

Control de Versiones

1.1 - Acerca del Control de Versiones

¿Qué es el control de versiones, y por qué debería importarte?

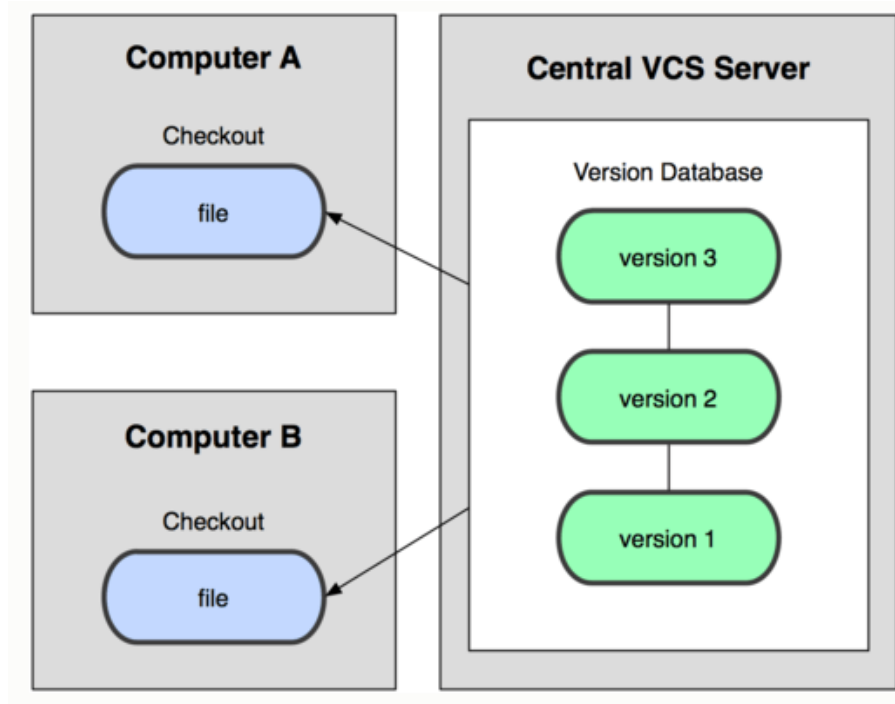
El control de versiones es un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante.

Si quieres mantener cada versión de una imagen o diseño (algo que sin duda quieres), un sistema de control de versiones (Version Control System o VCS en inglés) es una elección muy sabia. Te permite:

- Revertir archivos a un estado anterior
- Revertir el proyecto entero a un estado anterior
- Comparar cambios a lo largo del tiempo
- Ver quién modificó por última vez algo que puede estar causando un problema, quién introdujo un error y cuándo, y mucho más.
- Usar un VCS también significa generalmente que si fastidias o pierdes archivos, puedes recuperarlos fácilmente

Sistemas de control de versiones centralizados

El siguiente gran problema que se encuentra la gente es que necesitan colaborar con desarrolladores en otros sistemas. Para solventar este problema, se desarrollaron los sistemas de control de versiones centralizados (Centralized Version Control Systems o CVCSs en inglés). Estos sistemas, como CVS, Subversion, y Perforce, tienen un único servidor que contiene todos los archivos versionados, y varios clientes que descargan los archivos desde ese lugar central. Durante muchos años éste ha sido el estándar para el control de versiones.



Esta configuración ofrece muchas ventajas, especialmente frente a VCSs locales. Por ejemplo, todo el mundo puede saber (hasta cierto punto) en qué están trabajando los otros colaboradores del proyecto.

Sin embargo, esta configuración también tiene serias desventajas. La más obvia es el punto único de fallo que representa el servidor centralizado. Si ese servidor se cae durante una hora, entonces durante esa hora nadie puede colaborar o guardar cambios versionados de aquello en que están trabajando.

Sistemas de control de versiones distribuidos

Es aquí donde entran los sistemas de control de versiones distribuidos (Distributed Version Control Systems o DVCSs en inglés). En un DVCS (como Git, Mercurial, Bazaar o Darcs), los clientes no sólo descargan la última instantánea de los archivos: replican completamente el repositorio. Así, si un servidor muere, y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios de los clientes puede copiarse en el servidor para restaurarlo. Cada vez que se descarga una instantánea, en realidad se hace una copia de seguridad completa de todos los datos.

1.3 Empezando - Fundamentos de Git

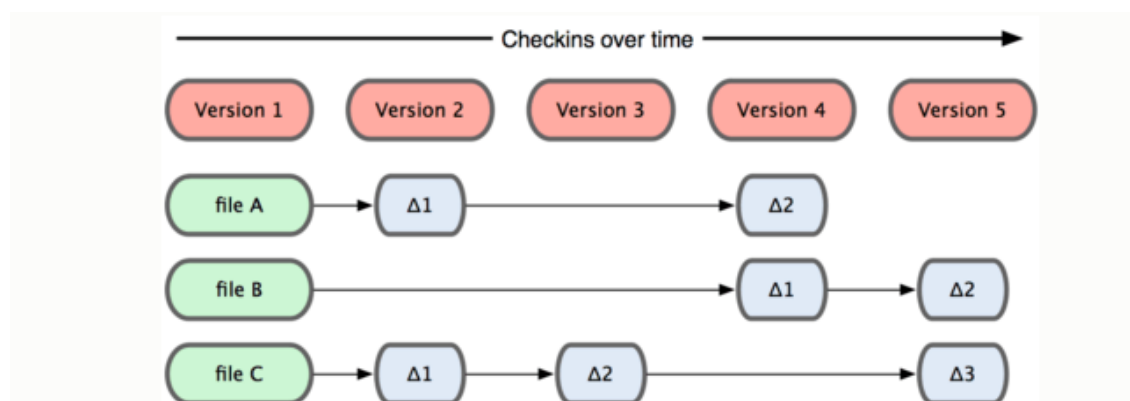


Figura 1-4. Otros sistemas tienden a almacenar los datos como cambios de cada archivo respecto a una versión base.

