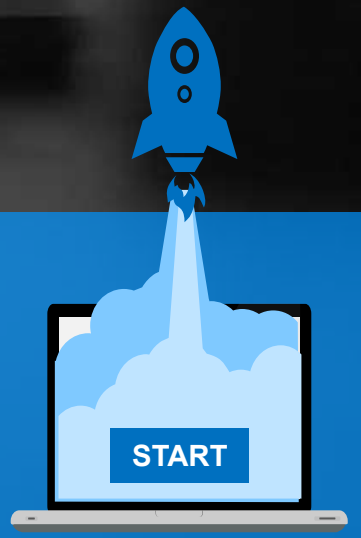
BIENVENIDOS

lsftn151 | AS | AYED II



Un Momento por favor.

Bienvenidos

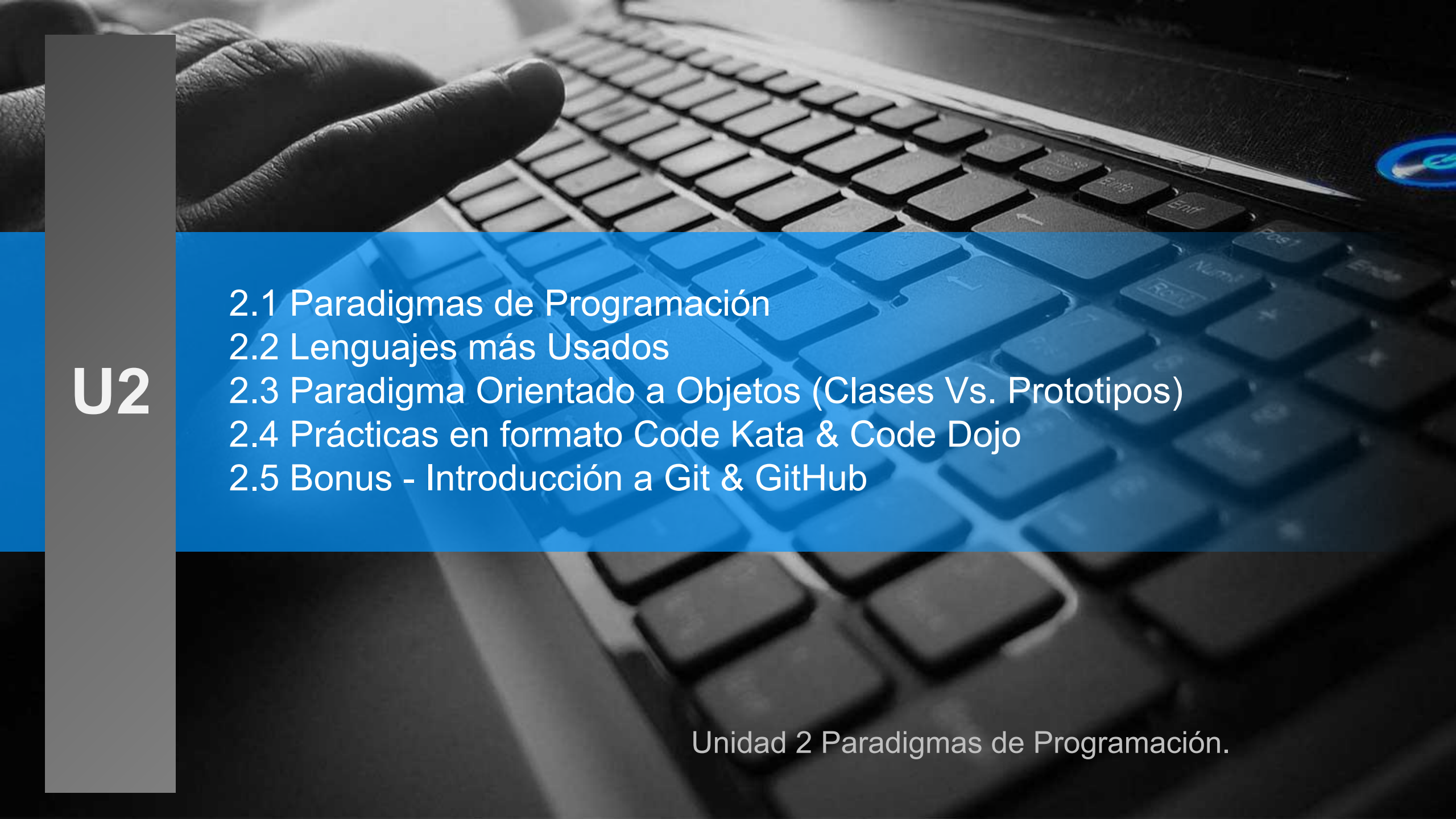
Hoy Comenzaremos este Tramo de formación juntos.
Sabemos que no será sencillo, pero material de sobra tenemos.

Vamos a poner a vuestra disposición una batería de conocimientos que sin dudarlo, les servirán para desempeñarse en el futuro laboral.

Acompañennos en este tramo poniendo lo mejor de Ustedes… para Ustedes.

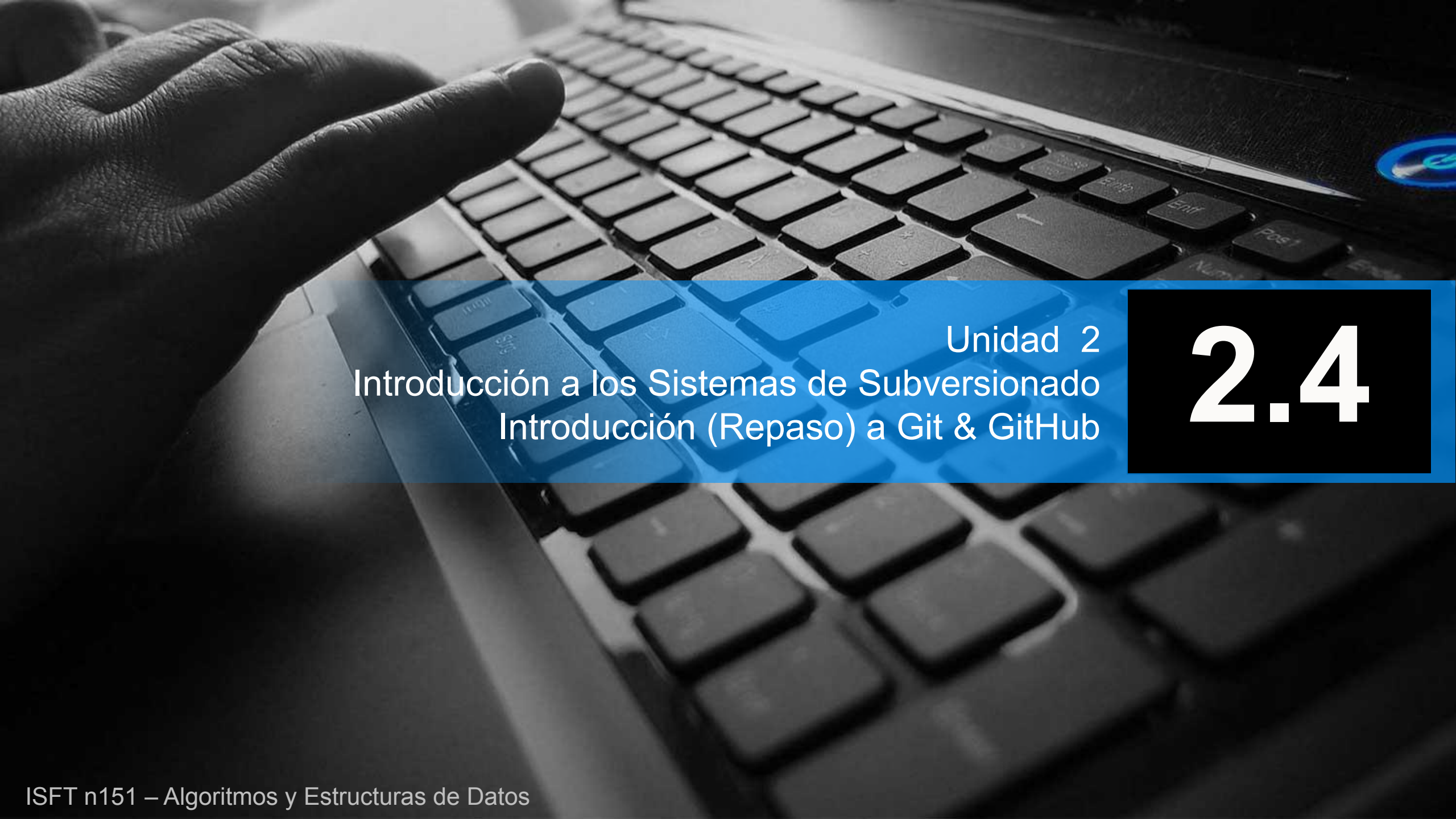
Unidad 2
Paradigmas de Programación

2



U2

2.1 Paradigmas de Programación
2.2 Lenguajes más Usados
2.3 Paradigma Orientado a Objetos (Clases Vs. Prototipos)
2.4 Prácticas en formato Code Kata & Code Dojo
2.5 Bonus - Introducción a Git & GitHub



Unidad 2

Introducción a los Sistemas de Subversionado
Introducción (Repaso) a Git & GitHub

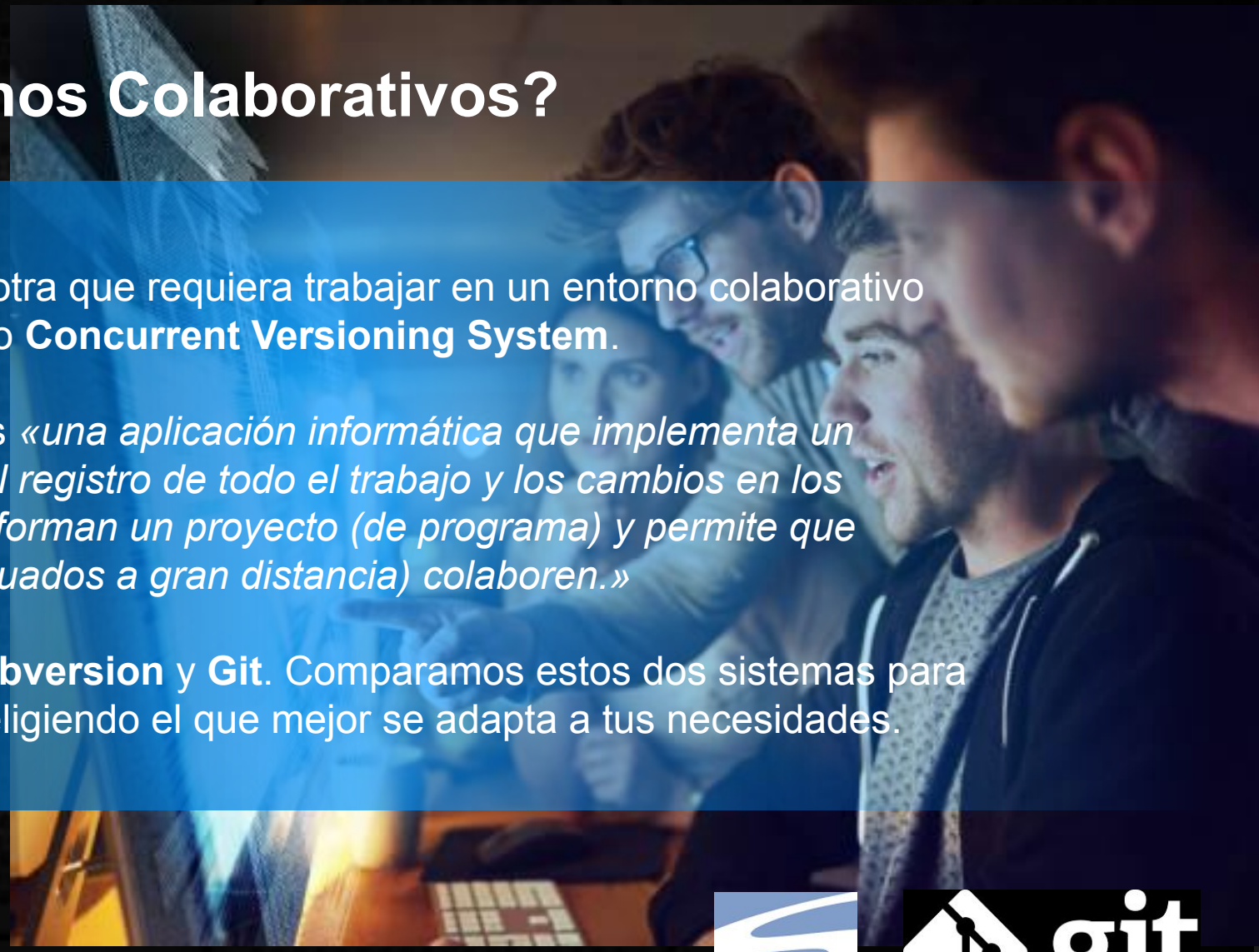
2.4

Como Desarrollo en entornos Colaborativos?

Si te dedicas a **programar aplicaciones o desarrollar páginas web en un entorno colaborativo**, seguramente te habrás preguntado:

- ¿Cómo planificar el trabajo?
- ¿Cómo realizar un control de las versiones?
- ¿Cómo sincronizar el trabajo?
- ¿Cómo facilitar la colaboración entre los distintos miembros que participan en el proyecto?

