SCM

75.52 Taller de Programación II - FIUBA

Pablo Rodriguez Manzi

Qué es?
Un SCM (Source Control Management System) o Revision Control System es un sistema que nos permite llevar registro de los cambios realizados a través del tiempo.

Para qué sirve? Qué nos ofrece?
- Repositorio
- Sincronización con el equipo
- Undo changes
- Track changes
- Branching and merging
- Tags

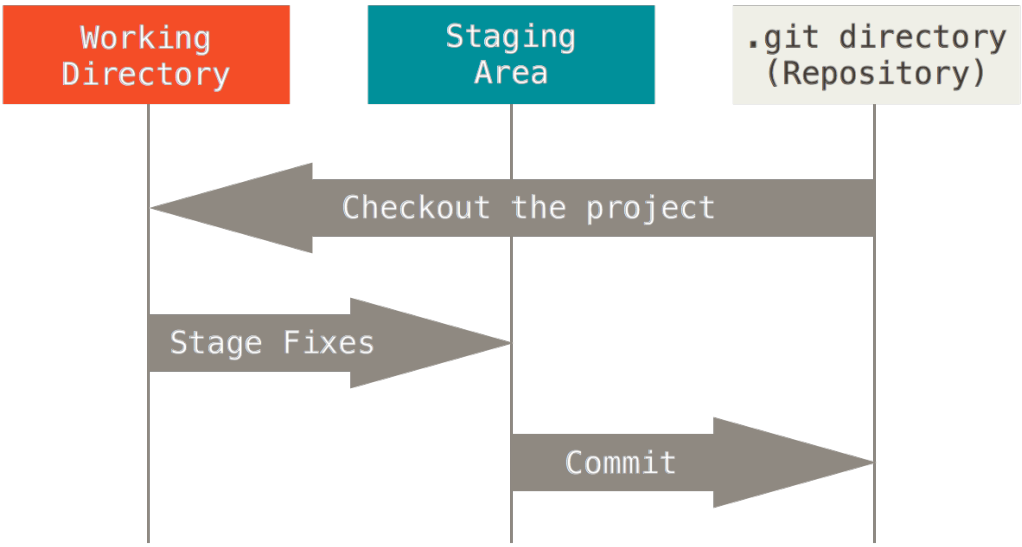
Centralizado vs Distribuido
Subversion vs Git o Mercurial
En un sistema centralizado existe un único repositorio en un servidor, el cual contiene toda la historia del mismo. Los clientes simplemente bajan la última versión del mismo y cada operación que realicen requieren un conexión con el servidor.
En un sistema distribuido no existe un único repositorio maestro, ya que cada cliente posee una copia exacta del repositorio.
Ventajas distribuido:
- Menos operaciones involucran una conexión a la red.
- Si se cae el server, uno puede seguir trabajando.
- Los datos están más seguros ya que no existe un único punto maestro.
- Es altamente escalable.

Subversion
- Es centralizado
- Almacena diferencias a un archivo base
- Cualquier cambio afecta a todos
- Pocos commits, pocos branches(es muy costoso)
- Poco escalable

Git in depth
- Desarrollado por Linus Torvalds en 2005 para desarrollar y mantener el kernel Linux
- Es distribuido
- Almacena snapshots, no diferencias
- Casi todas las operaciones se realizan de forma local

