسم خفيف (انظر أيضا annotated tag)
limit (v)	تقييد
list (n)	قائمة

list (v)	سرد
login	ولوج
log (n)	سجل
maintain, maintenance	تطوير، رعاية
merge commit	إيداع دمج
merged	مدموج
merge [to]	دمج [في]
metadata	بيانات وصفية
mirror [copy]	نسخة مقابلة
mirror [server]	خادوم مرآة(؟)
model	نموذج
modify	تعديل
mount, mounted	ضم، مضموم
namespace	مساحة أسماء
nonlinear	لاخطي (كلمة واحدة، بغير مسافة)
normalize, normalization	استنظام

open source	مصادر مفتوحة
output	ناتج
pack	عُلبة (ج: عُلب)
packfile	ملف عُلبة (ج: ملفات عُلبة)
pager	[برنامج] عارض
patch	رُقعة (ج: رُقع)، ترقيع
pipe (n)	قناة [يونكسية]
platform	منصة
pointer	إشارة (متحد مع ref)
policy	سياسة
port forwarding	توجيه منفذ
private	خصوصي (وليس «خاص»)
prompt	مُوَجِّه (ج: مُوَجِّهات)
proprietary	احتكاري
protocol	ميفاق (ج: موافيق)، بروتوكول
public	عمومي (وليس «عام»)

pull	جذب
pull request	طلب جذب، طلب دمج
push	دفع
read access	إذن القراءة
read/write access	إذن التحرير
rebase [on]	إعادة تأسيس [على]
reflog	سجل الإشارات
ref, reference	إشارة (انظر التفصيل في الملحق الآخر: إشارة الرأس وإشارات الكائنات)
release	إصدار، إصدارة
remote	[مستودع] بعيد
remote-tracking branch	فرع متعقب لبعيد
remove	إزالة
rename	تغيير اسم
repo, repository	مستودع
reset	إرجاع

restore	استعادة
revert	نقض
revision	مراجعة
revision control	التحكم في المراجعات
rollback	إعادة(؟)
root	جذر، جذري
root directory	مجلد جذر
script	بُرنِمج (ج: بُرنِمجات)
scripting	برمجة
section (in a Git project)	قسم
section (in the book)	فصل
serve	قدّم، يقدِّم، تقديم، يتيح، إتاحة
server	خادوم (ج: خواديم)
setup (arrangement)	ترتيب (ج: ترتيبات)، تركيب، تكوين
setup (installation)	تثبيت
set up, setting	إعداد

shell	صَدَفة، طرفية
shell access	وصول صَدَفي
sign [sth]	توقيع [شيء]
single point of failure	نقطة انهيار حاسمة
snapshot	لقطة
stage, staging, staged	يؤهل، تأهيل، مؤهل
staging area	منطقة التأهيل
standard	معيار
switch (v)	انتقال
system	نظام (ج: أنظمة)
tab (char)	مسافة جدولة
tab-completion	إكمال بزر الجدولة
tab (key)	زر الجدولة
tab (UI)	تبويب
tag, annotated	وسم معنون
tag, lightweight	وسم خفيف

team	فِرقة (ج: فِرَق) (؟)
teammate	زميل (ج: زملاء)
terminal [emulator]	طرفية
timestamp	خَتم زمني
track, tracking	تعقب، متعقِّب (انظر التفصيل في الملحق الآخر: التعقب والمتابعة: tracking)
transaction	معاملة
two factor authentication, 2FA	استيثاق ثنائي
undo	تراجع
untracked	غير متعقَّب
upload	رفع
upstream	منبع (انظر أيضا downstream)
verify [sth]	توثيق [شيء]، تحقق [شيء] (بغير «من»)
version	نسخة
version control	إدارة النسخ
version control system, VCS	نظام إدارة نسخ

versioned	مراقَب [للتغييرات]
virtual private network, VPN	شبكة وهمية خاصة
web	وب
workflow	[أسلوب] سير عمل
working copy	نسخة عمل
working tree	شجرة العمل

الملحق الخامس: المصطلحات والمفاهيم باللغتين

يضم هذا الملحق أوامر جيت مثل الملحق الثالث، ولكنه أيضا يضم مفاهيم جيت وأنظمة إدارة النسخ الموزعة، ويتناولها جميعًا بشرح موجز مع توضيح أسمائها باللغتين وأسباب هذه الأسماء، ويضم ملاحظات متفرقة للمترجم.

