Universidad de Santiago de Chile

Comunidad GNU/Linux USACH

Controlando Versiones con Git

Manual del Programador

Autor:
Daniel Gacitúa Vásquez
daniel.gacitua@usach.cl

14 de enero de 2016
v1.0

Índice

1. Introducción **2**
- 1.1. Cómo usar este Manual . . . 2
- 1.2. ¿Qué es Git? . . . 2
- 1.3. Requisitos para usar Git . . . 2
- 1.4. ¿Cómo funciona Git? . . . 3

2. Uso básico de Git **4**
- 2.1. Elementos básicos . . . 4
- 2.2. Zonas de trabajo . . . 5
- 2.3. Comandos esenciales . . . 5
  - 2.3.1. Crear proyectos Git . . . 5
  - 2.3.2. Crear y manejar commits . . . 6
  - 2.3.3. Estado actual del Repositorio . . . 6
  - 2.3.4. Comandos y Zonas de trabajo . . . 7
- 2.4. Consejos útiles . . . 7
- 2.5. Programación colaborativa con Git . . . 9

3. Instalando Git **10**
- 3.1. GNU/Linux . . . 10
- 3.2. Microsoft Windows . . . 11

4. Interacción con Servidores Públicos de Git **17**
- 4.1. BitBucket . . . 17
  - 4.1.1. Características . . . 17
  - 4.1.2. Interfaz web . . . 17
  - 4.1.3. Cargar Llaves SSH . . . 18
- 4.2. GitHub . . . 20
  - 4.2.1. Características . . . 20
  - 4.2.2. Interfaz web . . . 20
  - 4.2.3. Cargar Llaves SSH . . . 21

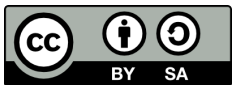
5. Uso avanzado de Git **23**
- 5.1. Merging: Fusionando cambios de código . . . 23
- 5.2. Rollbacking: Volviendo a estados anteriores del código . . . 25
- 5.3. Tagging: Identificando commits para publicación . . . 27
- 5.4. Stashing: Guardando código para el futuro . . . 28
- 5.5. Branching: El arte del código ramificado . . . 29

6. Miscelánea **31**
- 6.1. Acerca del autor . . . 31
- 6.2. Agradecimientos . . . 31
- 6.3. Ediciones al manual . . . 31

7. Bibliografía y Referencias **32**

1. Introducción

El objetivo de este documento es capacitar a su lector en el uso del sistema de control de versiones Git, tanto en ambientes de desarrollo GNU/Linux como Microsoft Windows. Este manual se distribuye bajo la licencia **Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)**.



Recuerda compartir este manual con todos tus colegas programadores. Un código versionado siempre es un código de mayor calidad, además de facilitar la edición por parte de múltiples usuarios y permitir la programación colaborativa.

Este documento es y será siempre un *work in progress*, con el fin de enseñar de la manera más simple y práctica posible los elementos propios de Git (que no son sencillos de aprender y asimilar). Cualquier sugerencia o mejora es bienvenida.

1.1. Cómo usar este Manual

Para asegurar la mejor experiencia de aprendizaje con este Manual, luego de terminar con la sección Introducción, continúa con Uso básico de Git (página 4), leéla bien hasta entenderla completamente. Luego procede a la sección Instalando Git (página 10) para instalar Git según tu sistema operativo y practicar los conceptos ya vistos. Si necesitas hostear en la nube tu proyecto, la sección Interacción con Servidores Públicos de Git (página 17) te proveerá de servidores públicos para dicha tarea. Finalmente, una vez que hayas entendido y sepas aplicar todo lo anterior, puedes ir a la sección Uso avanzado de Git (página 23) para conocer algunas técnicas que pueden ser útiles en proyectos con muchos programadores o muchas líneas de código. Por último, no olvides visitar la sección Miscelánea (página 31) para saber un poco más del Manual y su Autor.

1.2. ¿Qué es Git?

Git es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente. Hay algunos proyectos de mucha relevancia que ya usan Git, en particular, el grupo de programación del núcleo Linux[1].

1.3. Requisitos para usar Git

- Conocer los fundamentos básicos de algún lenguaje de programación (puede ser C, Java, Python o cualquier otro).
- Tener un ambiente de programación establecido en alguna distribución GNU/Linux o Microsoft Windows (esto quiere decir, contar con un IDE o Editor de Texto Plano para editar código).
- Contar con algún proyecto de código que pueda ser versionado con git (cualquier proyecto hecho en texto plano sirve).

1.4. ¿Cómo funciona Git?

Git se basa en el Modelo Cliente-Servidor para alojar repositorios de código (un repositorio viene a equivaler a un proyecto de código). El usuario (cliente git) tiene un repositorio local el cual aloja (dentro de su PC) diferentes snapshots (versiones) de su código. El usuario intercambia con un repositorio remoto (servidor git) sus snapshots para ser distribuidas a otros usuarios.

Ejemplos de servidores públicos de git son *BitBucket* y *GitHub*. Ejemplos de servidores implementables en un ambiente privado son *gitolite* y *gitosis*.

Los snapshots son conocidos en git como *commits*.

2. Uso básico de Git

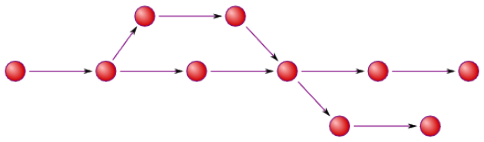
2.1. Elementos básicos

Los elementos básicos a considerar en cualquier proyecto versionado por Git son:

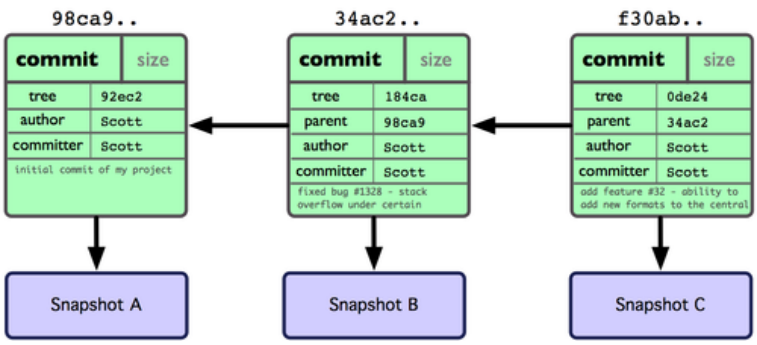
- El **commit**, es la unidad de trabajo básica de Git. Un commit es un snapshot o copia del estado actual de todo el código del proyecto.
- Las **zonas de trabajo**, son las encargadas de almacenar los diferentes commits realizados en el código, pueden ser zonas locales o remotas.
- Los **comandos de Git**, son acciones que nos permiten crear, modificar y mover de una zona a otra los commits generados por el usuario.

Respecto a los commits, estos se encadenan entre si como una lista enlazada a través del tiempo[2]:

En la imagen anterior, cada círculo representa un commit. Cabe destacar que en el progreso de los proyectos Git se pueden dar momentos en que la lista no es estrictamente lineal o tiene desvíos, como los siguientes:

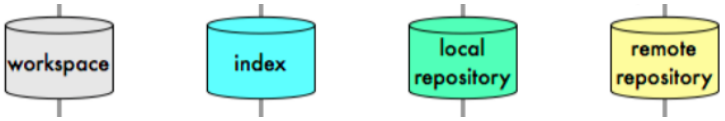


Además cada commit tiene un autor (usuario) y un timestamp (marca de tiempo) y un mensaje descriptivo. En base a éstos parámetros, se genera el *hash del commit*, que es un identificador alfanumérico único que nos permitirá referenciar dicho commit a futuro:



2.2. Zonas de trabajo

Las *zonas de trabajo* esenciales en Git son 4:



- El **Workspace** o **Directorio Local** es la carpeta local dentro de nuestro PC en la cual hacemos los cambios de nuestro código. Solo en esta zona de trabajo podemos editar el código (con nuestro Editor de Texto Plano o IDE de preferencia).
- El **Index** o **Índice** mantiene una copia observada de todos los archivos de código editados en el Workspace. Cada vez que editemos un archivo, debemos agregarlo al Index para que pueda ser añadido a un commit.
- El **Local Repository** o **Repositorio Local** agrupa todas las copias observadas en el Index en forma de commits, es decir, el Local Repository es la zona de trabajo en la que se empaquetan los cambios de código (en forma de commit) para luego ser intercambiados con ambientes remotos.
- El **Remote Repository** o **Repositorio Remoto** es la única zona de trabajo que se ubica remotamente (las 3 anteriores son locales). Es la encargada de almacenar los commits que son enviados desde los diferentes Repositorios Locales y se encarga de distribuir los cambios a todos los usuarios del proyecto Git (todo esto de manera transparente al usuario).

2.3. Comandos esenciales

A continuación mostraremos los comandos esenciales para crear y manejarte dentro de un proyecto Git:

2.3.1. Crear proyectos Git

Para crear un proyecto Git desde cero, nos posicionamos en una carpeta vacía, o que ya tenga código fuente, y ejecutamos la instrucción `git init`. Con esta instrucción la carpeta actual se transformará en un proyecto versionado por Git.

