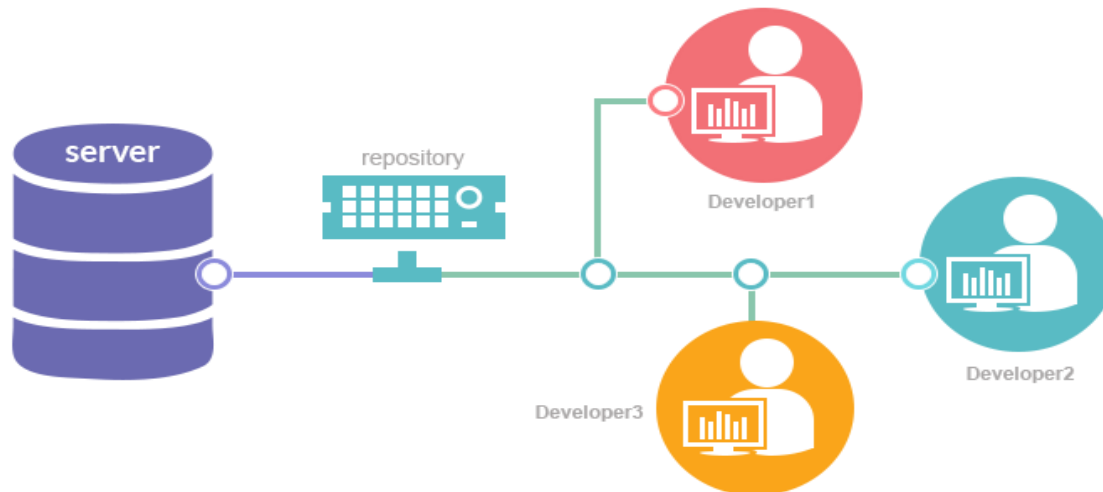


UT3 – CONTROL DE VERSIONES

Control de versiones

2

- “Se llama **control de versiones** a la gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración del mismo” (Wikipedia)
- Un **sistema de control de versiones (SCV)** es un programa que permiten gestionar un repositorio de archivos y sus distintas versiones
- Utilizan una **estructura cliente – servidor** para administrar el elemento principal: el **repositorio**



¿Por qué utilizar control de versiones?

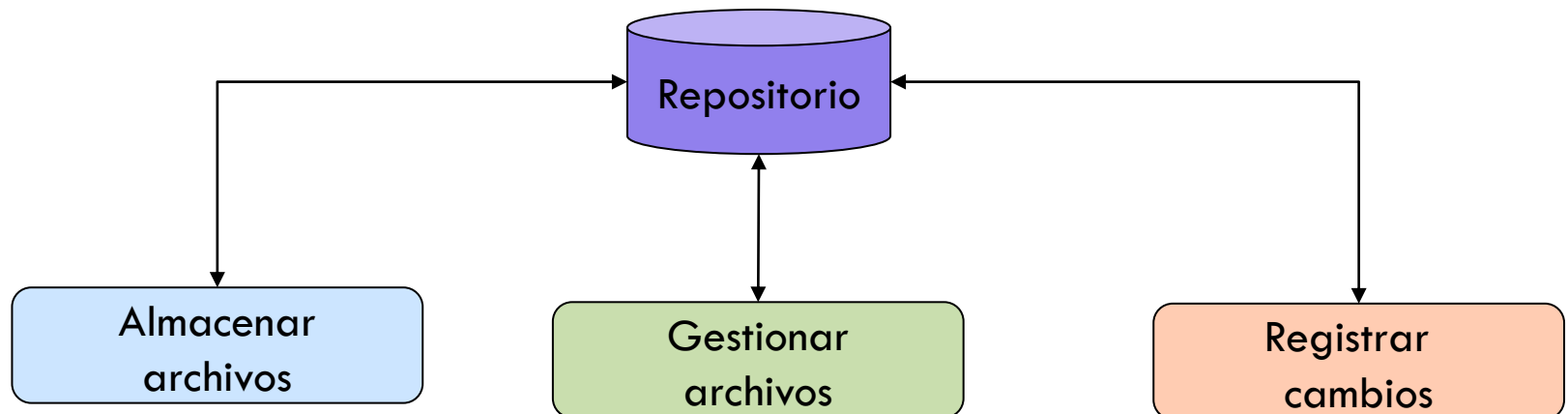
3

- Existen dos situaciones para trabajar con control de versiones:
 - ▣ **Desarrollo de un proyecto de manera individual** y se quiere tener un lugar donde poder tener un control de todos los cambios que se han ido realizando. De este modo si en un momento dado, algo deja de funcionar, podemos volver a la última versión que funcionaba correctamente.
 - ▣ **Desarrollo colaborativo de un proyecto:** más de una persona, trabajando sobre el mismo proyecto. Permite tener de una manera centralizada, la versión “correcta” y actualizada del código, así cómo un control de todos los cambios realizados.

¿Qué podemos hacer?

4

- Varios clientes pueden sacar copias del proyecto al mismo tiempo
- Realizar cambios a los ficheros manteniendo un histórico de los cambios
 - ▣ Deshacer los cambios hechos en un momento dado
 - ▣ Recuperar versiones pasadas
 - ▣ Ver históricos de cambios y comentarios
 - ▣ Los clientes pueden también comparar diferentes versiones de archivos
- Unir cambios realizados por diferentes usuarios sobre los mismos ficheros
- Actualizar una copia local con la última versión que se encuentra en el servidor
 - ▣ Esto elimina la necesidad de repetir las descargas del proyecto completo
- Mantener distintas ramas de un proyecto



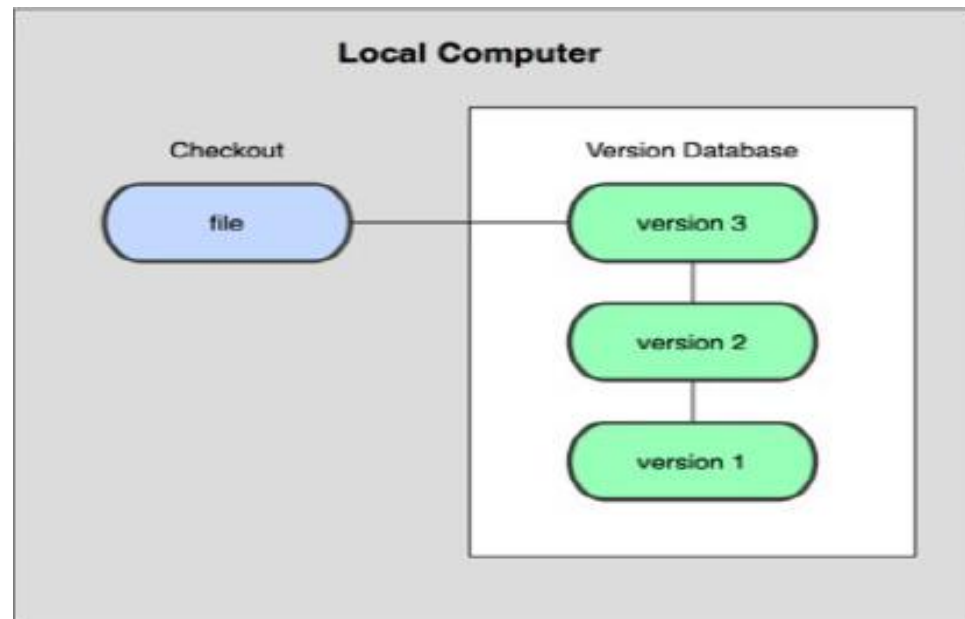
Clasificación de los SCV

5

- Podemos clasificar los sistemas de control de versiones según la arquitectura para almacenar la información en **locales**, **centralizados** o **distribuidos**.

