Presentación

Nombre y matrícula del estudiante

Dashel Humalba Eustaquio Muñoz (2021-1281)

Periodo académico

Cuatrimestral

Fecha de entrega

1 de abril de 2023

Nombre del profesor

Kelyn Tejafa Belliard

Nombre del tema de estudio

Programación lll

Git

Es un proyecto de código abierto maduro y con un mantenimiento activo que desarrolló originalmente **Linus Torvalds**, el famoso creador del kernel del sistema operativo Linux, en 2005. Un asombroso número de proyectos de software dependen de Git para el control de versiones, incluidos proyectos comerciales y de código abierto.

Los desarrolladores que han trabajado con Git cuentan con una buena representación en la base de talentos disponibles para el desarrollo de software, y este sistema funciona a la perfección en una amplia variedad de sistemas operativos e IDE (entornos de desarrollo integrados).

En lugar de tener un único espacio para todo el historial de versiones del software, como sucede de manera habitual en los sistemas de control de versiones antaño populares, como CVS o Subversion (también conocido como SVN), en Git, la copia de trabajo del código de cada desarrollador es también un repositorio que puede albergar el historial completo de todos los cambios.

Controlar las versiones del código

Obviamente nos sirve para el control de versiones de los proyectos y por ello permite saber todos los estados por los que han pasado cada uno de los archivos de código de un proyecto. Permite saber cuándo se modificó un archivo y qué cambios se realizaron, a lo largo de toda la existencia de ese archivo.

Trabajo en equipo

Git es la herramienta fundamental para trabajo en equipo, ya que permite que cada desarrollador trabaje con su propia copia local del proyecto y, cuando envíe cambios al repositorio global, se asegure que su código no machaca modificaciones introducidas por otros desarrolladores.

Además, gracias a Git cualquier desarrollador puede sincronizar el código del proyecto local con los cambios introducidos por otros desarrolladores, lo que facilita mucho el día a día del trabajo de todos los desarrolladores y la gestión de proyectos.

Revisar el código del proyecto en equipo

Otra de las ventajas de Git para equipos de trabajo es que podemos revisar cualquier actualización del código del proyecto entre varias personas. Un desarrollador envía un pull request y otro u otros desarrolladores pueden verificar que los cambios tienen buena pinta, antes de incorporarlos al proyecto.

Revisar la calidad del código y pasar las pruebas

Con Git podemos automatizar diversos procesos de comprobación de la calidad del código, como los que nos permiten los linters o las pruebas. Estas comprobaciones se pueden realizar con cualquier intento de enviar código al repositorio. Si no pasa la revisión del Linter o las pruebas, simplemente no se permitirá actualizar el código del proyecto.

Despliegue de aplicaciones web

Con Git puedes subir un sitio web al servidor en cuestión de instantes, sin usar el engorroso FTP. Además, cuando se actualiza el código del proyecto puedes usar Git para subir únicamente aquellos archivos que se han modificado, con un sencillo comando de consola.

Integración continua y despliegue continuo

Git además de todo permite organizar flujos de trabajo para la automatización de tareas repetitivas, como desplegar el proyecto en entornos de pruebas, desplegar en preproducción o producción.

Estos flujos de trabajo pueden ser tan complejos como se requiera según el proyecto. Todo empezará eventos realizados por el propio Git, cuando se envía por ejemplo código nuevo a una rama determinada. Entonces se podrán realizar todo tipo de acciones, como correr las pruebas, levantar servidores para pruebas de integración, enviar los cambios a servidores, ya sea para producción o preproducción, etc.

En todo sentido, Git sirve mucho más que solamente controlar las versiones de un software y resulta una herramienta fundamental para todos los procesos de automatización que se pueden realizar en todo tipo de proyectos de software.