Si contamos con una URL en SSH (o HTTPS) para nuestro repositorio remoto, podemos enlazarlo (como repositorio remoto de origen) a nuestro repositorio local con el comando `git remote add origin [URL]`. Un ejemplo sería (para SSH):

```
$ git remote add origin git@github.com:usuario/proyecto.git
```

Si el repositorio remoto del proyecto ya existe (y tienes acceso a él), puedes *clonar* dicho repositorio con todos sus commits actuales con el comando `git clone [URL]` (se creará un nuevo proyecto y se copiará todo el repositorio remoto en él, incluyendo el enlace al origen). Un ejemplo sería el siguiente:

```
$ git clone git@github.com:usuario/proyecto.git
```

IMPORTANTE: Los comandos `git init` y `git clone` son excluyentes, es decir, solo es necesario ejecutar uno de los dos para iniciar un proyecto Git.

2.3.2. Crear y manejar commits

Una vez creado (o clonado) el proyecto Git, podemos empezar a incorporar el código fuente al *Directorio Local* de trabajo. Supongamos que ya llevamos algo avanzado en un proyecto de C, y que hemos generado los archivos main.c y biblioteca.h, para marcar estos archivos como observados en nuestro proyecto (añadirlos al *Index*), usamos el comando `git add [ARCHIVO]` de la siguiente forma:

```
$ git add main.c
$ git add biblioteca.h
```

NOTA: Si se desea marcar como observados varios archivos, un atajo útil es `git add --all`, que nos permite agregar todos los archivos recientemente editados al *Index*.

Una vez que hemos agregado al *Index* todos los archivos necesarios, podemos agrupar estos cambios en un *commit* con el comando `git commit -m [MENSAJE]`. Un ejemplo sería el siguiente:

```
$ git commit -m "Agregados archivos main.c y biblioteca.h al proyecto"
```

NOTA: Es una buena costumbre en programación añadir un mensaje claro y conciso sobre los cambios de código que se incluyen en cada commit.

Una vez que hemos agrupado uno o más commits, y deseemos subirlos al *Repositorio Remoto*, utilizaremos el comando `git push [REMOTO] [RAMA]`. El uso más característico es el siguiente:

```
$ git push origin master
```

IMPORTANTE: Nótese acá que definimos a *origin* como repositorio remoto, que es la etiqueta por defecto. También definimos a *master* como la rama actual, que es la fijada por defecto. En muchos casos no es necesario cambiar ni la rama, ni el repositorio remoto. Si deseas aprender más sobre el desarrollo en múltiples ramas, termina de leer esta sección y dirígete a la página 29.

Por el contrario, si deseamos traer desde el *Repositorio Remoto* todos los commits a nuestro ambiente local, ejecutamos `git pull [REMOTO] [RAMA]`, usualmente de esta forma:

```
$ git pull origin master
```

NOTA: Otro elemento que se suele tomar en cuenta a este nivel es el tag HEAD, que es una referencia al último commit dentro de la rama actual (para saber más de los tags en Git, revisa la página 27).

2.3.3. Estado actual del Repositorio

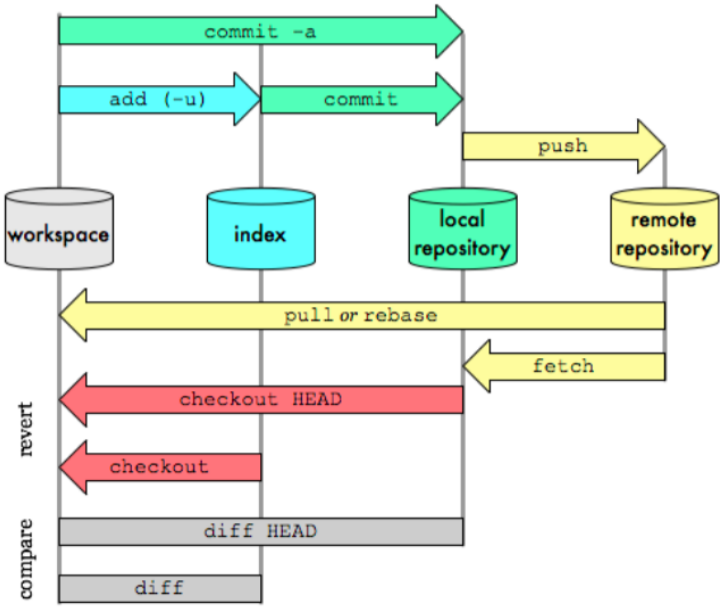
Un comando adicional muy útil es `git status`, puede ser ejecutado en cualquier momento y nos dará información útil como la siguiente:

- Archivos que están en el Index
- Archivos que no están en el Index
- Último commit en el Repositorio Local
- Si nuestro Repositorio Local está "adelantado"/"atrasado" respecto del Repositorio Remoto (útil para saber cuando hacer `push` o `pull`)

Por último, con el comando `git log` podemos ver una lista de nuestros últimos commits realizados, y un poco de información acerca de éstos.

2.3.4. Comandos y Zonas de trabajo

Una forma visual de ver los comandos recién explicados se resume en la siguiente imagen[3]:



En la imagen anterior vemos los comandos básicos que hemos enseñado en este manual, más algunos otros que quedan de tarea de investigación para quien esté interesado, aunque con lo que ya fue explicado acá basta para manejarse de forma básica en Git.

2.4. Consejos útiles

Recapitulando, hasta ahora hemos visto `git init` o `git clone` para inicializar Proyectos Git, los comandos `git add`, `git commit`, `git push` y `git pull` para manipular cambios y commits, y `git status` para verificar el estado de nuestro Repositorio Local.

A continuación se darán una serie de consejos útiles para mantenerte por el "buen camino" en Git:

- **Agregar a los Repositorios solo archivos de texto plano**, Git está optimizado solo para archivos de estas características, es recomendable no subir documentos formateados, ni archivos multimedia, ni binarios compilados.
- **Git guarda únicamente las diferencias entre commits**, es útil saber que una vez subido el commit a los Repositorios, Git automáticamente calcula las diferencias entre los archivos anteriores y los nuevos, y guarda estas diferencias en el disco como commits efectivos (en vez de guardar los archivos completos). Esto solo funciona si has respetado el punto anterior.
- **Una iteración de trabajo en Git es** `pull-add-commit-push`, cada vez que inicies una sesión de trabajo en git, importa los cambios remotos, haz los tuyos, genera nuevos commits y empújalos al Repositorio Remoto. Repite esta secuencia hasta acabar el proyecto.
- **Siempre hacer** `pull` **antes de** `push`, antes de comenzar a editar, importa los cambios remotos. También es recomendable importar cambios antes de subir los commits al Remoto (si es que ha pasado mucho tiempo desde el último `pull`).
- **Git fusiona automáticamente los cambios**, por si no te lo has preguntado aún, al hacer `pull`, Git automáticamente detecta las diferencias de código entre los Repositorios Local y Remoto y fusiona todos los cambios hechos, dejando como prioritarios los cambios más recientes. Si bien el algoritmo de fusionado (merge) por defecto de Git puede ser salvador en muchas ocasiones, no es perfecto y a veces hay que recurrir a Mergetools externos para fusionar cambios conflictivos (hablaremos más de esto en la página 23).
- **Siempre incluir un README**, es una buena costumbre en los proyectos Git incluir un archivo README.txt (en texto plano) o README.md (en formato Markdown) explicando el nombre del proyecto, sus autores, su función e instrucciones de compilación/ejecución. Dicho README debe ir ubicado en el directorio raíz del Proyecto Git.
- **Siempre incluir un GITIGNORE**, es una buena costumbre incluir un archivo de texto plano llamado .gitignore (nótese que comienza con punto) en el directorio raíz del proyecto Git. El archivo GITIGNORE incluye el nombre de los archivos y carpetas a excluir en los commits de nuestro proyecto (muy útil para excluir ejecutables y archivos multimedia) y es leído por el cliente Git cada vez que creamos un nuevo commit. Un ejemplo de su contenido es el siguiente:

  ```
  /build/*
  biblioteca.so
  *.class
  *.jpg
  *.wav
  ```

  Este GITIGNORE excluye de los commits del proyecto la carpeta build, el archivo biblioteca.so y todos los archivos con extensiones class, jpg y wav.

NOTA: Cada vez que iniciamos o clonamos un Proyecto Git, se crea en el directorio seleccionado una carpeta oculta .git que contiene todos los commits del repositorio. Es importante no editar bajo ningún motivo los contenidos de esta carpeta, y garantizar los derechos de lectura y escritura sobre la misma.

2.5. Programación colaborativa con Git

La gran ventaja de usar un sistema de control de versiones como Git reside en que permite a múltiples usuarios ver y editar el código en instancias simultáneas, lo que permite reducir los tiempos de programación al emplear varios desarrolladores en la tarea.