Locales

- La información se guarda en un ordenador o repositorio local con lo que no sirve para trabajar en forma colaborativa.
- Ejemplo: RCS (Revision Control System).

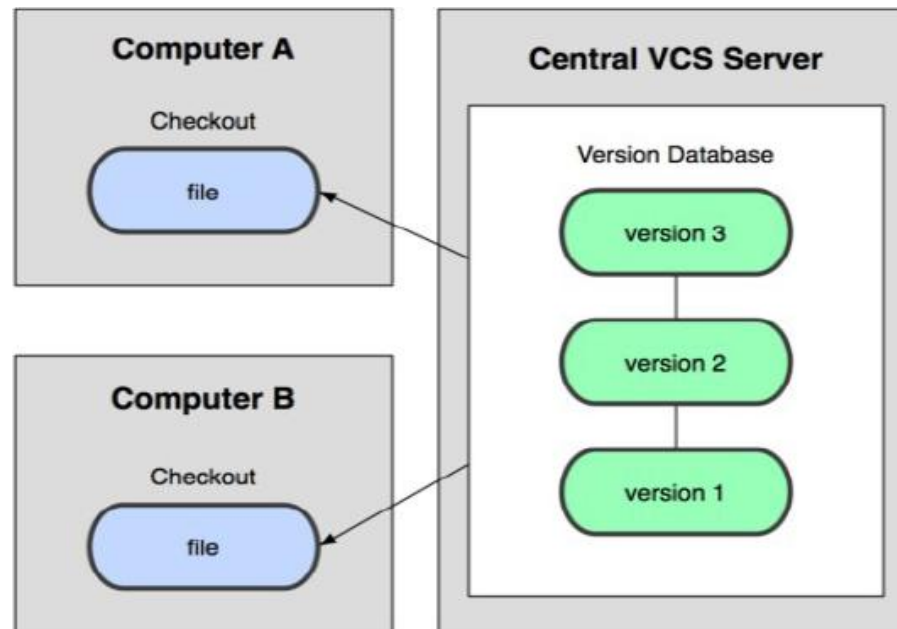


Clasificación de los SCV

6

Centralizados o CVCS (Centralized Version Control System)

- La información se guarda en un servidor dentro de un repositorio centralizado.
- Existe un usuario o usuarios responsables con capacidad de realizar tareas administrativas a cambio de reducir flexibilidad, necesitan la aprobación del responsable para realizar acciones, como crear una rama nueva.
- Ejemplos: *Subversion* y CVS.

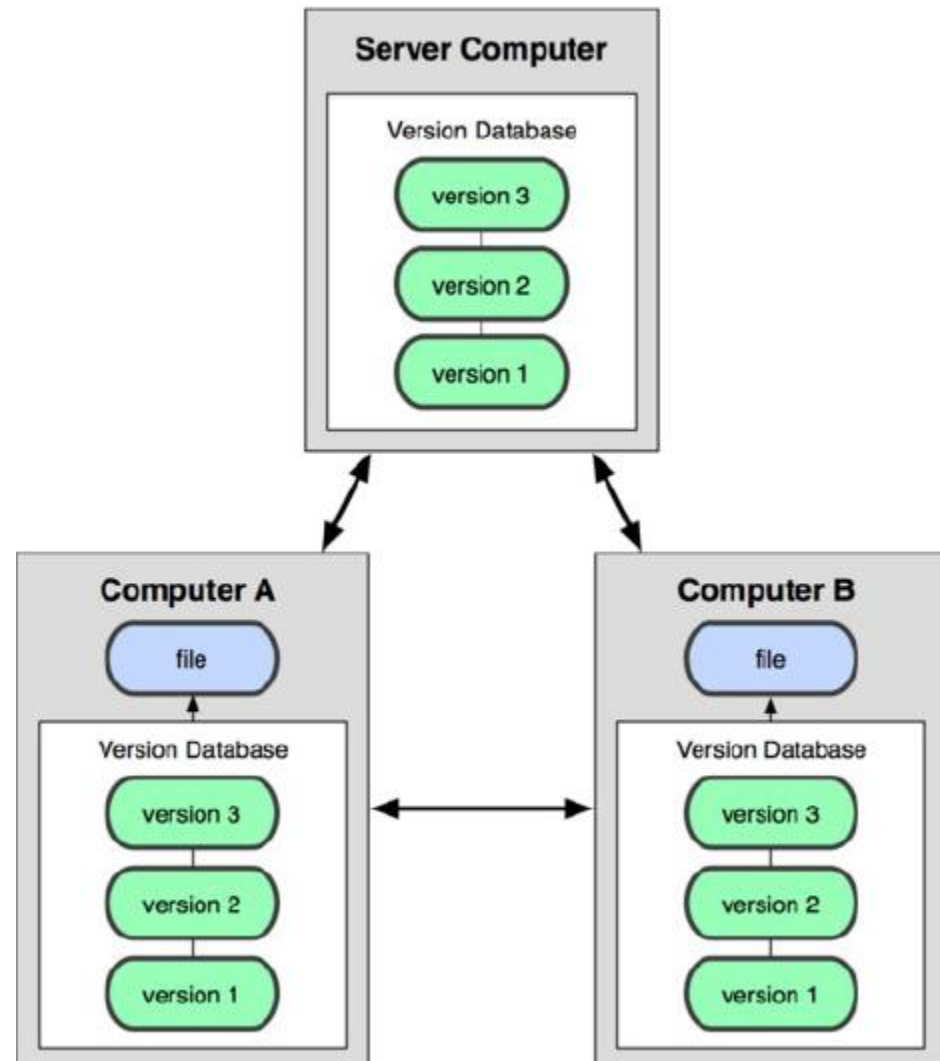


Clasificación de los SCV

7

Distribuidos o DVCS (Distributed Version Control System)

- Cada usuario tiene su propio repositorio.
- Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos.
- Es frecuente el uso de un repositorio, que está normalmente disponible, que sirve de punto de sincronización de los distintos repositorios locales.
- Ejemplos: *Git* y Mercurial



Subversion



8

- **Subversion** es un sistema de control de versiones con licencia OpenSource Apache.
- Utiliza una estructura cliente – servidor y es un sistema centralizado.
- Características:
 - ▣ Facilita el trabajo colaborativo.
 - ▣ Entrega historial de revisiones y copias.
 - ▣ Puede ser visto en una páginas web y hacer correcciones 'on the fly'.
 - ▣ Permite manejar varios tipos de archivos.
 - ▣ Manejo de branches y tags.