Función del comando Git init

El comando **Git init** crea un nuevo repositorio de git. Se utiliza para convertir un proyecto existente y sin versión en un repositorio de Git o para ya inicializar un nuevo repositorio vacío; Este suele ser el primer comando ejecutado a la hora de comenzar.

el comando *git init* permite crear de manera increíblemente sencilla nuevos proyectos con control de versiones. Con Git, no es necesario que crear un repositorio, importar los archivos ni extraerlos en una copia de trabajo. Además, Git

no precisa de la existencia previa de ningún servidor o privilegios de administrador. Basta con que utilizar el comando cd en el subdirectorio del proyecto y ejecutar git init para que tener el repositorio de Git totalmente funcional.

Las ramas en Git

Una rama representa una línea independiente de desarrollo. Las ramas sirven como una abstracción de los procesos de cambio, preparación y confirmación. Es como una forma de solicitar un nuevo directorio de trabajo, un nuevo entorno de ensayo o un nuevo historial de proyecto. Las nuevas confirmaciones se registran en el historial de la rama actual, lo que crea una bifurcación en el historial del proyecto.

El comando **git branch** es aquel que permite crear, enumerar y eliminar ramas, así como cambiar su nombre. No te permite cambiar entre ramas o volver a unir un historial bifurcado.

Opciones comunes

```
git branch
```

Enumera todas las ramas del repositorio.

```
git branch <branch>
```

*Crear una nueva rama llamada **<branch>**.*
Esto no extrae la nueva rama.

```
git branch -d <branch>
```

Elimina la rama especificada. Esta es una operación segura, ya que Git evita que elimines la rama si tiene cambios que aún no se han fusionado.

```
git branch -D <branch>
```

Fuerza la eliminación de la rama especificada, incluso si tiene cambios sin fusionar.

```
git branch -m <branch>
```

Cambia el nombre de la rama actual a **<branch>**

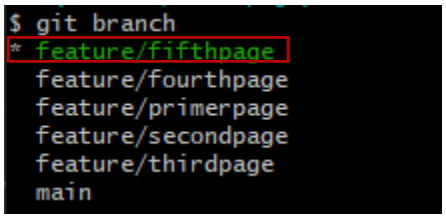
```
git branch -a
```

Enumera todas las ramas

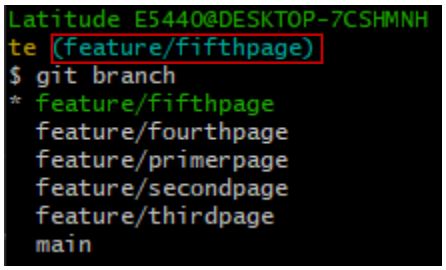
¿Cómo saber la rama en que estoy?

Hay dos maneras rápidas de saber la rama que se está utilizando:

1. Escribiendo el comando *git branch*: al ejecutarlo nos muestra una lista de todas las ramas creadas en el repositorio, el nombre de otro color seria la rama que uno está localizado.



2. Al lado de la dirección del repositorio: Es fácil de identificar y es mirando el nombre que tiene al lado la dirección que establece el repositorio en paréntesis.



Comandos más esenciales de Git

git init: este comando se utiliza para inicializar o crear un repositorio local vacío de Git.

git branch: con este comando se puede preguntar la posición del repositorio en donde nos localizamos; si estás en una rama específica, te saldrá la rama.

git branch -m main: con este comando se escoge la rama en donde te quieres localizar o hacer cambios.

git add: este comando se utiliza para agregar carpetas o documentos en el repositorio local.

git archive: este comando se utiliza para crear un archivo de documentos en un «árbol» específico.

git commit -m «primer commit»: este comando se utiliza para agregar un elemento al editor de texto de tu repositorio.

git log: con este comando puedes revisar el historial de commits de un programa.

git push -u origin main: este comando es muy importante, ya que es la forma de enviar las versiones que vas realizando en el proceso al repositorio remoto.

git clone: este comando se utiliza para clonar un repositorio en un nuevo directorio.

git switch: este comando se utiliza para saltar entre ramas de un repositorio.

git diff: se utiliza para saber los cambios que se han ejecutado entre commits.