Lo más importante y primordial para la programación colaborativa con Git, es que todos los comandos y acciones que hemos definido anteriormente son también válidos para un entorno Git multiusuario, con el único alcance que hay que tener algo más de cuidado a la hora de hacer `push` (para evitar romper el código en el Repositorio Remoto) y *pull* (para evitar traer cambios que puedan romper el Directorio Local). Cada programador del proyecto tendrá su Repositorio Local propio, pero el Repositorio Remoto será compartido, y será tarea del servidor Git ordenar los commits y distribuirlos a todos los miembros del proyecto.

Adicionalmente a los consejos vistos en la subsección anterior, se añaden los siguientes (para ambientes multiusuario):

- **Un `commit` debe ser siempre funcional**, hay dos características esenciales que un commit debe tener: Un commit debe compilar sin errores y además debe implementar características completas al proyecto. Si cualquiera de estos principios no se cumple, estamos entorpeciendo la labor del grupo de trabajo, ya que estaríamos aportando código con errores o haciendo "commits no funcionales" que incorporan funcionalidades a medias. Queda a criterio de cada programador definir que es una "característica completa".
- **Cuidado al hacer `pull`**, si estás desarrollando una característica nueva en el proyecto o arreglando algún bug importante, lo mejor es retrasar el `pull` hasta que dichos cambios sean concretados en un `commit`. Una vez que esté listo, podemos traer los cambios desde el repositorio remoto y fusionarlos con el commit recién hecho.
- **Cuidado al hacer `push`**, si tienes uno o más commits en tu Repositorio Local y deseas empujarlos al Repositorio Remoto, asegúrate que todos ellos sean "commits funcionales".
- **Medir la granularidad de los commits**, algunos programadores de Git prefieren los commits de *alta granularidad* (es decir, muchos commits con pocas líneas de código cada uno), otros prefieren la *baja granularidad* (pocos commits pero con muchas líneas de código cada uno). Es importante en los proyectos acordar de antemano la granularidad de los commits con los demás programadores, o en caso contrario saber cómo gestionar commits de distinta granularidad.

Recuerda que *solo la práctica hace al maestro*. Lo importante es que siempre se aprende algo nuevo al integrar Git con nuestros proyectos. Al principio es muy probable que hayan ocasiones en que se rompa el código o el repositorio entero, pero el enfrentar estas experiencias forman al buen programador. Como último "tip", recuerda siempre recurrir a buscadores web (como Google) para aquellos problemas o dudas que no puedas resolver, se recomienda hacer estas búsquedas en inglés para maximizar el número de soluciones posibles.

3. Instalando Git

3.1. GNU/Linux

Para instalar Git en distribuciones basadas en Debian (como Ubuntu y Linux Mint), ejecuta el siguiente comando en la terminal:

```
$ sudo apt-get install git
```

En otras distribuciones el paquete a instalar se suele llamar *git*. También es importante tener una Herramienta de Fusionado o Mergetool, en este caso instalaremos **KDiff3**[4] de la siguiente manera:

```
$ sudo apt-get install kdiff3-qt
```

Una vez instalado Git, se le debe indicar tu nombre de usuario y tu correo electrónico (para que todos tus commits sean identificados con estas credenciales) además de la Mergetool por defecto a utilizar (en este caso, KDiff3). Para ello introduce en la terminal estos 3 comandos (reemplaza el nombre y correo por tus identificadores propios):

```
$ git config --global user.name "Juan Pérez"
$ git config --global user.email juan.perez@usach.cl
$ git config --global merge.tool kdiff3
```

Git usa una llave SSH para validar los commits enviados a servidores remotos. Es importante crear esta llave en cada equipo que se vaya a utilizar en el desarrollo, y asegurar que el servidor remoto conozca la llave pública de cada equipo[5]. Para crear una llave SSH, primero ingresamos:

```
$ ssh-keygen -t rsa -b 4096 -C "juan.perez@usach.cl"
```

Reemplazando el correo por el usado en la configuración de credenciales. Luego nos pedirá crear una passphrase para la llave, usualmente se deja en blanco (o se puede definir una a gusto). Ahora activamos el Agente SSH y le añadimos la llave recién creada para que la almacene y utilice:

```
$ eval "$(ssh-agent -s)"
$ ssh-add ~/.ssh/id_rsa
```

Por último, le decimos (con el comando *cat*) que nos muestre la porción pública de la llave SSH, la cual copiaremos en el servidor git remoto a utilizar:

```
$ cat ~/.ssh/id_rsa.pub
```

¡Listo! Ahora podrás usar las instrucciones aprendidas en la sección anterior como comandos en la terminal para controlar el versionamiento de tu proyecto con Git.

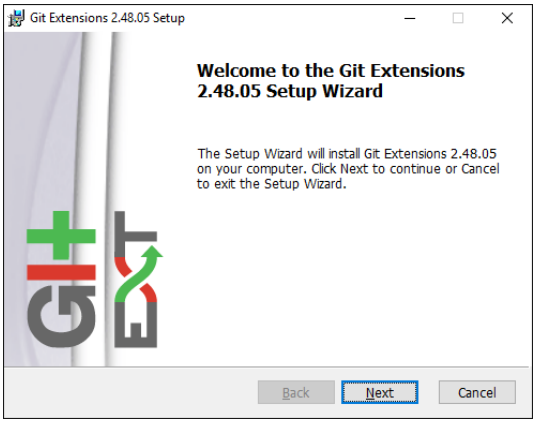
3.2. Microsoft Windows

Para Microsoft Windows es recomendable descargar el frontend **Git Extensions** para poder manejar Git desde una GUI.

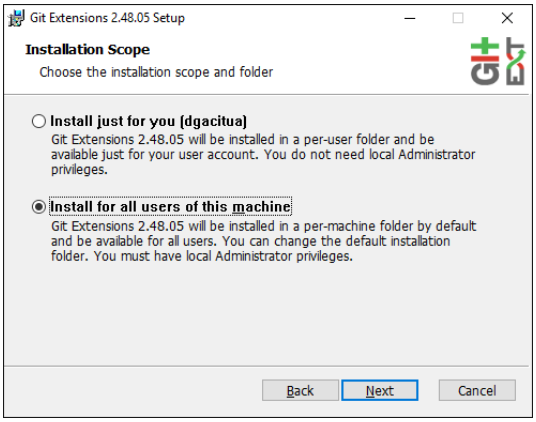
Descarga de Git: https://gitextensions.github.io/

IMPORTANTE: Se recomienda bajar la versión *Complete Setup* de Git Extensions, que trae todo lo necesario para establecer un ambiente de control de versiones (Cliente Git, Frontend y Mergetool[4]).

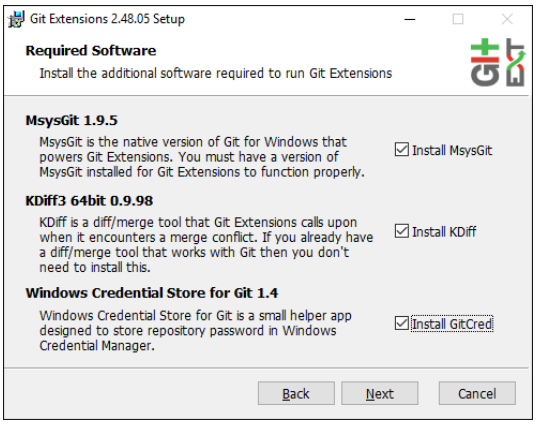
Iniciamos la instalación:



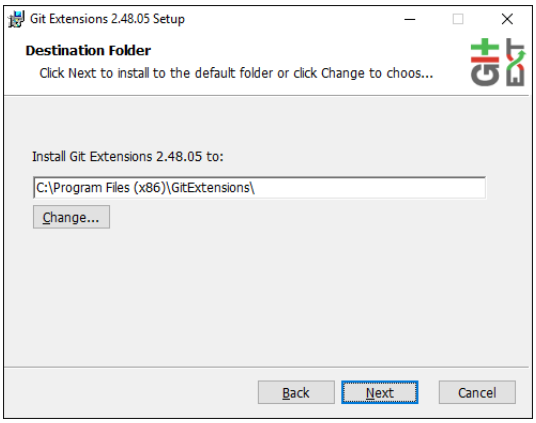
Escogemos si se va a instalar GitExtensions para todos los usuarios, o solo para el usuario actual:



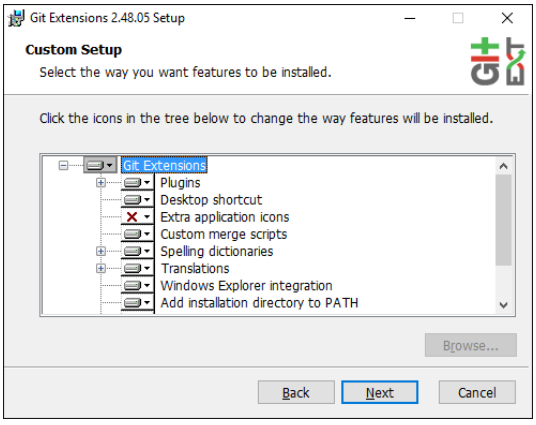
En este paso es importante marcar todo como software requerido:



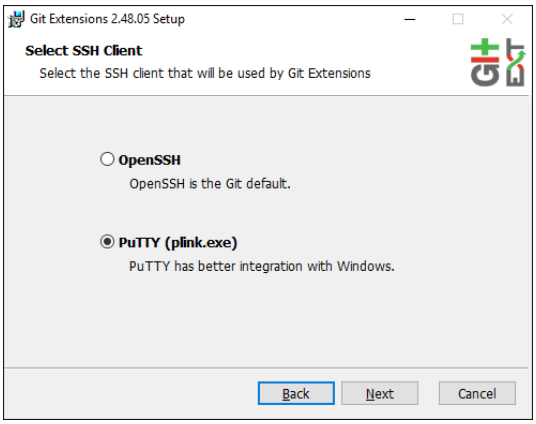
Escogemos el directorio donde se instalará (usualmente es mejor dejarlo por defecto):



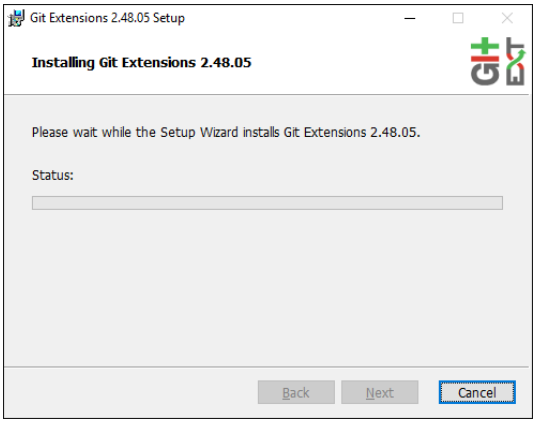
Acá vemos que features de Git Extensions vamos a instalar (si no estás seguro, mejor dejarlo por defecto):



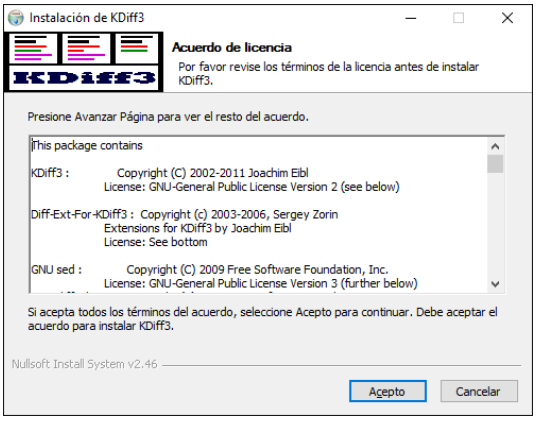
En este paso se recomienda usar PuTTY como cliente SSH para Windows (se instalará junto con Git Extensions):



Iniciamos la instalación:

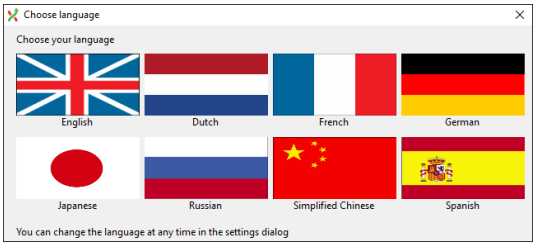


A mitad de instalación, se iniciará el instalador de KDiff3 (el Mergetool), se recomienda avanzar a través del instalador dejando todo por defecto:

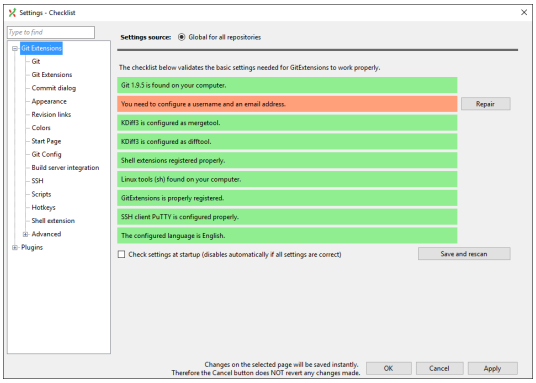


Luego vendrá el instalador de git-scm (el módulo principal del cliente Git), lo recomendable es hacer lo mismo que con KDiff3 y dejar que se instale con todas las configuraciones por defecto:

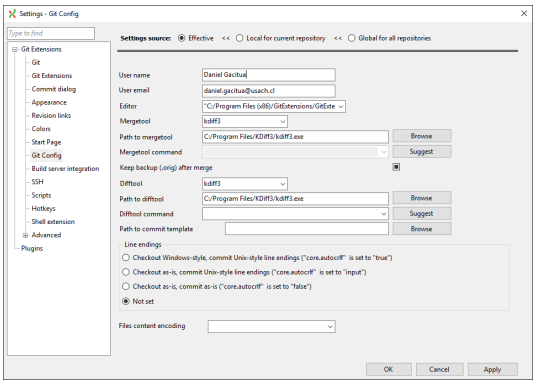
Una vez terminada la instalación, se abrirá Git Extensions. Se recomienda encarecidamente usar la interfaz en inglés, para usar los mismos conceptos (sin traducir) que se utilizan en Git para terminal GNU/Linux:



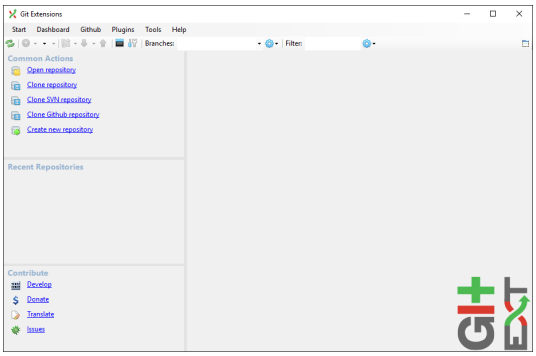
Se iniciará el checklist de configuraciones de Git Extensions, lo más probable es que solo falte configurar el Nombre y Correo del Autor de los commits:



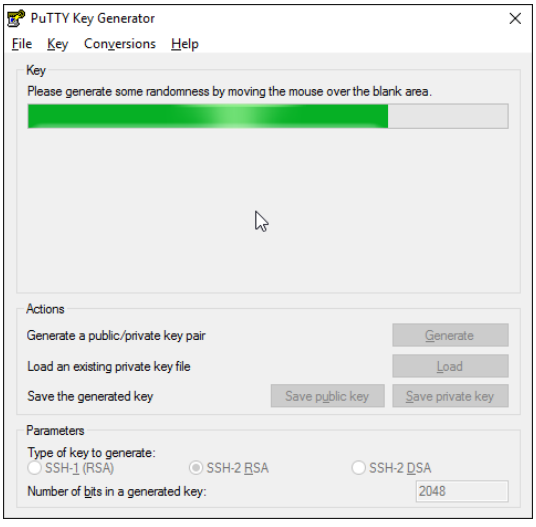
Hacemos click en *Repair* y añadimos ambos datos:



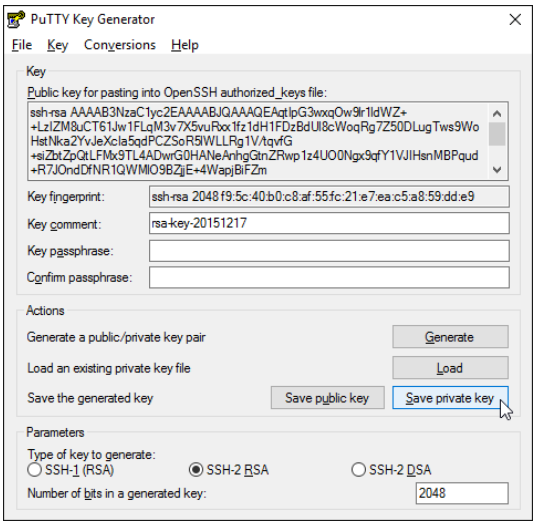
Luego tedremos nuestro ambiente de Git Extensions casi listo:



Para crear una llave SSH, vamos a *Tools*, luego a *PuTTY* y luego a *Generate or import key*. En la nueva ventana hacemos click en *Generate* para empezar a generar la llave:



La llave SSH se genera en base al movimiento del ratón, una vez generada, guardamos ambas partes de la llave con *Save public key* y *Save private key*:



Una vez guardadas ambas partes de la llave, abrimos con un Editor de Texto Plano el archivo con la llave pública, y copiamos su contenido en nuestro Servidor Remoto de Git.

4. Interacción con Servidores Públicos de Git

Gran parte de las veces trabajamos como freelancer o no contamos con servidores privados para poder levantar Proyectos Git, es por eso que a continuación documentamos como usar 2 de los más grandes proveedores de repositorios públicos (y gratuitos) de Git: **BitBucket** y **GitHub**.