Conceptos principales en Subversion

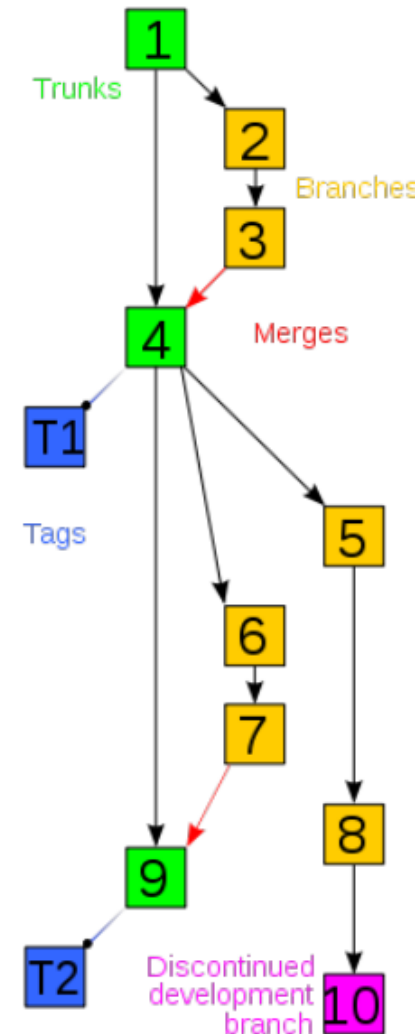
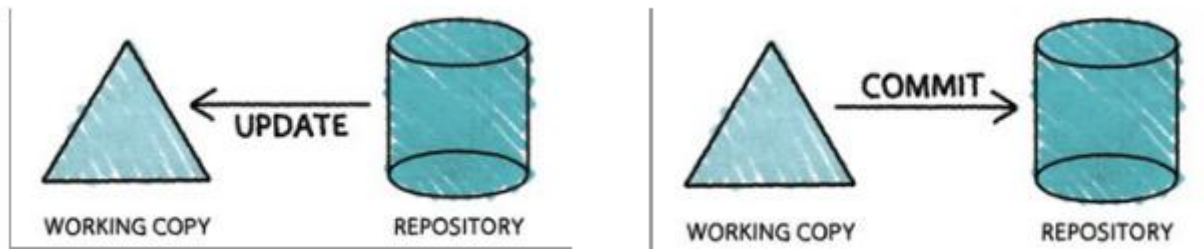
9

- **Repositorio:** lugar en el que se almacenan los datos actualizados e históricos de cambios (sistema de archivos en un disco duro, un banco de datos, etc).
- **Trunk:** rama de desarrollo principal
- **Revisión:** versión determinada de la información que se gestiona.
- **Tags:** rama de gestión de versiones. Reservado para versiones cerradas, por tanto no se desarrollará sobre esta rama. Permiten identificar de forma fácil revisiones importantes en el proyecto.
- **Working copy:** copia local del proyecto sobre el cual estamos trabajando
- **Conflicto:** cuando se han modificado por parte de dos clientes distintos un mismo trozo de código
 - A veces el conflicto lo arregla el sistema de control de versiones, y otras veces tiene que intervenir uno de los usuarios.

Operaciones principales en Subversion

10

- ❑ **Import:** primera vez que subimos el proyecto al repositorio. Sólo se realiza una vez
- ❑ **Checkout:** crea una copia de trabajo local desde el repositorio.
- ❑ **Update:** integra los cambios que han sido realizados en el repositorio en la copia de trabajo local.
- ❑ **Commit:** consiste en realizar un cambio local en el proyecto y luego almacenar dicho cambio en el repositorio.
- ❑ **Branch:** es una copia del proyecto aislada, de forma que los cambios realizados no afecten al resto del proyecto y viceversa.
- ❑ **Merge:** une dos grupos de cambios en un archivo (o grupo de archivos), generando una revisión unificada.



Operaciones habituales en Subversion

11

TRABAJO EN EQUIPO

- Situación en la que, al menos, dos personas modifican el código.
- En Subversion se permite modificación paralela.
 - ▣ Otras herramientas bloquean el código.
 - ▣ Si dos desarrolladores modificasen el mismo elemento a la vez, Subversion integrará los cambios de forma automática (si puede)
- ¿Cómo sincronizar?
 - ▣ Tres formas de sincronizar el código del entorno local con el del repositorio
 - Checkout
 - Update
 - Commit
- Dinámica de trabajo. Prácticas recomendadas:
 - ▣ Antes de comenzar: conseguir la última versión: **update** o **checkout**
 - ▣ Una vez resuelta la tarea: hacer otro **update** por si hay cambios al desarrollo actual.
 - ▣ Finalmente, se hace un **commit**.

- Git es un sistema de control de versiones distribuido y gratuito
- Fue diseñado por **Linus Torvalds** pensando en la eficiencia y confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos
- Basado en línea de órdenes... pero con clientes para quién no les guste 😊

Comandos básicos

13

□ Configuración

- Usado para establecer una configuración específica de usuario, como sería el caso del email, nombre de usuario y tipo de formato, etc...

- **git config**

- **git config --global user.email gmurod01@educantabria.es**

- **git config --global user.name "Gema Muro"**

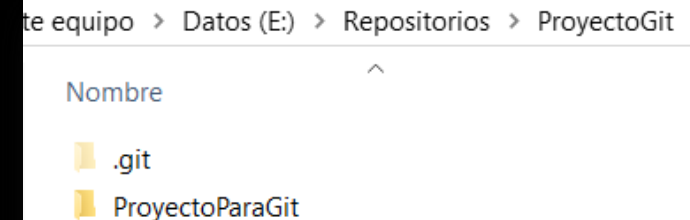
□ Crear un repositorio nuevo

- Si queremos iniciar el seguimiento en Git de un proyecto existente, vamos al directorio del proyecto y escribimos:

- **git init**

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

```
Iván@DESKTOP-7SCINFC MINGW64 ~  
$ cd E:/Repositorios/ProyectoGit  
  
Iván@DESKTOP-7SCINFC MINGW64 /e/Repositorios/ProyectoGit  
$ dir  
ProyectoParaGit  
  
Iván@DESKTOP-7SCINFC MINGW64 /e/Repositorios/ProyectoGit  
$ git init  
Initialized empty Git repository in E:/Repositorios/ProyectoGit/.git/  
  
Iván@DESKTOP-7SCINFC MINGW64 /e/Repositorios/ProyectoGit (master)  
$ |
```