Como Desarrollo en entornos Colaborativos?



Para realizar ese tipo de tareas o cualquier otra que requiera trabajar en un entorno colaborativo necesitarás un **CVS**, también conocido como **Concurrent Versioning System**.

Según la definición de Wikipedia, un CVS es *«una aplicación informática que implementa un sistema de control de versiones: mantiene el registro de todo el trabajo y los cambios en los ficheros (código fuente principalmente) que forman un proyecto (de programa) y permite que distintos desarrolladores (potencialmente situados a gran distancia) colaboren.»*

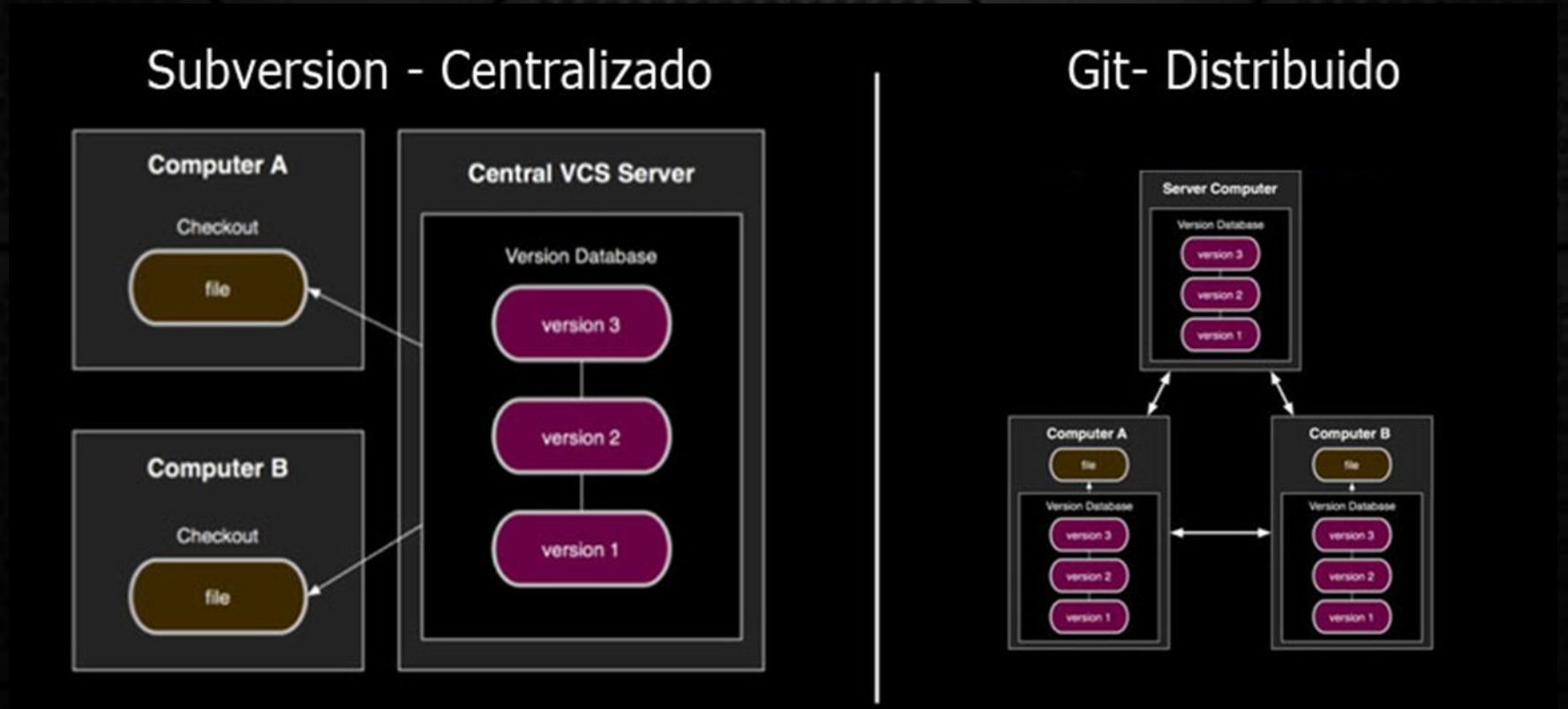
Los **CVS** mas conocidos y utilizados son **Subversion** y **Git**. Comparamos estos dos sistemas para que te sea más fácil escoger entre ambos, eligiendo el que mejor se adapta a tus necesidades.

SUBVERSION

git

Subversion vs Git | Sistema Centralizado vs Sistema Distribuido

Antes de conocer a cada una de las aplicaciones queremos dejar clara la principal **diferencia** entre los dos sistemas **CVS,** y es que mientras **Subversion** (SVN) es un sistema **centralizado**, **Git** es **distribuido**. Aunque luego explicaremos a que afecta este hecho a ambos sistemas así como sus características, en la imagen de abajo puedes observa dos gráficas una referente a un sistema centralizado como es el que tiene Subversion y otro un sistema distribuido como es el que tiene Git.



Subversion (SVN)

Subversion es un software libre que permite el control de versiones, permite el acceso al repositorio a través de redes ofreciendo la posibilidad de trabajar desde diferentes equipos, consiguiendo de esta manera la colaboración entre distintos miembros del proyecto.
El lanzamiento oficial fue en el 2000, pero su última actualización fue hace apenas unos meses, su uso es bajo licencia Apache y es multilengüe (disponible en Español). El funcionamiento de subversion se podría asimilar al de un gestor de ficheros de tipo repositorio.

Sistema Centralizado
Subversion se basa en un sistema centralizado, la principal **ventaja** de un **sistema centralizado** es su **simplicidad**, pero por el contrario no nos permite tener más de **un repositorio central** sobre el que trabajar, no podemos realizar **confirmaciones** (commit) si no estamos conectados al repositorio central y si estamos trabajando en un equipo de trabajo con muchos usuarios, la **colaboración** se complica.

Gi
t

Git vio la luz en 2005 y su última versión actualizada fue apenas unos meses. Desde su lanzamiento se ha convertido en un sistema de control de versiones con una funcionalidad plena (aunque su objetivo no era exactamente ese).
Se basa en un sistema distribuido, siendo perfecto para el trabajo colaborativo dada su enorme flexibilidad, ya que cada usuario dispone de edición a su propio repositorio y tiene capacidad de lectura de los repositorios de los otros usuarios.
Git identifica las acciones mediante una suma de comprobación (**checksum**) que impide cambiar los **contenidos** de cualquier archivo o directorio sin que se entere. No puedes perder **información** durante su transmisión o sufrir corrupción de archivos sin que Git lo detecte.
Como **Git** sólo añade información a su **base de datos interna** cuando realizas una acción o confirmas cambios, es difícil que el sistema haga algo que pueda ser deshecho o borre **información**. Esto hace que podamos experimentar sin el riesgo de estropear nuestro sistema de control de versiones.

Sistema Distribuido
En el caso de un **sistema distribuido** es posible **trabajar sin conexión al repositorio central** (trabajo offline), se facilita la colaboración, podemos tener tantos **repositorios externos** como queramos, la mayoría de las operaciones son locales por lo que el tiempo de ejecución de las mismas se reduce considerablemente y su instalación y configuración es muy sencilla en nuestro espacio de hosting.
Sus principales **desventajas** son que su **curva de aprendizaje** es alta y los flujos de trabajo (workflow) entre colaboradores pueden ser algo más complejos.

Diferencias entre Subversion y Git

Planteamos un ejemplo para que se entienda mejor la **diferencia entre ambos sistemas y nuestra comparativa Subversion vs Git**. Imaginémonos que somos un desarrollador que está fuera de la oficina y no tenemos conexión a Internet:

- Con **Subversion**, no podremos conectarnos al **repositorio central** y, por lo tanto no podremos realizar **confirmaciones** (commit) ni tener un control local de versiones del código fuente.
- Con **Git**, nuestra copia local es un **repositorio** y podemos hacer **confirmaciones** (commit) sobre éste y tener todas las ventajas del control de código fuente. Cuando volvamos a tener conexión con el repositorio central, podremos realizar confirmaciones sobre él.

- Mientras que **SVN** tiene la ventaja de que es más **sencillo** de aprender, **Git** se adapta mejor para desarrolladores que no están conectados continuamente al **repositorio central**.
- Además, **Git** es más **rápido** en ejecución que SVN y características avanzadas, como la creación de ramas de trabajo (**branching**) y la combinación de ramas (**merging**) están mejor definidas.

Esto explica que en **proyectos Open Source se utiliza principalmente Git**. Creamos una rama del proyecto principal, hacemos nuestros cambios sobre esa rama sin afectar a la versión en producción y generamos una petición al responsable del proyecto para que revise las modificaciones y las integre en producción.

SVN vs. Git: una comparación directa

Aunque muchos usuarios se preguntan cuál de los dos programas de control de versiones es mejor, no existe una respuesta general. La elección del sistema de control de versiones más adecuado para uno u otro proyecto dependerá de tus objetivos específicos. Ambos sistemas difieren en su estructura y en el proceso de trabajo resultante. La siguiente tabla resume sus principales diferencias:

	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

Resumen: Subversion vs Git

Git no es mejor que SVN, sólo trabajan de forma diferente. Si necesitamos un control de código fuente offline y tenemos la disposición de gastar más tiempo en aprender un sistema de control de versiones, Git es nuestra elección.
Si tenemos un sistema de control de código fuente estrictamente centralizado y estamos iniciándonos en el control de versiones, la **simplicidad de SVN** nos facilitará nuestro flujo de trabajo.

Estas son las principales ventajas de ambos sistemas:

Debes decantarte por Git cuando...

- ...no quieres depender de una conexión de red permanente, pues quieres trabajar en tu proyecto desde cualquier lugar.
- ...quieres seguridad en caso de fallo o pérdida de los repositorios principales.
- ...no necesitas contar con permisos especiales de lectura y escritura para los diferentes directorios (aunque, de ser así, será posible y complejo implementarlo).
- ...la transmisión rápida de los cambios es una de tus prioridades.

Subversion será la opción indicada, si...

- ...necesitas permisos de acceso basados en rutas de acceso para las diferentes áreas de tu proyecto.
- ...deseas agrupar todo tu trabajo en un solo lugar.
- ...trabajas con numerosos archivos binarios de gran tamaño.
- ...también quieres guardar la estructura de los directorios vacíos (estos son rechazados por Git, debido a que no contienen ningún tipo de contenido).

En caso de que las características enumeradas anteriormente no hayan sido decisivas para decantarse por alguno de los dos, siempre es recomendable realizar una prueba con cualquiera de los sistemas de control de versiones. En ambos casos encontrarás el apoyo de una gran comunidad, **proveedores de alojamiento de gran calidad como GitHub** y ofertas de soporte profesional

Introducción a GIT / GitHub

¿Qué es GIT?

Es un software de **control de versiones**, su propósito es llevar registro de los cambios en archivos de computadora y coordinar el trabajo que varias personas realizan sobre archivos compartidos (También puedes trabajar solo no hay problema 😃). Existe la posibilidad de trabajar de forma remota y una opción es GitHub

¿Qué es GitHub?

Es una plataforma de desarrollo colaborativo para alojar proyectos (en la nube) utilizando el sistema de control de versiones Git, Además cuenta con una herramienta muy útil que es GitHub Pages donde podemos publicar nuestros proyectos estáticos (HTML, CSS y JS) gratis.

¿Qué es GIT?

¿Qué es GIT?

Git se ha convertido en el estándar mundial para el control de versiones. Entonces, ¿qué es exactamente?

Git es un sistema de control de versiones distribuido, lo que significa que un clon local del proyecto es un repositorio de control de versiones completo. Estos repositorios locales plenamente funcionales permiten trabajar sin conexión o de forma remota con facilidad. Los desarrolladores confirman su trabajo localmente y, a continuación, sincronizan su copia del repositorio con la copia en el servidor. Este paradigma es distinto del control de versiones centralizado, donde los clientes deben sincronizar el código con un servidor antes de crear nuevas versiones.

La flexibilidad y popularidad de Git hacen que sea una excelente opción para cualquier equipo. Muchos desarrolladores y graduados universitarios ya saben cómo usar Git. La comunidad de usuarios de Git ha creado recursos para entrenar a desarrolladores y la popularidad de Git facilita la ayuda cuando sea necesario. Casi todos los entornos de desarrollo tienen compatibilidad con Git y las herramientas de línea de comandos de Git implementadas en cada sistema operativo principal.

git

¿Qué es GIT?

Ventajas de Git
Las ventajas de Git son muchas.

Desarrollo simultáneo
Todos tienen su propia copia local de código y pueden trabajar simultáneamente en sus propias ramas. Git funciona sin conexión, ya que casi todas las operaciones son locales.

Versiones más rápidas
Las ramas permiten el desarrollo flexible y simultáneo. La rama principal contiene código estable y de alta calidad desde el que se publica. Las ramas de características contienen trabajo en curso, que se combinan en la rama principal tras la finalización. Al separar la rama de versión del desarrollo en curso, es más fácil administrar código estable y enviar actualizaciones más rápidamente.

Integración integrada
Debido a su popularidad, Git se integra en la mayoría de las herramientas y productos. Cada IDE principal tiene compatibilidad integrada con Git y muchas herramientas admiten la integración continua, la implementación continua, las pruebas automatizadas, el seguimiento de elementos de trabajo, las métricas y la integración de características de informes con Git. Esta integración simplifica el flujo de trabajo diario.