Tres estados

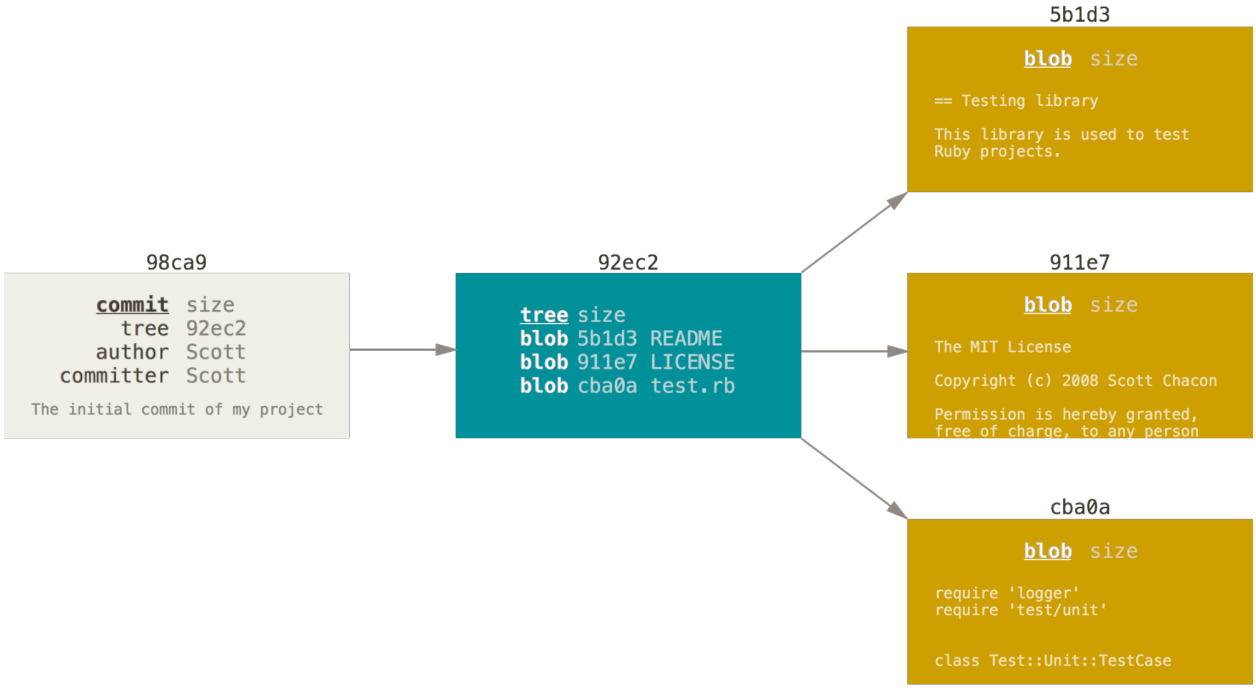
- *Working directory*: Es una copia de una versión particular de los archivos del repositorio para que usarlos o modificarlos. Es nuestro directorio de trabajo. Todos los archivos que modifiquemos residirán en este estado hasta que los agreguemos al index.
- *Index(staging area)*: Es el lugar dónde se colocan los archivos que queremos commitear al repositorio. Contiene snapshots de estos archivos.
- *Repositorio*: Contiene toda la historia de nuestros archivos. Las versiones aquí están seguras. Al hacer un commit, se toman los snapshots del index, y se los incluye en una nueva versión en el repositorio.



Cómo se almacena la información?

Toda la información del repositorio se almacena en el directorio .git
Existen 4 tipos de objetos que se almacenan en .git/objects

- Blob: Es el tipo de objeto más importante. Cada blob contiene a un snapshot de un archivo comprimido. Para almacenarlos se les computa el hash SHA-1 al contenido del archivo, y se utilizan los dos primeros caracteres como nombre de carpeta y los 38 restantes como nombre de archivo.
- Tree: Es similar a un directorio. Puede contener referencias a blobs y a otros trees. Se almacenan utilizando un hash de forma similar a los blobs.
- Commit: Al hacer un commit se genera un nuevo objeto “commit”. El mismo posee una referencia a un tree que refleja el estado del index al momento de realizar el commit, una referencia a un padre que es el commit anterior, un autor, un mensaje, fecha y hora.
- Tag: Dado que para referirse a los objetos hay que utilizar el hash, se le puede dar nombres a los objetos “commit” a través de Tags.



The Index

Qué hay en el index? Qué cambió?