Git modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea. Para ser eficiente, si los archivos no se han modificado, Git no almacena el archivo de nuevo, sólo un enlace al archivo anterior idéntico que ya tiene almacenado

Casi cualquier operación es local

La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para operar.

Por ejemplo, para navegar por la historia del proyecto, Git no necesita salir al servidor para obtener la historia y mostrártela, simplemente la lee directamente de tu base de datos local. Esto significa que ves la historia del proyecto casi al instante. Si quieres ver los cambios introducidos en un archivo entre la versión actual y la de hace un mes, Git puede buscar el archivo hace un mes y hacer un cálculo de diferencias localmente, en lugar de tener que pedirle a un servidor remoto que lo haga.

Git tiene integridad

Todo en Git es verificado mediante una suma de comprobación (checksum en inglés) antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma.

El mecanismo que usa Git para generar esta suma de comprobación se conoce como hash SHA-1.

```
$ sha1sum <nombre del archivo ISO>
```

Comprobamos si un archivo no ha sido modificado en la descarga por su hash. Si los hash coinciden es que la descarga no ha sufrido errores

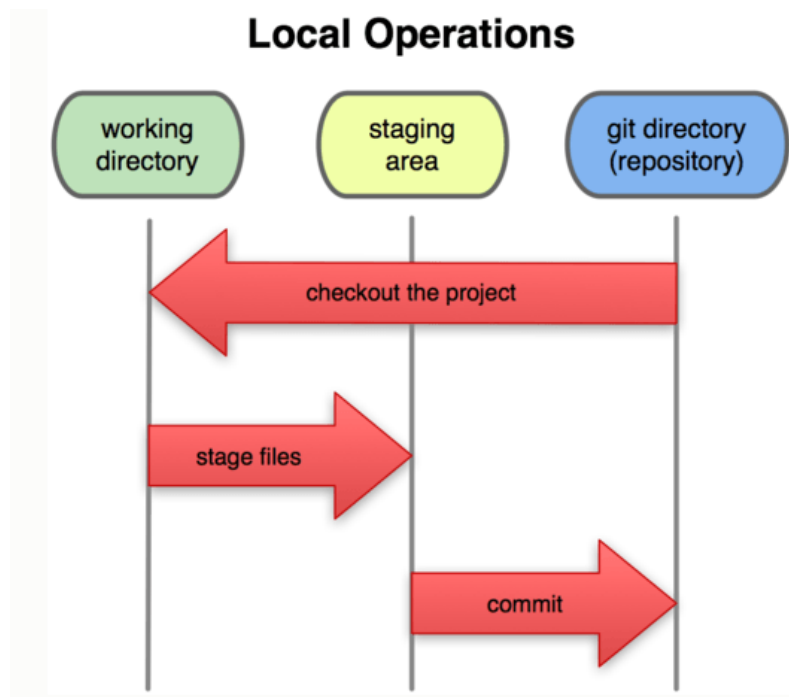
De hecho, Git guarda todo no por nombre de archivo, sino por el valor hash de sus contenidos.

Los tres estados

Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (committed), modificado (modified), y preparado (staged).

- **Confirmado significa** que los datos están almacenados de manera segura en tu base de datos local.
- **Modificado significa** que has modificado el archivo pero todavía no lo has confirmado a tu base de datos.
- **Preparado significa** que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

Esto nos lleva a las tres secciones principales de un proyecto de Git:



El directorio de Git es donde Git almacena los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, y es lo que se copia cuando clonas un repositorio desde otro ordenador.

El directorio de trabajo es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar.

El **área de preparación** es un sencillo archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación.

El flujo de trabajo básico en Git es algo así:

- Modificas una serie de archivos en tu directorio de trabajo.
- Preparas los archivos, añadiéndolos a tu área de preparación.
- Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena esas instantáneas de manera permanente en tu directorio de Git.

Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada (committed). Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está preparada (staged). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada (modified)

1.4 Empezando - Instalando Git

```
vicente@ubuntupc:~$ sudo apt-get update
```

```
vicente@ubuntupc:/var/lib/dpkg$ sudo apt-get install git
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
Se instalarán los siguientes paquetes adicionales:
  git-man liberror-perl
Paquetes sugeridos:
  git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk gitweb
  git-arch git-cvs git-mediawiki git-svn
Se instalarán los siguientes paquetes NUEVOS:
```

Instalando en Windows

Instalar Git en Windows es muy fácil. El proyecto msysGit tiene uno de los procesos de instalación más sencillos. Simplemente descarga el archivo exe del instalador desde la página de GitHub, y ejecútalo:

<http://msysgit.github.com/>

Una vez instalado, tendrás tanto la versión de línea de comandos (incluido un cliente SSH que nos será útil más adelante) como la interfaz gráfica de usuario estándar.

Nota para el uso en Windows: Se debería usar Git con la shell provista por msysGit (estilo Unix), lo cual permite usar las complejas líneas de comandos de este libro. Si por cualquier razón se necesitara usar la shell nativa de Windows, la consola de línea de comandos, se han de usar las comillas dobles en vez de las simples (para parámetros que contengan espacios) y se deben entrecomillar los parámetros terminándolos con el acento circunflejo (^) si están al final de la línea, ya que en Windows es uno de los símbolos de continuación.

1.5 Empezando - Configurando Git por primera vez

Configurando Git por primera vez

Ahora que tienes Git en tu sistema, querrás hacer algunas cosas para personalizar tu entorno de Git. Sólo es necesario hacer estas cosas una vez; se mantendrán entre actualizaciones. También puedes cambiarlas en cualquier momento volviendo a ejecutar los comandos correspondientes.

