



# **Prácticas Profesionalizantes 3**

# Objetivo de la materia

Desarrollar una aplicación web integrando distintas técnicas y herramientas tales como AJAX, ORM, MVC, APIs, entre otras.

Se usará un lenguaje interpretado en el servidor preferentemente (Ruby o Python)



# Tema de la clase

Sistemas de control de  
Versiones

# GIT

---

# ¿Qué es un sistema de control de versiones?

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.

Una versión, revisión o edición de un producto, es el estado en el que se encuentra el mismo en un momento dado de su desarrollo o modificación.

Por lo tanto un sistema de control de versiones es una herramienta que nos facilitara y organizara versiones de nuestro código o producto.





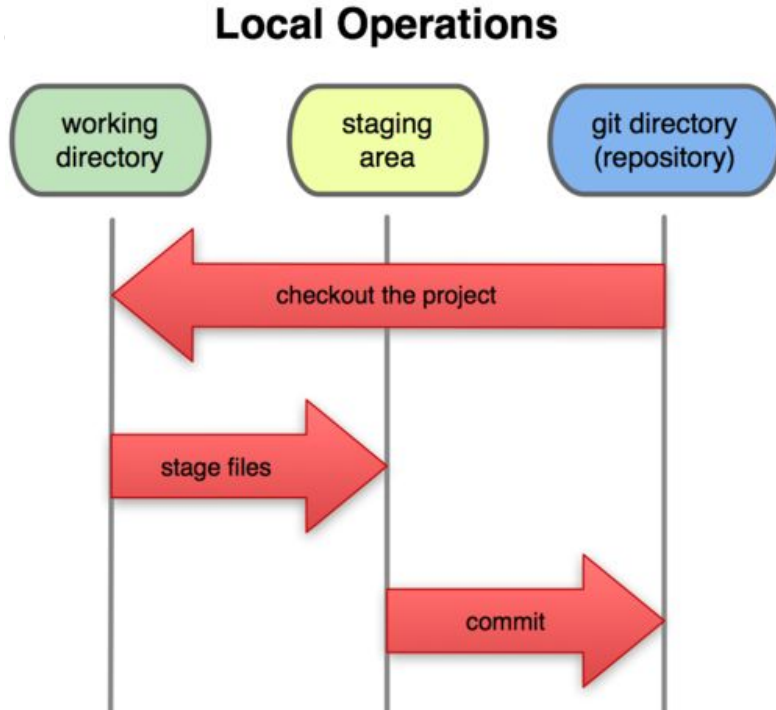
# ¿Qué es Git?

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. [Wikipedia](#)

## Características:

- Snapshots, no diferencias
- Casi todas las operaciones son locales
- Tiene integridad

# Los tres estados



- **WORKING DIRECTORY**
- **STAGING AREA**
- **GIT DIRECTORY**

# Obteniendo un repositorio de GIT

Inicializar un repositorio en un directorio existente

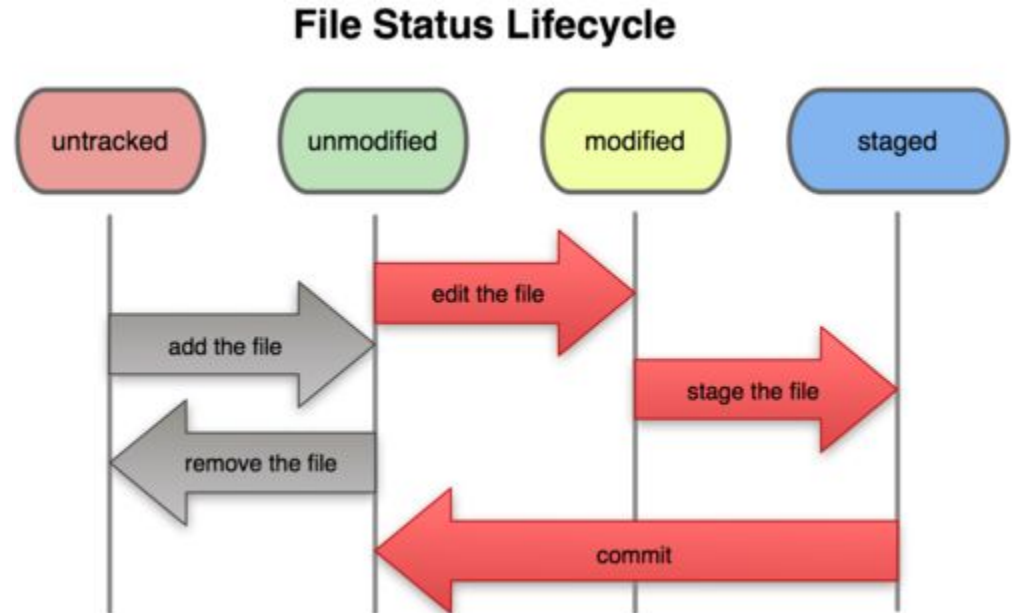
```
$ git init
```

Clonando un repositorio existente

```
$ git clone https://github.com/rails/rails.git
```