الإيداع والسحب

الغرض الأساسي في أنظمة إدارة النسخ هو أنك تحفظ النسخة الحالية من مجلد العمل كما تحفظ الأموال في المصرف، ثم عندما تحتاجها تأخذها منه. ولكنك لا تأخذها للأبد، بل «تستلفها» مؤقتا مثلما تستلف كتبا من المكتبة العامة.

فعملية السحب هذه لها اسم واحد شائع: check out. أما عملية الحفظ فلها اسمان في الأنظمة المختلفة: check in أو commit.

انظر أيضا: https://stackoverflow.com/q/12510574

وانظر: https://www.noureddin.dev/ysmu/link/commit

الدفع والجذب والاستحضار

لأن «سحب» محجوز لـcheck out، فكان علينا الإتيان بلفظ آخر ليعني pull.

العملية المرافقة لـpull هي push، وكلاهما يعرفان بالدفع والجذب، فكان هذان اللفظان مناسبين.

ولكن عملية الجذب pull عمليتان في الحقيقة، أولهما fetch، لإحضار الكائنات (objects) والإشارات (refs) من المستودع البعيد. فكان اللفظ المناسب لها تنزيل أو إحضار، فاخترت استحضار (طلب الحضور) لتمييزها عن الكلمة العامة «إحضار».

الإرجاع والاستعادة والنقض

يفرّق جيت بين الإرجاع reset، والاستعادة restore، والنقض revert، وطبعا إعادة التأسيس rebase.

وقد حاولت أن أجعل أسماءهم العربية متباعدة، تقليلا للخلط الأكيد بينهم.

والخلط بينهم وارد حتى إن دليل ("manpage") جت نفسه يخصص فصلا للفرق بينهم، ثم يشير إلى هذا الفصل في دليل كل أمر منهم. فنجد في دليل `git` فصلا بعنوان "Reset, restore and revert"، هذه ترجمته:

> في جت ثلاثة أوامر بأسماء متشابهة: الإرجاع `git reset`، والاستعادة `restore`، والنقض `git revert`.
>
> - أمر النقض `git revert` يصنع إيداعا جديدا ينقض (يعكس) فيه التعديلات التي قدّمتها إيداعات سابقة معينة.
> - أمر الاستعادة `git restore` يستعيد ملفات في شجرة العمل من الفهرس (منطقة التأهيل) أو من إيداع سابق. هذا الأمر لا يحدّث الفرع الحالي. يمكن استعمال هذا الأمر كذلك لاستعادة ملفات في الدليل من إيداع سابق.
> - أمر الإرجاع `git reset` يحدّث الفرع الحالي بتحريك رأس الفرع ليضيف أو يزيل إيداعات منه. هذه العملية تغيّر تاريخ الإيداعات.
>
>   يمكن استعمال أمر الإرجاع `git reset` لاستعادة الفهرس، وهو استعمال يشترك فيه مع أمر الاستعادة `git restore`.

ثم يذكر هذا في دليل أمر النقض `git revert` :

> لاحظ: يُستعمل أمر النقض `git revert` لتسجيل إيداعات جديدة تنقض (تعكس) تأثير إيداعات سابقة معينة (غالبا إيداعات خاطئة). إن أردت نبذ التعديلات غير المؤهلة جميعا من مجلد العمل، فانظر أمر الإرجاع `git reset`، تحديدا الخيار `--hard`. إذا أردت استخلاص ملفات معينة من إيداع سابق، فانظر أمر الاستعادة `git restore`، تحديدا الخيار `--source`. كن حذرا في استعمالك هذين البديلين، فكلاهما يلغي التعديلات غير المؤهلة التي في مجلد عملك.

لم أسمِّ أي أمر منهم باسم «إعادة» خشية خلطه على الناس مع «استعادة»، وكذلك لم أسمِّ أمرًا «تراجع» خشية خلطه مع «إرجاع»، بل آثرت جذرا مختلفا لكلٍ منهم.

أسماء أوامر جت

نسمي أوامر جت، وخصوصا الأوامر العلوية (انظر ~~الأوامر السفلية والعلوية (السباكة والبورسلين)~~)، بأسماء عربية، فمثلا أمر `git commit` اسمه أمر الإيداع، وأمر `git branch` اسم أمر التفريع، وهكذا.