Git trae una herramienta llamada `git config` que te permite obtener y establecer variables de configuración, que controlan el aspecto y funcionamiento de Git. Estas variables pueden almacenarse en tres sitios distintos:

- Archivo `/etc/gitconfig`: **Contiene valores para todos los usuarios del sistema y todos sus repositorios.** Si pasas la opción `--system` a `git config`, lee y escribe específicamente en este archivo.
- Archivo `~/.gitconfig` file: **Específico a tu usuario.** Puedes hacer que Git lea y escriba específicamente en este archivo pasando la opción `--global`.
- Archivo `config` en el directorio de Git (es decir, `.git/config`) **del repositorio que estés utilizando actualmente: Específico a ese repositorio.** Cada nivel sobrescribe los valores del nivel anterior, por lo que los valores de `.git/config` tienen preferencia sobre los de `/etc/gitconfig`.

Lo primero que deberías hacer cuando instalas Git es establecer tu nombre de usuario y dirección de correo electrónico. Esto es importante porque las confirmaciones de cambios (commits) en Git usan esta información, y es introducida de manera inmutable en los commits que envías:

```
vicente@ubuntupc:~/.gconf$ git config --global user.name "Vicente Pulido"
```

```
vicente@ubuntupc:~/NetBeansProjects/JavaApplication1$ git config --global user.email "vpulido13@gmail.com"
vicente@ubuntupc:~/NetBeansProjects/JavaApplication1$ cat /home/vicente/.gitconfig
[user]
  name = vpulido2
  editor = nano
  email = vpulido13@gmail.com
```

Nota: El archivo .gitconfig se crea cuando introducimos la primera clave: valor

Tu editor

Ahora que tu identidad está configurada, puedes elegir el editor de texto por defecto que se utilizará cuando Git necesite que introduzcas un mensaje. Si no indicas nada, Git usa el editor por defecto de tu sistema, que generalmente es Vi o Vim

```
vicente@ubuntupc:~$ git config --global user.editor emacs
vicente@ubuntupc:~$ cat /home/vicente/.gitconfig
[user]
    name = Vicente Pulido
    mail = vpulido13@gmail.com
    editor = emacs
vicente@ubuntupc:~$
```

Si quieres comprobar tu configuración, puedes usar el comando `git config --list` para listar todas las propiedades que Git ha configurado:

```
vicente@ubuntupc:~$ git config --list
user.name=Vicente Pulido
user.mail=vpulido13@gmail.com
user.editor=emacs
```

También puedes comprobar qué valor cree Git que tiene una clave específica ejecutando `git config {clave}`

1.6 Empezando - Obteniendo ayuda

Si alguna vez necesitas ayuda usando Git, hay tres formas de ver la página del manual (manpage) para cualquier comando de Git:

```
$ git help <comando>
$ git <comando> --help
$ man git-<comando>
```

Por ejemplo, puedes ver la página del manual para el comando config ejecutando:

```
$ git help config
```

Chapter 2

Fundamentos de Git

Al final del capítulo, deberías ser capaz de configurar e inicializar un repositorio, comenzar y detener el seguimiento de archivos, y preparar (stage) y confirmar (commit) cambios. También te enseñaremos a configurar Git para que ignore ciertos archivos y patrones, cómo deshacer errores rápida y fácilmente, cómo navegar por la historia de tu proyecto y ver cambios entre confirmaciones, y cómo enviar (push) y recibir (pull) de repositorios remotos.

2.1 Fundamentos de Git - Obteniendo un repositorio Git

Obteniendo un repositorio Git

Puedes obtener un proyecto Git de dos maneras.

- La primera toma un proyecto o directorio existente y lo importa en Git.
- La segunda clona un repositorio Git existente desde otro servidor.

En el caso de utilizemos la primera opción nos hemos de ubicar en el directorio del proyecto y ejecutar la siguiente orden:

```
vicente@ubuntupc:~/NetBeansProjects/JavaApplication1$ git init
Initialized empty Git repository in /home/vicente/NetBeansProjects/JavaApplication1/.git/
```

Esto crea un nuevo subdirectorio llamado `.git` que contiene todos los archivos necesarios del repositorio —un esqueleto de un repositorio Git. Todavía no hay nada en tu proyecto que esté bajo seguimiento.

```
vicente@ubuntupc:~/NetBeansProjects/JavaApplication1$ cd .git
vicente@ubuntupc:~/NetBeansProjects/JavaApplication1/.git$ ls
branches  config  description  HEAD  hooks  info  objects  refs
vicente@ubuntupc:~/NetBeansProjects/JavaApplication1/.git$
```

Clonando un repositorio existente

Si deseas obtener una copia de un repositorio Git existente —por ejemplo, un proyecto en el que te gustaría contribuir— el comando que necesitas es `git clone`.

Git recibe una copia de casi todos los datos que tiene el servidor. Cada versión de cada archivo de la historia del proyecto es descargado cuando ejecutas `git clone`.

```
vicente@ubuntupc:~/NetBeansProjects$ git clone https://github.com/pH-7/Simple-Java-Calculator.git
Clonar en «Simple-Java-Calculator»...
remote: Counting objects: 144, done.
remote: Total 144 (delta 0), reused 0 (delta 0), pack-reused 144
Receiving objects: 100% (144/144), 110.17 KiB | 129.00 KiB/s, done.
Resolving deltas: 100% (49/49), done.
Comprobando la conectividad... hecho.
vicente@ubuntupc:~/NetBeansProjects$ ls
JavaApplication1  Simple-Java-Calculator
vicente@ubuntupc:~/NetBeansProjects$
```