Soporte técnico sólido de la comunidad
Git es de código abierto y se ha convertido en el estándar de facto para el control de versiones. No hay escasez de herramientas y recursos disponibles para que los equipos aprovechen. El volumen de compatibilidad de la comunidad con Git en comparación con otros sistemas de control de versiones facilita la ayuda cuando sea necesario.

¿Qué es GIT?

Ventajas de Git
Las ventajas de Git son muchas.

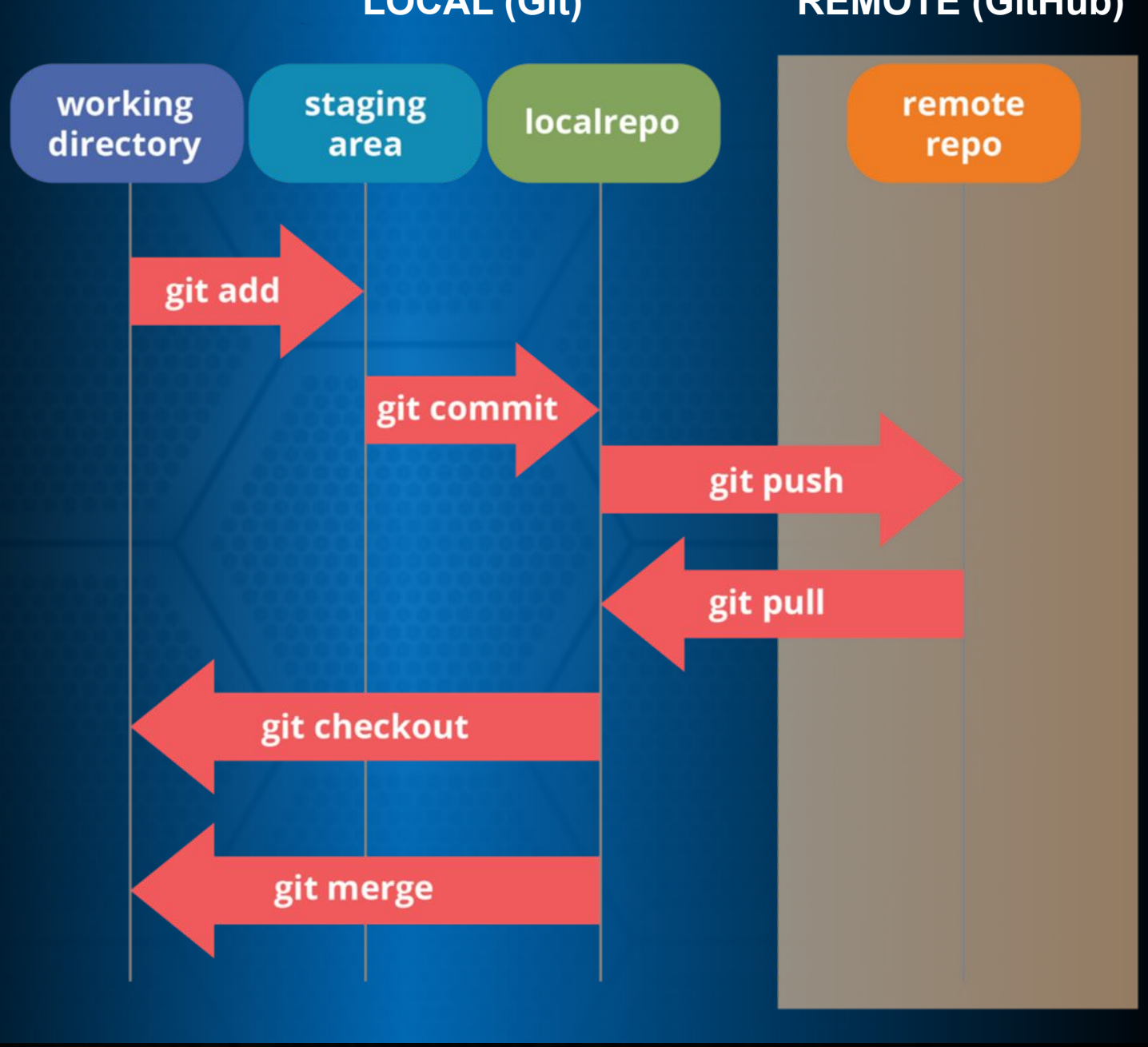
Git funciona con cualquier equipo
El uso de Git con una herramienta de administración de código fuente aumenta la productividad de un equipo fomentando la colaboración, aplicando directivas, automatizando procesos y mejorando la visibilidad y la rastreabilidad del trabajo. El equipo puede establecerse en herramientas individuales para el control de versiones, el seguimiento de elementos de trabajo y la integración e implementación continuas. O bien, pueden elegir una solución como GitHub o Azure DevOps que admita todas estas tareas en un solo lugar.

Solicitudes de incorporación de cambios
Use solicitudes de incorporación de cambios para analizar los cambios de código con el equipo antes de combinarlos en la rama principal. Las discusiones en las solicitudes de incorporación de cambios son valiosas para garantizar la calidad del código y aumentar el conocimiento en todo el equipo. Las plataformas como GitHub y Azure DevOps ofrecen una experiencia enriquecida de solicitudes de incorporación de cambios donde los desarrolladores pueden examinar los cambios de archivos, dejar comentarios, inspeccionar confirmaciones, ver compilaciones y votar para aprobar el código.

Directivas de rama
Teams puede configurar GitHub y Azure DevOps para aplicar flujos de trabajo y procesos coherentes en todo el equipo. Pueden configurar directivas de rama para asegurarse de que las solicitudes de incorporación de cambios cumplen los requisitos antes de la finalización. Las directivas de rama protegen ramas importantes mediante la prevención de inserciones directas, la necesidad de revisores y la garantía de compilaciones limpias.

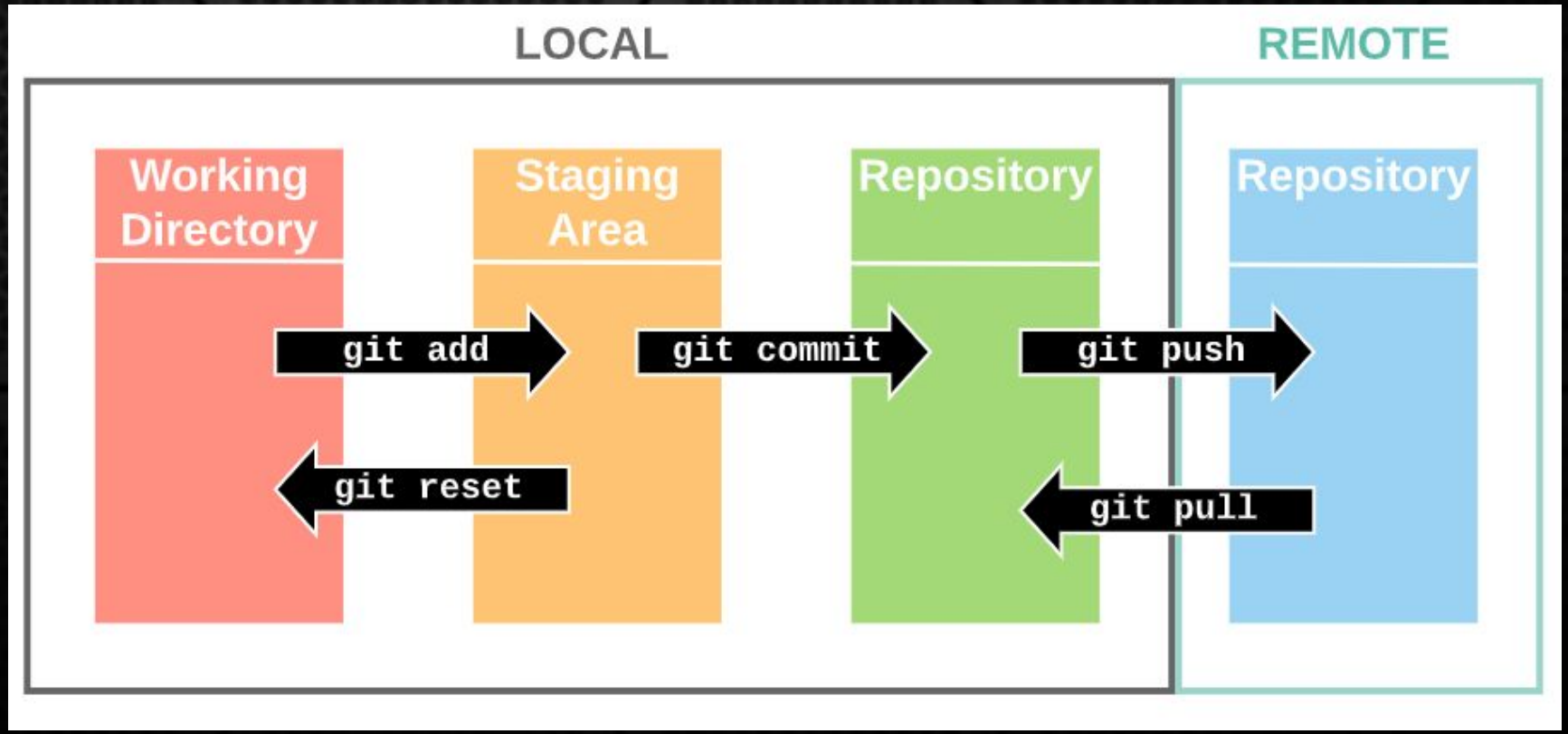
Flujo de trabajo de GIT

Tenemos nuestro directorio local (una carpeta en nuestro pc) con muchos archivos, Git nos irá registrando los cambios de archivos o códigos cuando nosotros le indiquemos, así podremos viajar en el tiempo retrocediendo cambios o restaurando versiones de código, ya sea en Local o de forma Remota (servidor externo). En la práctica quedará más claro.



Flujo de trabajo de GIT

- Git se basa en **snapshots** (instantáneas) del código en un estado determinado, que viene dado por el autor y la fecha.
- Un ***Commit*** es un conjunto de cambios guardados en el repositorio de *Git* y tiene un identificador *SHA1* único.
- Las **ramas** (*branches*) se pueden pensar como una línea de tiempo a partir de los *commit*. Hay siempre como mínimo una rama principal o predefinida llamada ***Master***.
- ***Head*** es el puntero al último commit en la rama activa.
- ***Remote*** se refiere a sitios que hospedan repositorios remotos como GitHub.



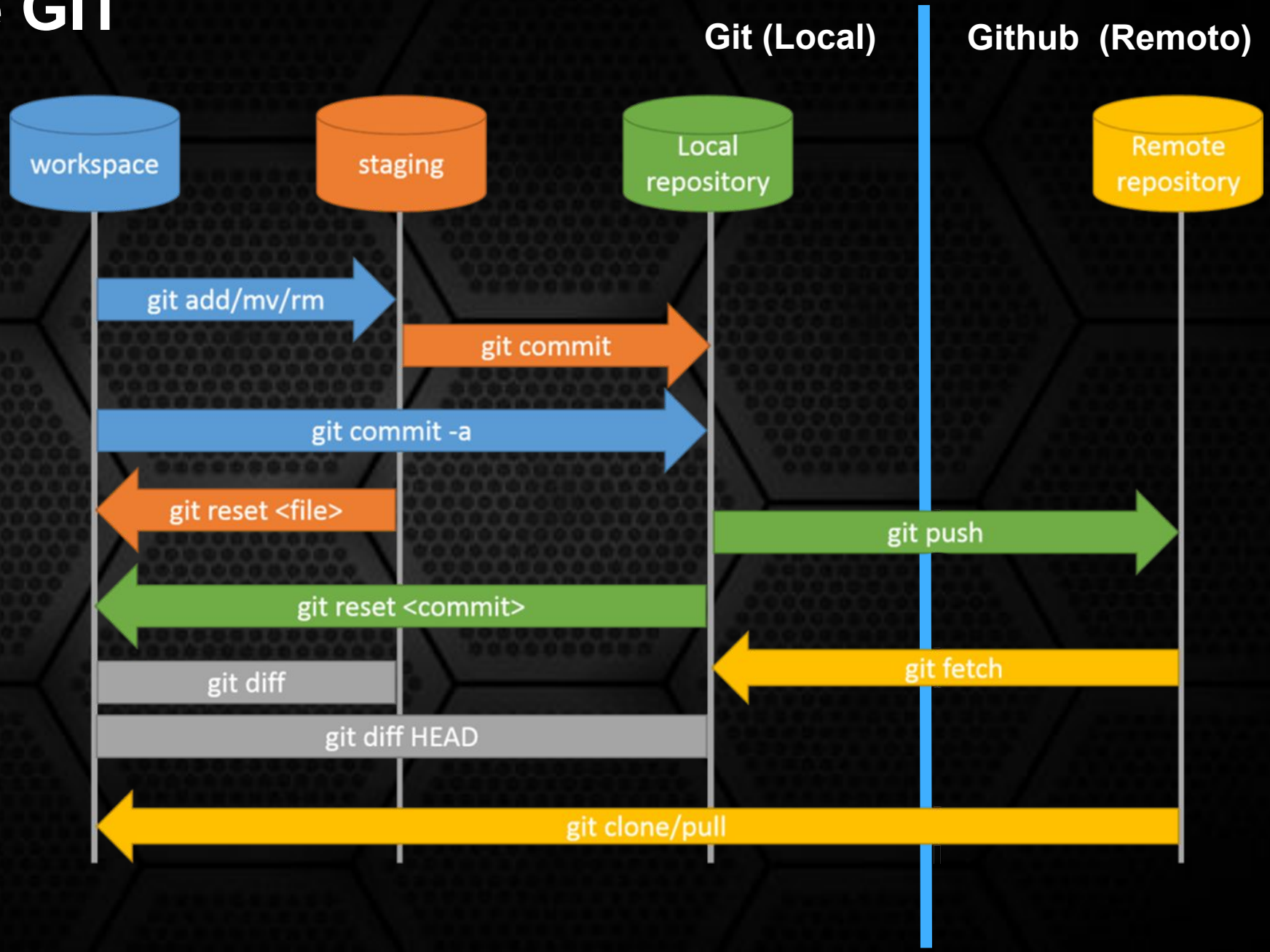
Flujo de trabajo de GIT

Si trabajamos en local (comenzamos en la imagen por la izquierda), inicializamos el directorio de trabajo (working directory). Podemos trabajar (editar ficheros) en el directorio de trabajo.