وأذكر أسماء الأوامر هنا مع وجودها في الملحق الثالث لسببين: الأول أن الملحق الثالث غير منشور لأنه غير مكتمل بعد (فذلك ما في الجدول الأول)، والآخِر لأن من الأوامر المذكورة في الكتاب ما لم يُذكر في الملحق الثالث (بعد)، مثل الأوامر الجديدة كأمر الاستعادة `git restore` وأمر الانتقال `git switch`، ومثل بعض الأوامر السفلية مثل أمر سرد الملفات `git ls-files` وأمر استعراض الملف `git cat-file` (وذلك ما في الجدول الآخِر).

add	إضافة
apply	تطبيق

archive	ضغط
bisect	تفتيش
blame	عتاب
branch	الفرع
checkout	سحب
cherry-pick	اصطفاء
clean	تنظيف
clone	استنساخ
commit	إيداع
config	تهيئة
describe	وصف
diff	الفرق
difftool	أداة الفرق
fast-import	استيراد سريع
fetch	استحضار
format-patch	تنسيق رقعة

fsck	فحص نظام الملفات
gc	جامع المهملات
grep	بحث
help	مساعدة
init	ابتداء
log	السجل
merge	دمج
mergetool	أداة الدمج
mv	نقل
pull	جذب
push	دفع
rebase	إعادة تأسيس
reflog	سجل الإشارات
remote	البعيد
request-pull	طلب جذب
reset	استعادة

revert	إرجاع
rm	إزالة
send-email	أرسل بريد
shortlog	السجل الموجز
show	إظهار
stash	تخبئة
status	الحالة
submodule	وحدة فرعية
tag	وسم

من الأوامر غير المذكورة في الملحق الثالث:

cat-file	استعراض الملف
daemon	العفريت
ls-files	سرد الملفات
ls-remote	سرد البُعداء
ls-tree	سرد الأشجار

restore	استعادة
switch	انتقال

إشارة الرأس وإشارات الكائنات

يقول جيت أحيانا pointer وأحيانا ref أو reference، لكن خلافا لبعض لغات البرمجة، هذه جميعا تعني الشيء نفسه(https://github.com/progit/progit2/issues/1460) ، وهو الشيء الذي «يُشير» إلى كائن أو شيء آخر.

عند التحدث عن الفأرة مثلا، فكلمة «مؤشر» (pointer أو cursor) صحيحة لأنها اسم الفاعل من الفعل «يؤشر» أيْ ذلك الشيء الذي «يضع إشارة». أما عند التحدث عن البرمجة، فإن pointer **لا تعني** «مؤشر»، لإن الpointer لا يضع إشارة على شيء، بل يشير إلى شيء، فاللفظ الصحيح هو «مُشِير». وهو اللفظ المستخدم في لغة كلمات.

أما كلمة reference، ففي سياق الكتب تعني الكتاب الذي نرجع إليه للبحث عن معلومة، فترجمته إلى «مرجع» عندئذٍ صحيحة. لكن في سياق البرمجة، الreference لا يَرجع إلى شيء، بل يُرجعنا نحن أو يُحيلنا إلى شيء (refer to)، فترجمته إلى «مُحيل» أفضل.

ولكن المشير والمحيل اسمان لمسمى واحد في جت، فالأفضل توحيد الاسم.

استعملت «مشيرًا» في البدء، ثم وجدت أن الأقرب في الاستعمال هو «إشارة»، فهي ما استعملت في الكتاب.

ولمنع اللبس، أقول «إشارة الرأس» غالب الوقت للفظ HEAD.

ولكن جت يفرق بين head وHEAD؛ انظر `man gitglossary` أو على الشابكة (https://git-scm.com/docs/gitglossary#Documentation/gitglossary.txt-aiddefhashahash) .

التعقب والمتابعة: tracking

يستعمل الفعل tracking في المشاريع البرمجية استعمالين رئيسيين، ويترجم بلفظ مختلف حسب استعماله:

١. «المتابعة»، وهي أن يتابع الإنسانُ العللَ (bugs) والمسائل (issues) والأهداف (milestones) وغير ذلك. ومنها متابع العلل (bug tracker) أو متابع المسائل (issue tracker).

٢. «التعقب»، وهي (١) أن يتعقب جت ملفًا، أي أن يتابع تغييراته ويرصدها ويسجلها، (٢) وأن تجعل جت يتعقب فرعًا بعيدًا، أن أي يجعل جت فرعا محليا (يسمى «فرعا متعقِّبا») أو إشارة محلية (تسمى «فرعا متعقِّبا لبعيد») تتابع التغييرات الحادثة في الفرع البعيد. (انظر الفصل التالي لتفصيل هذا الأمر.)

الفروع البعيدة والفروع المتعقّبة لبعيد والفروع المتعقّبة

يفرق جت وكتاب احترف جت بين الفروع المتعقبة (tracking branch) والفروع المتعقبة لبعيد (remote-tracking branch)؛ انظر تعقب الفروع.