Esto crea un directorio llamado "Simple-Java-Calculator", inicializa un directorio .git en su interior, descarga toda la información de ese repositorio, y saca una copia de trabajo de la última versión. Si te metes en el nuevo directorio "Simple-Java-Calculator", verás que están los archivos del proyecto, listos para ser utilizados.

```
vicente@ubuntupc:~/NetBeansProjects$ cd Simple-Java-Calculator/
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ ls
build.xml  license.txt  manifest.mf  nbproject  README.md  Screenshots  SimpleJavaCalculator.jar  src
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ cd src
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator/src$ ls
simplejavacalculator
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator/src$ cd simplejavacalculator/
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator/src/simplejavacalculator$ ls
Calculator.java  SimpleJavaCalculator.java  UI.java
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator/src/simplejavacalculator$
```

2.2 Fundamentos de Git - Guardando cambios en el repositorio

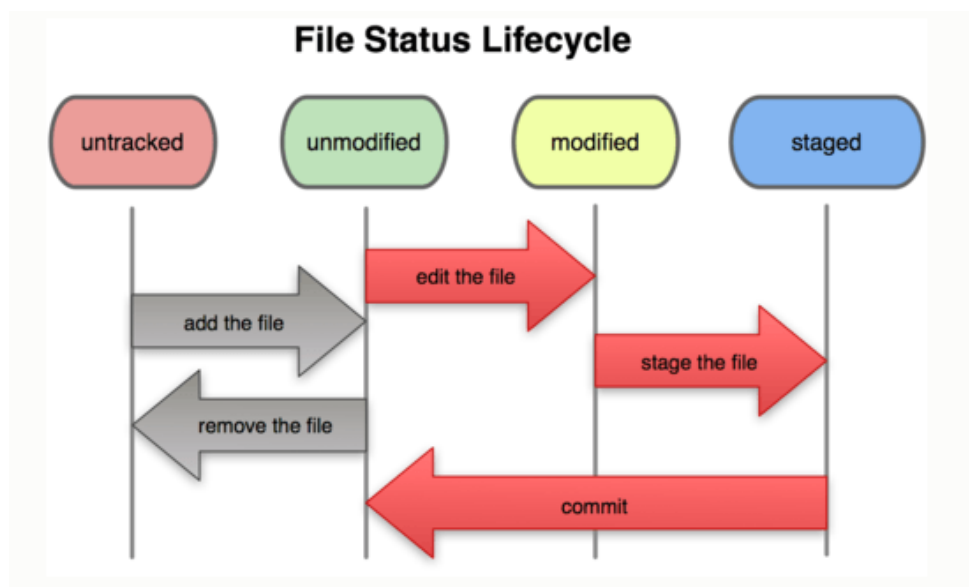
Guardando cambios en el repositorio

Tienes un repositorio Git completo, y una copia de trabajo de los archivos de ese proyecto. Necesitas hacer algunos cambios, y confirmar instantáneas de esos

cambios a tu repositorio cada vez que el proyecto alcance un estado que desees grabar.

Recuerda que cada archivo de tu directorio de trabajo puede estar en uno de estos dos estados: bajo seguimiento (tracked), o sin seguimiento (untracked). Los archivos bajo seguimiento son aquellos que existían en la última instantánea; pueden estar sin modificaciones, modificados, o preparados. Los archivos sin seguimiento son todos los demás.

A medida que editas archivos, Git los ve como modificados, porque los has cambiado desde tu última confirmación. Preparas estos archivos modificados y luego confirmas todos los cambios que hayas preparado, y el ciclo se repite.



[Comprobando el estado de tus archivos](#)

Tu principal herramienta para determinar qué archivos están en qué estado es el comando `git status`. Si ejecutas este comando justo después de clonar un repositorio, deberías ver algo así:

```
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git status
En la rama master
Su rama está actualizada con «origin/master».
Cambios no preparados para el commit:
  (use «git add <archivo>...» para actualizar lo que se confirmará)
  (use «git checkout -- <archivo>...» para descartar cambios en el directorio de trabajo)

    modificado:   nbproject/private/private.properties

no hay cambios agregados al commit (use «git add» o «git commit -a»)
```

Aquí vemos que tenemos un archivo modificado, no hay archivos bajo seguimiento.
Estas en la rama master, que es la predeterminada

Digamos que añades un nuevo archivo a tu proyecto, un sencillo archivo README. Si el archivo no existía y ejecutas `git status`, verás tus archivos sin seguimiento así:

```
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git status
En la rama master
Su rama está actualizada con «origin/master».
Cambios no preparados para el commit:
  (use «git add <archivo>...» para actualizar lo que se confirmará)
  (use «git checkout -- <archivo>...» para descartar cambios en el directorio de trabajo)

    modificado:   nbproject/private/private.properties

Archivos sin seguimiento:
  (use «git add <archivo>...» para incluir en lo que se ha de confirmar)

    README
```

Sin seguimiento (Untracked Files) significa básicamente que Git ve un archivo que no estaba en la instantánea anterior; Git no empezará a incluirlo en las confirmaciones de tus instantáneas hasta que se lo indiques explícitamente.

Seguimiento de nuevos archivos

Para empezar el seguimiento de un nuevo archivo se usa el comando `git add`.

Iniciaremos el seguimiento del archivo README ejecutando esto:

```
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git add README
```

Si vuelves a ejecutar el comando `git status`, verás que tu README está ahora bajo seguimiento y preparado:

```
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git status
En la rama master
Su rama está actualizada con «origin/master».
Cambios para hacer commit:
  (use «git reset HEAD <archivo>...» para sacar del stage)

    nuevo archivo: README

Cambios no preparados para el commit:
  (use «git add <archivo>...» para actualizar lo que se confirmará)
  (use «git checkout -- <archivo>...» para descartar cambios en el directorio de trabajo)

    modificado:   nbproject/private/private.properties
```

Puedes ver que está preparado porque aparece bajo la cabecera “Cambios a confirmar” (“Changes to be committed”). Si confirmas ahora, la versión del archivo en el momento de ejecutar `git add` será la que se incluya en la instantánea.