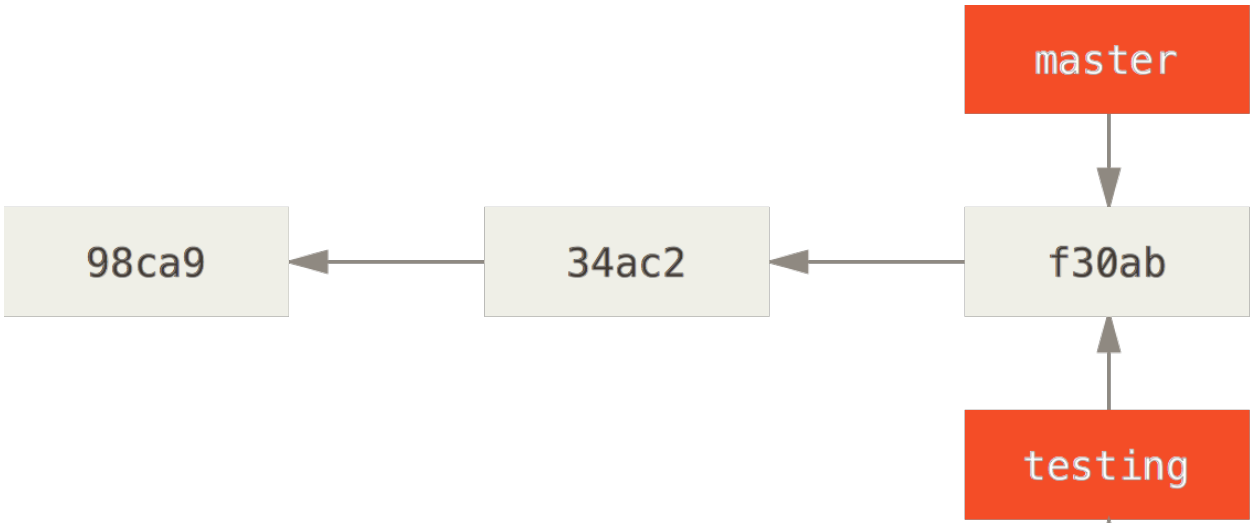
- git ls-files: Lista los archivos que se encuentren en el index
- git status: Muestra el estado del working directory
- git diff: Muestra las diferencias entre el working directory y el index. (Con --cached muestra entre el index y el último commit, y con HEAD muestra las diff entre el wd y el último commit).

Commits

- git commit: Se utiliza para crear un nuevo “commit”. Al ejecutarse se genera un nuevo tree que refleja el estado del index al momento y un nuevo commit con la referencia a dicho tree, a el/los padre/s, autor, fecha, hora y comentario.
- git log: Muestra la historia de todos los commits
- git show: Muestra la información de un commit y las diferencias introducidas.

Branching

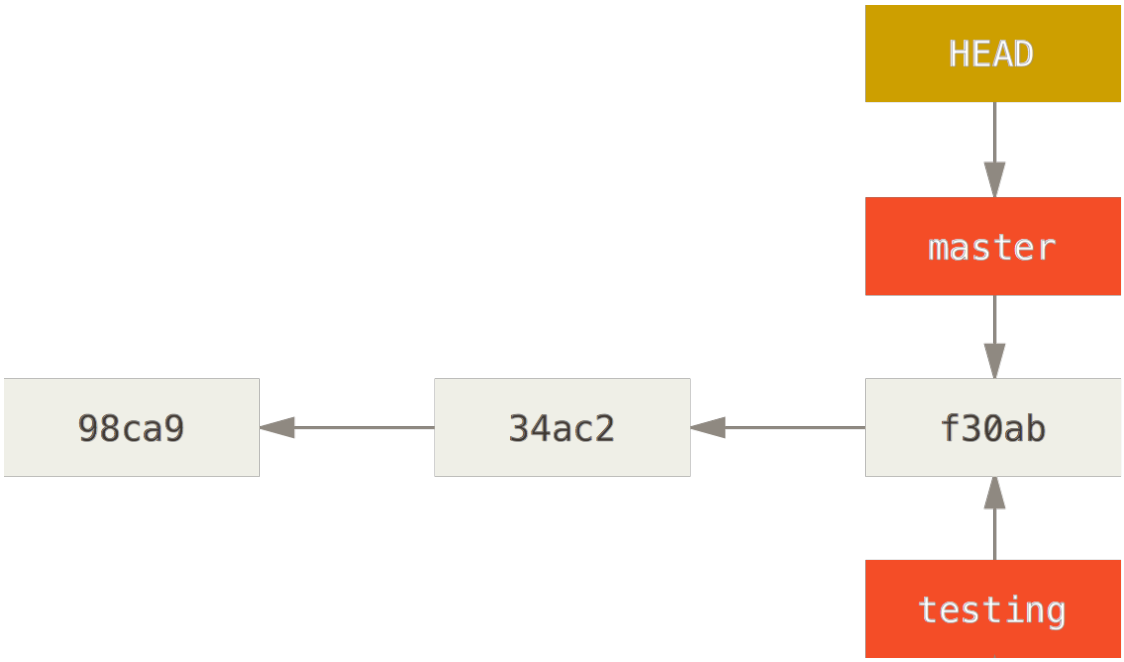
En git crear nuevos branches es realmente simple y eficiente. Un branch es simplemente un puntero a un commit. Por defecto, cuando iniciamos un repositorio se crea automáticamente el branch “master”. Para crear nuevos branches



Para crear un nuevo branch: git branch branchName
Para listar los branches: git branch -v

