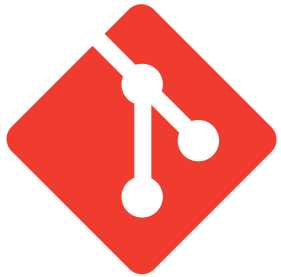
Taller de Git

Clase 2

ComCom

DC - FCEyN - UBA

22 de abril de 2019



taller de git

Recapitulando

Recapitulando

```
git status
```

Recapitulando

```
git status

   Changes to be commited:
```

Recapitulando

```
git status
```

```
Changes to be commited:
```

La clase anterior, aprendimos cómo:

Recapitulando

```
git status
```

```
Changes to be commited:
```

La clase anterior, aprendimos cómo:

- Obtener una copia local de un repositorio (`git...`).

Recapitulando

```
git status
```

```
Changes to be commited:
```

La clase anterior, aprendimos cómo:

- Obtener una copia local de un repositorio (`git clone`).

Recapitulando

```
git status
```

```
Changes to be commited:
```

La clase anterior, aprendimos cómo:

- Obtener una copia local de un repositorio (`git clone`).
- Iniciar un repositorio vacío (`git...`),

Recapitulando

`git status`

`Changes to be commited:`

La clase anterior, aprendimos cómo:

- Obtener una copia local de un repositorio (`git clone`).
- Iniciar un repositorio vacío (`git init`),

Recapitulando

`git status`

`Changes to be commited:`

La clase anterior, aprendimos cómo:

- Obtener una copia local de un repositorio (`git clone`).
- Iniciar un repositorio vacío (`git init`), y vincularlo a un repositorio remoto (`git...`).

Recapitulando

```
git status
```

```
Changes to be commited:
```

La clase anterior, aprendimos cómo:

- Obtener una copia local de un repositorio (`git clone`).
- Iniciar un repositorio vacío (`git init`), y vincularlo a un repositorio remoto (`git remote`).

Recapitulando

```
git status
```

```
Changes to be commited:
```

La clase anterior, aprendimos cómo:

- Obtener una copia local de un repositorio (`git clone`).
- Iniciar un repositorio vacío (`git init`), y vincularlo a un repositorio remoto (`git remote`).
- Marcar cambios como preparados o *staged* (`git...`),

Recapitulando

```
git status
```

```
Changes to be commited:
```

La clase anterior, aprendimos cómo:

- Obtener una copia local de un repositorio (`git clone`).
- Iniciar un repositorio vacío (`git init`), y vincularlo a un repositorio remoto (`git remote`).
- Marcar cambios como preparados o *staged* (`git add`),

Recapitulando

```
git status
```

```
Changes to be commited:
```

La clase anterior, aprendimos cómo:

- Obtener una copia local de un repositorio (`git clone`).
- Iniciar un repositorio vacío (`git init`), y vincularlo a un repositorio remoto (`git remote`).
- Marcar cambios como preparados o *staged* (`git add`), y confirmar estos cambios (`git...`).

Recapitulando

```
git status
```

```
Changes to be commited:
```

La clase anterior, aprendimos cómo:

- Obtener una copia local de un repositorio (`git clone`).
- Iniciar un repositorio vacío (`git init`), y vincularlo a un repositorio remoto (`git remote`).
- Marcar cambios como preparados o *staged* (`git add`), y confirmar estos cambios (`git commit`).

Recapitulando

```
git status
```

```
Changes to be commited:
```

La clase anterior, aprendimos cómo:

- Obtener una copia local de un repositorio (`git clone`).
- Iniciar un repositorio vacío (`git init`), y vincularlo a un repositorio remoto (`git remote`).
- Marcar cambios como preparados o *staged* (`git add`), y confirmar estos cambios (`git commit`).
- Ver el estado actual de nuestros cambios (`git...`).

Recapitulando

```
git status
```

```
Changes to be commited:
```

La clase anterior, aprendimos cómo:

- Obtener una copia local de un repositorio (`git clone`).
- Iniciar un repositorio vacío (`git init`), y vincularlo a un repositorio remoto (`git remote`).
- Marcar cambios como preparados o *staged* (`git add`), y confirmar estos cambios (`git commit`).
- Ver el estado actual de nuestros cambios (`git status`).

Recapitulando

```
git status
```

```
Changes to be commited:
```

La clase anterior, aprendimos cómo:

- Obtener una copia local de un repositorio (`git clone`).
- Iniciar un repositorio vacío (`git init`), y vincularlo a un repositorio remoto (`git remote`).
- Marcar cambios como preparados o *staged* (`git add`), y confirmar estos cambios (`git commit`).
- Ver el estado actual de nuestros cambios (`git status`).
- Enviar nuestros cambios a un repositorio remoto (`git...`)

Recapitulando

```
git status
```

```
Changes to be commited:
```

La clase anterior, aprendimos cómo:

- Obtener una copia local de un repositorio (`git clone`).
- Iniciar un repositorio vacío (`git init`), y vincularlo a un repositorio remoto (`git remote`).
- Marcar cambios como preparados o *staged* (`git add`), y confirmar estos cambios (`git commit`).
- Ver el estado actual de nuestros cambios (`git status`).
- Enviar nuestros cambios a un repositorio remoto (`git push`)

Recapitulando

```
git status
```

```
Changes to be commited:
```

La clase anterior, aprendimos cómo:

- Obtener una copia local de un repositorio (`git clone`).
- Iniciar un repositorio vacío (`git init`), y vincularlo a un repositorio remoto (`git remote`).
- Marcar cambios como preparados o *staged* (`git add`), y confirmar estos cambios (`git commit`).
- Ver el estado actual de nuestros cambios (`git status`).
- Enviar nuestros cambios a un repositorio remoto (`git push`) y bajarnos los cambios a nuestro repositorio local (`git...`).

Recapitulando

```
git status
```

```
Changes to be commited:
```

La clase anterior, aprendimos cómo:

- Obtener una copia local de un repositorio (`git clone`).
- Iniciar un repositorio vacío (`git init`), y vincularlo a un repositorio remoto (`git remote`).
- Marcar cambios como preparados o *staged* (`git add`), y confirmar estos cambios (`git commit`).
- Ver el estado actual de nuestros cambios (`git status`).
- Enviar nuestros cambios a un repositorio remoto (`git push`) y bajarnos los cambios a nuestro repositorio local (`git pull`).

Recapitulando

```
git status
```

```
Changes to be commited:
```

La clase anterior, aprendimos cómo:

- Obtener una copia local de un repositorio (`git clone`).
- Iniciar un repositorio vacío (`git init`), y vincularlo a un repositorio remoto (`git remote`).
- Marcar cambios como preparados o *staged* (`git add`), y confirmar estos cambios (`git commit`).
- Ver el estado actual de nuestros cambios (`git status`).
- Enviar nuestros cambios a un repositorio remoto (`git push`) y bajarnos los cambios a nuestro repositorio local (`git pull`).
- Resolver los conflictos que pueden presentarse al trabajar de forma colaborativa.