Nombre
.git
ProyectoParaGit

Comandos básicos

14

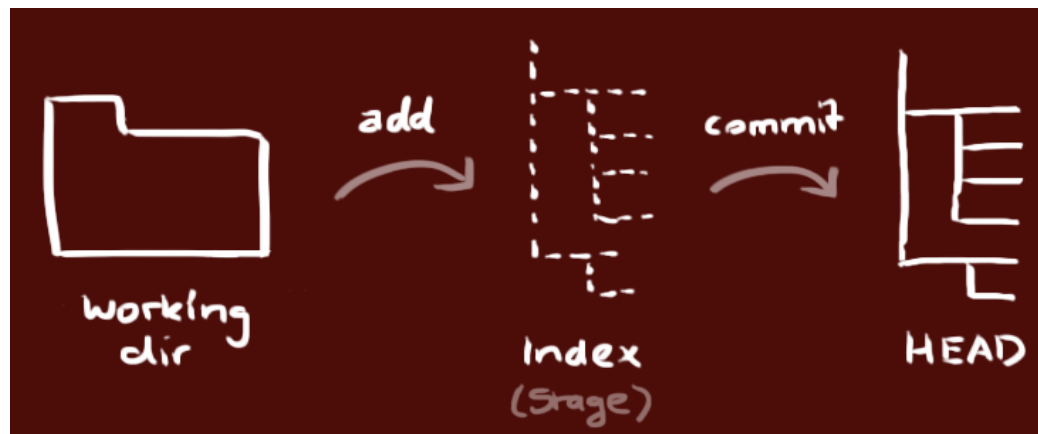
- **Clonar un repositorio**
 - **git clone /path/to/repository**
 - **git clone username@host:/path/to/repository** (en caso de usar un repositorio remoto)
 - Se obtienen los ficheros de trabajo (código) y toda la información de control de versiones (historia)
 - Para obtener una copia de un repositorio Git existente (por ejemplo, un proyecto en el que nos gustaría contribuir)
 - Al contrario que con git init, con git clone no es necesario crear un directorio para el proyecto. Al clonar se creará un directorio con el nombre del proyecto dentro del que te encuentres al llamar a la orden.
 - A diferencia de Subversion, el comando es clone y no checkout.

```
Iván@DESKTOP-7SCINF6 MINGW64 /e/Repositorios
$ git clone https://gitlab.com/ivanlorenzo/repo-ejemplo.git
Cloning into 'repo-ejemplo'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 9 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (9/9), done.
```

Flujo de trabajo en git

15

- Tu repositorio local esta compuesto por tres "árboles" administrados por git.
 - ▣ El primero es tu **Directorio de trabajo** que contiene los archivos, los modificas, los borras, creas nuevos, etc.
 - Git sabe que tiene que controlar ese directorio, pero no lo hace hasta que se lo digas expresamente
 - ▣ El segundo es el **Index** que actúa como una zona intermedia
 - Se preparan los archivos que le indiques poniéndolos en una especie de lista virtual a la que llamamos el "Index". En Index ponemos los archivos que hemos ido modificando, pero las cosas que están en el "Index" aun no han sido archivadas por git.
 - ▣ El último es el **HEAD** que apunta al último commit realizado



Manteniendo nuestro repositorio al día

(add)

16

- Cuando tenemos el repositorio iniciado (o clonado) empezamos a trabajar creando ficheros, editándolos, modificándolos, etc.
- Para que git sepa que tiene que empezar a tener en cuenta un archivo (a esto se le llama preparar un archivo), usamos la orden `git add` de este modo:
 - ▣ `git add NOMBRE_DEL_ARCHIVO`
- Esto, como vimos antes, añadirá el archivo indicado al Index. No lo archivará realmente en el sistema de control de versiones ni hará nada. Solo le informa de que debe tener en cuenta ese archivo para futuras instrucciones
- Una función interesante es `git status`
 - ▣ Da un resumen de cómo están las cosas ahora mismo respecto a la versión del repositorio (concretamente, respecto al HEAD): qué archivos se han modificado, qué hay en el Index, etc.
 - ▣ Cada vez que no tengas muy claro qué has cambiado y qué no, consulta `git status`.

```
Iván@DESKTOP-7SCINF6 MINGW64 /e/Repositorios/repo-ejemplo (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

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

        modified:   index.html

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


Manteniendo nuestro repositorio al día

(add)

17

```
Iván@DESKTOP-7SCINFC MINGW64 /e/Repositorios/repo-ejemplo (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

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

        modified:   index.html

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

Iván@DESKTOP-7SCINFC MINGW64 /e/Repositorios/repo-ejemplo (master)
$ git add index.html

Iván@DESKTOP-7SCINFC MINGW64 /e/Repositorios/repo-ejemplo (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   index.html
```

Manteniendo nuestro repositorio al día

(commit)

18

- Cuando se han hecho los cambios necesarios y se ha puesto en el Index todo lo que se quiera poner bajo el control de versiones, llega el momento de "hacer commit".
- Esto significa mandar al HEAD los cambios que tenemos en el Index
- **Para hacer commit** a estos cambios:
 - ▣ **git commit -m "Mensaje del commit"**
- También se podrán confirmar todos los cambios que haya en el directorio de trabajo aunque no hayan sido preparados (es decir, aunque no se haya hecho add)
 - ▣ **git commit -a**
 - ▣ Esto incluye tanto los ficheros modificados como los eliminados, con lo que sería equivalente a hacer un **git add -A** seguido de un **git commit**.

```
Iván@DESKTOP-7SCINFC MINGW64 /e/Repositorios/repo-ejemplo (master)
$ git commit -m "Pequeño cambio de ejemplo"
[master 5bf55b5] Pequeño cambio de ejemplo
1 file changed, 1 deletion(-)
```

Envío de cambios (push)

19

- Ahora el archivo está incluido en el **HEAD**, pero aún no en el repositorio remoto, sólo tenemos una copia local.
- **Para enviar estos cambios al repositorio remoto:**
 - ▣ **git push origin master**
 - ▣ *master* es la rama del repositorio donde vamos a hacer los cambios. Habrá que reemplazar *master* por la rama a la que queramos enviar los cambios.
 - ▣ *origin* es el repositorio remoto

```
Iván@DESKTOP-7SCINFC MINGW64 /e/Repositorios/repo-ejemplo (master)
$ git remote -v
origin https://gitlab.com/ivanlorenzo/repo-ejemplo.git (fetch)
origin https://gitlab.com/ivanlorenzo/repo-ejemplo.git (push)

Iván@DESKTOP-7SCINFC MINGW64 /e/Repositorios/repo-ejemplo (master)
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 328 bytes | 328.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To https://gitlab.com/ivanlorenzo/repo-ejemplo.git
   edb9ffa..5bf55b5  master -> master
```

Envío de cambios (pull)

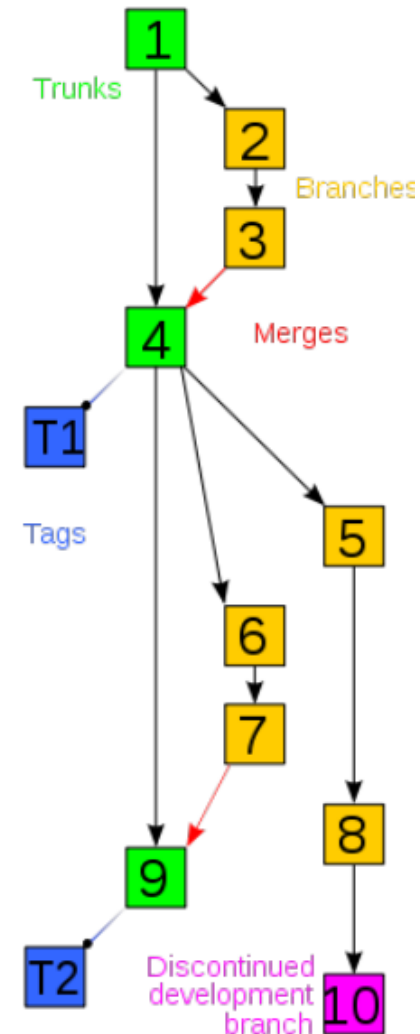
20

- Para actualizar el repositorio local al commit más nuevo:
 - ▣ **git pull**
 - ▣ Hay que realizarlo en el directorio de trabajo para bajar y fusionar los cambios remotos

Ramificación de código (branch)

21

- Existen situaciones en las que el ciclo de vida de un proyecto implica una evolución paralela de su código. Para ello se crean distintas ramas que al final se pueden fusionar todas ellas.
- Motivos para bifurcar:
 - ▣ Seguir evolucionando un software mientras se corrigen bugs de una versión puesta en producción. (Branch evolutivo y otro correctivo)
 - ▣ Muchas modificaciones que dejan al repositorio inestable, se crea un Branch para finalizar dichas modificaciones
 - ▣ Dos evoluciones de distinta naturaleza y no sea conveniente desarrollarlas de forma conjunta
- La ramificación no tiene mucho coste, pero no conviene abusar
- Mejor sólo cuando no haya alternativa.
- El problema es que crece de manera exponencial y es difícil la gestión de todas las versiones del proyecto.
- Los branches se consideran **ramas de vida limitada que deben terminar con un cierre de version (Tag)** o una fusión de cambios a la rama de la que se bifurco.



Ramas

22

- Las ramas (branch) son utilizadas para desarrollar funcionalidades aisladas unas de otras.
- La rama *master* es la rama "por defecto" cuando se crea un repositorio.
- Ya sabemos que se pueden crear nuevas ramas durante el desarrollo y fusionarlas a la rama principal cuando terminemos
- Para crear una nueva rama llamada "nuevaRama" y cambiarte a ella:
 - ▣ `git checkout -b nuevaRama`
- Para volver a la rama principal
 - ▣ `git checkout master`
- Para borrar la rama
 - ▣ `git branch -d nuevaRama`
- Una rama nueva no estará disponible para los demás a menos que se suba (push) la rama al repositorio remoto
 - ▣ `git push origin nuevaRama`