HEAD

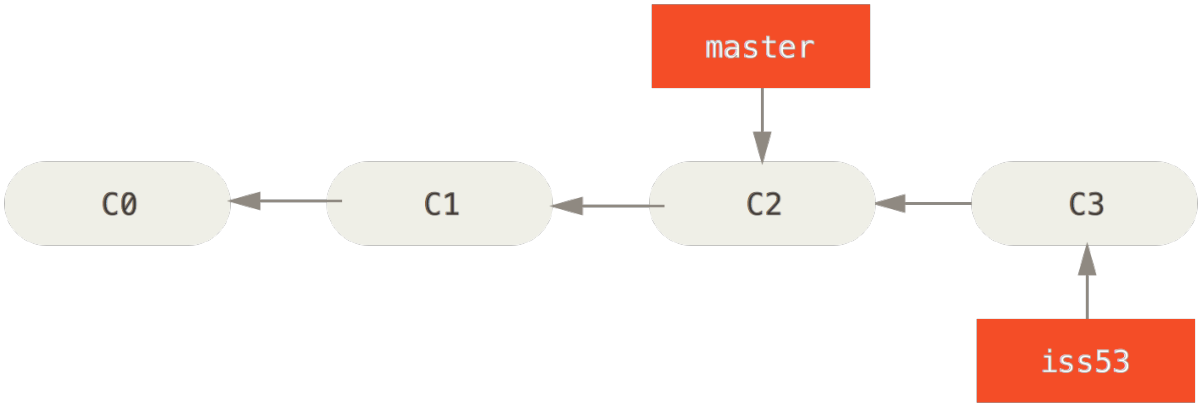
HEAD es una referencia a un branch o a un commit particular. Indica el estado actual del working directory.



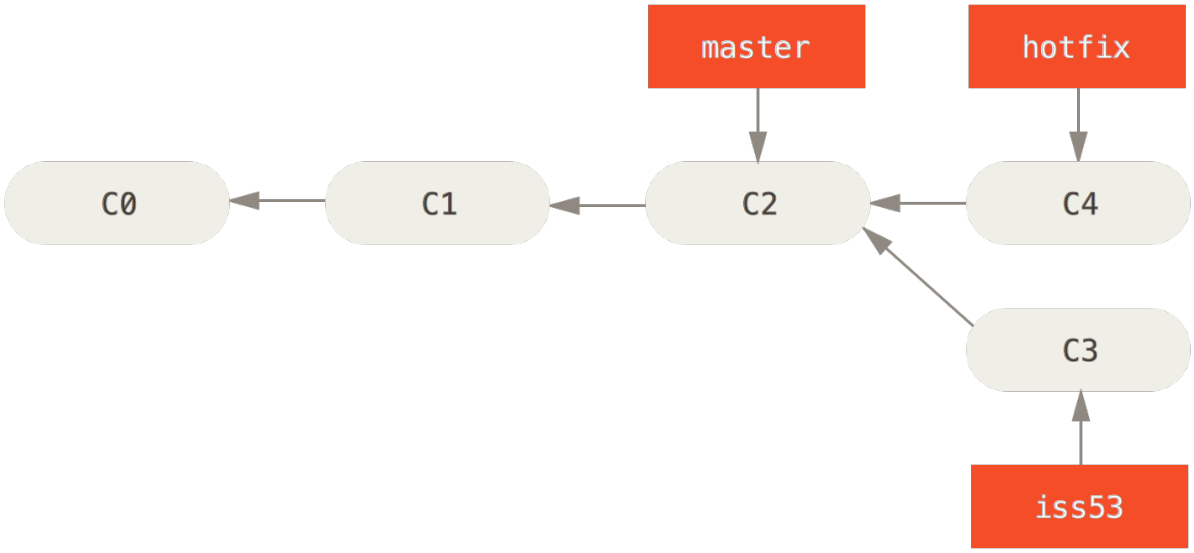
El valor de HEAD se encuentra almacenado en el archivo HEAD en el directorio .git.
El comando “git checkout” modifica su valor, lo que nos permite facilmente cambiar de branch, o volver al estado de un commit anterior.
Utilizando git “log --decorate” muestra a qué commit apuntan los branches y HEAD.