El comando `git add` recibe la ruta de un archivo o de un directorio; si es un directorio, añade todos los archivos que contenga de manera recursiva.

Preparando archivos modificados

En la imagen anterior vemos que hay un archivo, *private.properties*, aparece bajo la cabecera “Cambios no preparados para el Commit” (“Changes not staged for commit”) —esto significa que un archivo bajo seguimiento ha sido modificado en el directorio de trabajo, pero no ha sido preparado todavía

Para prepararlo, ejecuta el comando `git add` para preparar el archivo *private.properties* y volvemos a ejecutar `git status`:

```
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git add nbproject/private/private.properties
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git status
En la rama master
Su rama está actualizada con «origin/master».
Cambios para hacer commit:
  (use «git reset HEAD <archivo>...» para sacar del stage)

    nuevo archivo: README
    modificado:   nbproject/private/private.properties

vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$
```

Si modificas un archivo después de haber ejecutado `git add`, tendrás que volver a ejecutar `git add` para preparar la última versión del archivo:

En el siguiente ejemplo se ilustra el caso. Modificamos el README y vemos que sale en cambios para hacer el commit y cambios no preparados para hacer el COMMIT. Tenemos que ejecutar un `git add README` para que desaparezca de cambios no preparados para hacer el COMMIT:

```

vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ echo 'System.out.println("Nova edicio");' >> README
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ cat README
prova
System.out.println("Nova edicio");
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git status
En la rama master
Su rama está actualizada con «origin/master».
Cambios para hacer commit:
  (use «git reset HEAD <archivo>...» para sacar del stage)

    nuevo archivo: README
    modificado:    nbproject/private/private.properties

Cambios no preparados para el commit:
  (use «git add <archivo>...» para actualizar lo que se confirmará)
  (use «git checkout -- <archivo>...» para descartar cambios en el directorio de trabajo)

    modificado:    README

vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git add README
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git status
En la rama master
Su rama está actualizada con «origin/master».
Cambios para hacer commit:
  (use «git reset HEAD <archivo>...» para sacar del stage)

    nuevo archivo: README
    modificado:    nbproject/private/private.properties
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$

```

Para ver lo que has modificado pero aún no has preparado, escribe *git diff*:

En este caso si que vemos las líneas que hemos modificado:

Si añadimos una nueva línea en el archivo *nbproject/private/private.properties* podemos observar los cambios en verde.

```

vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git diff
diff --git a/nbproject/private/private.properties b/nbproject/private/private.properties
index 95be30b..8072d0c 100644
--- a/nbproject/private/private.properties
+++ b/nbproject/private/private.properties
@@ -1,2 +1,3 @@
 compile.on.save=true
 user.properties.file=/home/vicente/.netbeans/8.2/build.properties
+user.properties.file=/home/vicente/prova
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$

```

Ese comando compara lo que hay en tu directorio de trabajo con lo que hay en tu área de preparación. El resultado te indica los cambios que has hecho y que todavía no has preparado.

Si quieres ver los cambios que has preparado y que irán en tu próxima confirmación, puedes usar `git diff --cached`. (A partir de la versión 1.6.1 de Git, también puedes usar `git diff --staged`).

```
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..833bf1a
--- /dev/null
+++ b/README
@@ -0,0 +1,2 @@
+prova
+System.out.println("Nova edicio");
diff --git a/nbproject/private/private.properties b/nbproject/private/private.properties
index 80a1a4e..95be30b 100644
--- a/nbproject/private/private.properties
+++ b/nbproject/private/private.properties
@@ -1,2 +1,2 @@
 compile.on.save=true
-user.properties.file=/Users/archieg/Library/Application Support/NetBeans/8.0.2/build.properties
+user.properties.file=/home/vicente/.netbeans/8.2/build.properties
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$
```

En la imagen anterior podemos observar los cambios que hemos realizado desde que nos descargamos el proyecto en verde.

Volvemos a poner todos los archivos en el area de preparacion o cambios para hacer commit:

```
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git add nbproject/private/private.properties
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git status
En la rama master
Su rama está actualizada con «origin/master».
Cambios para hacer commit:
  (use «git reset HEAD <archivo>...» para sacar del stage)

    nuevo archivo: README
    modificado:     nbproject/private/private.properties
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$
```

Y observaremos que no ahora no hay ninguna diferencia:

```
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git diff
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$
```

Confirmando tus cambios

Ahora que el área de preparación está como tú quieres, puedes confirmar los cambios. Recuerda que cualquier cosa que todavía esté sin preparar —cualquier archivo que hayas creado o modificado, y sobre el que no hayas ejecutado `git add` desde su última edición— no se incluirá en esta confirmación. Se mantendrán como modificados en tu disco.

```
$ git commit
```

Para un recordatorio todavía más explícito de lo que has modificado, puedes pasar la opción `-v` a `git commit`.

¡Acabas de crear tu primera confirmación! Puedes ver que el comando `commit` ha dado cierta información sobre la confirmación: a qué rama has confirmado (master), cuál es su suma de comprobación SHA-1 de la confirmación (463dc4f), cuántos archivos se modificaron, y estadísticas acerca de cuántas líneas se han añadido y cuántas se han eliminado.