Recapitulando

```
git status

   Untracked files:
```

Recapitulando

```
git status
```

```
Untracked files:
```

Hoy vamos a ver:

- Comandos para mover y eliminar archivos.
- Comandos para inspeccionar cambios anteriores.
- Ramificaciones (o *branches*).
- Algunos comandos un poco más avanzados.
- *Bonus track*.

Otros comandos útiles

git rm

git mv

Otros comandos útiles

git rm

Permite borrar un archivo y marcar este cambio como *staged*. La sintaxis es `git rm [archivo]`.

git mv

Otros comandos útiles

git rm

Permite borrar un archivo y marcar este cambio como *staged*. La sintaxis es `git rm [archivo]`.

git mv

Permite mover/renombrar un archivo y marcar este cambio como *staged*. La sintaxis es `git mv [archivo] [nuevo nombre/ubicación]`.

Inspeccionando los cambios

git diff

Inspeccionando los cambios

git diff

Muchas veces es conveniente ver cuáles son los cambios realizados antes de pasarlos a *staged*.

Para esto tenemos el comando `git diff`, que muestra las diferencias entre el estado actual de los archivos y la última vez que hicimos `git add` (cambios marcados como *staged*).

Si, en cambio, queremos ver las diferencias entre los cambios marcados como *staged* y los que confirmamos en el último *commit*, podemos usar `git diff --staged`.

git diff

Muchas veces es conveniente ver cuáles son los cambios realizados antes de pasarlos a *staged*.

Para esto tenemos el comando `git diff`, que muestra las diferencias entre el estado actual de los archivos y la última vez que hicimos `git add` (cambios marcados como *staged*).

Si, en cambio, queremos ver las diferencias entre los cambios marcados como *staged* y los que confirmamos en el último *commit*, podemos usar `git diff --staged`.

Ejercicio

Modificar un archivo de algún repo de la clase pasada y ejecutar `git diff`.

Viendo la historia de los *commits*

git log

Viendo la historia de los *commits*

git log

Después de haber hecho varios *commits*, o si acabamos de clonar un repositorio de Internet, puede que queramos ver el historial de *commits* para saber cuáles son las modificaciones que se hicieron. Esto se puede lograr utilizando el comando `git log`.

Viendo la historia de los *commits*

git log

Después de haber hecho varios *commits*, o si acabamos de clonar un repositorio de Internet, puede que queramos ver el historial de *commits* para saber cuáles son las modificaciones que se hicieron. Esto se puede lograr utilizando el comando `git log`.

Ejercicio

Ejecutar `git log` en algún repo de la clase pasada.

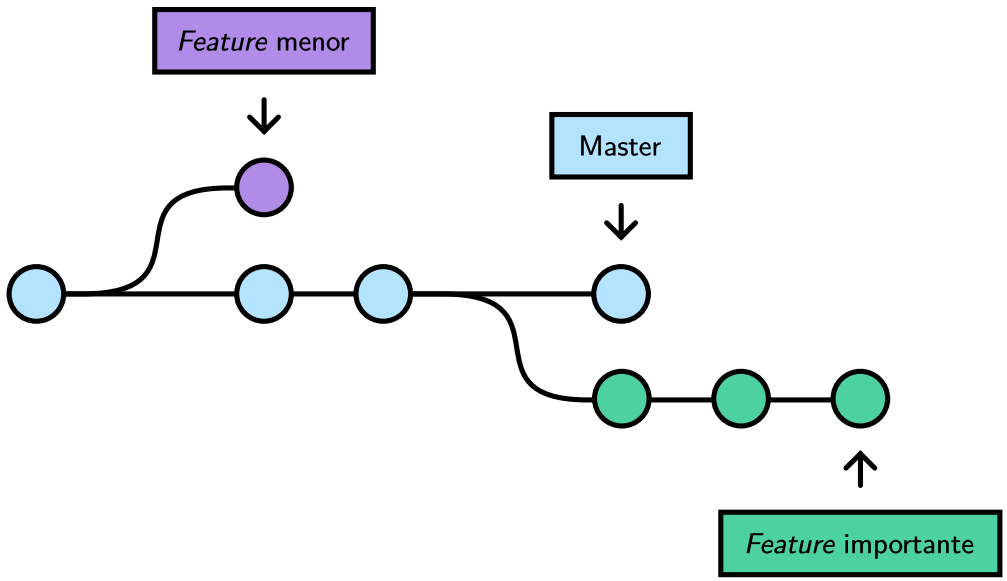
Ramificaciones en Git

Una **rama** (o *branch*) en Git representa una línea independiente de desarrollo. Al crear nuevas ramas, podemos pensar que nuestro proyecto diverge en dos distintos: los cambios que hagamos en uno no impactan al otro.

Ramificaciones en Git

Una **rama** (o *branch*) en Git representa una línea independiente de desarrollo. Al crear nuevas ramas, podemos pensar que nuestro proyecto diverge en dos distintos: los cambios que hagamos en uno no impactan al otro.

Un ejemplo visual:



Creando ramas

git branch

Creando ramas

git branch

Para crear una rama nueva, podemos ejecutar `git branch [nombre de la rama]`.

Nótese que este comando no nos mueve a la nueva rama, solo la crea.

Para ver las ramas de nuestro repositorio local, podemos ejecutar `git branch`.

git branch

Para crear una rama nueva, podemos ejecutar `git branch [nombre de la rama]`.

Nótese que este comando no nos mueve a la nueva rama, solo la crea.

Para ver las ramas de nuestro repositorio local, podemos ejecutar `git branch`.

Ejercicio

Crear una rama llamada "prueba" en algún repo de la clase pasada.

Cambiando de rama

git checkout

Cambiando de rama

git checkout

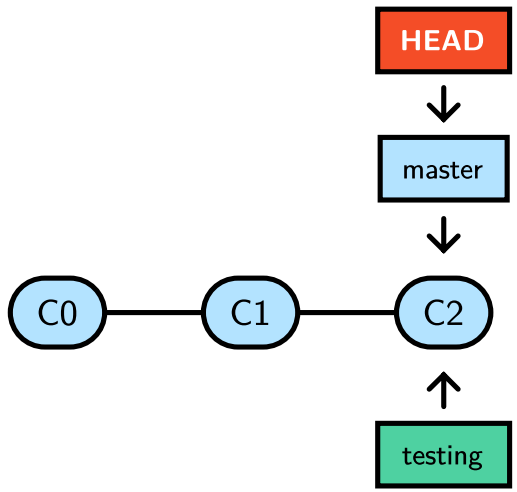
Para cambiar de rama, podemos ejecutar `git checkout [nombre de la rama]`.