Merge

Supongamos que en un punto de nuestro trabajo decidimos crear un nuevo branch para solucionar un issue específico, el repositorio se verá de la sig forma:

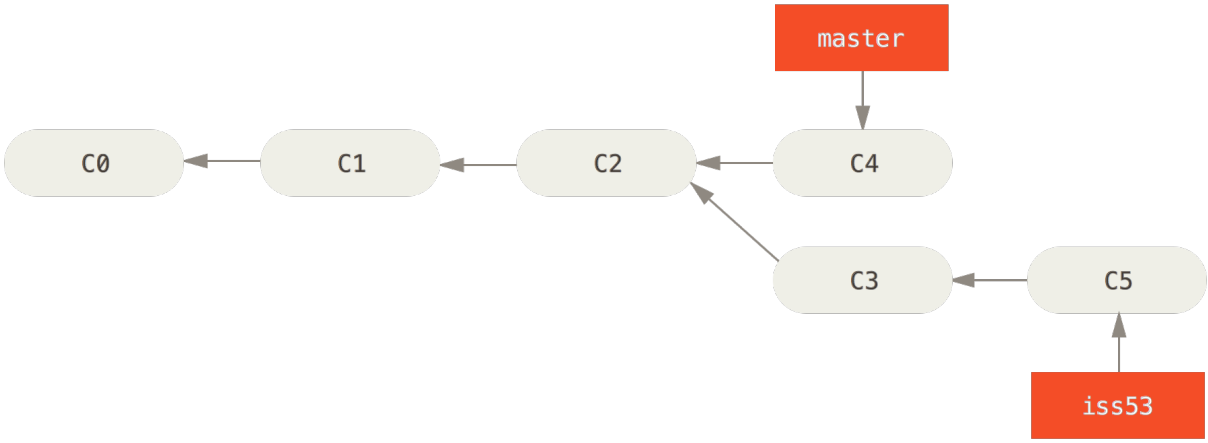


Supongamos que mientras estamos trabajando en este issue debemos solucionar inmediatamente un bug, por lo tanto dejamos este branch y creamos otro branch para solucionar dicho bug:



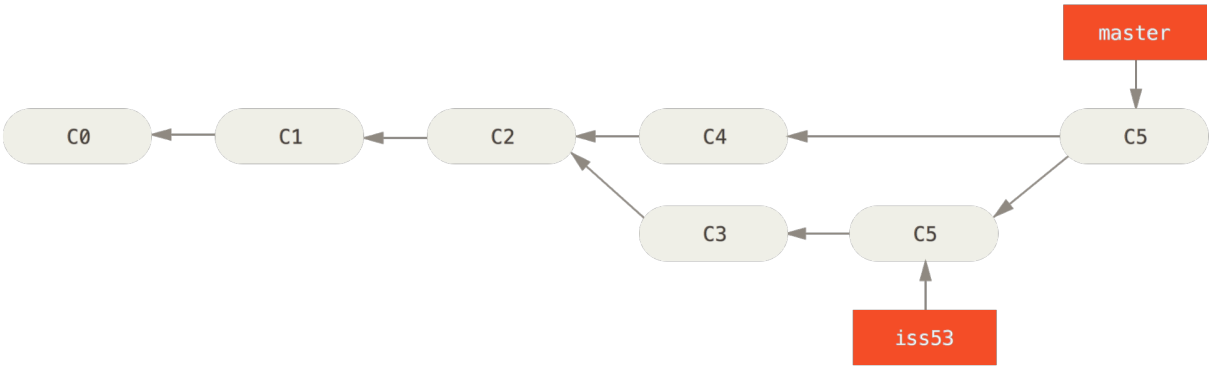
En este momento la historia de nuestro proyecto a divergido.
Una vez que nuestro bug se encuentra solucionado queremos incorporar dichos cambios a la rama master, para eso hacemos:
-git checkout master
-git merge hotfix

Este es el tipo de merge más simple de todos, porque lo único que se debe hacer es adelantar el puntero de master al commit C4, a esto se lo denomina “fast-foward”:



Ahora podemos seguir trabajando en nuestro issue, una vez terminado queremos mergear estos cambios en master. Para hacerlo procedemos de la misma forma:
git checkout master
git merge iss53

Pero en este caso lo que se debe hacer es diferente, git debe mergear la snapshot C5 con la C4, para esto genera un nuevo objeto C6 que contiene el resultado del merge.