لكن لا يبدو أن الكتاب يفرق بين الفروع البعيدة (remote branch) والفروع المتعقبة لبعيد (remote-tracking branch)، إلا قليلا، فكلاهما نظرتان للشيء نفسه: الفرع `origin/master` مثلا هو الفرع الرئيس في المستودع البعيد، فهو في مستودعي المحلي فرع متعقب لبعيد، لكنه يشير إلى الفرع البعيد نفسه. فيجوز قول «الفرع البعيد» للفرع المتعقب لبعيد من باب المجاز المرسل أغلب الوقت. وهذا استعمال الكتاب إلا قليلا. (من أمثلة هذا القليل: حذف فروع بعيدة.)

إذنا القراءة والتحرير

أقترح تعريب "read-only" بـ«القراءة»، و"read/write" بـ«التحرير»، والاسم "access" المرتبط بهما غالبا بـ«إذن» (وجمعه «أذون»).

واخترت هذين الفعلين لأنهما متعديان بغير حرف جر، فلا نحتاج أن نقول «الكتابة على/إلى/في المستودع»، بل نقول «تحرير المستودع» بغير وسيط. ومثله في القراءة.

و«التحرير» أقوى من «الكتابة» المجردة، فهو يعني التعديل والتقويم، وحديثا يشمل المراجعة والإنشاء. فمعناه واضح فيه أنه "read/write"، فغالبا لا يوجد إذن «كتابة فقط»، فالأنسب كلمة تدل على القراءة والكتابة معا.

وقد استعملت وقتًا قصيرا كلمة «اطلاع» تعريبًا لـ"read"، لكني عدلت عنها إلى «قراءة» لحاجة «اطلاع» إلى حرف جر، وعدم وجود منفعة من «اطلاع» (مثل شمول معنى «تحرير»، فصار أنفع من «كتابة»)، ولشهرة «قراءة».

تعريب كلمة كود

لهذه الكلمة معانٍ كثيرة في غير البرمجة، منها رمز ورقم ومعيار وغيرهم.

ومعناها في البرمجة شديد الاتساع ويستعصي على النقل إلى لغة أخرى، فأبى أكثر الناس إلا أن يأخذوا هذه الكلمة بكل معانيها البرمجية. فمنهم من نقلها صوتيا («كود») ومنهم من أتى بكلمة من جذر الكلمة الإنجليزية الأصلية («رمز») في لغتهم، فقال السوريون «رِماز» (على وزن «كتاب») وقال الصينيون 代码 (رمز تبديل(؟)).

ولا أرى في هذا خيرا كثيرا، فهذا لفظ أعجمي في أصله ومعناه واستعماله، وليس فيه من العربية إلا حرفه.

ففي هذا الكتاب نحاول تعريب هذه الكلمة في مواضعها المختلفة تعريبا يفهمه العربي بغير شرح وبغير معرفة الاستعمال الإنجليزي وبغير معرفة الاصطلاح السوري.

وصعوبة الأمر أن هذه الكلمة شديدة العموم، فنحتاج إلى تخصيصها في كل سياق قبل تعريبها، فغالبا نعتبرها صفة (=«برمجية») لاسم محذوف، ونرد هذا الاسم عند الترجمة.

فهذه أشهر تعريباتها حسب السياق:

- «مصدر برمجي» (= "source code") أو «مصدر» فحسب ←
  - «قاعدة المصدر» = "code base" (أو «مصدر برمجي» أيضا)
- «مشروع برمجي» (مثل العبارة: "hosting your code" وما في معناها)
- «برمجيات» ←
  - «محرر برمجيات» = "code editor"
- «تعليمات برمجية» ←
  - «ثغرة تنفيذ تعليمات برمجية عشوائية» = "arbitrary code execution vulnerability"
  - «تعليمات برمجية خبيثة» = "malicious code"

ولم نفرغ من هذه الكلمة بعد.

إصلاحات لغوية عامة ونصائح أسلوبية

بعض هذه النصائح إصلاحات حقيقية، وبعضها ليست سوى اقتراحات لاتساق الترجمة.

- قل «يستعمل» أغلب الوقت، ولا تقل «يستخدم» إلا للضرورة، فالاستخدام يكون للعاقل. (وتبقى "user" «مستخدما».) ويستثنى من ذلك ما يقدّم خدمة، فنستخدم جت‌هب، ونستخدم الخواديم، ونستخدم جت نفسه تشبيهًا له بالعاقل. ولا بأس من استعمال «استعمال» مع أيٍّ منهم. ولكن قلل من كليهما، لأن ~~استعمال~~ هذا الفعل يكثر في الإنجليزية المعاصرة بغير حاجة. ومثله الفعل «يستطيع».
- لا تقل «نفس الشيء» وقل «الشيء نفسه». (أو «شيء واحد» أو «الشيء الواحد» عند إرادة الإبهام، مثل «يمكنكم الآن التعاون في مشروع واحد» أو «ستبيت كل مشاريعك البرمجية على جهاز واحد».)
- لا تقل «بسيط» إلا عندما تعني «غير معقد»، وقلل منها عموما.