Con el comando *Git add* enviamos los cambios a *staging*, que es un estado intermedio en el que se van almacenando los archivos a enviar en el *commit*. Finalmente con *commit* lo enviamos al repositorio local.

Si queremos colaborar con otros, con *push* subimos los archivos a un repo remoto y mediante *pull* podríamos traer los cambios realizados por otros en remoto hacia nuestro directorio de trabajo.

Si comenzamos trabajando en remoto, lo primero que hacemos es un clon de la información en el directorio local.



Comandos GIT / GitH

Basic Git Commands:

Git: configurations

```
$ git config --global user.name "FirstName LastName"
$ git config --global user.email "your-email@email-provider.com"
$ git config --global color.ui true
$ git config --list
```

Git: starting a repository

```
$ git init
$ git status
```

Git: staging files

```
$ git add <file-name>
$ git add <file-name> <another-file-name> <yet-another-file-name>
$ git add .
$ git add --all
$ git add -A
$ git rm --cached <file-name>
$ git reset <file-name>
```

RubyGarage.org,
Via: @ml.india

Git: committing to a repository

```
$ git commit -m "Add three files"
$ git reset --soft HEAD^
$ git commit --amend -m <enter your message>
```

Git: pulling and pushing from and to repositories

```
$ git remote add origin <link>
$ git push -u origin master
$ git clone <clone>
$ git pull
```

Git: branching

```
$ git branch
$ git branch <branch-name>
$ git checkout <branch-name>
$ git merge <branch-name>
$ git checkout -b <branch-name>
```

GIT - Comandos básicos

```js
// Conocer la versión de git instalada
git version
```

```js
// Ayuda sobre los comandos
git help
```

```js
// Iniciar un nuevo repositorio
// Crear la carpeta oculta .git
git init
```

```js
// Ver que archivos no han sido registrados
git status
```

GIT - Comandos básicos

```js
// Agregar todos los archivos para que esté pendiente de los cambios
git add .
```

```js
// Crear commit (fotografía del proyecto en ese momento)
git commit -m "primer commit"
```

```js
// Muestra la lista de commit del mas reciente al más antigüo
git log
```

En resumidas cuentas nosotros realizamos cambios en nuestros archivos, el comando status verificará que archivos han sidos modificados. Cuando deseemos registrar esos cambios tendremos que agregarlos con add . así ya estará listo para poder hacer un commit. El commit realiza la copia de ese instante para poder volver en el tiempo si es que es necesario.

GIT – Trucos & Comandos básicos

```js
// Muestra en una línea los commit realizados
git log --oneline
```

```js
// Muestra en una línea los commit realizados pero más elegante
git log --oneline --decorate --all --graph
```

```js
// Solo muestra los archivos modificados
git status -s
```

```
// Vemos información de la rama maestra
git status -s -b
git status -sb //Hace lo mismo que el comando anterior
```

Diferencias entre -- y -

`--decorate` hace referencia a una palabra

`-s` hace referencia al comando o a varios comandos, `-sa` serían dos comandos diferentes

Creando alias globales

Los alias nos sirven para crear atajos de comandos, podemos guardar diferentes alias de forma global y quedarán guardados en la configuración de git.

```js
// Guardamos el alias "lg" que ejecutará todo lo que está entre comillas
git config --global alias.lg "log --oneline --decorate --all --graph"
```

```js
// Para ver el archivo config con los alias creados
git config --global -e
```

Vim es el editor de código en la línea de comandos

> **Salir del modo edición "Vim"**
>
> Para salir del modo edición de la líneas de comando precionar `:q` , en caso de realizar algún cambio sin guardar escribir `:qa`
>
> `:q!` también sirve para salir sin guardar

Creando alias globales

```js
// Modo lectura sin poder modificar
git config --global -l
```

```js
// Realiza el add . y commit más mensaje al mismo tiempo
git commit -am "más comandos agregados"
```

```js
// Para editar un commit, como por ej: el mensaje
git commit --amend
```

Trucos de editor Vim

- `i` para comenzar a editar
- `esc` para salir del modo edición
- `wq` para guardar y salir
- `q!` salir sin guardar cambios

Viajes a través de los commit

Vamos a conocer como podemos movernos entre los diferentes commit que tengamos registrados, supongamos que tenemos los siguientes commit:

- f82f457 (HEAD -> master) mas comandos agregados
- f52f3da nuevos comandos en fundamentos.md
- e4ab8af mi primer commit

```js
// Viajamos al commit en específico f52f3da
git reset --mixed f52f3da
```

```js
// Viajamos al commit en específico f52f3da y eliminamos los cambios futuros
git reset --hard f52f3da
```

```js
// Muestra todos los cambios incluso si borramos los commit
git reflog
```

Viajes a través de los commit

```js
// Viajamos al commit en específico f52f3da y podemos restaurar los archivos
git reset --hard f52f3da
```

Si no hicimos un commit pero aún así queremos revertir los cambios en un archivo específico podríamos utilizar el siguiente comando:

```
git checkout -- nombreArchivo.conExtensión
```

Si deseamos destruir todos los cambios sin haber realizado un commit podemos utilizar:

```
git reset --hard
```

Renombrar archivos

Puede que queramos renombrar un archivo, es recomendable hacerlo directamente en la línea de comandos para registrar los cambios con git.

```js
// Cambiar nombre
git mv nombreOriginal.vue nombreNuevo.vue
```

Recuerden hacer el commit para registrar los cambios en git

Eliminar archivos

```js
// Cambiar nombre
git rm nombreArchivo.vue
```

También recordar hacer el commit para salgar cambios en git

Ignorando Archivos

Para no hacer seguimiento de carpetas o archivos, debemos crear el siguiente archivo:

- .gitignore

Su estructura de ejemplo sería así:

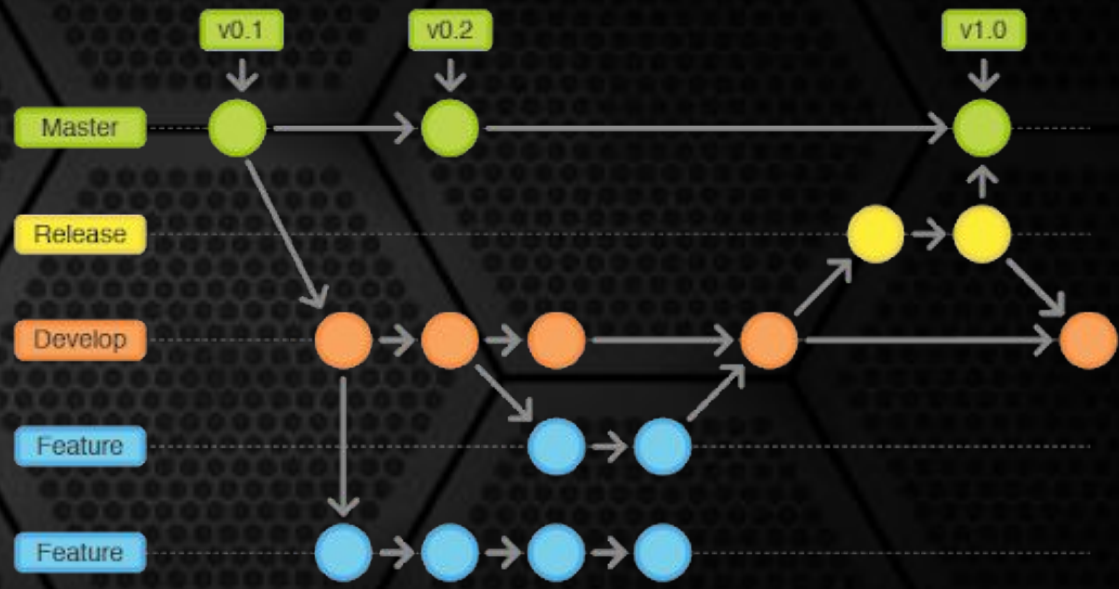
```js
arhivo.js // Ignora el archivo en cuestion
*.js // Ignora todos los arhivos con extensión .js
node_modules/ //Ignora toda la carpeta
```

Comandos GIT Branch

Branch (Ramas) GIT / GitHub

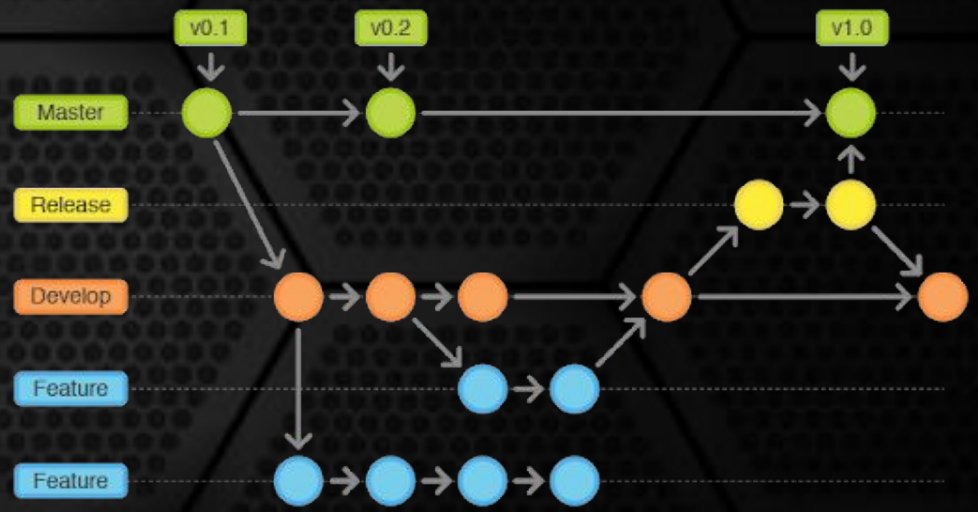
En el día a día del trabajo con Git una de las cosas útiles que podemos hacer es trabajar con ramas. Las ramas son caminos que puede tomar el desarrollo de un software, algo que ocurre naturalmente para resolver problemas o crear nuevas funcionalidades. En la práctica permiten que nuestro proyecto pueda tener diversos estados y que los desarrolladores sean capaces de pasar de uno a otro de una manera ágil.

Ramas podrás usar en muchas situaciones. Por ejemplo imagina que estás trabajando en un proyecto y quieres implementar una nueva funcionalidad en la que sabes que quizás tengas que invertir varios días. Posiblemente sea algo experimental, que no sabes si llegarás a incorporar, o bien es algo que tienes claro que vas a querer completar, pero que, dado que te va a ocupar un tiempo indeterminado, es posible que en medio de tu trabajo tengas que tocar tu código, en el estado en el que lo tienes en producción.



Branch (Ramas) GIT / GitHub

.
Bajo el supuesto anterior lo que haces es crear una rama. Trabajas dentro de esa rama por un tiempo, pero de repente se te cuelga el servidor y te das cuenta que hay cosas en el proyecto que no están funcionando correctamente. Los cambios de la rama no están completos, así que no los puedes subir. En ese instante, lo que las ramas Git te permiten es que, lanzando un sencillo comando, poner de nuevo el proyecto en el estado original que tenías, antes de empezar a hacer esos cambios que no has terminado. Perfecto! solucionas la incidencia, sobre el proyecto original y luego puedes volver a la rama experimental para seguir trabajando en esa idea nueva.
Llegará un momento en el que, quizás, aquellos cambios experimentales los quieras subir a producción. Entonces harás un proceso de fusionado entre la rama experimental y la rama original, operación que se conoce como merge en Git. Las aplicaciones de las ramas son, como puedes imaginar, bastante grandes. Espero haber podido explicar bien una de ellas y que te hayas podido hacer una idea antes de seguir la lectura de este artículo.