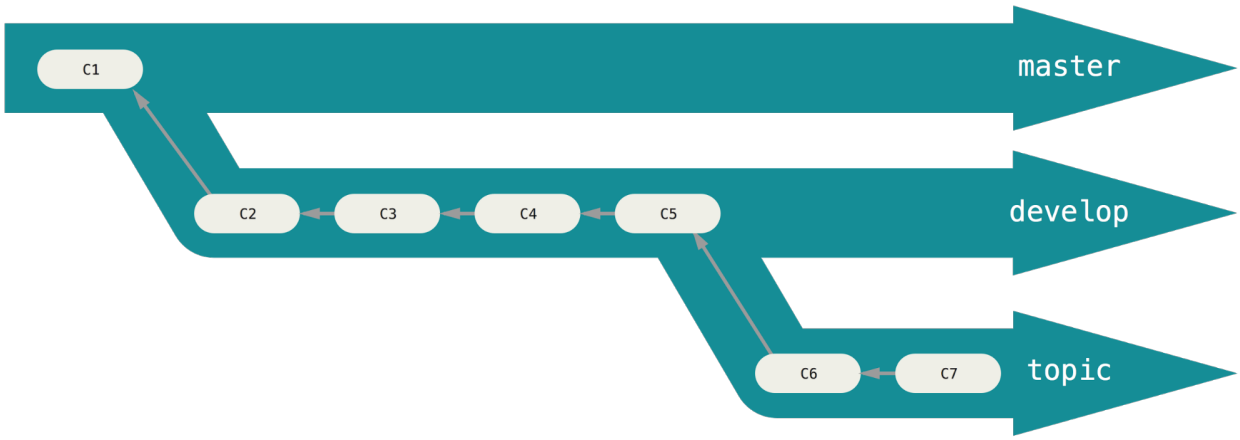


Merge Conflicts

Si al mergear dos branches, fueron modificados en ambos las mismas partes de los mismos archivos, entonces git no podrá resolver el merge ya que no sabrá cuál es la versión correcta. En ese caso al realizar un “git merge” se le deja al usuario la tarea de resolver estos conflictos.
Una vez resueltos los conflictos se deberá realizar un commit.

Branching Strategy

Master: Es el branch principal de nuestro proyecto. Siempre debe contener una versión totalmente estable y testeada del mismo.
Develop: Se utiliza para reunir el trabajo de los “Topic branches”. No se trabaja directamente en él. La idea es que se mantenga lo más estable posible, una vez alcanzado un estado realmente estable, se lo puede mergear con master.
Topic Branches: Dado que en git crear y mergear branches es algo realmente fácil, resulta realmente útil crear un branch por cada feature en el que estemos trabajando. Son branches de corta duración, una vez que el feature está terminado y testeado se lo puedo mergear con Develop y eliminar el branch. Esto nos permite switchear fácilmente si de pronto nos vemos obligados a trabajar en distintas partes al mismo tiempo. Además es muy fácil descartar cambios en caso de que no nos convenza los que estamos haciendo.



Remotes

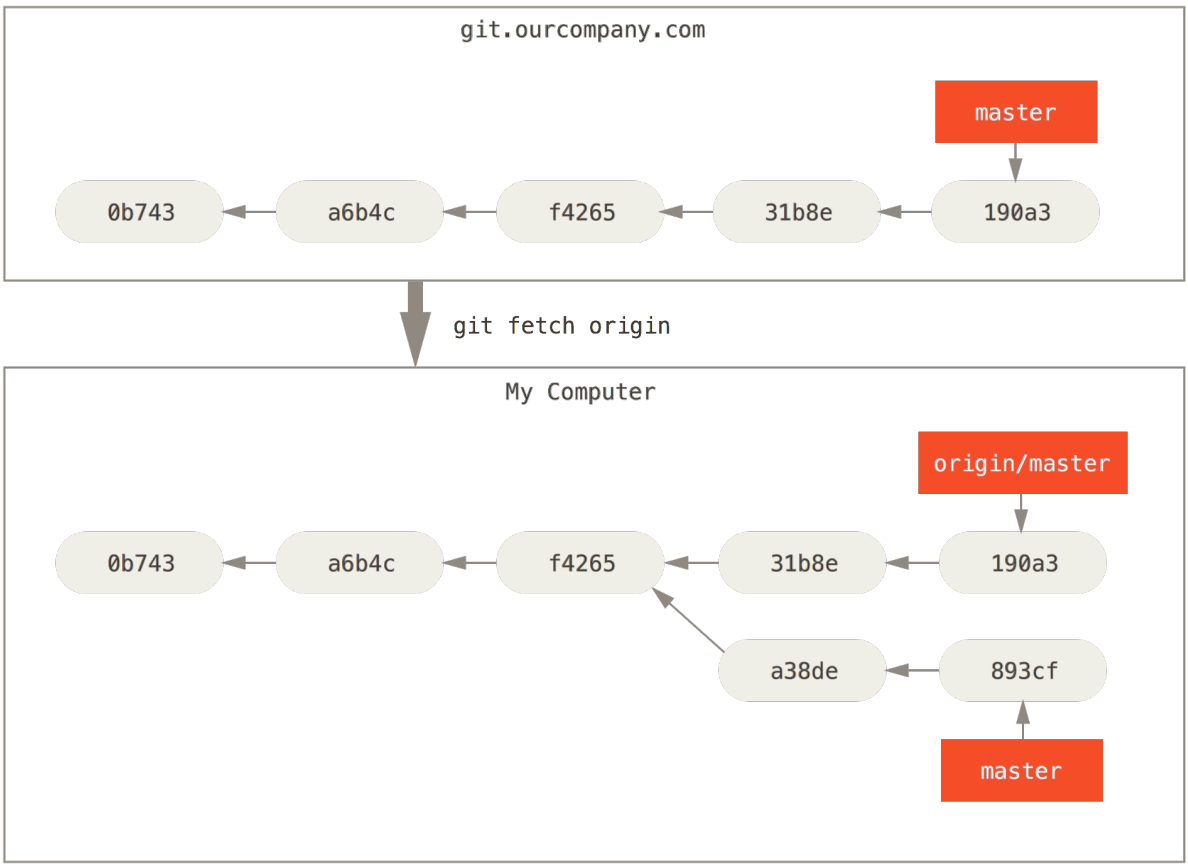
Hasta ahora todo lo que hemos hecho ha sido en nuestro repositorio local. Los “remotes” nos permiten colaborar con otros repositorios.