```
Iván@DESKTOP-7SCINFC MINGW64 /e/Repositorios/borrar (master)
$ git checkout -b nuevaRama
Switched to a new branch 'nuevaRama'

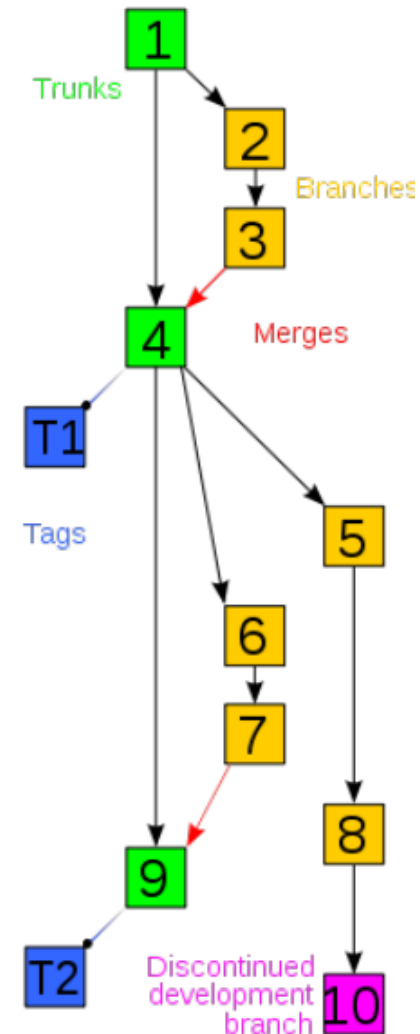
Iván@DESKTOP-7SCINFC MINGW64 /e/Repositorios/borrar (nuevaRama)
$ git checkout master
Switched to branch 'master'

Iván@DESKTOP-7SCINFC MINGW64 /e/Repositorios/borrar (master)
$ git branch -d nuevaRama
Deleted branch nuevaRama (was 3330bc3).
```

Fusión de cambios (merge)

23

- Se usa después de una ramificación para aplicarlo a algún desarrollo en paralelo.
- Se utiliza el comando **Merge**, que aplica los cambios producidos entre dos revisiones en una rama a otra rama.
- En caso de rama evolutiva y rama correctiva, se fusionan los cambios realizados en la correctiva con los surgidos simultáneamente en la evolutiva.
- Se realizará cuando se finalice un desarrollo paralelo.
- Mejor con Merge que intentar hacerlo a mano.



Actualizar y fusionar

24

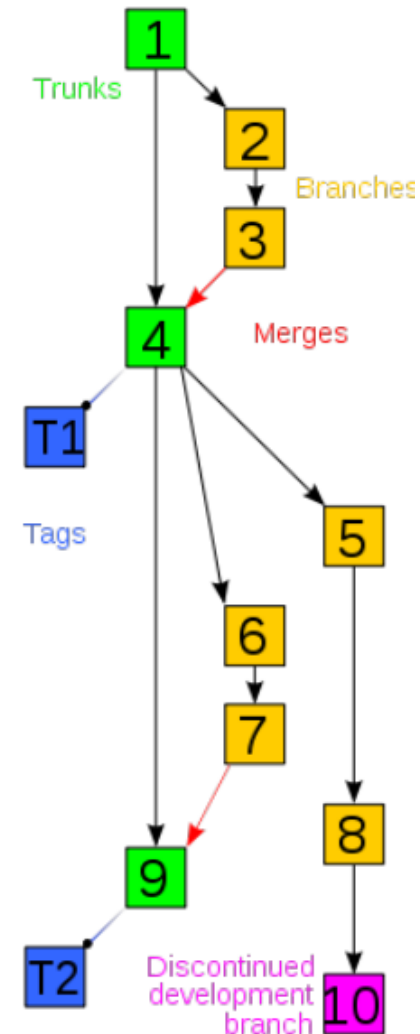
- Para fusionar otra rama a la rama activa (por ejemplo master):
 - ▣ `git merge <nombreRama>`
- En ambos casos git intentará fusionar automáticamente los cambios. Desafortunadamente, no siempre será posible y se podrán producir **conflictos**.
- En última instancia habrá que evaluar manualmente estos conflictos editando los archivos mostrados por git.
- Después de modificarlos, habrá que marcarlos como fusionados con
 - ▣ `git add <filename>`
- Antes de fusionar los cambios podemos revisarlos usando:
 - ▣ `git diff <source_branch> <target_branch>`

```
Iván@DESKTOP-7SCINFC MINGW64 /e/Repositorios/borrar (master)
$ git diff master nuevaRama
diff --git a/index.html b/index.html
index fa15663..fa807de 100644
--- a/index.html
+++ b/index.html
@@ -3,7 +3,7 @@
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
-    <title>git - la guía sencilla MASTER</title>
+    <title>git - la guía sencilla CREANDO UNA RAMA</title>
    <link href='http://fonts.googleapis.com/css?family=Chelsea+Market' rel='stylesheet' type='text/css'>
    <link rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/normalize/0/normalize.min.css" type="text/css">
    <link rel="stylesheet" href="css/style.css" type="text/css">
```


Cierre de versión (creación de tag)

25

- Puede ser conveniente cerrar versiones de un proyecto. Se sigue desarrollando aunque la primera versión ya está en producción.
- A veces hay que volver atrás a resolver un bug de una versión ya acabada y seguir desarrollando el resto de versiones.
- El cierre de versión se denomina crear un Tag de la versión desarrollada. Esto implica **llevar una copia de la versión a cerrar a la rama de gestión de versiones**.
- **No** es costoso hacer Tags de un proyecto, así que se pueden hacer cada vez que se llegue a un objetivo del proyecto (hito).
- OJO: no modificar nunca un Tag, porque, aunque Subversion lo permite, se estaría perdiendo la referencia a la versión que en su momento se decidió congelar.
- Se debe utilizar el Master o bien un Branch para la evolución de la siguiente versión.



Etiquetas (tag)

26

- Se recomienda crear etiquetas para cada nueva versión publicada de un software.
- Este concepto no es nuevo, ya que estaba disponible en SVN.
- Puedes crear una nueva etiqueta llamada 1.0.0 ejecutando
 - ▣ **git tag 1.0.0 1b2e1d63ff**
 - ▣ 1b2e1d63ff se refiere a los 10 caracteres del commit id al cual quieres referirte con tu etiqueta.
- Puedes obtener el commit id con:
 - ▣ **git log**