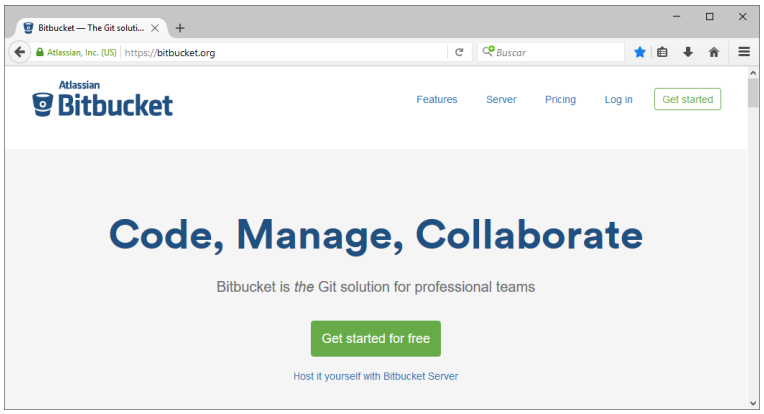
4.1. BitBucket

4.1.1. Características

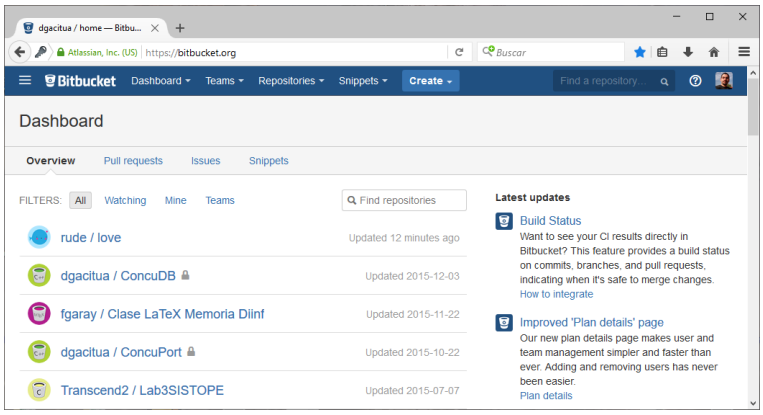
- Repositorios públicos y privados ilimitados.
- Control de Ramas, Issue Tracking y Wiki.
- Fácil integración con JIRA, Confluence y HipChat.
- Recomendado para proyectos personales y trabajos prácticos académicos.

4.1.2. Interfaz web

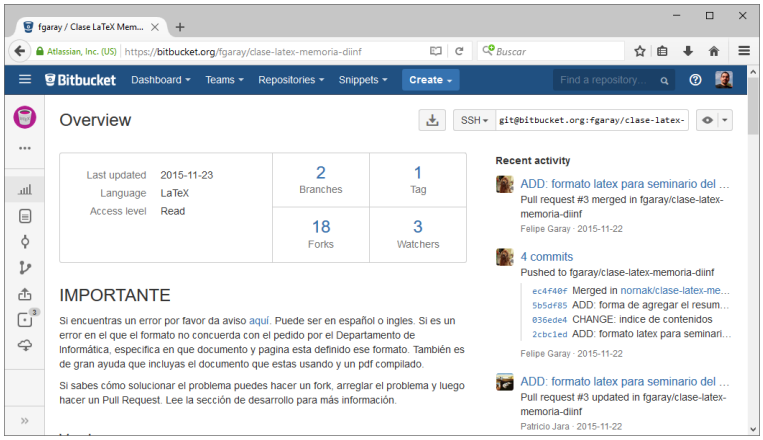
Accedemos primero a https://bitbucket.org/ y somos bienvenidos por la siguiente pantalla:



Una vez que hayamos creado una cuenta en BitBucket (proceso muy sencillo) y nos hayamos logueado con esta, nos recibe la siguiente vista, que muestra todos los proyectos que tengamos:

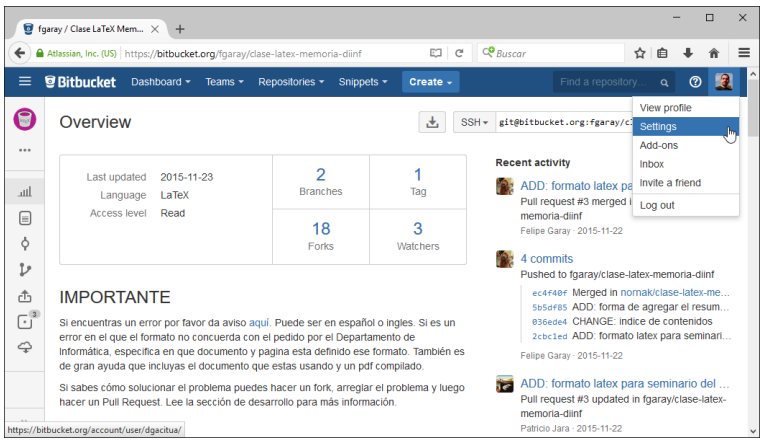


Al clickear cualquier proyecto, nos aparece la siguiente vista, en el menú de la izquierda podremos elegir ver la descripción del proyecto, sus commits, sus ramas y otras configuraciones:

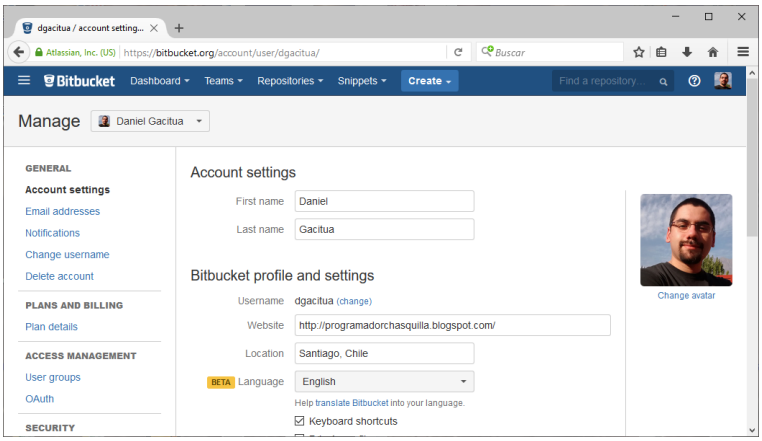


4.1.3. Cargar Llaves SSH

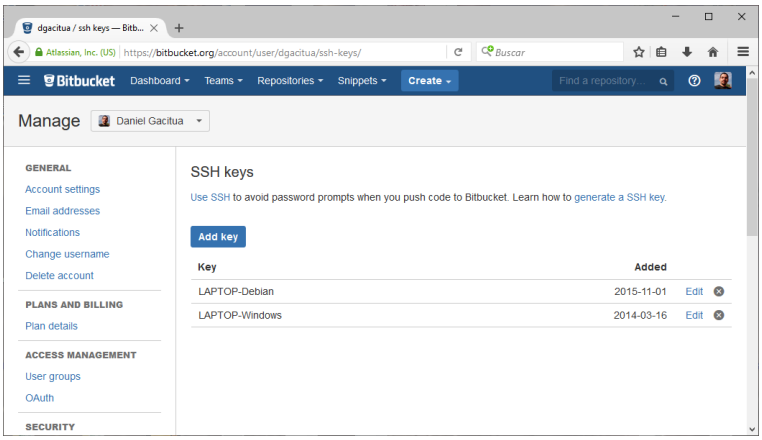
Para insertar la llave pública SSH de nuestro equipo local en BitBucket, hacemos click en el símbolo de nuestro avatar (esquina superior derecha) y escogemos *Settings*:



Acá podemos ajustar nuestras opciones de usuario, pero nos interesa buscar en el menú de la izquierda la opción *SSH keys*:



Dentro de *SSH Keys*, podremos asociar las llaves públicas SSH a nuestra cuenta de BitBucket con el botón *Add key*:



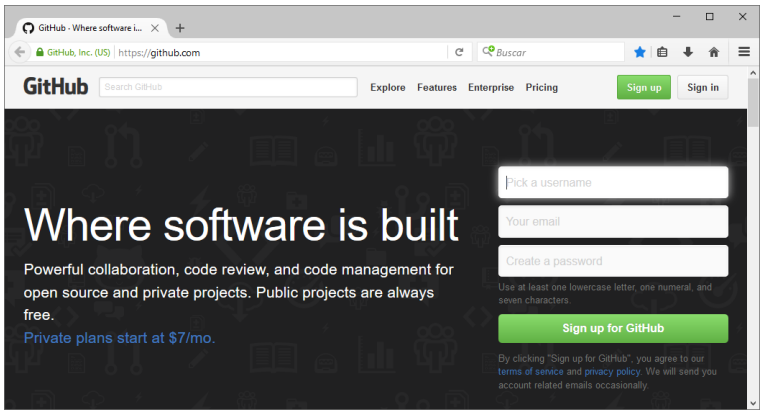
4.2. GitHub

4.2.1. Características

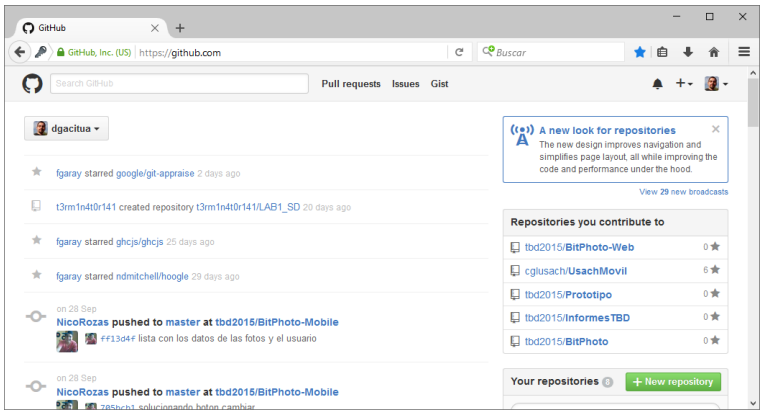
- Repositorios públicos ilimitados.
- Control de Ramas, Issue Tracking y Wiki.
- Capacidad para manejar Organizaciones y Teams de Trabajo.
- Recomendado para proyectos Open Source y proyectos públicos de gran escala.