Git Flow

Es un flujo de trabajo basado en Git que brinda un mayor control y organización en el proceso de integración continua.

Se implementa por varias razones:

- Aumenta la velocidad de entrega de código terminado al equipo de pruebas.
- Disminuyen los errores humanos en la mezcla de las ramas.
- Elimina la dependencia de funcionalidades al momento de entregar código para ser puesto en producción.

Es recomendado cuando:

- El equipo de trabajo está conformado por más de dos (2) personas.
- Se emplean metodologías ágiles.

- El proyecto tiene cambios frecuentes y se requiere actualizar el ambiente de producción garantizando continuidad en la operación.
- El proyecto tiene un nivel de complejidad considerable.
- Se desea tener un proceso de soporte a errores efectivo con actualizaciones rápidas.

En git Flow se organizan en dos ramas principales:

Rama master: cualquier commit que pongamos en esta rama debe estar preparado para *subir a producción.*

Rama develop**:** rama en la que está el código que conformará la siguiente versión planificada del proyecto.

Además de estas dos ramas, se proponen ramas auxiliares:

Feature or topic branches: Estas ramas se utilizan para desarrollar nuevas características de la aplicación que, una vez terminadas, se incorporan a la rama develop.

Release branches: Estas ramas se utilizan para preparar el siguiente código en producción. En estas ramas se hacen los últimos ajustes y se corrigen los últimos bugs antes de pasar el código a producción incorporándolo a la rama master.

Hotfix branches: Esas ramas se utilizan para corregir errores y bugs en el código en producción. Funcionan de forma parecida a las Releases Branches, siendo la principal diferencia que los hotfixes no se planifican.

En git flow cada vez que queramos hacer algo en el código, se tendrá que crear la rama que corresponda, trabajar en el código, incorporar el código donde corresponda y cerrar la rama. A lo largo de nuestra jornada de trabajo necesitaremos ejecutar varias veces al día los comandos git, merge, push y pull así como hacer checkouts de diferentes ramas, borrarlas, etc. Git-flow son un conjunto de extensiones que nos ahorran bastante trabajo a la hora de ejecutar todos estos comandos, simplificando la gestión de las ramas de nuestro repositorio.

Trunk Based Development

Es una estrategia de Git donde existe un trunk (un branch principal, usualmente llamado master/main) en el cual todo el equipo colabora he integra directamente (hace push), siguiendo estas consideraciones:

1.No haber branches de larga duración. No se da el mantenimiento a ningún Branch.

Ejemplo: En el caso de crear un Branch reléase y este se encuentra un error, debe ser reparado en el trunck y luego hacer un nuevo Branch de reléase, pero nunca se le da mantenimiento a ese Branch.

2. Hacer commits mínimo una vez al día. Lo que se busca con esto es eliminar la distancia entre los desarrolladores cuando se empieza a codear algo nuevo.

3.Todo commit creado es código funcional, implicando que se cumpla la definición de hecho (definition of done), se haya creado las pruebas necesarias y todo lo que sea requerido para asegurar que el código no introduzca un bug. Esto significa que el equipo debe de tener un grado de madurez y responsabilidad alto al momento de entregar el código.

4. El estado del trunck siempre debe estar verde y optimo, es decir, que este listo para hacerle reléase.

Beneficios:

- Release por demanda, dado que el codigo en el trunk siempre esta listo y en buen estado, deberiamos poder hacer release en cualquier momento.
- Nos evitamos esos grandes conflictos y esos "merges" extra, ej. Cuando se le da mantenimiento al branch de release y esos cambios tienen que ser movidos a develop, es código que ya paso por un proceso de aprobación, volver a tener que hacer code review no tiene sentido, esto representa un gasto de tiempo. Trunk-based development resuelve este problema.

- Los desarrolladores siempre van poder trabajar con el código más reciente.
- El equipo se vuelve más eficiente y más ágil para entregar código.