```
Iván@DESKTOP-7SCINFC MINGW64 /e/Repositorios/borrar (master)
$ git tag 1.0.0

Iván@DESKTOP-7SCINFC MINGW64 /e/Repositorios/borrar (master)
$ git log
commit 894f2b805e7b72804407fe6c9cf456a2936a549d (HEAD -> master, tag: 1.0.0)
```

Remplazar cambios locales

27

- En caso de hacer algo algo mal (lo que seguramente nunca suceda 😊) se puede reemplazar los cambios locales usando el comando
 - ▣ **git checkout -- <filename>**
- Este comando reemplaza los cambios en el directorio de trabajo con el último contenido de HEAD.
- Los cambios que ya han sido agregados al Index, así como también los nuevos archivos, se mantendrán sin cambio.
- Si lo que queremos es borrar los commits y dejar uno anterior podemos usar el comando:
 - ▣ **git reset --hard COMMIT_ID**

```
Iván@DESKTOP-7SCINFC MINGW64 /e/Repositorios/borrar (master)
$ git log --oneline
894f2b8 (HEAD -> master, tag: 1.0.0) cambios en MASTER
f596838 master
4878fed (nuevaRama) cambios en la nueva rama
3330bc3 primer commit

Iván@DESKTOP-7SCINFC MINGW64 /e/Repositorios/borrar (master)
$ git reset --hard 3330bc3
HEAD is now at 3330bc3 primer commit
```

- Herramienta web para gestionar nuestros repositorios.
- Gratis si los repositorios son abiertos.
- De pago si queremos tener repositorios privados y múltiples colaboradores...
- Wiki para cada proyecto y página web para cada proyecto
- Gráfico para ver cómo los desarrolladores trabajan en sus repositorios y bifurcaciones del proyecto
- Funcionalidades como si se tratase de una red social, como por ejemplo: seguidores
- Herramienta para **trabajo colaborativo** entre programadores.



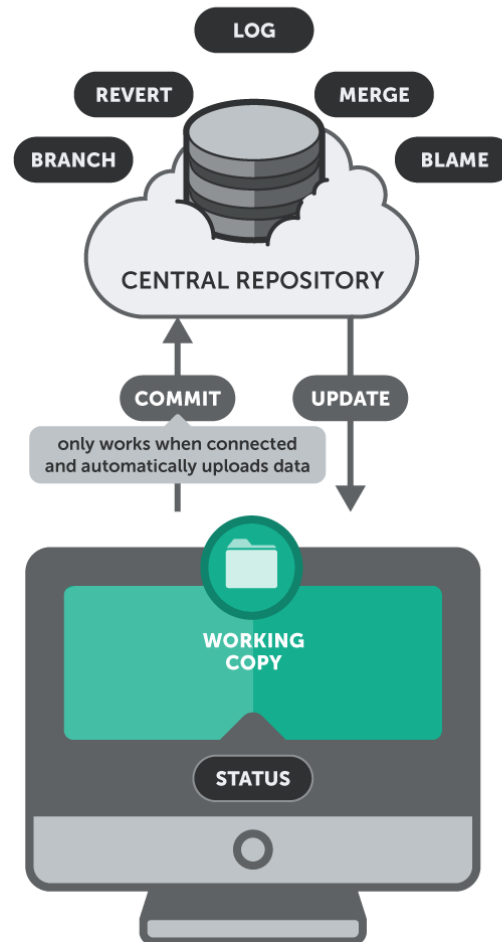
- ¿Si queremos algo sistema similar en nuestro propio servidor, orientado a organizaciones que dependen del almacenamiento de sus proyectos?
- ¿Y si queremos que nuestros proyectos sean privados inicialmente, sin tener que pagar por la cuenta mínima del servicio?
- **GitLab** es un proyecto de **código libre** que se puede **instalar en nuestro servidor** y que nos permite tener repositorios privados sin coste alguno
- Además podremos utilizar sus servidores al igual que en GitHub permitiéndonos tener repositorios **ilimitados privados sin coste**.

SVN vs Git

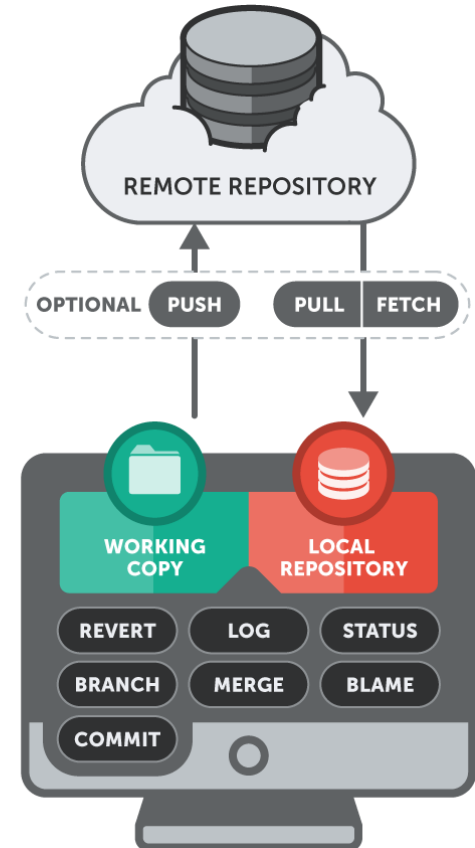
30

- **Centralizado vs Distribuido**
- **Operaciones locales**
 - La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para operar.
 - Por lo general no se necesita información de ningún otro ordenador de tu red.
 - En SVN la mayoría de las operaciones tienen el retardo de la red.
 - Git es **mucho más rápido**. Al tener toda la historia del proyecto en el disco local, la mayoría de las operaciones parecen prácticamente inmediatas.