Cambiando de rama

git checkout

Para cambiar de rama, podemos ejecutar `git checkout [nombre de la rama]`.

Un ejemplo visual:

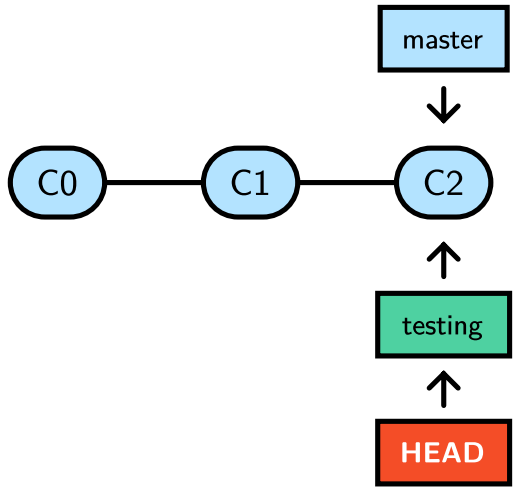


Cambiando de rama

git checkout

Para cambiar de rama, podemos ejecutar `git checkout [nombre de la rama]`.

Un ejemplo visual:



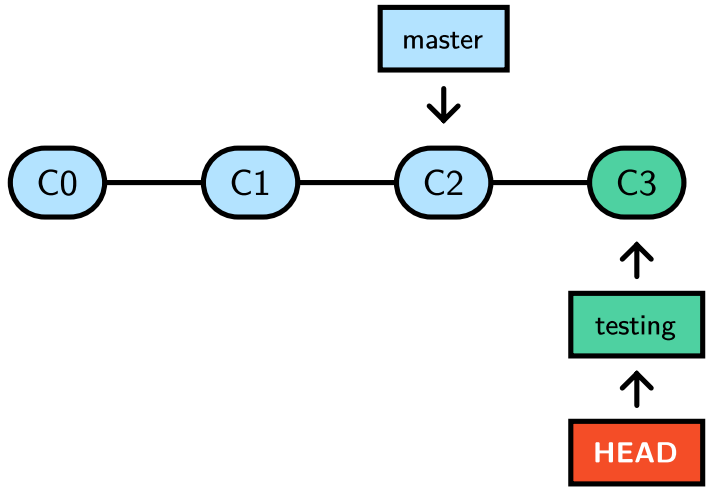
Acá cambiamos a la rama "testing", ejecutando `git checkout testing`.

Cambiando de rama

git checkout

Para cambiar de rama, podemos ejecutar `git checkout [nombre de la rama]`.

Un ejemplo visual:



Y los siguientes *commits* serán agregados a la rama "testing".

Fusionando ramas

git merge

git merge

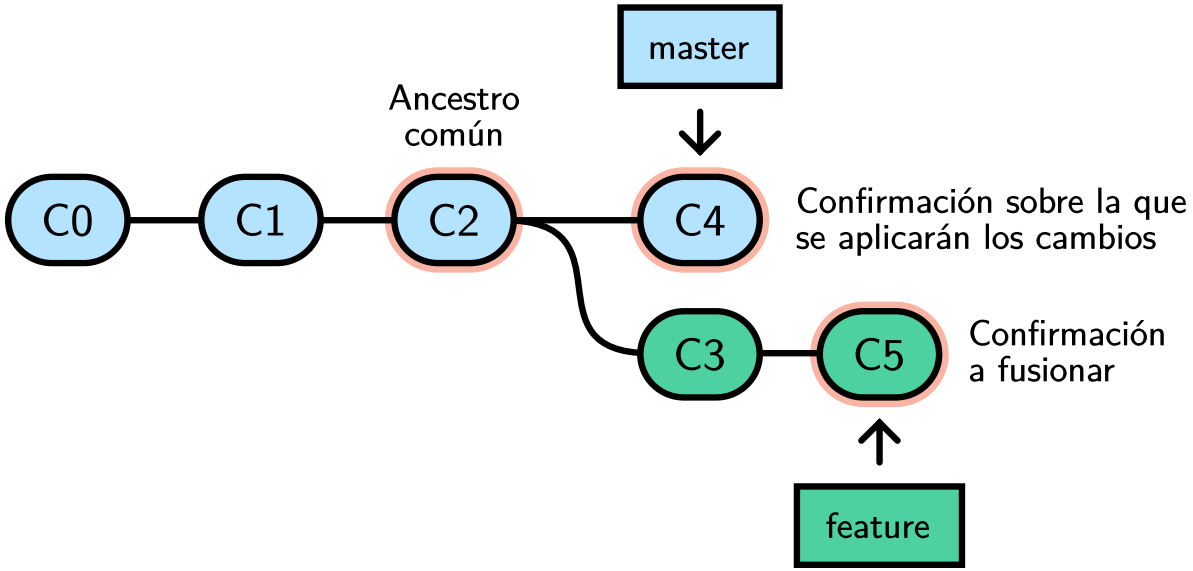
Nos permite fusionar las historias de dos ramas distintas (podría haber conflictos).

La sintaxis es: `git merge [nombre de la rama a fusionar]`.

Importante: este comando fusiona la rama que le decimos **en la rama en la que estamos parados**.

git merge

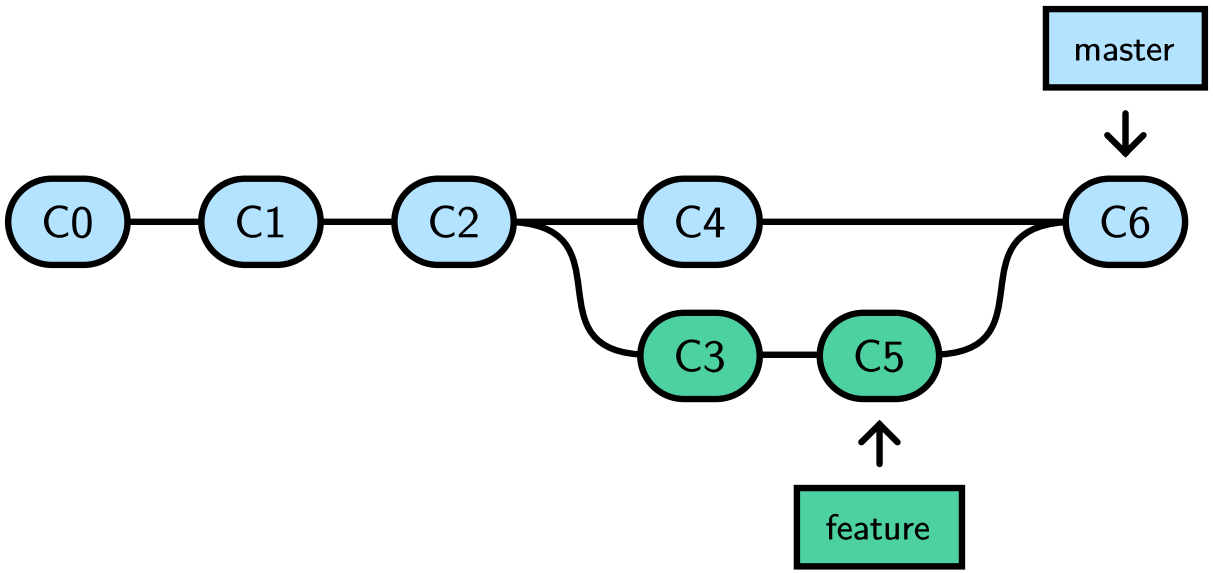
Un ejemplo visual:



Antes del *merge*.

git merge

Un ejemplo visual:



Después de pararnos en la rama "master" (`git checkout master`) y haber fusionado la rama "feature" (`git merge feature`).