- git clone: Clona el respositorio indicado, creando una copia local. Agrega automáticamente como remote al repositorio fuente llamado “origin”
- git remote -v: Lista los remotes
- git remote add: Agrega un nuevo remote para colaborar.

Working with Remotes

- git fetch: Trae los últimos cambios del remote y branch indicados
- git push: Publica los cambios en el remote y branch indicados.

Al trabajar con Remotes, los branches provenientes de Remotes llevan la nomenclatura del remote previo al nombre del branch para diferenciarlos de los branches del repositorio local.



Tracking Branches

Se pueden configurar los branches locales para automáticamente “trackear” a determinados branches de los distintos remotes. Esto llevará la cuenta de las diferencias entre los branches locales y los branches trackeados. Por defecto al clonar un repositorio el branch master trackeará al branch origin/master

- git checkout --track origin/branchName: Crea un branch que trackea a origin branchName.
- git branch -vv: Muestra los branches trackeados y su estado.
- git pull: Realiza un fetch y a continuación un merge del branch trackeado.

Workflows

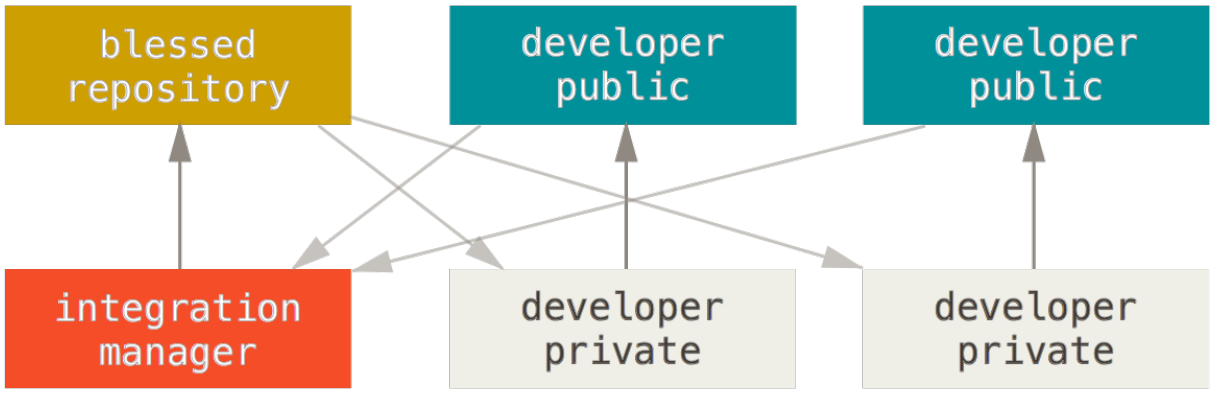
A continuación veremos algunos flujos de trabajo:

Centralizado

En este flujo de trabajo existe un único repositorio público al cual todos los desarrolladores tienen acceso de lectura y escritura, y realizan todos los cambios directamente en él. Es el flujo de trabajo típico para sistemas no distribuidos como Subversion. Sirve para equipos pequeños, pero no es escalable.