SUBVERSION



GIT



SVN vs Git

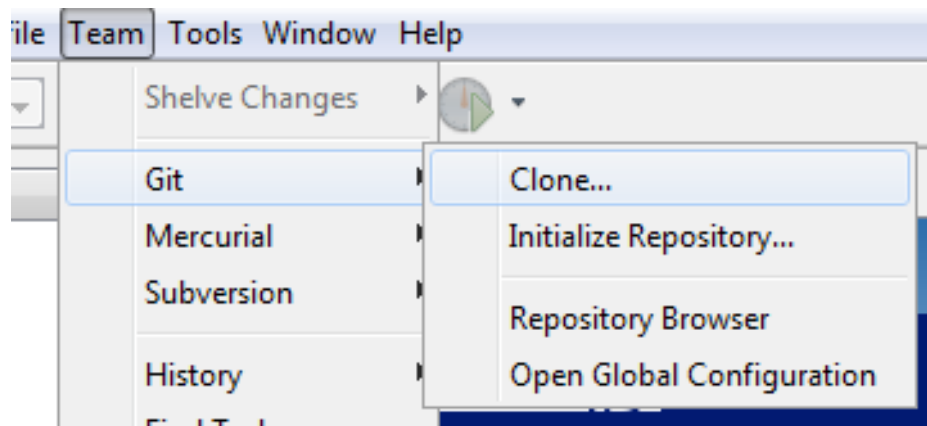
31

	SVN	Git
Control de versiones	Centralizada	Distribuida
Repositorio	Un repositorio central donde se generan copias de trabajo	Copias locales del repositorio en las que se trabaja directamente
Autorización de acceso	Dependiendo de la ruta de acceso	Para la totalidad del directorio
Seguimiento de cambios	Basado en archivos	Basado en contenido
Historial de cambios	Solo en el repositorio completo, las copias de trabajo incluyen únicamente la versión más reciente	Tanto el repositorio como las copias de trabajo individuales incluyen el historial completo
Conectividad de red	Con cada acceso	Solo necesario para la sincronización

Git & NetBeans

32

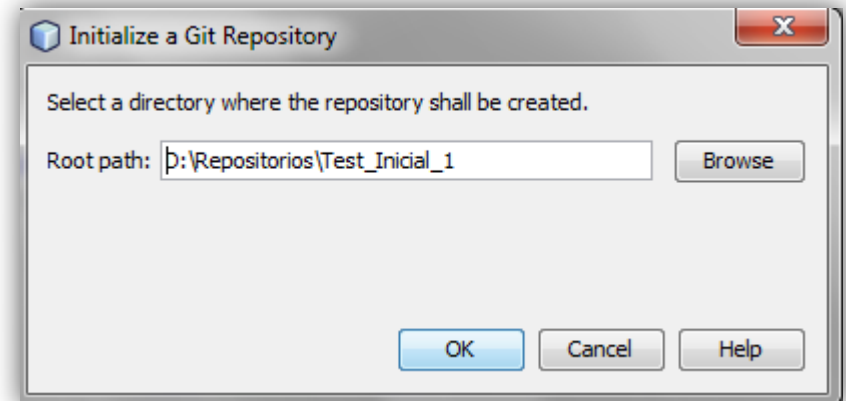
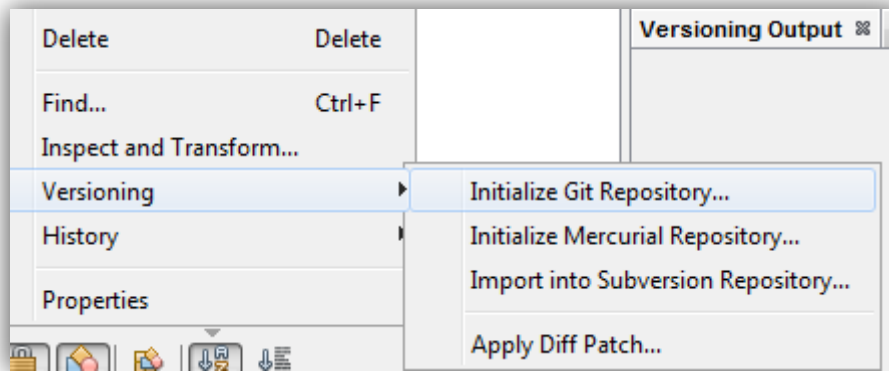
- Netbeans cuenta con soporte nativo para Git
- Para utilizar Git desde Netbeans podemos:
 - ▣ Inicializar un repositorio en algún proyecto existente
 - ▣ Clonar un proyecto que ya está versionado desde Netbeans.



Inicializar un repositorio Git

33

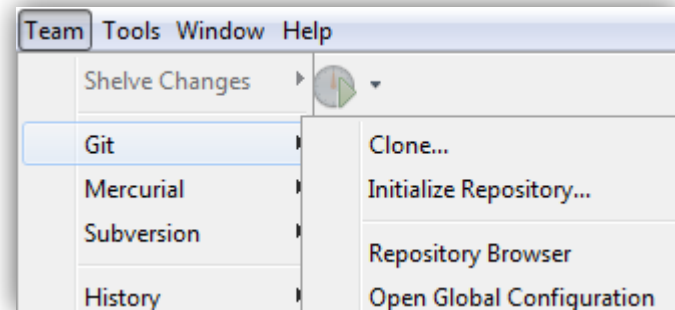
- Si ya tenemos iniciado algún proyecto en Netbeans y queremos comenzar a versionarlo con Git, hacemos clic derecho sobre nuestro proyecto (En el explorador de proyectos) → *Versioning* → *Inicialize Git Respository*.
- Luego nos pedirá que seleccionemos un directorio para el repositorio. Por defecto nos pondrá el directorio en el que se almacena nuestro proyecto. Utilizaremos este directorio.



Clonar un repositorio externo desde NetBeans (I)

34

- ¿Y si ya tenemos algún proyecto alojado en un repositorio externo?
- ¿O si queremos descargar el código de algún proyecto de código libre que utilice git para versionar su código?
- Podemos hacerlo desde Team → Git → Clone...



- A continuación nos pedirá la URL del repositorio que pretendemos clonar. Tengo un proyecto subido en <https://gitlab.com/ivanlorenzo/repo-ejemplo.git> que utilizaré

Clone Repository

Steps

1. Remote Repository
2. Remote Branches
3. Destination Directory

Remote Repository

Specify Git Repository Location:

Repository URL:

User: (leave blank for anonymous access)

Password: ☒ Save Password

[Proxy Configuration...](#)

Specify Destination Folder:

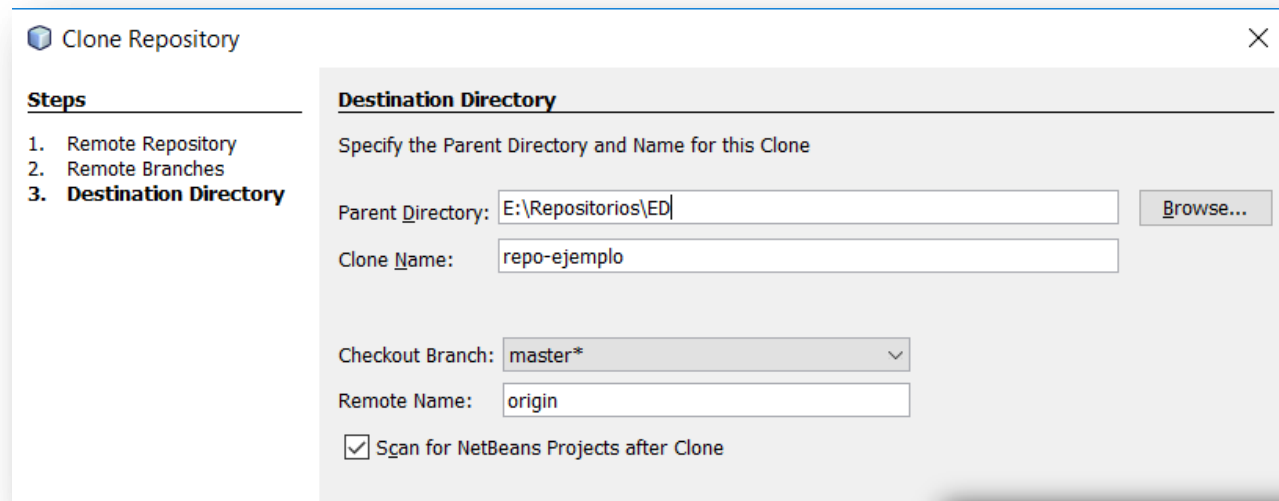
Clone into: [Browse...](#)

(Leave empty to specify the destination later)

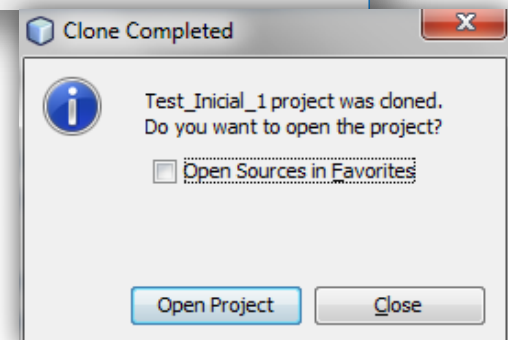
Clonar un repositorio externo desde NetBeans (II)

35

- A continuación elegimos la rama que deseamos clonar (en mi caso 'master')