¡A practicar!

¡A practicar!

Pero antes de empezar...

Para hacer el siguiente ejercicio, vamos a trabajar con un **fork** de un repositorio.

¡A practicar!

Pero antes de empezar...

Para hacer el siguiente ejercicio, vamos a trabajar con un **fork** de un repositorio. *¿Y eso?*

¡A practicar!

Pero antes de empezar...

Para hacer el siguiente ejercicio, vamos a trabajar con un **fork** de un repositorio. *¿Y eso?*

El concepto de *fork* no es propio de Git. En el ámbito del desarrollo de software, el término se refiere a un proyecto que surge a partir de otro, y adopta un curso de desarrollo independiente.

¡A practicar!

Pero antes de empezar...

Para hacer el siguiente ejercicio, vamos a trabajar con un **fork** de un repositorio. *¿Y eso?*

El concepto de *fork* no es propio de Git. En el ámbito del desarrollo de software, el término se refiere a un proyecto que surge a partir de otro, y adopta un curso de desarrollo independiente.

Los servidores de Git, como GitLab, nos permiten hacer *fork* de proyectos. Al hacer esto creamos una copia de un repositorio ajeno en nuestra propia cuenta, sobre la cual podemos trabajar libremente. Los cambios que hagamos en nuestra copia **no** impactarán en el repositorio original.

¡A practicar!

Pero antes de empezar...

Para hacer el siguiente ejercicio, vamos a trabajar con un **fork** de un repositorio. *¿Y eso?*
El concepto de *fork* no es propio de Git. En el ámbito del desarrollo de software, el término se refiere a un proyecto que surge a partir de otro, y adopta un curso de desarrollo independiente.
Los servidores de Git, como GitLab, nos permiten hacer *fork* de proyectos. Al hacer esto creamos una copia de un repositorio ajeno en nuestra propia cuenta, sobre la cual podemos trabajar libremente. Los cambios que hagamos en nuestra copia **no** impactarán en el repositorio original.

Ahora sí...

¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): 👾 y 👽

¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): 👾 y 👽

1. 👽: Hacer un *fork* en su cuenta de GitLab de este repositorio: `https://gitlab.com/talleres-comcom/taller-git-ejercicio2`, y darle permiso a 👾 para hacer *push*.

¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): 👾 y 👽

1. 👽: Hacer un *fork* en su cuenta de GitLab de este repositorio: `https://gitlab.com/talleres-comcom/taller-git-ejercicio2`, y darle permiso a 👾 para hacer *push*.
2. 👾 y 👽: Obtener una copia local del repositorio de 👽.

¡A practicar!

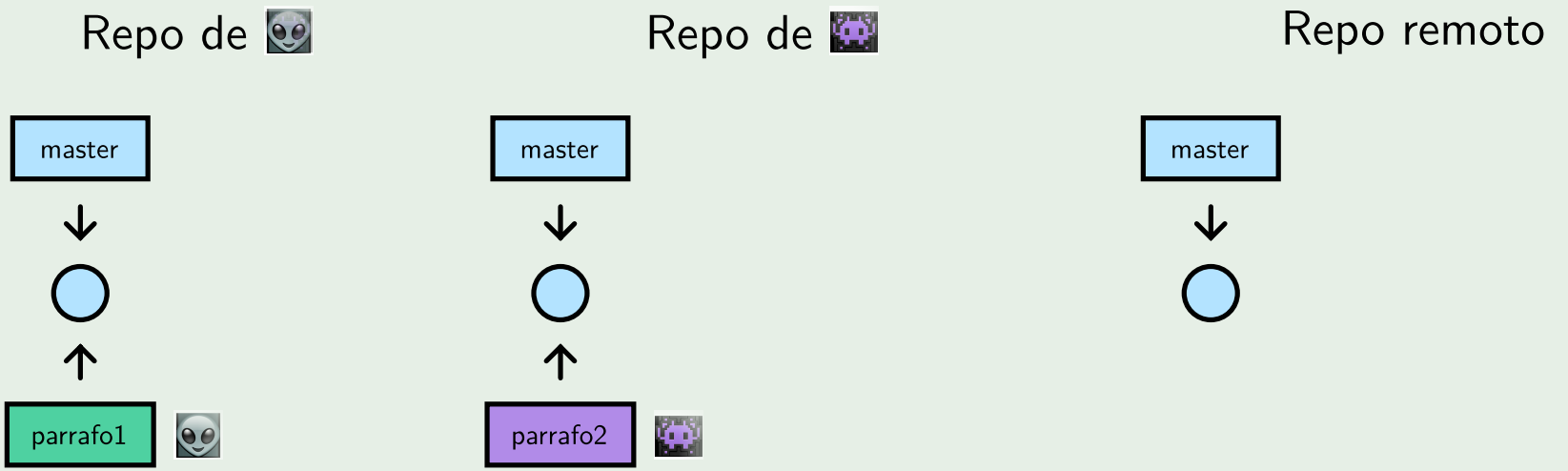
Ejercicio de a 2 máquinas (preferiblemente 2 personas): 👾 y 👽

1. 👽: Hacer un *fork* en su cuenta de GitLab de este repositorio: https://gitlab.com/talleres-comcom/taller-git-ejercicio2, y darle permiso a 👾 para hacer *push*.
2. 👾 y 👽: Obtener una copia local del repositorio de 👽.
3. 👾 y 👽: El repo tiene una única rama, "master", donde van a encontrar un fragmento incompleto, de un cuento de Borges. Repartirse el trabajo: cada uno deberá completar un parrafo.

¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): 👾 y 👽

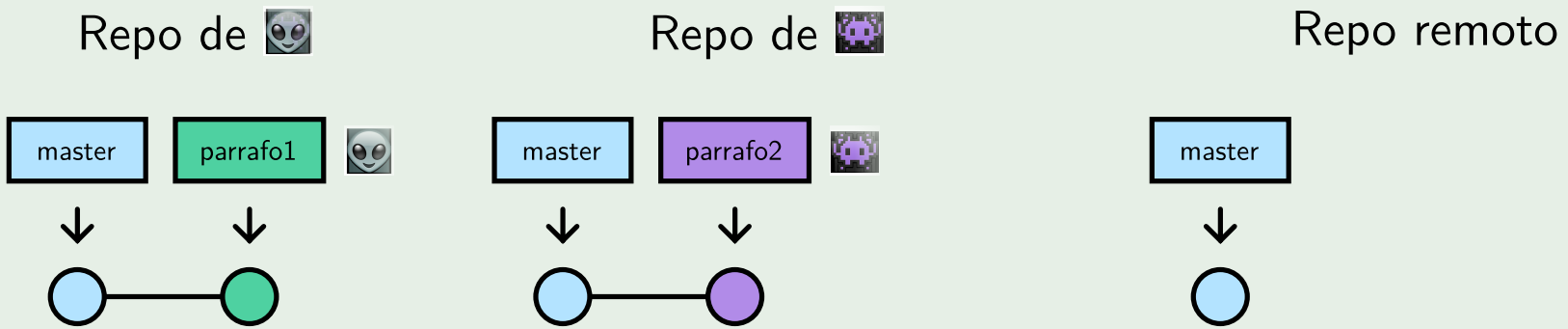
1. 👽: Hacer un *fork* en su cuenta de GitLab de este repositorio: https://gitlab.com/talleres-comcom/taller-git-ejercicio2, y darle permiso a 👾 para hacer *push*.
2. 👾 y 👽: Obtener una copia local del repositorio de 👽.
4. 👾 y 👽: Crear, cada uno, una rama propia donde harán sus modificaciones. Posicionarse en la rama recién creada.



¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): 👾 y 👽

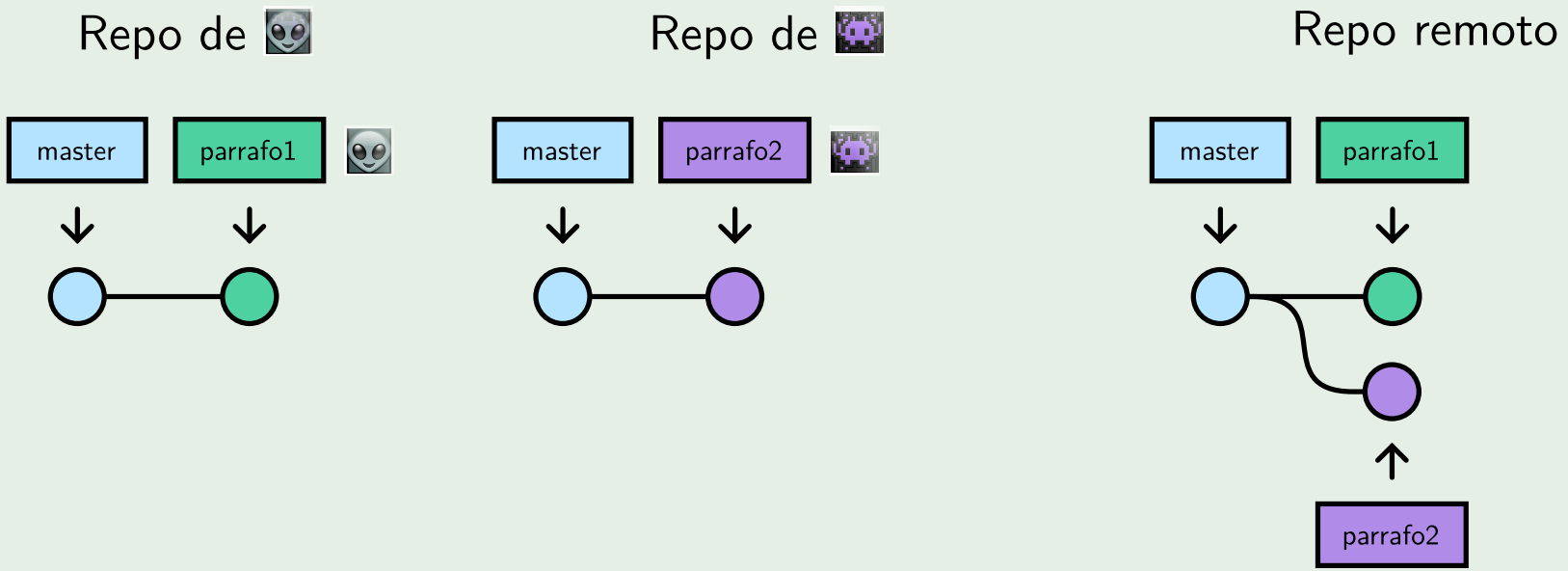
1. 👽: Hacer un *fork* en su cuenta de GitLab de este repositorio: https://gitlab.com/talleres-comcom/taller-git-ejercicio2, y darle permiso a 👾 para hacer *push*.
2. 👾 y 👽: Obtener una copia local del repositorio de 👽.
5. 👾 y 👽: Completar la parte elegida del cuento, y hacer *push* de estos cambios en el repositorio remoto.



¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): 👾 y 👽

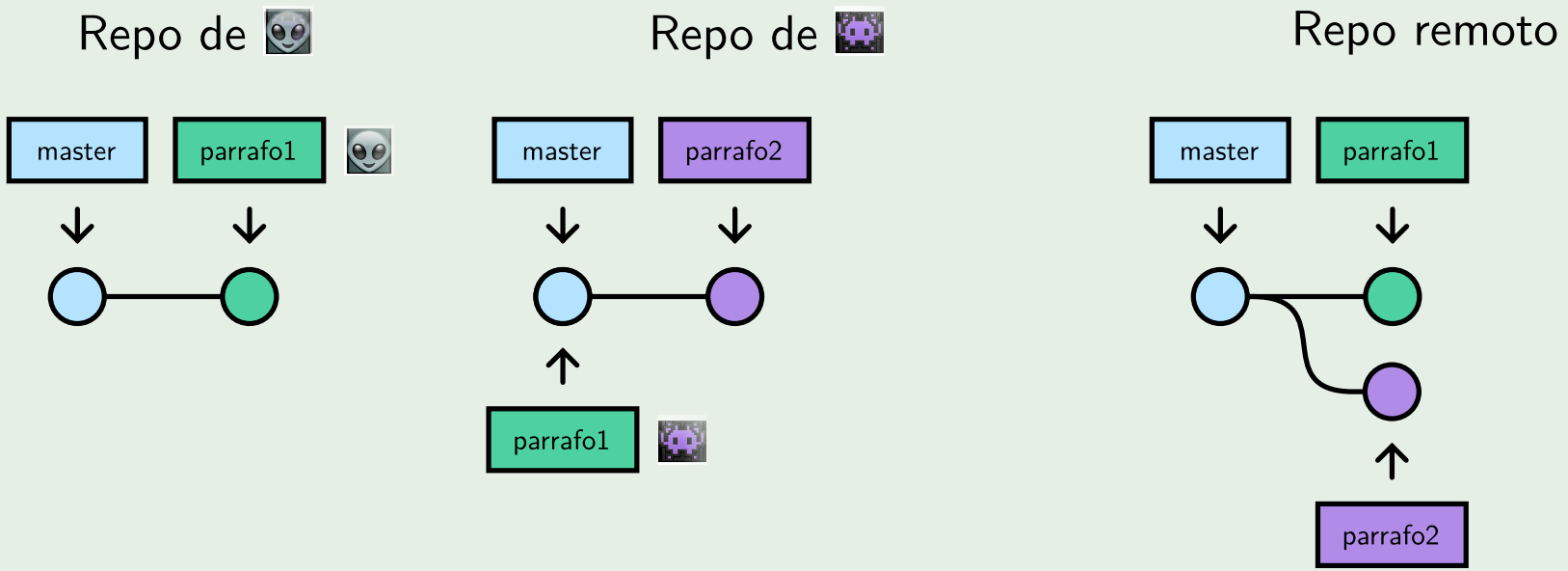
1. 👽: Hacer un *fork* en su cuenta de GitLab de este repositorio: https://gitlab.com/talleres-comcom/taller-git-ejercicio2, y darle permiso a 👾 para hacer *push*.
2. 👾 y 👽: Obtener una copia local del repositorio de 👽.
5. 👾 y 👽: Completar la parte elegida del cuento, y hacer *push* de estos cambios en el repositorio remoto.



¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): 👾 y 👽

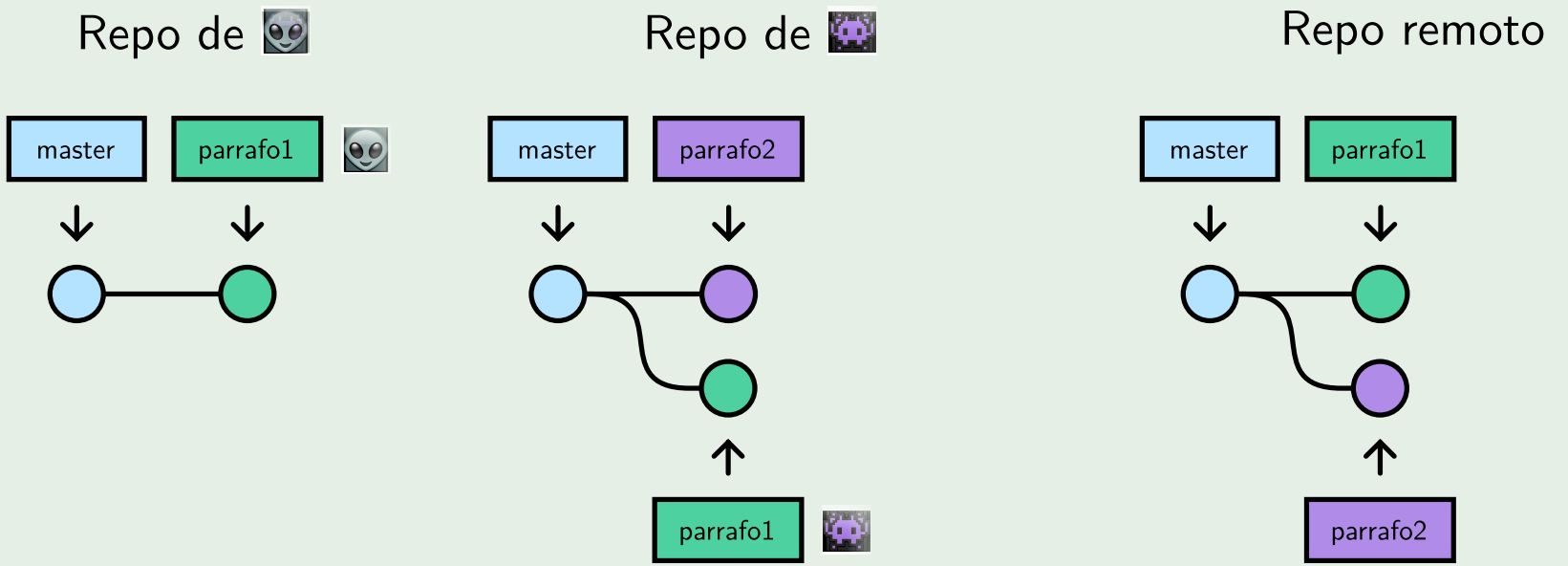
1. 👽: Hacer un *fork* en su cuenta de GitLab de este repositorio: https://gitlab.com/talleres-comcom/taller-git-ejercicio2, y darle permiso a 👾 para hacer *push*.
2. 👾 y 👽: Obtener una copia local del repositorio de 👽.
6. 👾: Crear una nueva rama **a partir de "master"** con el nombre que 👽 usó para la suya, y traerse (*pull*) a esta rama los cambios de 👽.



¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): 👾 y 👽

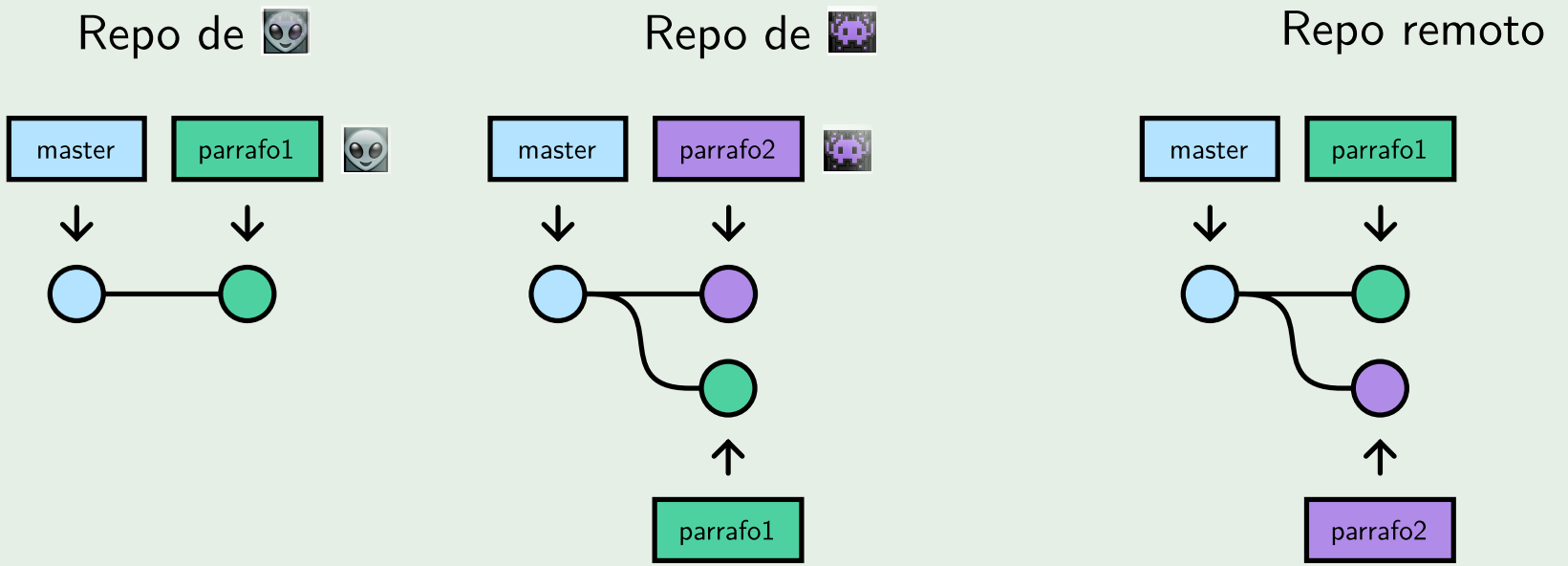
1. 👽: Hacer un *fork* en su cuenta de GitLab de este repositorio: https://gitlab.com/talleres-comcom/taller-git-ejercicio2, y darle permiso a 👾 para hacer *push*.
2. 👾 y 👽: Obtener una copia local del repositorio de 👽.
6. 👾: Crear una nueva rama **a partir de "master"** con el nombre que 👽 usó para la suya, y traerse (*pull*) a esta rama los cambios de 👽.



¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): 👾 y 👽

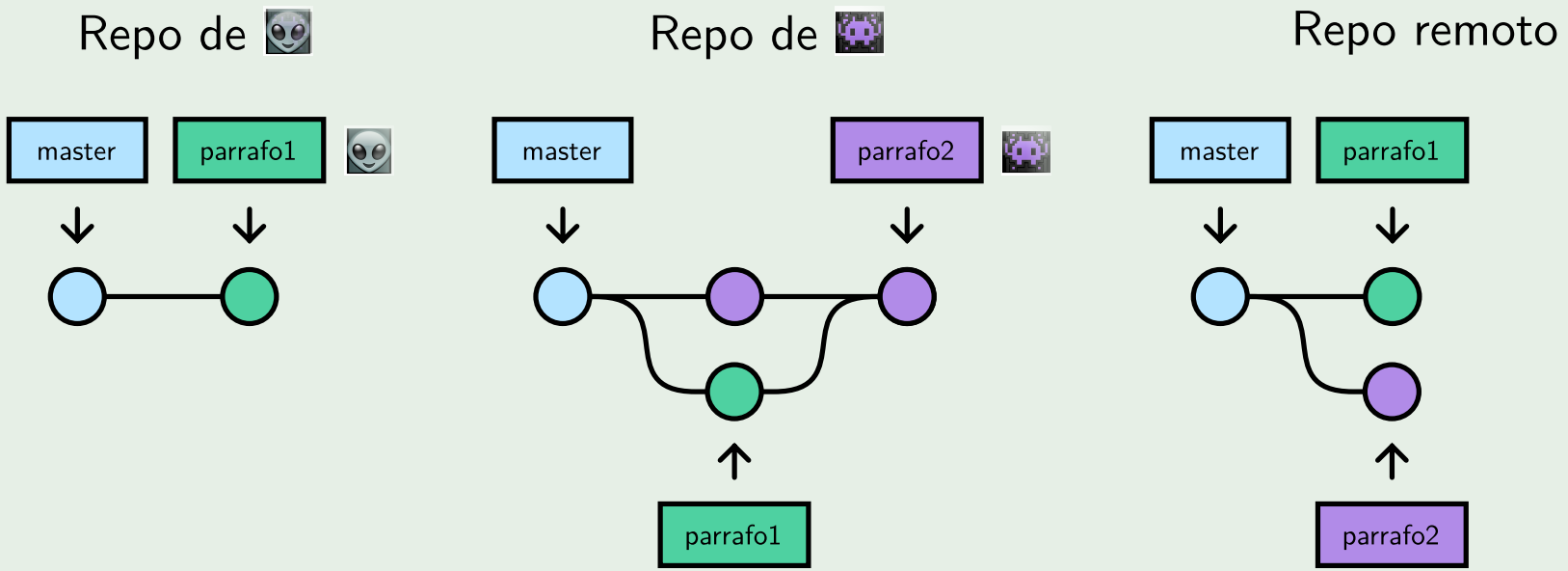
1. 👽: Hacer un *fork* en su cuenta de GitLab de este repositorio: https://gitlab.com/talleres-comcom/taller-git-ejercicio2, y darle permiso a 👾 para hacer *push*.
2. 👾 y 👽: Obtener una copia local del repositorio de 👽.
7. 👾: Volver a su rama de trabajo y fusionar en ella los cambios de 👽. Enviar estos cambios al repositorio remoto.



¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): 👾 y 👽

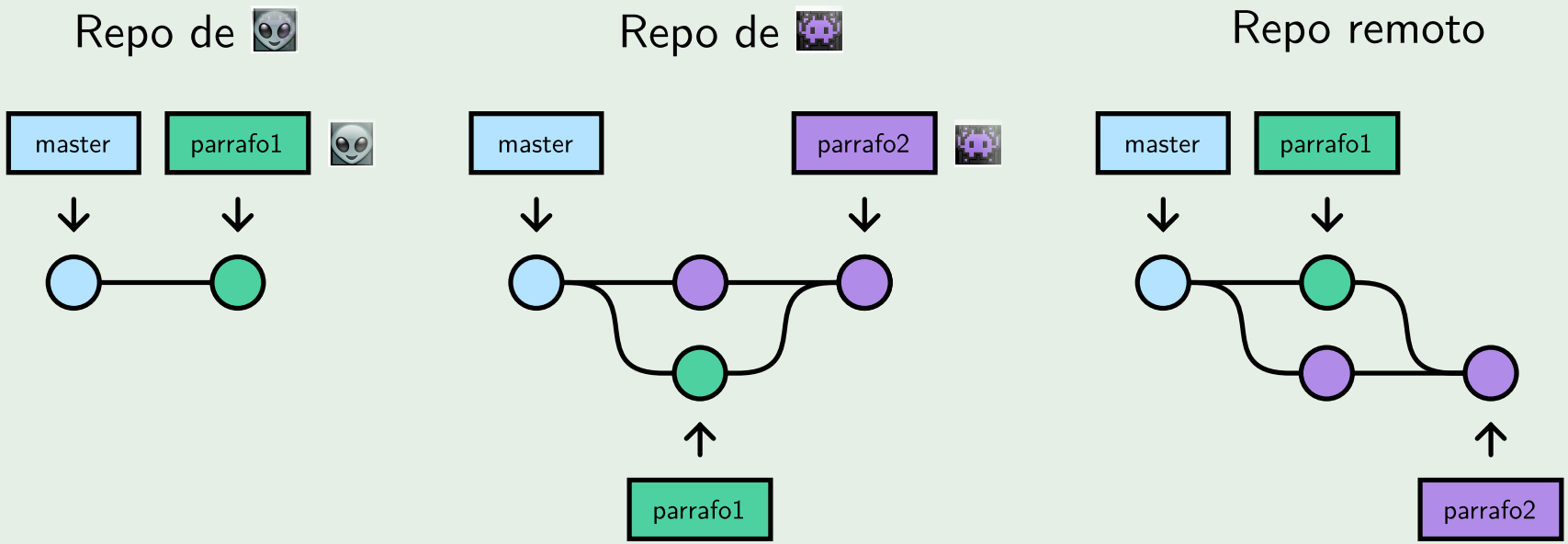
1. 👽: Hacer un *fork* en su cuenta de GitLab de este repositorio: https://gitlab.com/talleres-comcom/taller-git-ejercicio2, y darle permiso a 👾 para hacer *push*.
2. 👾 y 👽: Obtener una copia local del repositorio de 👽.
7. 👾: Volver a su rama de trabajo y fusionar en ella los cambios de 👽. Enviar estos cambios al repositorio remoto.



¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): 👾 y 👽

1. 👽: Hacer un *fork* en su cuenta de GitLab de este repositorio: https://gitlab.com/talleres-comcom/taller-git-ejercicio2, y darle permiso a 👾 para hacer *push*.
2. 👾 y 👽: Obtener una copia local del repositorio de 👽.
7. 👾: Volver a su rama de trabajo y fusionar en ella los cambios de 👽. Enviar estos cambios al repositorio remoto.



¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): 👾 y 👽

1. 👽: Hacer un *fork* en su cuenta de GitLab de este repositorio: `https://gitlab.com/talleres-comcom/taller-git-ejercicio2`, y darle permiso a 👾 para hacer *push*.
2. 👾 y 👽: Obtener una copia local del repositorio de 👽.
8. 👽: Seguir los dos pasos anteriores, pero con los cambios de 👾.

¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): 👾 y 👽

1. 👽: Hacer un *fork* en su cuenta de GitLab de este repositorio: https://gitlab.com/talleres-comcom/taller-git-ejercicio2, y darle permiso a 👾 para hacer *push*.
2. 👾 y 👽: Obtener una copia local del repositorio de 👽.
3. 👾 y 👽: El repo tiene una única rama, "master", donde van a encontrar un fragmento incompleto, de un cuento de Borges. Repartirse el trabajo: cada uno deberá completar un parrafo.
4. 👾 y 👽: Crear, cada uno, una rama propia donde harán sus modificaciones. Posicionarse en la rama recién creada.
5. 👾 y 👽: Completar la parte elegida del cuento, y hacer *push* de estos cambios en el repositorio remoto.
6. 👾: Crear una nueva rama **a partir de "master"** con el nombre que 👽 usó para la suya, y traerse (*pull*) a esta rama los cambios de 👽.
7. 👾: Volver a su rama de trabajo y fusionar en ella los cambios de 👽. Enviar estos cambios al repositorio remoto.
8. 👽: Seguir los dos pasos anteriores, pero con los cambios de 👾.

Revirtiendo cambios

git commit –amend

git revert

Revirtiendo cambios

git commit –amend

Podemos usarlo para **arreglar**, por ejemplo, el mensaje del último *commit* que hicimos: `git commit --amend -m [nuevo mensaje]`.

git revert

Revirtiendo cambios

git commit –amend

Podemos usarlo para **arreglar**, por ejemplo, el mensaje del último *commit* que hicimos: `git commit --amend -m [nuevo mensaje]`.

git revert

Permite **revertir** exactamente los cambios introducidos por un *commit*. Buscamos el *hash* del *commit* en cuestión usando `git log`, y luego ejecutamos `git revert [hash]`.

Otro comando útil

git stash

Otro comando útil

git stash

Al ejecutar `git stash`, se guarda el estado actual de los archivos modificados y nos deja el directorio limpio.

Para volver a mostrar los cambios guardados ejecutamos `git stash apply`.

Extras: Ignorando archivos

El archivo .gitignore

Es común tener archivos generados automáticamente que no queremos agregar al repositorio. Por ejemplo, archivos compilados: .pdf, .exe, .log, .pyc, etc. Sin embargo, es bastante molesto verlos todo el tiempo al ejecutar `git status`.
Para arreglar esto podemos crear un archivo especial llamado `.gitignore`, que le indica a Git qué archivos **ignorar** por completo.

Extras: Ignorando archivos

El archivo .gitignore

Es común tener archivos generados automáticamente que no queremos agregar al repositorio. Por ejemplo, archivos compilados: .pdf, .exe, .log, .pyc, etc. Sin embargo, es bastante molesto verlos todo el tiempo al ejecutar `git status`.
Para arreglar esto podemos crear un archivo especial llamado `.gitignore`, que le indica a Git qué archivos **ignorar** por completo.

Por ejemplo, para ignorar todos los archivos con extensión *.pyc*:

1. Crear un archivo llamado `.gitignore` en el directorio principal del proyecto.
2. Adentro escribir: **.pyc*

Más ejemplos de `.gitignore`:
https://github.com/github/gitignore.

Extras: Más comandos

- `git fetch [remote repository]`: Para traer todos los datos de un repositorio remoto.
- `git reset`: Permite deshacer cambios; sirve para revertir modificaciones en el área de trabajo, pero también puede eliminar por completo *commits* anteriores. ¡Usar con mucho cuidado!
- `git rebase [branch]`: Aplica todos los *commits* que difieren entre una rama y aquella en la que estamos parados. Así podemos incorporar cambios realizados en otras ramas manteniendo lineal el historial de la rama actual.
- `git blame [archivo]`: Para ver qué *commit* modificó por última vez cada línea de un archivo, y quién fue su autor. ¡Así, podemos saber a quién *culpar* cuando haya problemas!
- `git bisect`: Encuentra cuál fue el *commit* que introdujo cierto error, haciendo búsqueda binaria en el historial de *commits*.

Extras: Servidores

Bibliografía

- Git Community book, disponible online y en español: https://git-scm.com/book/es/v2
- `git help [command]` para ver la documentación de cualquier comando de Git.
- A visual Git reference: http://marklodato.github.io/visual-git-guide/index-es.html
- Try Git online: https://try.github.io

Feedback

¡Nos interesa saber que les pareció el taller!

Pueden completar el formulario:
https://forms.gle/ARMtesbt9PygqUpp9

También nos pueden tirar ideas para futuros talleres.