Ramas o branch

Hasta el momento solo hemos trabajado en la rama "master" pero puede que necesitemos crear diferentes ramas para los seguimientos de git.

```js
// Crea una nueva rama
git branch nombreRama
```

```js
// Nos muestra en que rama estamos
git branch
```

```js
// Nos movemos a la nueva rama
git checkout nombreRama
```

Ramas o branch

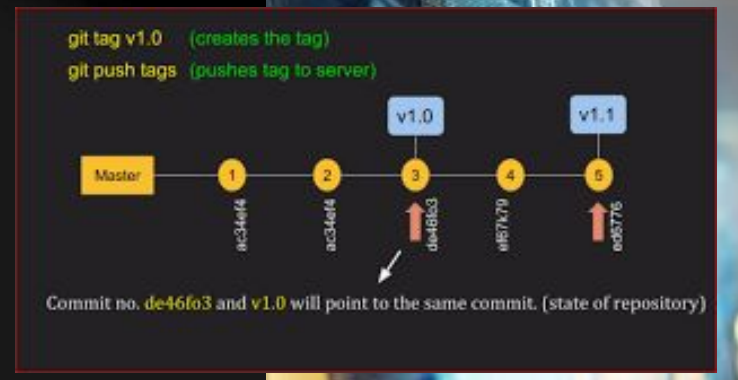
Podemos unir la rama master con la nueva, para eso tenemos que estar en la master para ejecutar el siguiente comando:

```js
// Nos movemos a la nueva rama
git merge nombreRama
```

```js
// Eliminar una rama
git branch -d nombreRama
```

> **Atajos**
>
> Podemos utilizar `git checkout -b nuevaRama` para crear la nuevaRama y viajar a ella.

Comandos GIT Tags



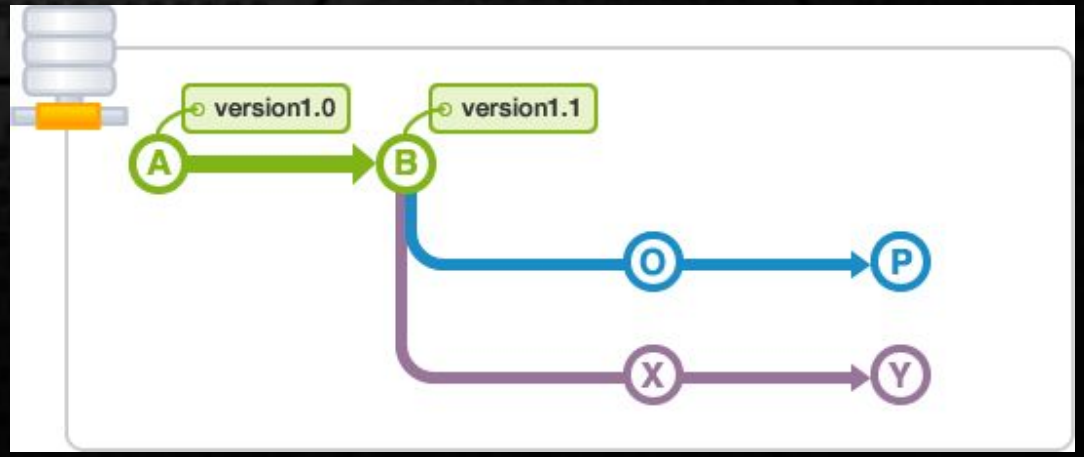
Tags (Versiones) GIT / GitHub

Aprende a usar Git Tag. Los tag son una manera de etiquetar estados de tu repositorio, que se usa comúnmente para indicar las versiones o releases de un proyecto mantenido con Git.

Git tiene la posibilidad de marcar estados importantes en la vida de un repositorio, algo que se suele usar habitualmente para el manejo de las releases de un proyecto. A través del comando "git tag" podemos crear etiquetas, en una operación que se conoce comúnmente con el nombre de "tagging".

Numeración de las versiones
Git es un sistema de control de versiones. Por tanto permite mantener todos los estados por los que ha pasado cualquier de sus archivos. Cuando hablamos de **Git tag** no nos referimos a la versión de un archivo en particular, sino de todo el proyecto de manera global. Sirve para etiquetar con un tag el estado del repositorio completo, algo que se suele hacer cada vez que se libera una nueva versión del software.

Las versiones de los proyectos las define el desarrollador, pero no se recomienda crearlas de manera arbitraria. En realidad la recomendación sería darle un valor semántico.
Esto no tiene nada que ver con Git, pero lo indicamos aquí porque es algo que consideramos interesante que sepas cuando empiezas a gestionar tus versiones en proyectos.



Tags

Con los tags podemos hacer versiones de nuestro proyecto.

```js
// Crear un tags
git tag versionAlpha -m "versión alpha"
```

```js
// Listar tags
git tag
```

```js
// Borrar tags
git tag -d nombreTags
```

```js
// Hacer una versión en un commit anterior ej: f52f3da
git tag -a nombreTag f52f3da -m "version alpha"
```

```js
// Mostrar información del tag
git show nombreTag
```

¿Qué es GitHub?

GITHUB

Github es un portal creado para alojar el código de las aplicaciones de cualquier desarrollador, y que fue comprada por Microsoft en junio del 2018. La plataforma está creada para que los desarrolladores suban el código de sus aplicaciones y herramientas, y que como usuario no sólo puedas descargarte la aplicación, sino también entrar a su perfil para leer sobre ella o colaborar con su desarrollo.

Como su nombre indica, la web utiliza el sistema de control de versiones Git diseñado por Linus Torvalds. Un sistema de gestión de versiones es ese con el que los desarrolladores pueden administrar su proyecto, ordenando el código de cada una de las nuevas versiones que sacan de sus aplicaciones para evitar confusiones. Así, al tener copias de cada una de las versiones de su aplicación, no se perderán los estados anteriores cuando se va a actualizar.

Así pues, Git es uno de estos sistemas de control, que permite comparar el código de un archivo para ver las diferencias entre las versiones, restaurar versiones antiguas si algo sale mal, y fusionar los cambios de distintas versiones. También permite trabajar con distintas ramas de un proyecto, como la de desarrollo para meter nuevas funciones al programa o la de producción para depurar los bugs.

Las principales características de la plataforma es que ofrece las mejores características de este tipo de servicios sin perder la simplicidad, y es una de las más utilizadas del mundo por los desarrolladores. Es multiplataforma, y tiene multitud de interfaces de usuario.

Así pues, Github es un portal para gestionar las aplicaciones que utilizan el sistema Git. Además de permitirte mirar el código y descargarte las diferentes versiones de una aplicación, la plataforma también hace las veces de red social conectando desarrolladores con usuarios para que estos puedan colaborar mejorando la aplicación.

Crear una nuevo repositorio

Para subir nuestro proyecto debemos crear un nuevo repositorio, al momento de la creación nos mostrará una serie de comandos para subir el proyecto.

```
git remote add origin https://github.com/bluuweb/tutorial-github.git
git push -u origin master
```

Al ejecutar estas líneas de comando te pedirá el usuario y contraseña de tu cuenta de github.

```js
// Nos muestra en que repositorio estamos enlazados remotamente.
git remote -v
```

Subir los tags

Por defecto si creaste un proyecto con diferentes versiones no subirá los tags, para eso tenemos el siguiente comando.

```
git push --tags
```

Push

Al ejecutar el comando git push estaremos subiendo todos los cambios locales al servidor remoto de github, ten en cuenta que tienes que estar enlazado con tu repositorio, para eso puedes utilizar git remote -v luego ejecuta:

```
git push
```

Pull

Cuando realizamos cambios directamente en github pero no de forma local, es esencial realizar un pull, donde descargaremos los cambios realizados para seguir trabajando normalmente.
Es importante estar enlazados remotamente, puedes verificar con: git remote -v, luego ejecuta:

```
git pull
```

Fetch

Este comando hace la comparación de nuestros archivos locales con los del servidor, si existiera alguna diferencia nos pediría realizar un get pull para realizar un match de nuestros arhivos locales.

```
git fetch
```

Clonar repositorio

Para descargar un repositorio completo basta con tomar la url ej: https://github.com/bluuweb/tutorial-github.git y ejecutar el siguiente comando en alguna carpeta de su computadora.

```
git clone https://github.com/bluuweb/tutorial-github.git nombreCarpeta
```

¿Qué es GitHub Pages?

¿Qué es GitHub Pages?

Las páginas web estáticas se han vuelto de moda y con buena razón, son increíblemente rápidas y con un número cada vez mayor de servicios de alojamiento compatibles, bastante fáciles de configurar.

Github Pages, la cual permite alojar sitios web estáticos sin necesidad de tener conocimientos en servidores. GitHub pages permite dos modalidades de publicación:

La primera es **“User site”** (solo se podrá tener un sitio de este tipo por cuenta); en este caso el sitio web será publicado en ***username.github.io*** (siendo username el nombre de usuario de la cuenta).

Ej por sitio: http://jloemig.github.io

La segunda opción es **“Project site”** (proyectos ilimitados) el cual será publicado en ***username*.github.io/*repository*** (siendo repository el nombre del repositorio).

Ej por proyecto: http://jloemig.github.io/GH2

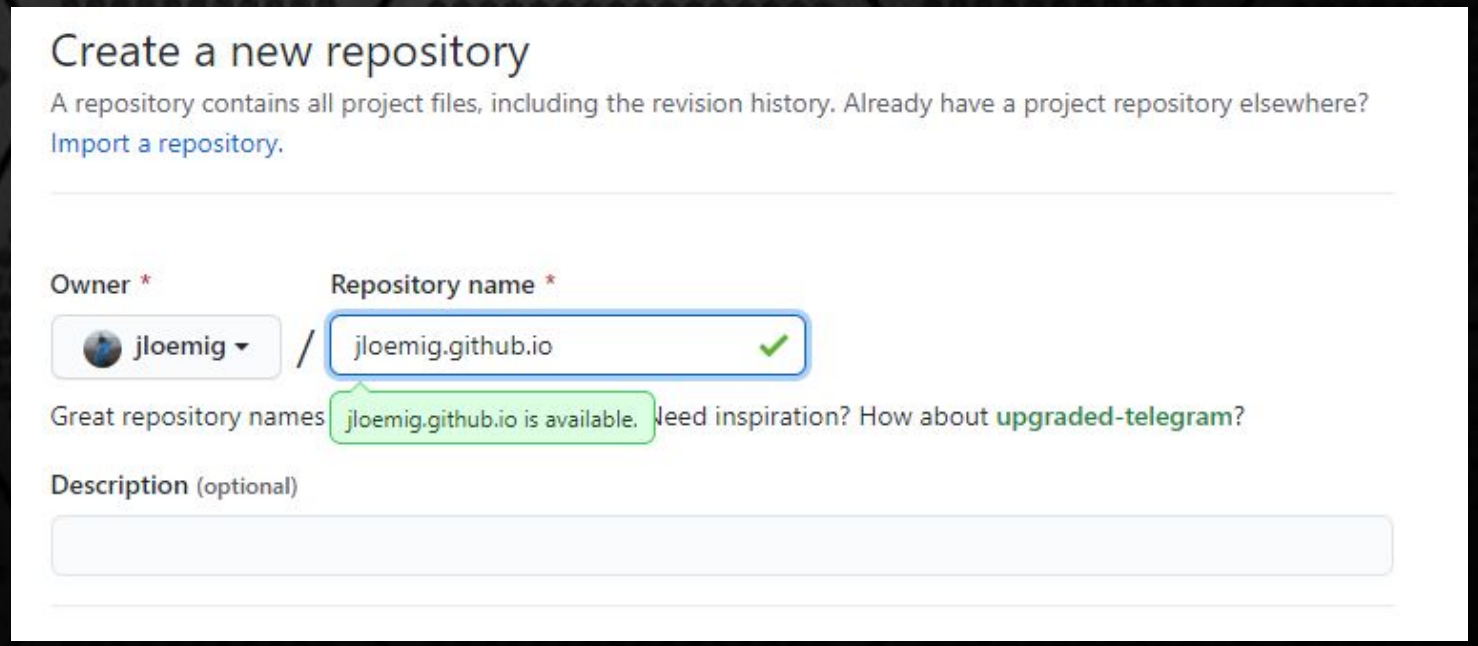
Nuestra GitHub Pages

Las páginas web estáticas se han vuelto de moda y con buena razón, son increíblemente rápidas y con un número cada vez mayor de servicios de alojamiento compatibles, bastante fáciles de configurar.

Solo Admite HTML CSS y JavaScript !!!