4.2.2. Interfaz web

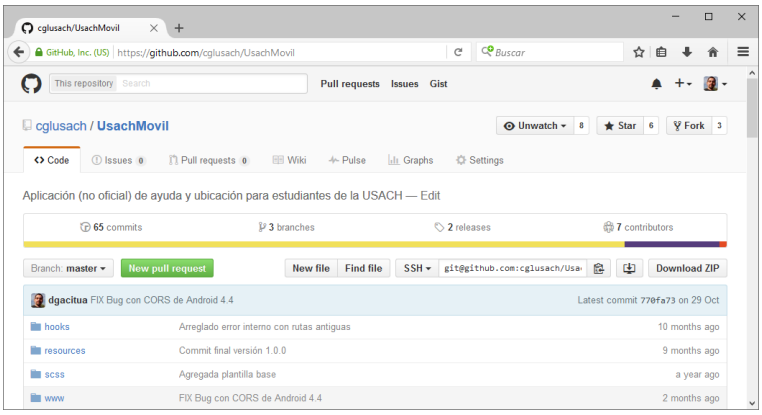
Accedemos a https://github.com/ y seremos recibidos por la siguiente vista:



Una vez creada nuestra cuenta en el menú *Sign up*, nos logeamos con la opción *Sign in*, y podremos ver una lista con las últimas noticias de nuestros repositorios en GitHub:

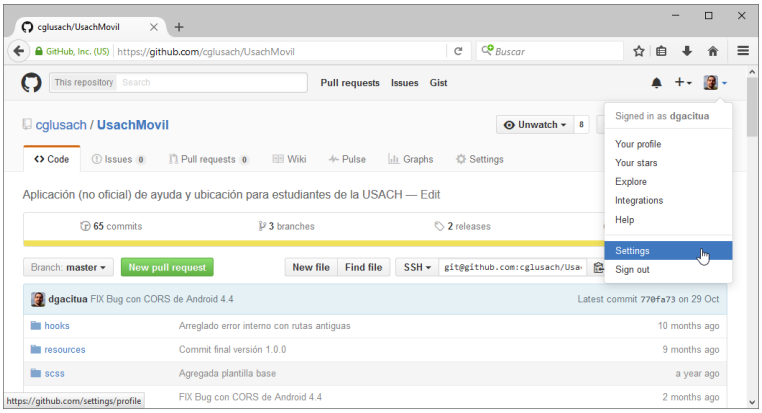


En los menús de la derecha podemos ver nuestros últimos proyectos en los que hemos trabajado, al seleccionar uno de éstos, seremos recibidos por la vista de repositorio (los menús de commits, issues, ramas y otros ajustes del repo está en la barra superior debajo del título del proyecto):

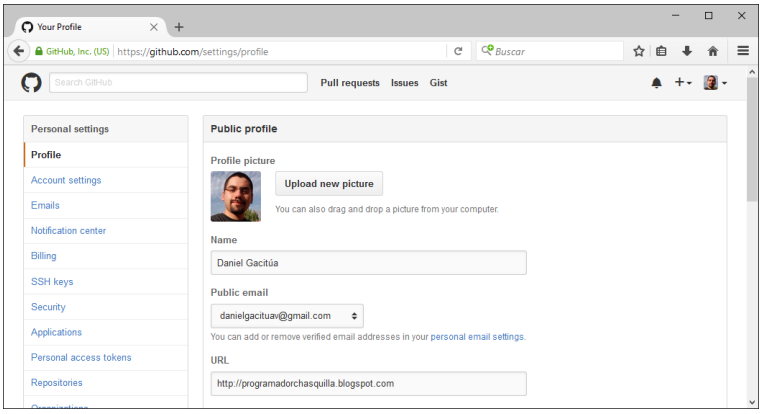


4.2.3. Cargar Llaves SSH

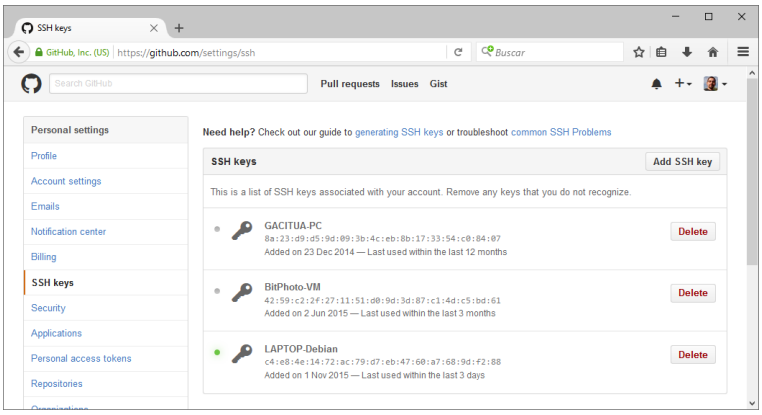
Para insertar las llaves públicas SSH de nuestros equipos en GitHub, hacemos click en el ícono de nuestro avatar (esquina superior derecha), y ahí hacemos click en *Settings*:



Acá podremos ajustar nuestras configuraciones de usuario, si lo deseamos. Pero para la insertar la llave SSH nos interesa el menu *SSH keys*, ubicado en el menú de la izquierda:



Dentro del menu *SSH Keys* podremos asociar una nueva llave SSH con GitHub con el botón *Add SSH key*, o podremos borrar aquellas llaves que no necesitemos con *Delete*:



5. Uso avanzado de Git

Una vez comprendidos y dominados los aspectos básicos de Git, es hora de pasar a conceptos ligeramente más complejos, pero de mucha utilidad a la hora de gestionar proyectos Git (se utilizará la sintaxis de Git de terminal GNU/Linux para expresar los nuevos comandos).

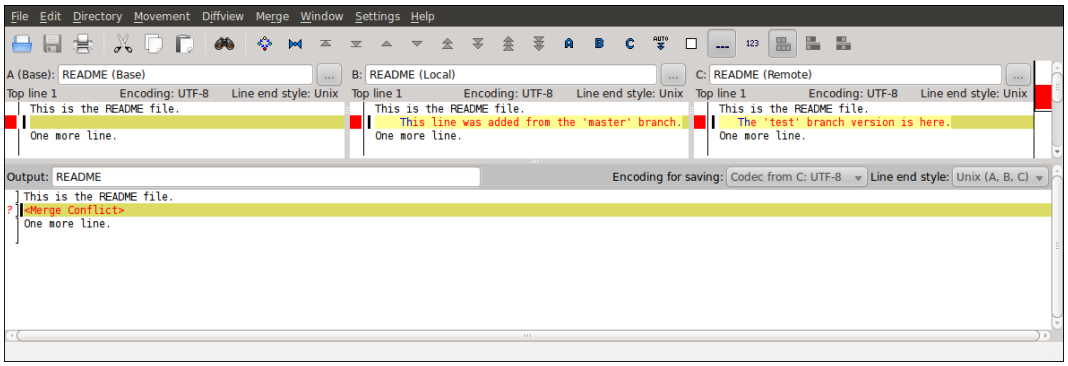
5.1. Merging: Fusionando cambios de código

Muchas veces necesitaremos tirar y fusionar commits que no son nuestros con nuestro Directorio Local. El cliente Git por defecto trae algunos algoritmos automáticos para lidiar con fusiones de código, pero para aquellos cambios conflictivos es necesario contar con una Mergetool (herramienta de fusionado) externa. En la sección INSTALANDO GIT (página 10) se instaló y configuró **KDiff3** como mergetool, asi que trabajaremos con esta herramienta para cualquier conflicto de fusionado.