Como alternativa, puedes escribir tu mensaje de confirmación desde la propia línea de comandos mediante la opción `-m`:

```
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git commit -m "first commit"
[master a2c7959] first commit
2 files changed, 4 insertions(+), 1 deletion(-)
create mode 100644 README
```

Eliminando archivos

Imaginar que queremos borrar un archivo. Por ejemplo el README que se encuentra en el area de stage:

```
Cambios para hacer commit:
(use «git reset HEAD <archivo>...» para sacar del stage)

    modificado:    README
```

Si lo borramos:

```
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git status
En la rama master
Su rama está delante de «origin/master» para 1 commit.
(use "git push" to publish your local commits)
Cambios para hacer commit:
(use «git reset HEAD <archivo>...» para sacar del stage)

    modificado:    README

Cambios no preparados para el commit:
(use «git add/rm <archivo>...» para actualizar lo que se confirmará)
(use «git checkout -- <archivo>...» para descartar cambios en el directorio de trabajo)

    borrado:       README
```

Como podemos observar no irá al area stage hasta que no ejecutemos el siguiente comando:

```
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git rm README
rm 'README'
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git status
En la rama master
Su rama está delante de «origin/master» para 1 commit.
(use "git push" to publish your local commits)
Cambios para hacer commit:
(use «git reset HEAD <archivo>...» para sacar del stage)

    borrado:       README
```

Si confirmamos se perderá

Para recuperlo completamente y volver a tenerlo en el area de trabajo, haríamos un:

```
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git reset HEAD README
Unstaged changes after reset:
D      README
```

Y un:

```
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git checkout README
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ ls
build      license.txt  nbproject  README.md  SimpleJavaCalculator.jar
build.xml  manifest.mf  README     Screenshots  src
```

Moviendo archivos

```
$ git mv README.txt README
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    README.txt -> README
```

2.3 Fundamentos de Git - Viendo el histórico de confirmaciones

Antes de nada vamos a clonar un proyecto para demostración:

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

Si quieres mirar atrás para ver qué modificaciones se han llevado a cabo

```
vicente@ubuntupc:~/NetBeansProjects/simplegit-progit$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the verison number

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

    removed unnecessary test code

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

    first commit
```

Como puedes ver, este comando lista cada confirmación con su suma de comprobación SHA-1, el nombre y dirección de correo del autor, la fecha y el mensaje de confirmación.

Una de las opciones más útiles es `-p`, que muestra las diferencias introducidas en cada confirmación. También puedes usar la opción `-2`, que hace que se muestren únicamente las dos últimas entradas del histórico:

```
vicente@ubuntupc:~/NetBeansProjects/simplegit-progit$ git log -p -2
```

Como puedes ver, la opción `--stat` imprime tras cada confirmación una lista de archivos modificados, indicando cuántos han sido modificados y cuántas líneas han

sido añadidas y eliminadas para cada uno de ellos, y un resumen de toda esta información.

```
vicente@ubuntupc:~/NetBeansProjects/simplegit-progit$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the verison number

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

La opción más interesante es `format`, que te permite especificar tu propio formato:

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

```
vicente@ubuntupc:~/NetBeansProjects/simplegit-progit$ git log --pretty=format:"%h - %an, %ar - %s"
ca82a6d - Scott Chacon, hace 10 años - changed the verison number
085bb3b - Scott Chacon, hace 10 años - removed unnecessary test code
a11bef0 - Scott Chacon, hace 10 años - first commit
```

Modificando tu última confirmación

Por ejemplo, si confirmas y luego te das cuenta de que se te olvidó preparar los cambios en uno de los archivos que querías añadir, puedes hacer algo así:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```


2.5 Fundamentos de Git - Trabajando con repositorios remotos

Los repositorios remotos son versiones de tu proyecto que se encuentran alojados en Internet o en algún punto de la red. Puedes tener varios, cada uno de los cuales puede ser de sólo lectura, o de lectura/escritura, según los permisos que tengas

Vamos a clonar nuestro repositorio que creamos en github

`https://github.com/vpulido2/dawbio_calculator.git`

Mostrando tus repositorios remotos

Para ver qué repositorios remotos tienes configurados, puedes ejecutar el comando `git remote` o `git remote -v`

```
vicente@ubuntupc:~/NetBeansProjects/dawbio_calculator$ git remote
origin
vicente@ubuntupc:~/NetBeansProjects/dawbio_calculator$ git remote -v
origin https://github.com/vpulido2/dawbio_calculator.git (fetch)
origin https://github.com/vpulido2/dawbio_calculator.git (push)
vicente@ubuntupc:~/NetBeansProjects/dawbio_calculator$
```

"origin" —es el nombre predeterminado que le da Git al servidor del que clonaste—:

Añadiendo repositorios remotos

```
$ git remote  
origin  
$ git remote add pb git://github.com/paulboone/ticgit.git  
$ git remote -v  
origin  git://github.com/schacon/ticgit.git  
pb      git://github.com/paulboone/ticgit.git
```

Ahora puedes usar la cadena "pb" en la línea de comandos, en lugar de toda la URL. Por ejemplo, si quieres recuperar toda la información de Paul que todavía no tienes en tu repositorio, puedes ejecutar:

```
$ git fetch pb
```

Recibiendo de tus repositorios remotos

Este comando recupera todos los datos del proyecto remoto que no tengas todavía. Después de hacer esto, deberías tener referencias a todas las ramas del repositorio remoto, que puedes unir o inspeccionar en cualquier momento

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

Si clonas un repositorio, el comando añade automáticamente ese repositorio remoto con el nombre de "origin". Por tanto, `git fetch origin` recupera toda la información enviada a ese servidor desde que lo clonaste (o desde la última vez que ejecutaste `fetch`).

Es importante tener en cuenta que el comando `fetch` sólo recupera la información y la pone en tu repositorio local —no la une automáticamente con tu trabajo ni modifica aquello en lo que estás trabajando.

Al ejecutar `git pull`, por lo general se recupera la información del servidor del que clonaste, y automáticamente se intenta unir con el código con el que estás trabajando actualmente.