# Ciclo de vida de un archivo

- UNTRACKED
- UNMODIFIED
- MODIFIED
- STAGED



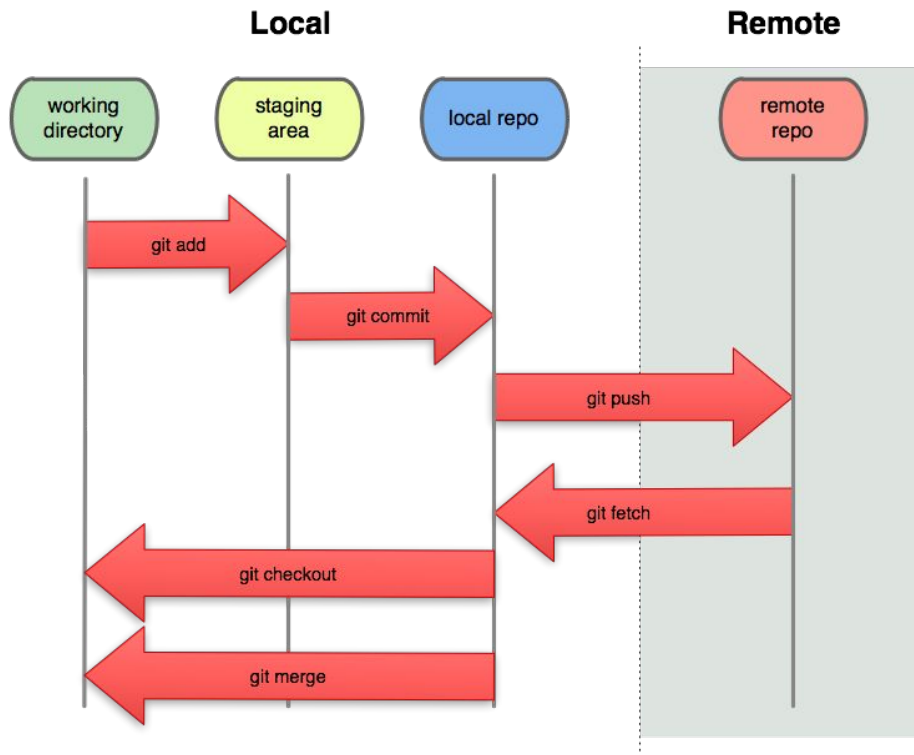


# Repositorios Remotos

Son repositorios externos (ejemplo: de compañeros de trabajo).

Puede haber una cantidad N de repositorios remotos

El flujo de trabajo (workflow) completo se ilustra en el gráfico



# Directorio .git/

Cada repositorio Git es almacenado en la carpeta .git del directorio en el cual el repositorio ha sido creado.

Este directorio contiene la historia completa del repositorio. El archivo .git/config contiene la configuración local del repositorio.

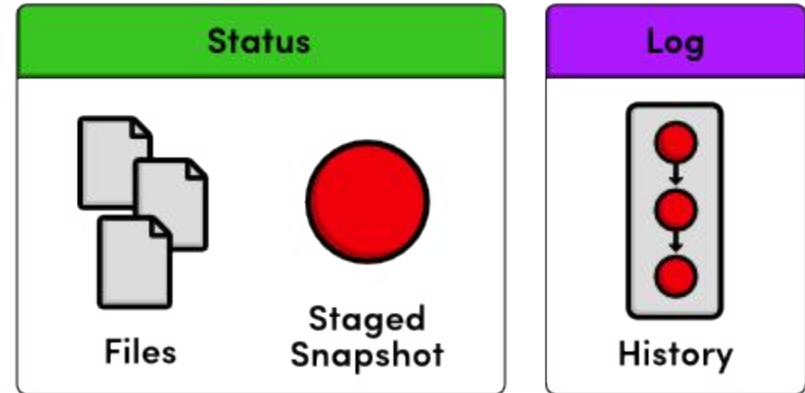


# Comprobando el estado del repositorio

```
$ git status
```

```
# On branch master
```

```
nothing to commit (working directory clean)
```