Cada vez que Git de terminal GNU/Linux indique el siguiente error (usualmente tras un `pull` o un `merge`), será necesario usar la mergetool:

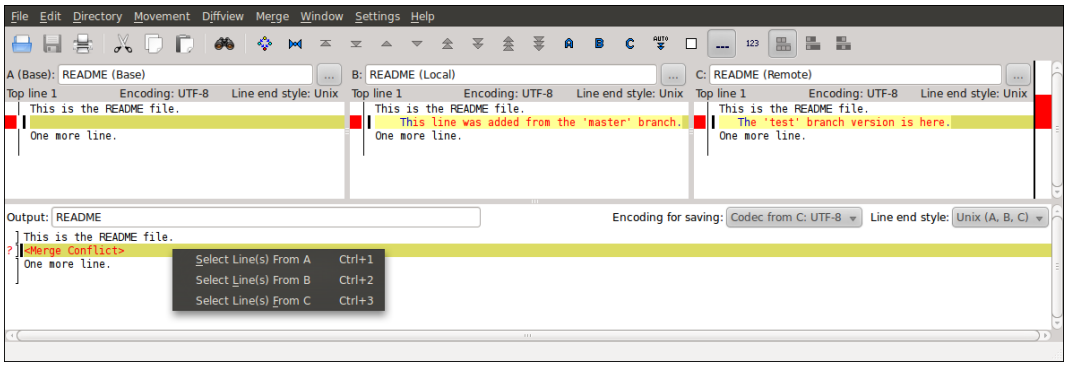
```
Automatic merge failed; fix conflicts and then commit the result.
```

Como ya está configurada la herramienta de fusionado, bastará con ejecutar `git mergetool` para que inicie el GUI de KDiff3[6]:

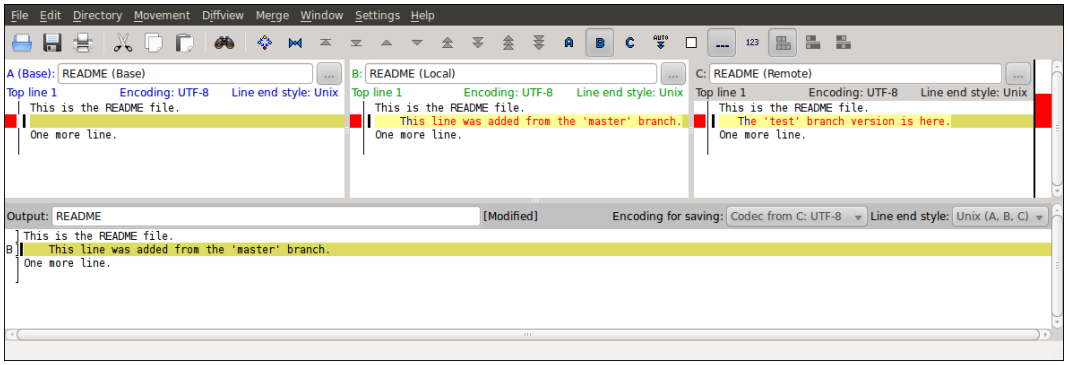


Acá vemos el layout dividido en 4 regiones. Las 3 regiones superiores corresponden al ancestro en común del archivo (columna A), la versión en el Repositorio Local (columna B) y la versión que estamos tirando desde un Repositorio Remoto (columna C). La región inferior muestra el archivo final tras las fusiones.

Para arreglar el conflicto, buscaremos las lineas marcadas como *Merge Conflict* en la región inferior, haremos click derecho en éstas y escogeremos cual de las 3 versiones deseamos implementar (columna A, B o C). Podemos escoger más de una versión, o ninguna, de ser necesario:



Nótese que acá estamos lidiando manualmente con los conflictos de fusionado, se recomienda tener discreción (estar seguro de lo que se está haciendo) al escoger los cambios a concretar. Una vez que se hayan solucionado todos los conflictos de fusionado de este archivo, lo guardaremos (click en el ícono de diskette en la barra de herramientas) y cerraremos KDiff3:



Si hay conflictos de fusionado en más de un archivo, Git nos consultará si deseamos abrir KDiff3 de nuevo, pero para solucionar los conflictos de otro archivo, y asi procederemos de la misma manera (uno a uno) para todos los archivos con conflicto de fusionado.

Luego de haber terminado con los conflictos de fusionado, ejecutaremos `git commit` (tal cual) para hacer un nuevo commit guardando los cambios recién realizados.

5.2. Rollbacking: Volviendo a estados anteriores del código

Una de las características más apreciadas de Git (junto con facilitar la programación colaborativa) es la acción de guardar las versiones de código (snapshots) en commits. En esta parte enseñaremos como hacer una "vuelta atrás" o *rollback* a un commit anterior dentro del proyecto.

Muchas veces se hace necesario recurrir a esta técnica ya que algún miembro del proyecto puede introducir (sin querer) commits que rompen el código.

Hay varias formas de hacer un rollback en Git. Si deseamos deshacer los cambios de uno o más commits, utilizaremos `git revert`. Si queremos deshacer toda la historia de una serie de commits, usaremos `git reset`.

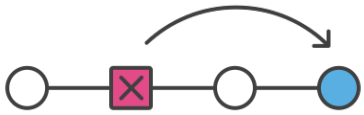
Primero que todo, supongamos que tenemos el siguiente historial en nuestro proyecto de Git (podemos verlo asi usando el comando `git log --oneline`):

```
1f1a4ae Continuar haciendo cosas locas
5e05d51 Intentando algo loco
4f88cf8 Hechos cambios importantes sobre main.c
2ec1363 Creado main.c
548217b Commit inicial
```

El comando `git revert` permite deshacer los cambios de uno o más commits, sin alterar el resto de la historia del proyecto. El comando se utiliza como `git revert [HASH]`, como en el siguiente ejemplo:

```
$ git revert 5e05d51
```

Se generará un nuevo commit en el HEAD del proyecto deshaciendo todos los cambios de los commits indicado por su hash (en este caso, el commit `5e05d51`)[7].



O también puede funcionar con rangos respecto a HEAD, en el siguiente ejemplo, se hace utilizamos una variante de revert para deshacer los últimos 3 commits:

```
$ git revert --no-commit HEAD~3..HEAD
$ git commit -m "Rollback de los últimos 3 commits"
```

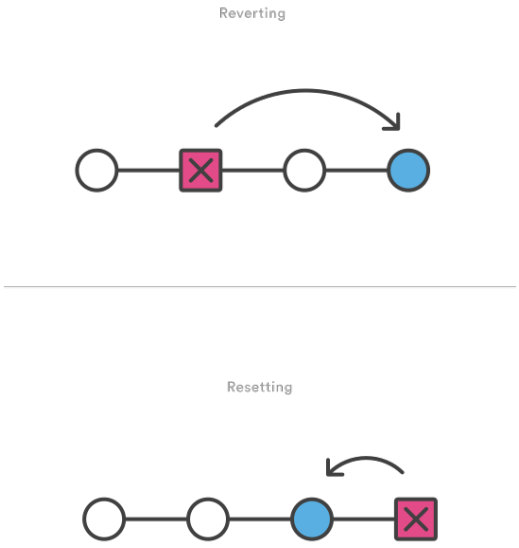
Esta es la forma "suave" de deshacer cambios para llegar a estados anteriores. La forma "dura" es usando `git reset [HASH]`. Este comando borra permanentemente todos los cambios hasta llegar al commit con el hash indicado. Por ejemplo, si queremos volver al estado del commit `4f88cf8` deshaciendo toda la historia posterior, ejecutamos lo siguiente:

```
$ git reset 4f88cf8
```

Un uso alternativo para `git reset` es cuando se quieren deshacer permanentemente todos los cambios locales (no agrupados en commit) y regresar al último commit en el Repositorio Local:

```
$ git reset --hard HEAD
```

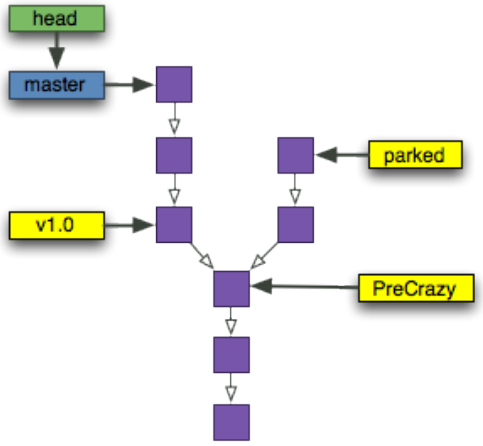
Una vez regresado al estado anterior con `git reset`, se pueden hacer nuevos cambios y añadirlos al repositorio. Es importante notar que siempre se recomienda usar `revert` en vez de `reset`, y si es necesario utilizar este último, usarlo exclusivamente a nivel de Repositorio Local para evitar problemas con el resto del equipo de trabajo.



5.3. Tagging: Identificando commits para publicación

En secciones anteriores hemos visto el tag HEAD, que representa el último commit en la rama actual, este es un tag por defecto y no administrable por el usuario. ¿Pero qué pasa si queremos crear y administrar nuestros propios tags? Un tag en Git es muy útil para el versionado del código, además podemos marcar nuestros commits para publicarse con un tag y así facilitar la tarea a otros usuarios que necesiten descargar nuestro proyecto.