- En la siguiente ventana elegimos en qué directorio deseamos que se clone y el nombre que le daremos a nuestro repositorio (lo dejo por defecto)
- Activamos la casilla que dice: *'Scan for netbeans projects after Clone'* y finalizamos



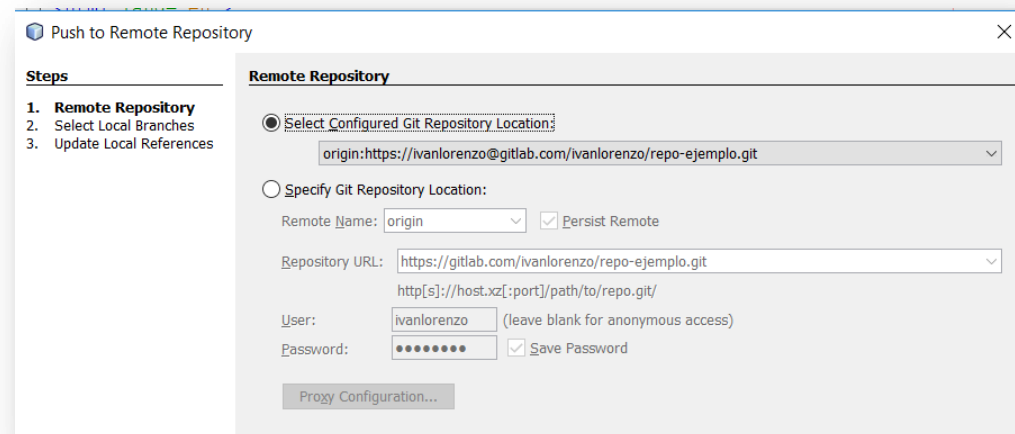
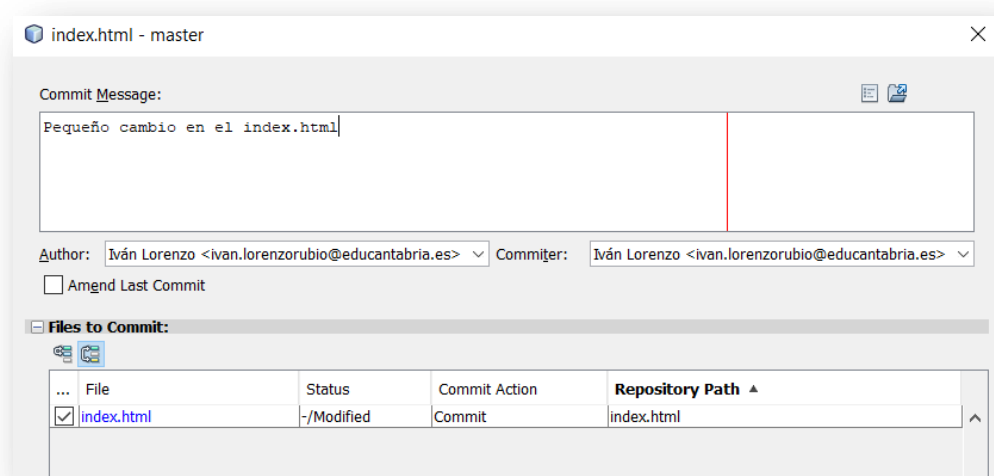
- Después podemos abrir nuestro proyecto



Subir archivos y cambios al repositorio remoto

36

- Para subir nuevos cambios habrá que realizar dos pasos sencillos: Commit y Push
- **Commit**
 - Clic con el botón derecho sobre el proyecto → Git → Commit. Podemos elegir los archivos nuevos o modificados a enviar. Tras introducir un comentario podemos enviarlo al repositorio local (al HEAD).
- **Push**
 - En caso de querer enviarlos ahora al repositorio remoto hacemos clic con el botón derecho sobre el proyecto → Git → Remote → Push
 - Aquí debemos introducir el repositorio remoto, nuestro usuario y nuestra contraseña



Descargar cambios a nuestro repositorio local

37

- Si alguien más ha subido cambios al repositorio remoto y queremos actualizar nuestro repositorio local, debemos hacer un pull.
 - ▣ Clic con el botón derecho sobre el proyecto → Git → Remote → Pull

Pull from Remote Repository

Steps

1. **Remote Repository**
2. Remote Branches

Remote Repository

☒ Select Configured Git Repository Location

origin:https://ivanlorenzo@gitlab.com/ivanlorenzo/repo-ejemplo.git

☐ Specify Git Repository Location:

Remote Name: origin ☒ Persist Remote

Repository URL: https://gitlab.com/ivanlorenzo/repo-ejemplo.git
http[s]://host.xz[:port]/path/to/repo.git/

User: ivanlorenzo (leave blank for anonymous access)

Password: •••••••• ☒ Save Password

Proxy Configuration...

Más opciones de Git en NetBeans

38

- ☐ Add
- ☐ Diff
- ☐ Branch
- ☐ Merge
- ☐ Tag
- ☐ Checkout

- ☐ Más información en:
<https://netbeans.org/kb/docs/ide/git.html>

¿Cómo trabajamos en equipo?

39

- Para que los cambios se vean reflejados se debe hacer un commit en **Team → Commit**.
- Si trabajamos en conjunto con otros desarrolladores siempre se recomienda antes de hacer nuevos cambios realizar un Update **Team → Remote → Pull**.
- Podemos revisar las diferencias entre nuestras copias locales y las del repositorio con **Team → Diff**

Buenas prácticas

40

- Para evitar errores se recomienda:
 - ▣ Tener bien definida la estructura del proyecto (master, tags, branches).
 - ▣ **Siempre** hacer un **pull** antes de empezar a trabajar.
 - ▣ Realizar commits sólo al final de la jornada o cuando se estime que es necesario reflejar cambios.
 - ▣ Realizar commits de archivos relevantes ("de a uno")
 - ▣ Agregar pequeños comentarios a los Commits para describir lo que se realizó.