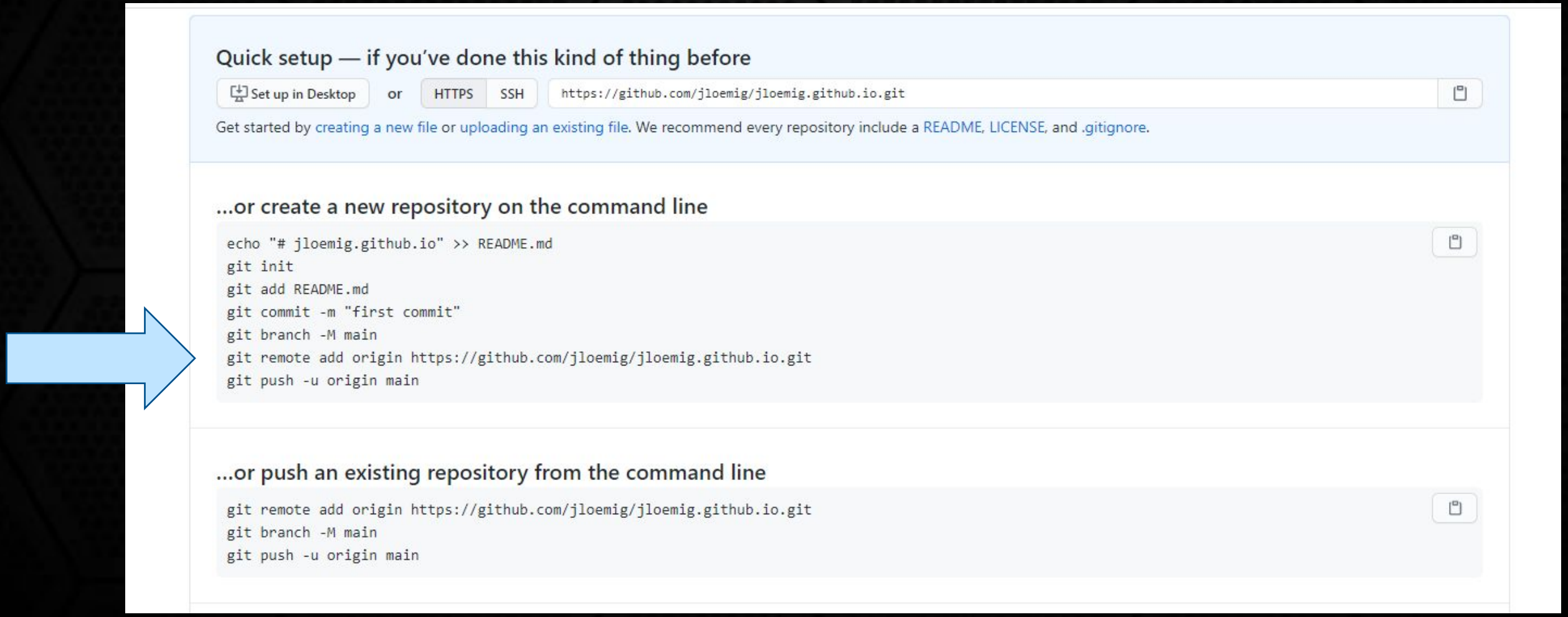
Ejemplo página de aterrizaje: http://jloemig.github.io/GH2 (puede ser por site o proyect)

Create un Nuevo Repo



Nuestra GitHub Pages

Create un Nuevo Repo



Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH https://github.com/jloemig/jloemig.github.io.git

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

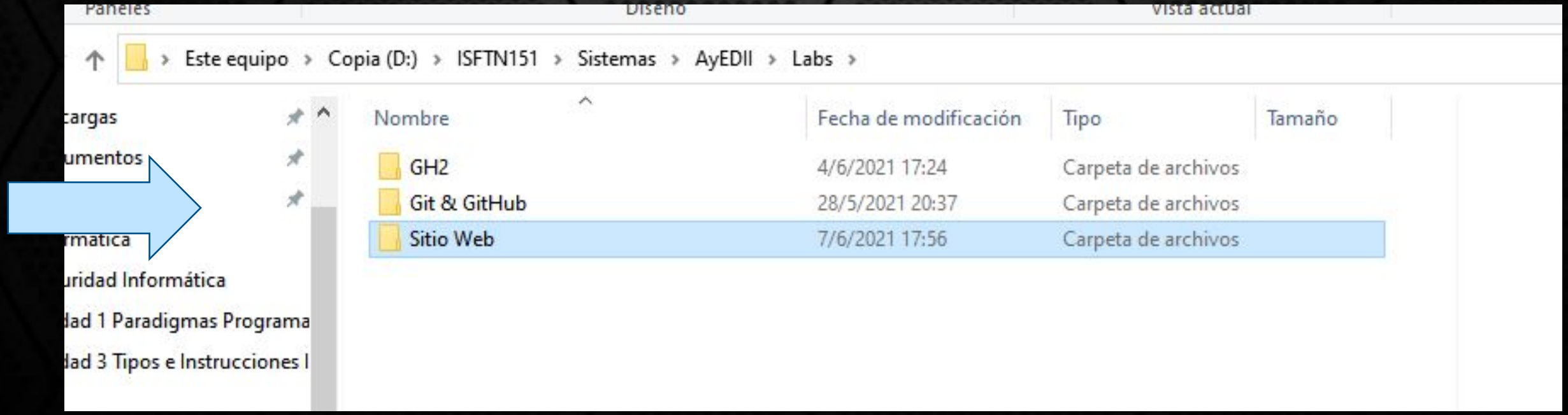
```
echo "# jloemig.github.io" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/jloemig/jloemig.github.io.git
git push -u origin main
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/jloemig/jloemig.github.io.git
git branch -M main
git push -u origin main
```

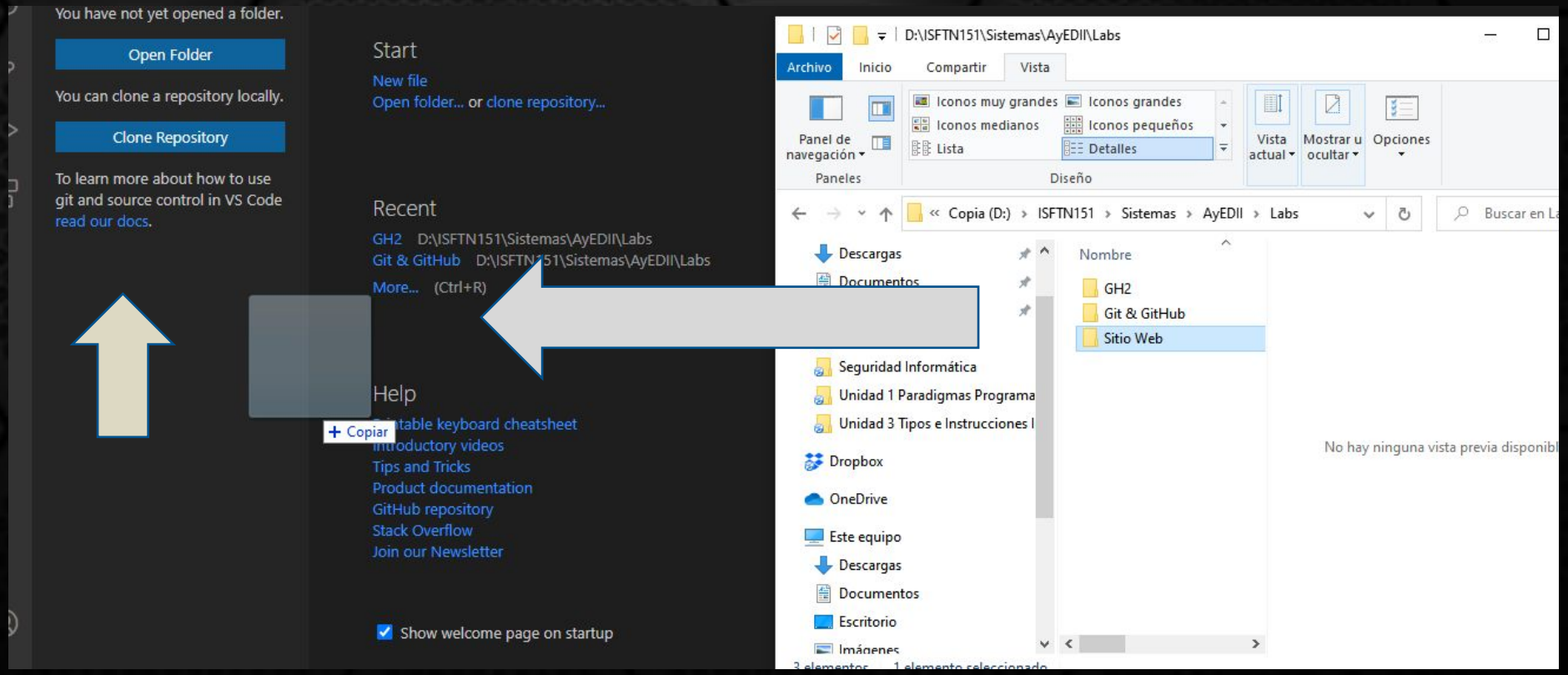
Nuestra GitHub Pages

Creamos una Carpeta para la Pagina del “Sitio”
La “drageamos” al Visual Studio Code



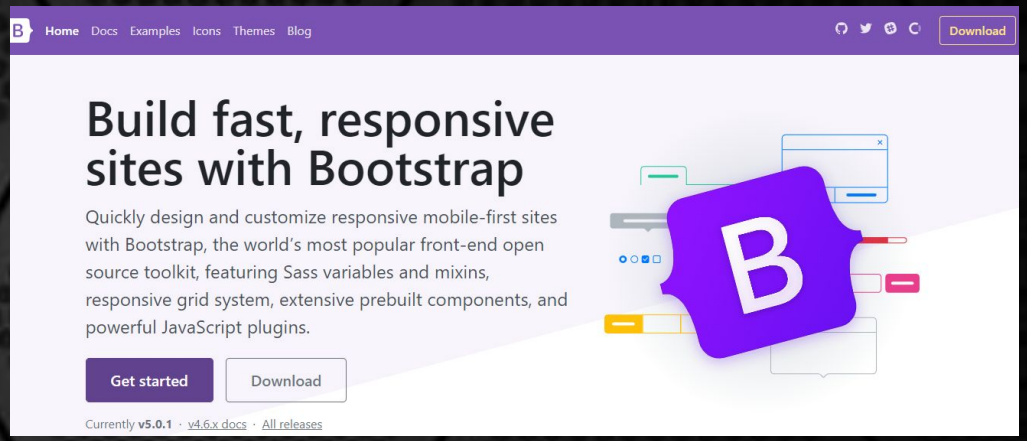
Nuestra GitHub Pages

Creamos una Carpeta para la Pagina del “Sitio”
La “dragemaos” al VS Code



Nuestra GitHub Pages

1. Creamos un Index.html
2. Ir a https://getbootstrap.com/docs/5.0/getting-started/introduction/l
3. Copy started template
4. Guardar el archivo
5. Abrir terminal



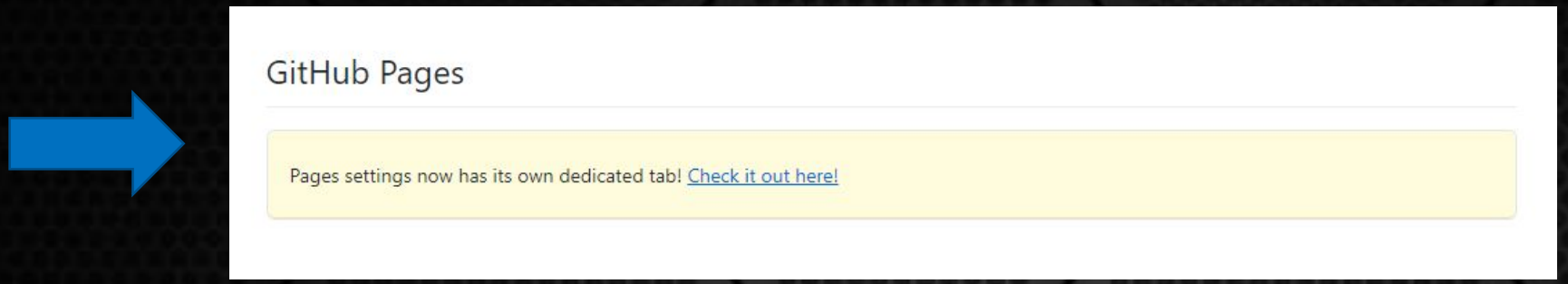
6. Git init
7. git add ..
8. git config user.email jloemig@gmail.com
9. gir commit -m "Web page inicio"
10. git remote add origin https://github.com/jloemig/jloemig.github.io.git
11. git push -u origin master

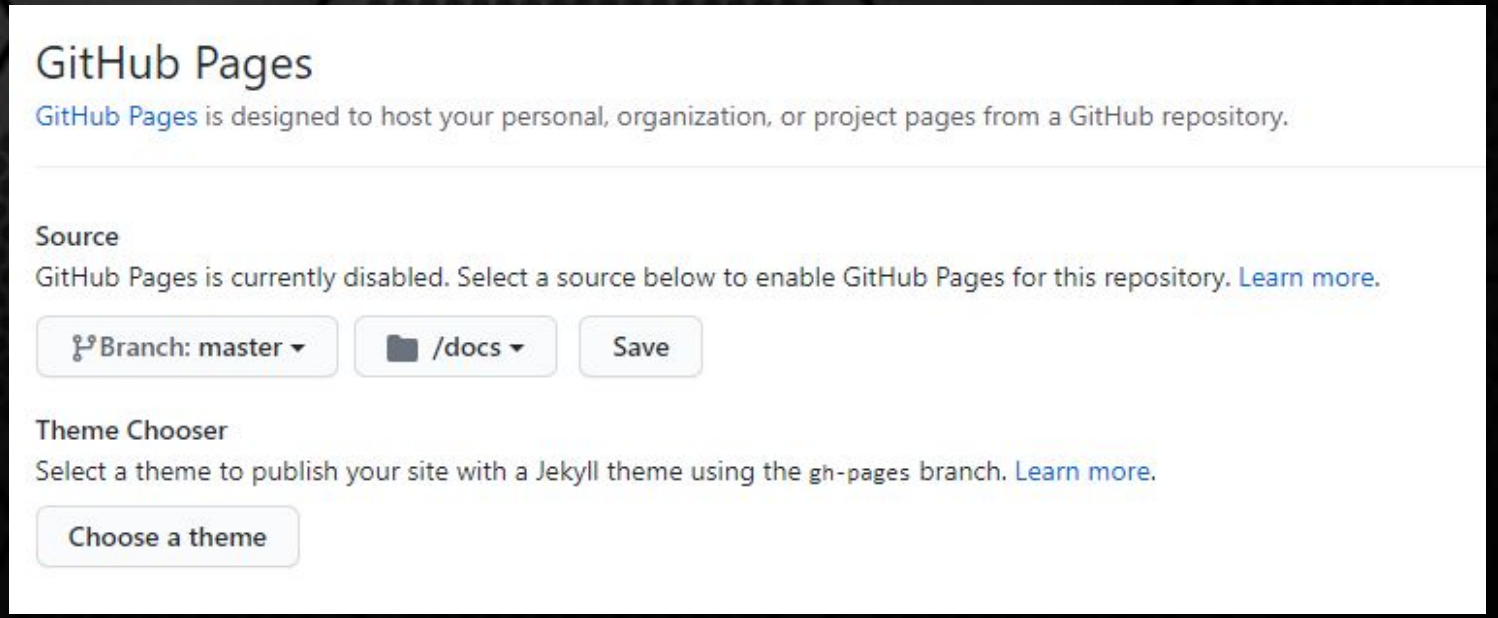
Nuestra GitHub Pages x repositorio

1. Volvemos la repo GH2
2. Creamos una carpeta docs (ahí esta la pag d eaterrizaje del proyecto)
3. Creamos una pagina.
4. Copiamos https://getbootstrap.com/docs/5.0/getting-started/introduction/
5. Copiamos jumbotron https://mdbootstrap.com/docs/standard/extended/jumbotron/ o cualquiera..