En la siguiente imagen vemos un ejemplo de la aplicación de tags (representados en amarillo) sobre los commits[8]:



Un tag en Git es un alias para los hash de commits, usualmente para facilitar la identificación y legibilidad de éste. Un tag está compuesto por un número de versión y un mensaje. Para crear un tag sobre el último commmit (el HEAD) usaremos el comando `git tag -a [VERSIÓN] -m [MENSAJE]` de la siguiente manera:

```
$ git tag -a v1.0 -m "Release final del producto"
```

Para crear un tag sobre cualquier otro commit del proyecto, usaremos el comando `git tag -a [VERSIÓN] -m [MENSAJE] [HASH]`. Por ejemplo, para aplicar un tag sobre el commit `1f1a4ae`, lo hacemos de la siguiente forma:

```
$ git tag -a v1.1 -m "Parche para bug en la GUI principal" 1f1a4ae
```

Por defecto, todos los tags creados quedan en tu Repositorio Local, para llevarlos al Repositorio Remoto puedes usar `git push [REMOTO] [VERSIÓN]` para empujarlos uno a uno, o `git push [REMOTO] --tags` para empujarlos todos a la vez. Recuerda que el Repositorio Remoto por defecto en Git es `origin`.

Para listar todos los tags del proyecto, usa `git tag` (tal cual). Para ver el commit y la información asociada al tag, usa `git show [VERSIÓN]`.

5.4. Stashing: Guardando código para el futuro

Esta es una situación que puede pasar más de alguna vez cuando se está trabajando con proyectos Git: Estamos desarrollando tranquilamente, cuando uno de nuestros compañeros de equipo incorpora una característica importante que afecta a nuestro desarrollo. Siendo que nuestra característica aún está incompleta, no corresponde hacer un commit aún, pero tampoco queremos perder nuestro desarrollo volviendo al último commit para poder traer los nuevos cambios.

Aquí viene un nuevo concepto, el *stash*. Un *stash* es una instancia especial que permite agrupar y almacenar cambios de código sin implícitamente realizar un commit. Al crear un *stash* todos los cambios sobre el Directorio Local son almacenados en éste, y luego se vuelve automáticamente al último commit (el HEAD). Una vez estando en el HEAD podemos realizar los cambios necesarios, para luego re-aplicar los cambios del *stash* sobre el Directorio Local cuando se estime conveniente.

Antes de crear un *stash* conviene hacer `git status` para saber que cambios serán escritos en el nuevo *stash*. Luego ejecutamos `git stash` para crear el nuevo *stash*. Podemos generar más de un *stash* de ser necesario.

```
$ git stash
Saved working directory and index state \
  "WIP on master: 1f1a4ae Continuar haciendo cosas locas"
HEAD is now at 1f1a4ae Continuar haciendo cosas locas
(To restore them type "git stash apply")
```

Para ver la lista actual de stashes, usamos `git stash list`:

```
$ git stash list
stash@{0}: WIP on master: 1f1a4ae Continuar haciendo cosas locas
stash@{1}: WIP on master: 5e05d51 Intentando algo loco
stash@{2}: WIP on master: 4f88cf8 Hechos cambios importantes sobre main.c
```

Para restaurar el último stash se usa `git stash apply`, o si se quiere recuperar un stash anterior en la lista, procedemos de la siguiente forma:

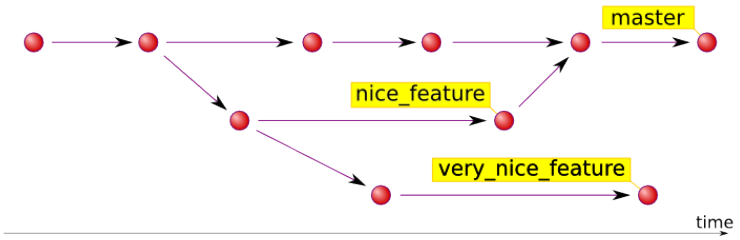
```
$ git stash apply stash@{2}
```

Para eliminar un stash que ya no sea necesario, se procede asi:

```
$ git stash drop stash@{2}
```

5.5. Branching: El arte del código ramificado

En secciones anteriores se vio que Git emplea a `master` como rama por defecto para todos los proyectos Git. Una de las grandes ventajas de Git por sobre otros sistemas de versionamiento de código es el poder crear múltiples *branches* o ramas que nos permitan "dividir y conquistar" cualquier proyecto de software, sin importar el tamaño.



Para crear una nueva rama, usamos `git branch [RAMA]`, de la siguiente forma:

```
$ git branch nice_feature
```

Para cambiar de rama, usamos `git checkout [RAMA]`, de la siguiente forma:

```
$ git checkout nice_feature
```

Nótese que el HEAD cambiará al último commit de la nueva rama. Como acabamos de crear `nice_feature`, el commit al que apunta el HEAD será el mismo que el último commit de la rama anterior. Una vez en la nueva rama, podemos trabajar de la misma manera en la que lo hacíamos en `master`, pero sin alterar los commits de las otras ramas. Esto es muy útil a la hora de incorporar nuevas características sin afectar el código que ya está estable.

Una vez que nuestra nueva característica ya está implementada, y deseemos fusionar su contenido con la rama `master` (o la que estimemos conveniente), ejecutamos `git checkout` a la rama donde deseemos importar el código y luego `git merge` de la siguiente forma:

```
$ git checkout master
$ git merge nice_feature
```

Nótese aquí dos cosas: Primero, al hacer merge, es posible que los cambios se fusionen automáticamente o sea necesario hacerlo manualmente con un mergetool (ver página 23). Segundo, para fusionar cambios entre ramas, siempre es necesario hacer `checkout` a la rama destino y luego hacer `merge` para traer los cambios de la rama de origen.

La fusión de ramas será representada en un nuevo commit al finalizar. Las ramas pueden seguir siendo usadas tras una fusión (es más, es posible fusionar múltiples veces una rama con otra, siempre que hayan cambios desde la útima fusión).

Para un "manejo inteligente" de las ramas, se recomienda dejar la rama `master` para todo el código estable y listo para ser distribuido (se recomienda también apoyarse en los tags para esto) y dejar las ramas para nuevas características para implementar al código (así tambien como bugfixes y mejoras menores). Hay variados esquemas para organizar las ramas en Git, sin embargo Git en si no impone ningún esquema en específico, es responsabilidad del desarrollador utilizar o crear un esquema de ramas que le convenga y facilite la tarea.

6. Miscelánea

6.1. Acerca del autor

Daniel Gacitúa Vásquez es un estudiante de Ingeniería de Ejecución en Computación e Informática de la Universidad de Santiago de Chile (USACH). Miembro del Staff de la Comunidad GNU/Linux USACH. Ha dictado en varias ocasiones el *Taller de Programación Colaborativa con Git* en dicha casa de estudios.

6.2. Agradecimientos

- Al Staff de la **Comunidad GNU/Linux USACH** por brindarme su apoyo para elaborar este texto.
- A **Felipe Garay** por enseñarme lo básico para iniciar en el mundo del versionado con Git.
- Al **Departamento de Ingeniería Informática de la USACH** por proveerme de las herramientas tecnológicas para poder construir este Manual.

6.3. Ediciones al manual

- **17-12-2015 (v0.1)** Inicio de la creación del Manual
- **20-12-2015 (v0.2)** Elaborado esquema base del Manual
- **24-12-2015 (v0.3)** Funcionalidad básica del Manual terminada
- **26-12-2015 (v0.4)** Primera versión preliminar del Manual completada
- **13-01-2016 (v0.5)** Segunda versión preliminar del Manual completada
- **14-01-2016 (v1.0)** Primera versión pública

7. Bibliografía y Referencias

[1] *Git*. Wikipedia. 2015. URL: https://es.wikipedia.org/wiki/Git (visitado 17-12-2015).

[2] *Git is Your Friend not a Foe*. The House of Hades. 2010. URL: http://hades.github.io/2010/01/git-your-friend-not-foe/ (visitado 22-12-2015).

[3] *Git Workflows*. Yan Pritzker. 2015. URL: http://documentup.com/skwp/git-workflows-book (visitado 19-12-2015).

[4] *KDiff3*. KDE Applications. 2015. URL: https://www.kde.org/applications/development/kdiff3/ (visitado 18-12-2015).

[5] *Generating SSH Keys*. GitHub Help. 2015. URL: https://help.github.com/articles/generating-ssh-keys/ (visitado 17-12-2015).

[6] *Merging with a GUI*. GitGuys. 2015. URL: http://www.gitguys.com/topics/merging-with-a-gui/ (visitado 24-12-2015).

[7] *Undoing Changes*. Atlassian Git Tutorials. 2015. URL: https://www.atlassian.com/git/tutorials/undoing-changes (visitado 25-12-2015).

[8] *Git objects: The tag*. 365Git. 2010. URL: http://365git.tumblr.com/post/497500602/git-objects-the-tag (visitado 25-12-2015).