```
vicente@ubuntupc:~/NetBeansProjects/dawbio_calculator$ git pull
Already up-to-date.
vicente@ubuntupc:~/NetBeansProjects/dawbio_calculator$
```

[Enviando a tus repositorios remotos](#)

Cuando tu proyecto se encuentra en un estado que quieres compartir, tienes que enviarlo a un repositorio remoto. El comando que te permite hacer esto es sencillo: `git push [nombre-remoto][nombre-rama]`. Si quieres enviar tu rama maestra (`master`) a tu servidor origen (`origin`), ejecutarías esto para enviar tu trabajo al servidor:

```
vicente@ubuntupc:~/NetBeansProjects/dawbio_calculator$ git push origin master
Username for 'https://github.com': vpulido2
Password for 'https://vpulido2@github.com':
Everything up-to-date
vicente@ubuntupc:~/NetBeansProjects/dawbio_calculator$
```

Imaginar que no hemos creado el archivo README.rd en nuestro repositorio. Lo primero de todo crearemos un nuevo repositorio en GitHub. Una vez creado enviaremos nuestro propio archivo README:

```
1) echo "# dawbio_calculator" >> README.md
2) git init
3) git add README.md
4) git commit -m "first commit"
5) git remote add origin https://github.com/vpulido2/nuevo_repositorio.git
6) git push -u origin master
```

Inspeccionando un repositorio remoto

Si quieres ver más información acerca de un repositorio remoto en particular, puedes usar el comando `git remote show [nombre]`

```
vicente@ubuntupc:~/NetBeansProjects/Simple-Java-Calculator$ git remote show origin
* remote origin
Fetch URL: https://github.com/pH-7/Simple-Java-Calculator.git
Push URL: https://github.com/pH-7/Simple-Java-Calculator.git
HEAD branch: master
Remote branch:
  master tracked
```

2.6 Fundamentos de Git - Creando etiquetas

Creando etiquetas

Como muchos VCSs, Git tiene la habilidad de etiquetar (tag) puntos específicos en la historia como importantes.

Listando tus etiquetas

Listar las etiquetas disponibles en Git es sencillo, Simplemente escribe `git tag`:

```
$ git tag
```

Creando etiquetas

Se recomienda usar las etiquetas anotadas .Tienen:

- suma de comprobación
- contienen el nombre del etiquetador
- correo electrónico y fecha
- tienen mensaje de etiquetado

Generalmente se recomienda crear etiquetas anotadas para disponer de toda esta información

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

Puedes ver los datos de la etiqueta junto con la confirmación que fue etiquetada usando el comando `git show`:

```
$ git show v1.4
```

Etiquetando más tarde

Puedes incluso etiquetar confirmaciones después de avanzar sobre ellas. Supón que tu historico de confirmaciones se parece a esto:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fc9bc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Ahora, supón que olvidaste etiquetar el proyecto en v1.2, que estaba en la confirmación "updated rakefile". Puedes hacerlo ahora. Para etiquetar esa confirmación específica la suma de comprobación de la confirmación (o una parte de la misma) al final del comando:

```
$ git tag -a v1.2 -m 'version 1.2' 9fceb02
```

Puedes ver que has etiquetado la confirmación:

```
$ git tag
```

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

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

    updated rakefile
```

Ignorando archivos

A menudo tendrás un tipo de archivos que no quieras que Git añada automáticamente o te muestre como no versionado. Suelen ser archivos generados automáticamente, como archivos de log, o archivos generados por tu compilador.

Para estos casos puedes crear un archivo llamado `.gitignore`, en el que listas los patrones de nombres que deseas que sean ignorados.

```
$ cat .gitignore
*.lo
*~
```

La primera línea le dice a Git que ignore cualquier archivo cuyo nombre termine en `.lo` o `.a` —archivos objeto que suelen ser producto de la compilación de código—.

La segunda línea le dice a Git que ignore todos los archivos que terminan en tilde (`~`), usada por muchos editores de texto, como Emacs, para marcar archivos temporales.

He aquí otro ejemplo de archivo .gitignore:

```
# a comment - this is ignored
# no .a files
*.a
# but do track lib.a, even though you're ignoring .a files above
!lib.a
# only ignore the root TODO file, not subdir/TODO
/TODO
# ignore all files in the build/ directory
build/
# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt
# ignore all .txt files in the doc/ directory
doc/**/*.txt
```


Annexo 1

4.9 Git en un servidor - Git en un alojamiento externo

GitHub

GitHub contempla espacios de nombres para los proyectos. En lugar de estar focalizado en los proyectos, GitHub gira en torno a los usuarios. Esto significa que, cuando alojo mi proyecto 'grit' en GitHub, no lo encontraras bajo 'github.com/grit', sino bajo 'github.com/schacon/grit'

Para albergar proyectos públicos de código abierto, cualquiera puede crear una cuenta gratuita. Vamos a ver cómo hacerlo.

Configurando una cuenta de usuario

1.- Nos vamos a :

<https://github.com/pricing>

2.- Clicas sobre el botón "Registro" ("Sign Up")

3.- Rellenamos el formulario con un : user, mail y passwd

Create your personal account

Username

vpulido2



This will be your username — you can enter your organization's username next.

Email Address

vpulido13@gmail.com



You will occasionally receive account related emails. We promise not to share your email with anyone.

Password

.....



Use at least one lowercase letter, one numeral, and seven characters.


By clicking on "Create an account" below, you are agreeing to the [Terms of Service](#) and the [Privacy Policy](#).