- قلل من «فقط» واستعمل الاستثناء أو «إنما» متى أمكن.

- قل «يبقى» و«يظل» و«لا يزال» و«ما زال»، لكن لا تقل «لا زال»، فاستعمال «لا» مع الماضي يعني الدعاء، مثل «لا أراك الله مكروها».

- لا تقل «استبدل» لشيوع الخطأ فيه، وقل «أبدل القديم بالجديد»، فلا خلاف فيه، أو «أبدل من القديم الجديدَ»، أو ائت بفعل من جذر آخر مثل «يحل محل». (الصحاح في مادة بدل: «والأَبْدالُ: قومٌ من الصالحين لا تخلُو الدنيا منهم، إذا مات واحدٌ أَبْدَلَ الله مكانَهُ بآخر.»)

- لا تستعمل حرف الجر الكاف إلا للتشبيه، وعلامة ذلك استقامة المعنى وثبوته عندما تبدلها بـ«مثل»، وإلا فغيّر تركيب الجملة واستعمل شيئا غير الجر بالكاف، كالتمييز والحال والمفعول به.

- لا تقل «بداية»، فهي عامية، وفصيحها «بَداءة» لكنه غير مألوف، فقل «بَدء» أو «ابتداء» أو «أول». (قال فيها المصباح المنير: «و"البِدَايَةُ" بالياء مكان الهمز عامي نصّ عليه ابن بري وجماعة». وقال العباب الزاخر: «وقَوْلُ العامَّةِ: البِدايَةُ - مؤازاةً للنِّهاية: لَحْنٌ، ولا تُقاس على الغَدَايا والعَشايا؛ فإنَّها مَسْموعةٌ بخلاف البِداية».)

- لا تقل «تحقّق من [شيء أو فِعل أو أن تفعل]»؛ قل «تحقّق ...» بغير «من»، أو قل «تفقّد ...».

- لا تقل «كل ما عليك هو»؛ إنما قل «ليس عليك سوى/إلا/غير».

- لا تقل «لا داعٍ لـ...»، بل قل «لا داعيَ إلى ...»، بالياء وبـ«إلى».

- لا تبدأ جملة فرعية بـ«مما»، فهي غالبا خاطئة. وعلامة ذلك اختلال المعنى إذا وضعت مكانها «مِن الذي» أو اسم موصول آخر. أمثلة:
  - «يتيح جت أيضا خيارا للحالة الموجزة، مما يتيح لك رؤية تعديلاتك بإيجاز» ← «...، لترى ...»
  - «إضافة [شيء] تجعل جت [يفعل شيئا]، مما يتيح لك تخطي مرحلة الإضافة» ← «...، لتتخطى ...»
  - «لأن طريقة التفريع في جت خفيفة [...]، مما يجعل إنشاء فرع ...» ← «...، فتجعل ...»
  - «ثم تستطيع حذف الفرعين [...]، مما يجعل تاريخك ...» ← «...، فيصير تاريخك ...»

- «تذكر أنه غير مستوثَق، مما يعني أن أي شيء تتيحه ...» ← «...، فأي شيء ...»
- «وقتئذٍ ستبيت كل مشاريعك البرمجية على جهاز واحد، مما يزيد من احتمال فقد البيانات فقدا كارثيا.» ← «...، وهذا يُزيد احتمال ...»
- «لم تقترب حاله مما صار عليه اليوم» ← صحيحة
- «فبدلا مما رأينا، ...» ← صحيحة

- «مِيزة»/«مِيزات» تعني الاختلاف (التمايز، وفصل الشيء من الشيء) ولا تعني التفضيل. فإذا أردت معنى الفضل (التمام والكمال)، فقل «مَزِيَّة»/«مَزَايَا». فترجم "feature" بـ«ميزة» وترجم "advantage" بـ«مزية».
- قدّم الفعل على فاعله ما لم يسبب ذلك خلطا على القارئ.
- انظر موارد معجم يسمو(/https://www.noureddin.dev/ysmu/notes) ، وبالأخص كتاب نحو إتقان الكتابة باللغة العربية (https://web.archive.org/web/20100914143217/http://www.reefnet.gov.sy/Arabic_Proficiency/Arabic_Proficiency_Index.htm).