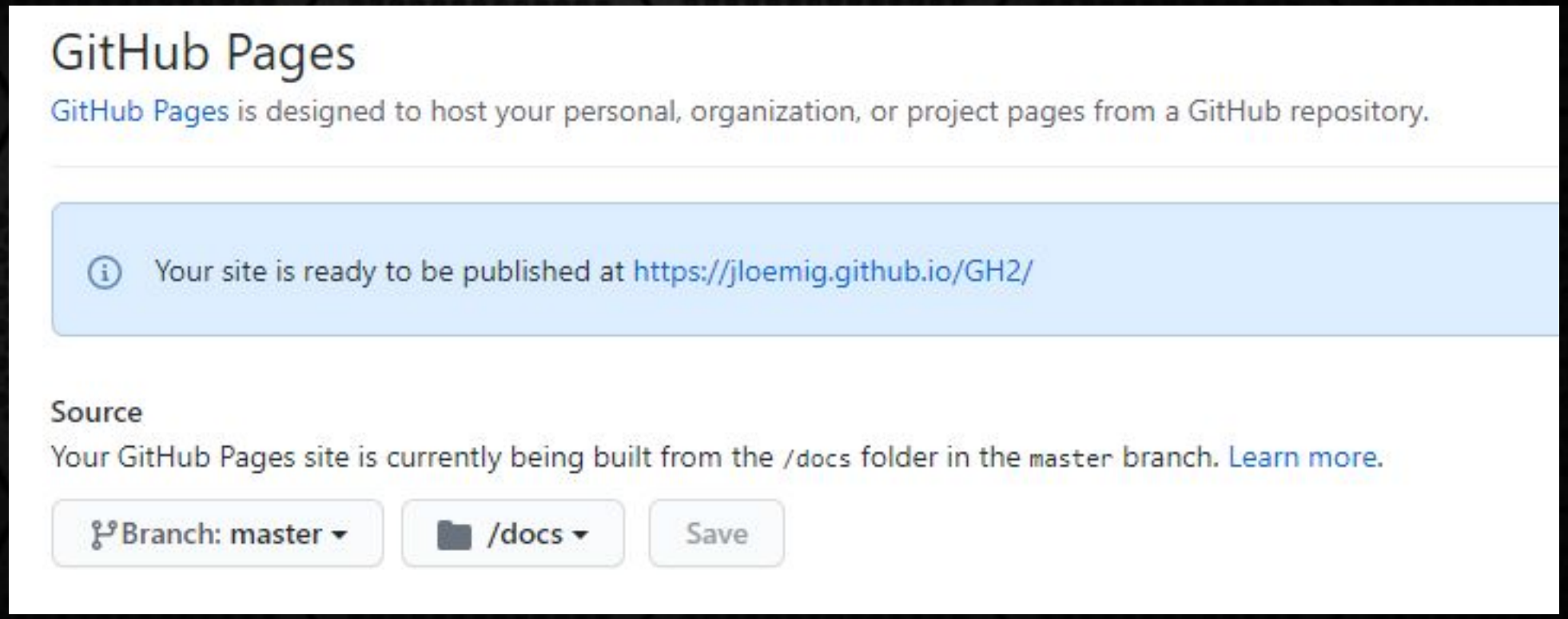
Nuestra GitHub Pages x repositorio

Settings - Pages

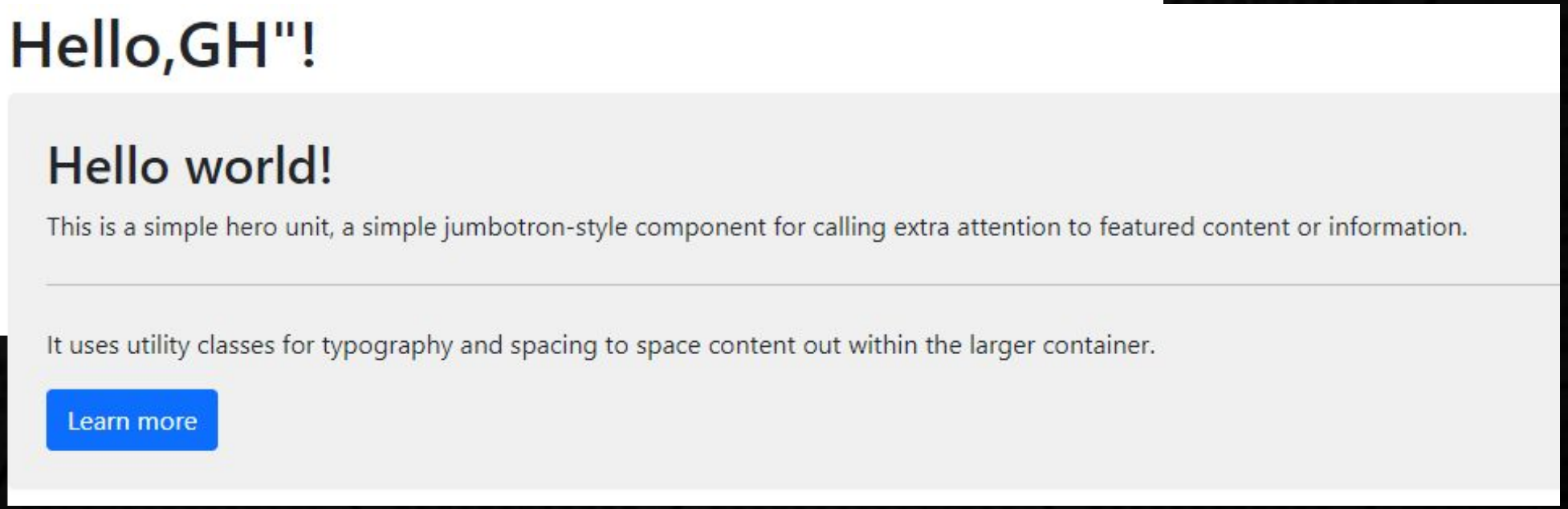




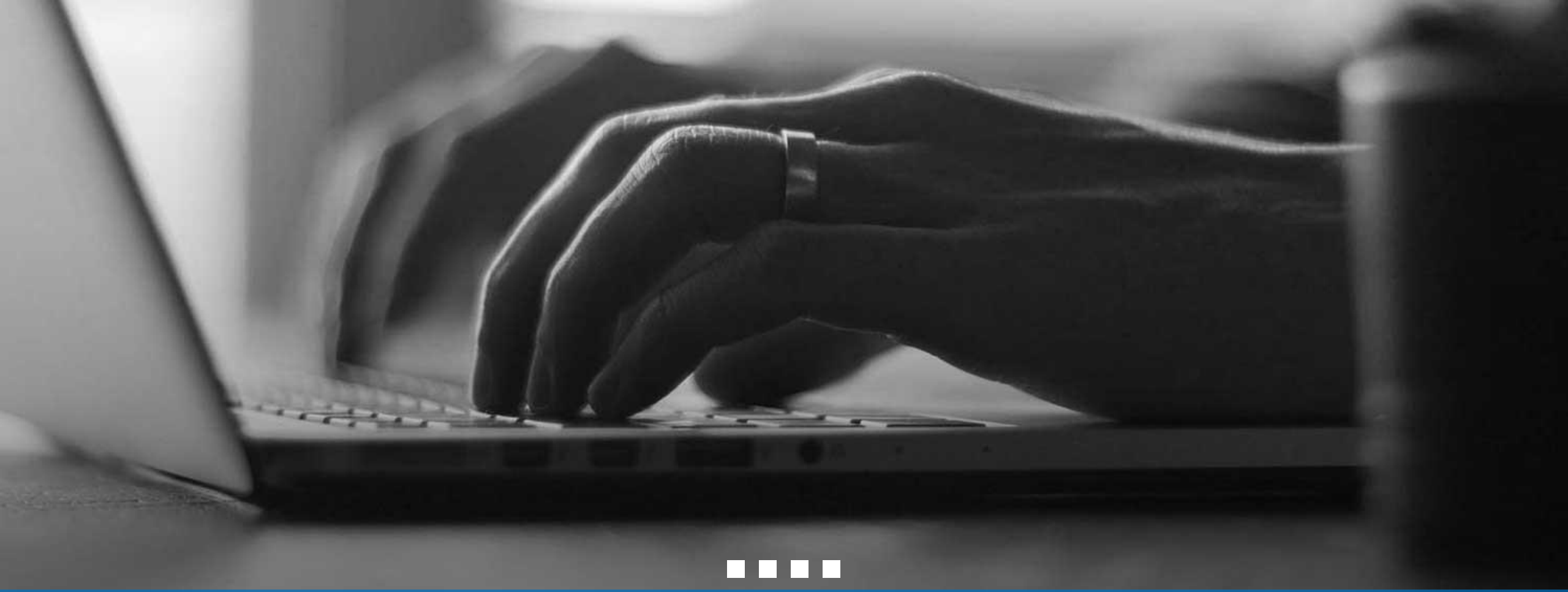
Nuestra GitHub Pages x repositorio



Hello, world!



Nuestras Páginas de Aterrizaje están Listas !!!!



MUCHAS GRACIAS

{Hasta la Próxima}

Unidad 2

Code Kata & Code Dojo
Una Práctica Habitual en AyED II

2.5

¿Qué es un Coding Dojo?

CODING DOJO

Kata
Code

Un **Coding Dojo** se trata de una **reunión de programadores para trabajar en un "reto" de programación** durante unas horas, una tarde o incluso un día entero. Además los programadores que codifiquen en ese momento están rodeados por una audiencia (que participará o no posteriormente) observa el proceso de codificación, si es posible técnicamente a través de un proyector. En definitiva, una forma divertida de aprender técnicas nuevas con programadores que posiblemente en otras circunstancias no tengamos la oportunidad de trabajar.

https://inlab.fib.upc.edu/es/blog/te-entrenas-para-ser-mejor-programador-descubre-los-coding-dojo

Principios básicos del Coding Dojo

- Fundamentalmente **entender que adquirir nuevas habilidades de programación es un proceso continuo** en la que como programadores debemos estar abiertos siempre a aprender nuevas técnicas y tecnologías.