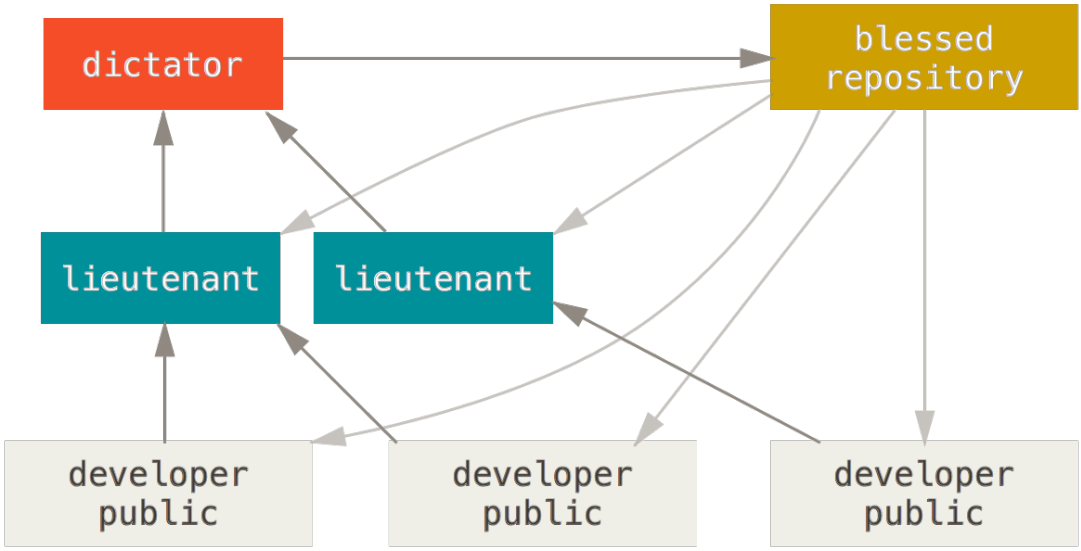
Integration-Manager Workflow

En este caso cada developer tiene su propio repositorio público (además del local) en el cuál publica sus cambios. A su vez existe un repositorio maestro el cuál contiene la versión “oficial” del proyecto. Cada vez que un desarrollador quiere introducir cambios, publica en su repositorio y le avisa al Integration Manager el cual descarga los cambios, chequea que todo esté correcto y los publica en el repositorio maestro.



Dictator and Lieutenants Workflow

Es similar al Integration-Manager, pero se utiliza para proyectos grandes (es el que utiliza Linux por ejemplo). En vez de existir un único Integration Manager, existen varios Lieutenants que revisan los cambios de los desarrolladores y los reportan al Dictator, el cual finalmente publica los mismo en el repositorio maestro.



Forks y Pull Requests

Para facilitar la colaboración, GitHub y Bitbucket nos ofrecen el concepto de “Fork” de un repositorio. Es similar a un clone de un repositorio, pero en vez de realizarse una copia local, se realiza una copia a nuestro usuario público. De esta manera cuando realicemos modificaciones y queramos que las mismas se incluyan en el repositorio original, podemos realizar un “Pull request”. El administrador del repo original recibirá una notificación, podrá evaluar nuestros cambios, realizar una revisión de nuestro código y eventualmente mergear los mismo en el repo original.

Git Reset vs Git Checkout vs Git Revert

Estos comandos nos permiten volver a versiones anteriores y deshacer cambios:

- git revert *commit*: Genera un nuevo commit que revierte los cambios introducidos por el commit en particular. No altera la historia de nuestro proyecto.
- git checkout *branch/commit* : Mueve el puntero HEAD al branch o commit especificado. Actualiza el working directory para coincidir con dicho commit.
- git checkout *commit file*: No mueve el HEAD. Actualiza el *file* en el working directory para coincidir con el commit especificado.
- git reset *commit*:
    - --soft : Mueve el puntero del branch actual al commit especificado
    - --mixed: Además actualiza el index para coincidir con el commit especificado. (Se utiliza generalmente para sacar archivos de la staging area)
    - --hard: Además actualiza el working directory para coincidir con el commit especificado. Se utiliza para descartar cambios locales.
- git reset *commit file*: Actualiza el index para que el *file* coincida con el commit especificado.