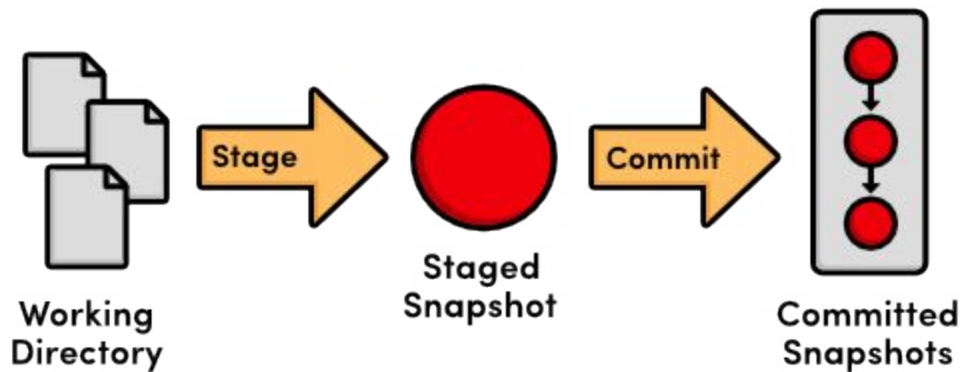
# Creando y modificando contenido

```
# Creamos contenido  
# Agregamos todo (archivos y directorios) al repositorio
```

```
$ git add .  
# Hacemos un commit al repositorio
```

```
$ git commit -m "Initial commit"  
# Muestra el log (un historial)
```

```
$ git log
```



# Revirtiendo cambios (checkout)

```
$ git checkout
```

Este comando saca contenido del repositorio y lo pone en el área de trabajo.

De esta forma permite descartar los cambios no commiteados (los agregados con git add al área de staging).

Además puede utilizarse para "moverse" de rama.

Este comando NO altera la historia de commits





# Ejemplos con checkout

Descartando cambios en un archivo:

```
$ git checkout <file>
```

Cambiando de rama:

```
$ git checkout <rama>
```

Crear nueva rama y cambiar a ésta:

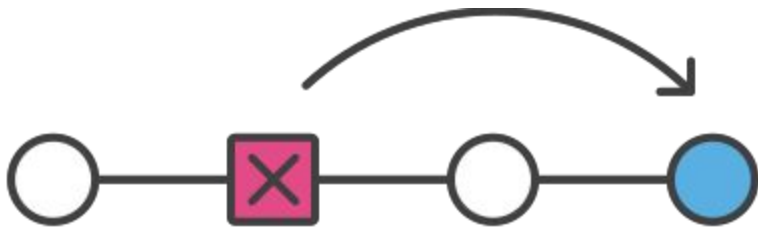
```
$ git checkout -b <rama>
```

# Revirtiendo (revert)

Este comando crea un nuevo commit que deshace los cambios introducidos por un commit previo.

```
$ git revert
```

Agrega nueva historia al proyecto, NO modifica lo ya existente.



# Ejemplos con revert

Revertir un commit (produciendo un nuevo commit con los cambios contrarios):

```
$ git revert <commit>
```

Aplica los cambios necesarios para revertir el commit sin generar un nuevo commit. Afecta el directorio de trabajo y el área de staging.

```
$ git revert --no-commit <commit>
```





# Reiniciar (reset)

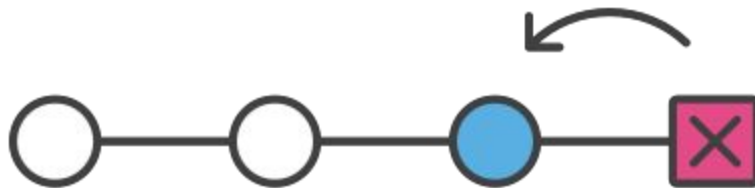
Dependiendo cómo es utilizado este comando realiza operaciones muy distintas.

```
$ git reset
```

Modifica el área de staging.

Permite modificar qué commit es el último (HEAD) de una rama.

Este comando modifica la historia del repositorio.



# Ejemplos con reset

Descartar todos los cambios locales en tu directorio de trabajo:

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

Sacar todos los archivos del área de pruebas (es decir, deshacer el último git add):

```
$ git reset HEAD
```





# Más ejemplos con reset

Reestablecer tu puntero HEAD a un commit anterior y descartar todos los cambios desde entonces:

```
$ git reset --hard <commit>
```

Reestablecer tu puntero HEAD a un commit anterior y preservar todos los cambios en el área de pruebas (stage area):

```
$ git reset <commit>
```

Reestablecer tu puntero HEAD a un commit anterior y preservar los cambios locales sin confirmar (uncommitted changes):


```
$ git reset --keep <commit>
```

# ¿Cuándo utilizar checkout, revert o reset?

Si modificamos erróneamente un archivo en nuestra área de trabajo sin haber commiteado los cambios, se debe utilizar git **checkout** para traer una copia anterior del archivo desde el repositorio local.

Si en la historia del proyecto se produjo un commit que luego se decide que fue erróneo, se debe utilizar git **revert**. Va a deshacer los cambios provocados por el commit erróneo, generando un nuevo commit que registra esta corrección.

Si realizamos un commit erróneo, pero no lo hemos compartido con nadie más (todos los commits son locales). Se puede utilizar git **reset** para reescribir la historia como si nunca se hubiera realizado el commit con errores.





¿Preguntas?