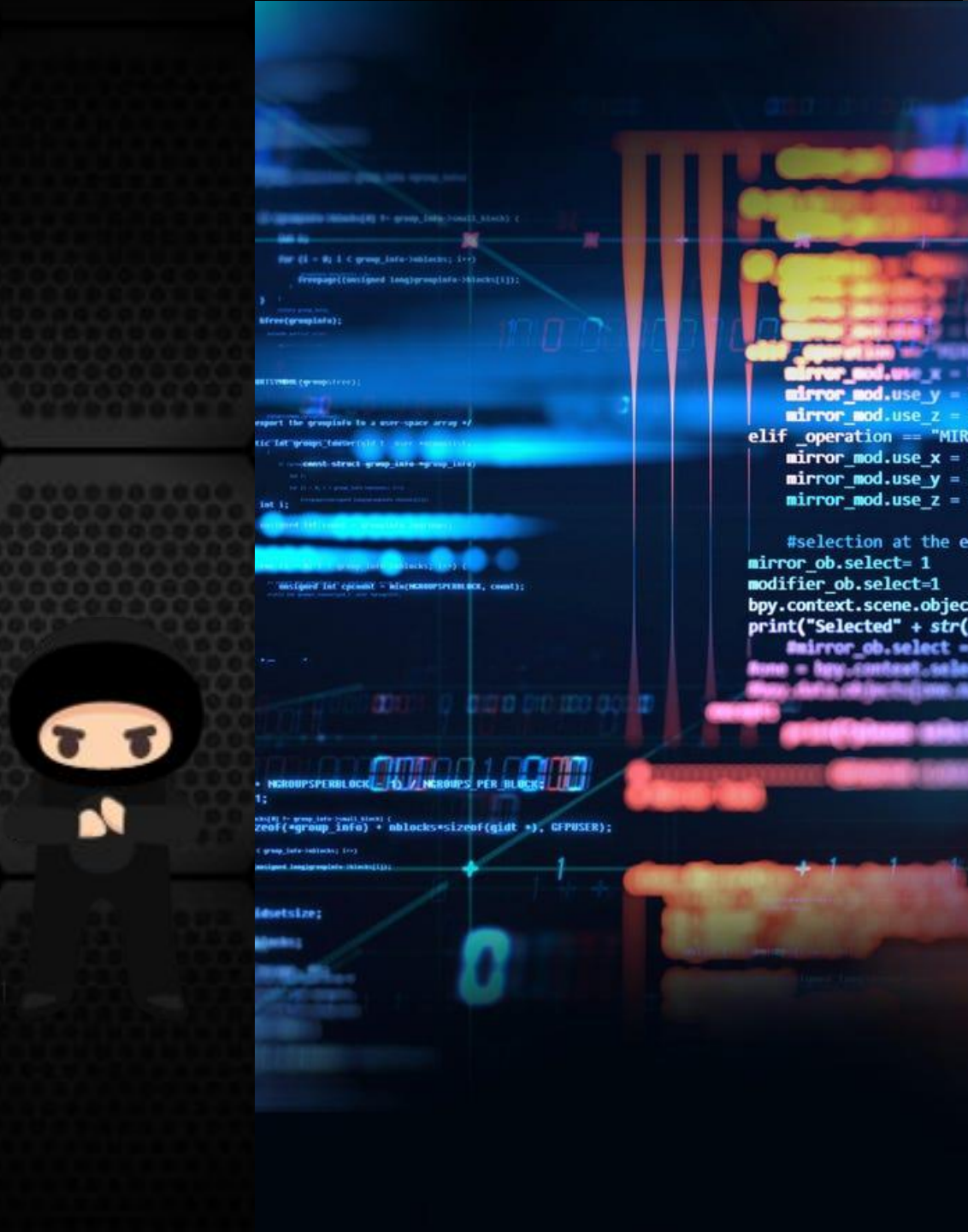
- El entorno de **Coding Dojo es colaborativo y no competitivo** en el que todos los programadores aportan indiferentemente del nivel que tengan.

- **Deber ser divertido** ya que estamos para pasar un buen rato a parte de aprender. Se puede tener a alguien que actúe como speaker para que, de forma distendida, vaya narrando el proceso. Si consigue que ver a un compañero codificando en eclipse sea divertido, mucho mejor.

- Es importante **estar abierto a nuevas ideas** y nuevas formas de abordar un problema.

Modalidades de Coding Dojo

- Codekata
- Randori



Modalidades de Coding Dojo

Codekata

En esta modalidad **un participante exhibe una solución a un desafío concreto** (por ejemplo, pasar de número latinos a romanos) la cual ha sido preparada con anterioridad, ensayada y refinada. El concepto de kata se extrae de las artes marciales en las que se ensaya una técnica constantemente hasta conseguir la perfección.

En el **Coding Dojo básicamente lo que se intenta es refinar nuestro código**, incluso ver diferentes soluciones a un mismo problema usando diferentes lenguajes. Se usa TDD para alcanzar la solución. Por supuesto, al final todos podemos participar proponiendo mejoras en el código de la Kata final para refinarla aún más. Todos aprendemos de todos.

```
#include<iostream>
using namespace std;

int main()
{
 cout<<"Hello World";
 return 0;
}
```

Modalidades de Coding Dojo

Git
Para trabajar en equipo utilizaremos como tecnología de repositorio Git, como metodología: branch per feature / pull requests

Randori

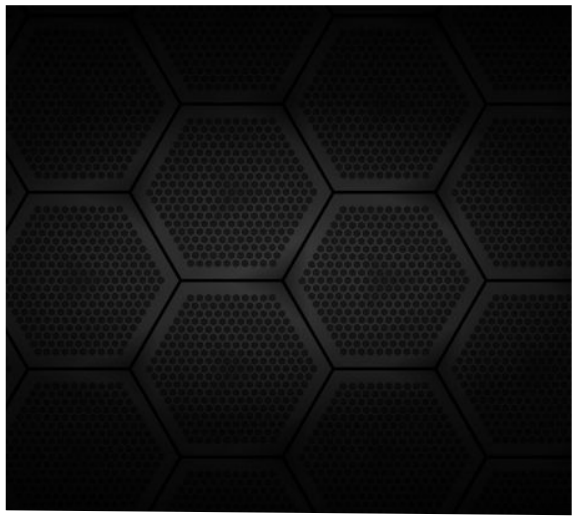
El reto se intenta solucionar por parejas, *pair programming*, en la que uno hace de piloto con el teclado y otro de copilo sobre un mismo ordenador resolviendo el problema. Se turnan según un *time box*, tiempo decidido que puede ser el tiempo de un pomodoro (25x5) como medida, rotando las parejas pasando el copiloto a piloto y entrando uno nuevo como copiloto proviniente de la audiencia.
Eventualmente la audiencia puede opinar y dar sugerencias de cómo resolver el problema, pero en ocasiones la pareja puede decidir no ser interrumpida. Ante todo siempre se tiene que ir explicando cada paso que se realice para que la audiencia pueda seguir el proceso y poder entrar en marcha sin problemas. Normalmente se elije un tiempo de interacción concreto para los turnos y se aprovecha el cambio de pareja para analizar la evolución del proceso y el feedback que va dando la audiencia a modo de retrospectiva.

Lo mejor de todo es animarse entre un grupo de compañeros, aprovechar los coding dojo que se suelen organizar en las universidad o si tienes la oportunidad en tu empresa de mezclar a varios desarrolladores de diferentes equipos para crear un proyecto express de un día o un par de horas. Practicar unas katas de programación o realizar un miniproyecto utilizando la técnica Randori para dinamizar la participación de todo el equipo en el proyecto. Incluso se pueden formar varios equipos que trabajen en paralelo en diferentes soluciones.

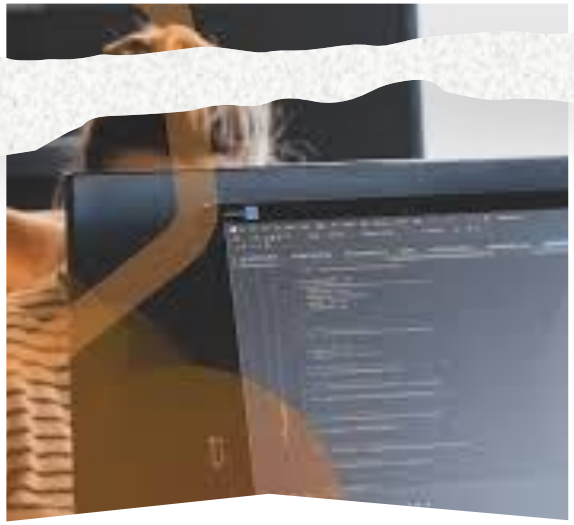
Kata Code

Un noche de enero del 2007 Dave Thomas, uno de los autores del magnífico libro **The Pragmatic Programmers**, estaba haciendo un programa en Ruby para implementar un pequeño contador de ciertas palabras que aparecían en los artículos de su blog. Para comprobar el rendimiento creo un test y se puso a **probar diferentes técnicas de búsqueda y comparación**, observando las enormes diferencias de rendimiento entre distintas formas de solucionar el problema.
Supongo que la mayoría de nosotros nos hubiéramos quedado bastante contentos con haber optimizado el código y que funcionara correctamente. Pero no así Dave, el cual escribió esta magnífica anotación en su blog: **A menudo, el verdadero valor de algo no es el que tiene en sí mismo, sino que lo es la actividad que lo creó.**

Lo que hace una buena ducha
Otro de los Pragmatic Programmers, Chad Fowler, cogio esta frase y escribió un post utilizando a los músicos como ejemplo de **la importancia de la práctica de la ejecución por encima del resultado final**. Finalmente Brian Marick, uno de los firmantes del Manifiesto Agile, se basó en el artículo de Flower para escribir una entrada en su blog en donde hacía énfasis sobre **el valor de la práctica en el aprendizaje de procesos creativos**, como es el desarrollo de software.
Dave, como nos cuenta en su blog, estando en la ducha se da cuenta que su práctica de 45 minutos la noche anterior era muy similar al ensayo de un músico. Se pregunta qué es lo que había hecho que la ha convertido en una sesión de práctica, y las conclusiones a las que llega las comparte con el resto de la comunidad.

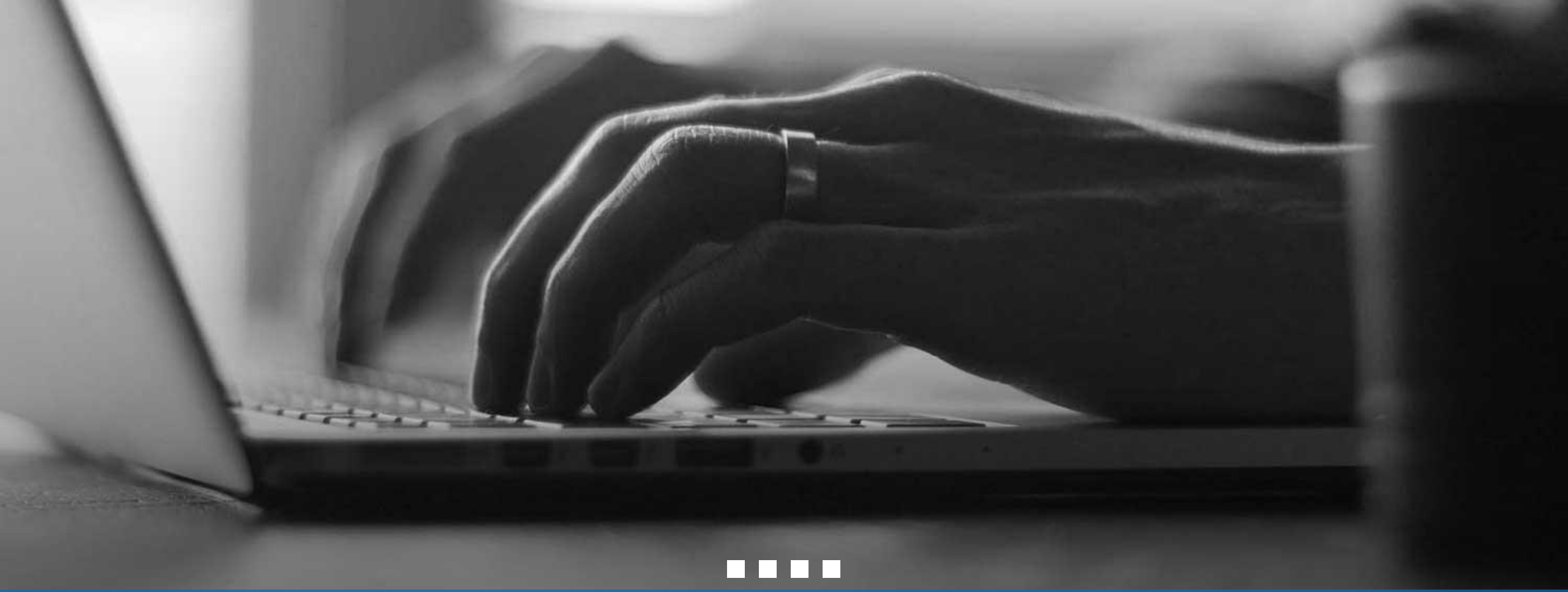



result == 'Fizz'
def test_five_should_return_buzz(fizzBuz
result = fizzBuzz.filter(5)
assert result == 'Buzz'
def test_fifteen_should_return_fizzbuzz(
result = fizzBuzz.filter(15)
assert result == 'FizzBuzz'



La principal ventaja de esta manera de prácticar es que nos acostumbra a atacar los problemas de múltiples formas, sin prisas, sin estar pendiente de los errores o de romper nuestro trabajo o el trabajo de algún compañero. Sin pensar en documentación, análisis o metodología.
Disfrutar del reto intelectual, como si fuera un crucigrama o una buena lectura. También el tener una lista interminable de Katas publicadas en la Web, nos permite enfrentarnos a retos a los cuales no estamos habituados en nuestro trabajo diario o, simplemente, no se nos hubiera ocurrido.
La llegada de TDD le ha dado una vuelta de tuerca a las Code-Katas ya que le sumamos el ciclo red-green-refactor y utilizamos la práctica para hacernos a esta nueva técnica de desarrollo que es tan extraña para quien no está acostumbrada a ella.
La única precaución que se debe tener en cuenta es que estamos haciendo prácticas con ejemplos ideales y dirigidos. Es decir, muy raramente nos vamos a encontrar
problemas tan simples y sencillos en el mundo real. Y, como dice el dicho, la realidad siempre supera la ficción. También hay que no caer en la tentación de igualar Code-Kata con TDD.
Se puede hacer una Code-Kata sin tan siquiera test unitarios, siempre que el feedback sea el suficiente.

Practicar, practicar, practicar.



MUCHAS GRACIAS

{Hasta la Próxima}

Muchas Gracias

Nos veremos en la Proxima.

ISFT n151 – Algoritmos y Estructuras de Datos