4.- Clicamos create account

5.- Seleccionamos la primera opción y 

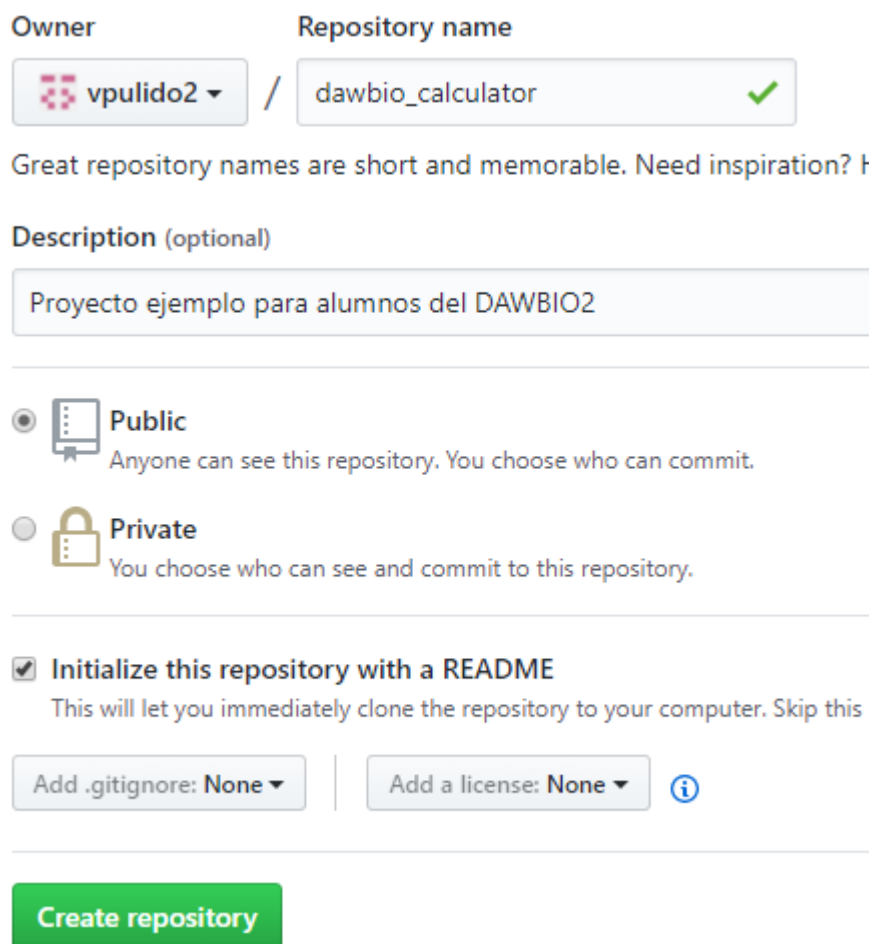
- ☒ Unlimited public repositories for free.
- ☐ Unlimited private repositories for \$7/month.

Don't worry, you can cancel or upgrade at any time.

6.- Creamos un nuevo repositorio desplegando el boton  que encontramos arriba a la derecha.

7.- Verificamos el mail entrando con nuestro user y passwd y después actualizamos la página para ver el formulario New Repository

8.- Insertamos el nombre del repositorio y si queremos inicializamos el repositorio con un README. A continuación Create Repository



The screenshot shows the GitHub 'Create repository' form. At the top, there are two input fields: 'Owner' with a dropdown menu showing 'vpulido2' and a 'Repository name' field containing 'dawbio_calculator' with a green checkmark. Below these is a hint: 'Great repository names are short and memorable. Need inspiration?'. The 'Description (optional)' field contains 'Proyecto ejemplo para alumnos del DAWBIO2'. Under the 'Visibility' section, 'Public' is selected with a radio button, and 'Private' is unselected. Below this, the checkbox 'Initialize this repository with a README' is checked. At the bottom, there are two dropdown menus: 'Add .gitignore: None' and 'Add a license: None', followed by an information icon. A large green 'Create repository' button is at the bottom.

Owner: vpulido2 / Repository name: dawbio_calculator ✓

Great repository names are short and memorable. Need inspiration? [Learn more](#)

Description (optional): Proyecto ejemplo para alumnos del DAWBIO2

☒ Public
Anyone can see this repository. You choose who can commit.


☐ Private
You choose who can see and commit to this repository.

☒ Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this

Add .gitignore: None | Add a license: None ⓘ

[Create repository](#)

9.- Si ponemos atención veremos un botón [Clone or download](#) que si lo desplegamos podemos ver la URL http que necesitamos para clonar nuestro proyecto (solo lectura) En nuestro caso vemos que el path del proyecto se forma concatenando el propietario del proyecto + nombre + .git, dentro del dominio github.com

Owner	Repository name
 vpulido2 ▼	/ dawbio_calculator ✓

Obteniendo la URL al proyecto: https://github.com/vpulido2/dawbio_calculator.git

10.- También tenemos la opción de usar ssh con la siguiente URL pero para lectura y escritura: `git@github.com:vpulido2/dawbio_calculator.git`

11.- Ahora queremos inicializar un nuevo proyecto y enviarlo al servidor github. Pues bien, lo primero que tenemos que hacer, ya que lo que vamos a hacer es escribir en el mismo, es configurar una clave pública/privada ssh. Seguir **Annexo 2**

12.- Si hubieramos realizado correctamente los pasos del **annexo 2** y comprobado nuestra conexión ssh ya podríamos inicializar nuestro proyecto local. A continuación se muestran los pasos:

Figura 4-7. Instrucciones para un nuevo repositorio.

Estas instrucciones son similares a las que ya hemos visto. Para inicializar un proyecto, no siendo aún un proyecto Git, sueles utilizar:

```
$ git init
$ git add .
$ git commit -m 'initial commit'
```

Una vez tengas un repositorio local Git, añádele el sitio GitHub como un remoto y envía (push) allí tu rama principal:

```
$ git remote add origin git@github.com:testinguser/iphone_project.git
$ git push origin master
```

Annexo 2

Como crear una clave pública/privada con ssh para poder escribir en el servidor github.

<https://help.github.com/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/#platform-linux>.

<https://help.github.com/articles/adding-a-new-ssh-key-to-your-github-account/#platform-linux>

<https://help.github.com/articles/testing-your-ssh-connection/>