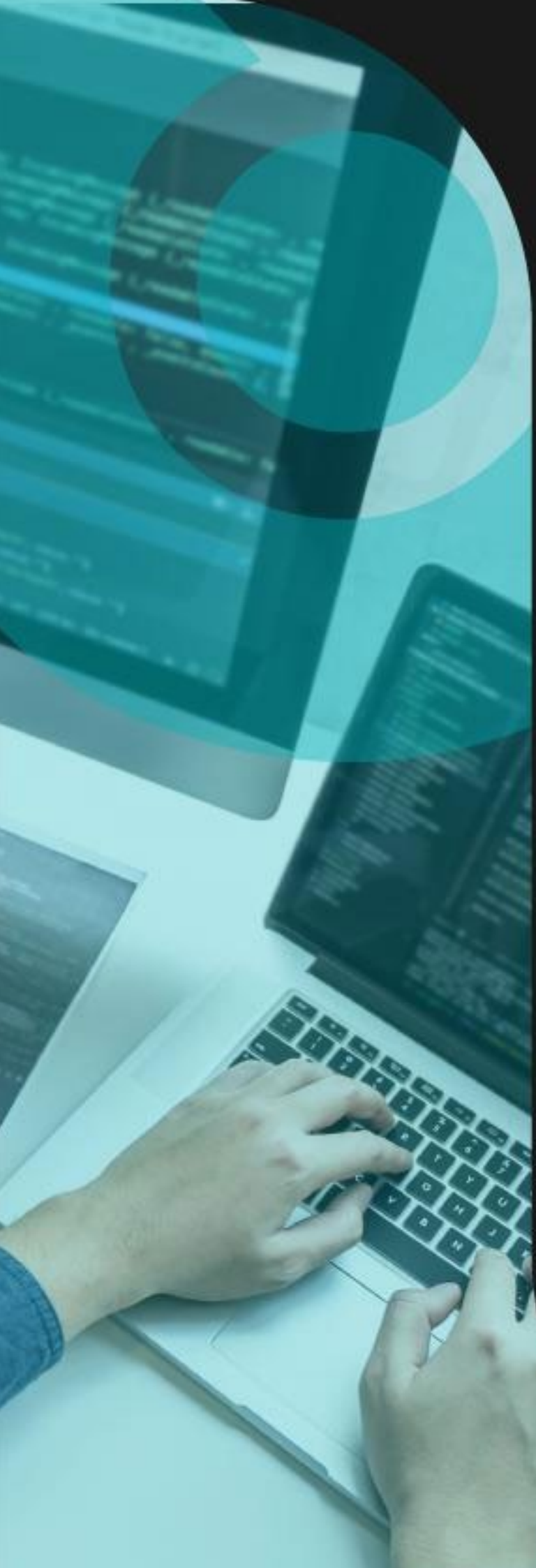

AWAKELAB

BASECAMP

Ciencia de Datos

Módulo: Fundamentos de Programación en Python

Aprendizaje Esperado

- 6. Gestionar el código fuente utilizando GitHub para mantener un repositorio de código remoto seguro y permitir trabajo concurrente.

Una breve historia de Git

Git fue creado por Linus Torvalds en 2005 como una alternativa de código abierto al sistema de control de versiones BitKeeper, que se utilizaba en el desarrollo del kernel de Linux. Cuando BitKeeper dejó de ser gratuito, Torvalds decidió crear su propio sistema de control de versiones para Linux y Git fue el resultado.

Desde entonces, Git se ha convertido en uno de los sistemas de control de versiones más populares del mundo y se utiliza en una amplia variedad de proyectos, desde pequeñas aplicaciones hasta grandes proyectos de software de código abierto y empresas líderes.

Git es distribuido, lo que significa que cada desarrollador tiene una copia completa del repositorio en su propia máquina. Esto significa que los desarrolladores pueden trabajar de forma independiente y realizar cambios en el código fuente sin necesidad de una conexión constante a un servidor central.

Git utiliza un modelo de ramas, lo que permite a los desarrolladores trabajar en diferentes características de un proyecto de forma aislada y luego fusionar sus cambios en una sola rama principal. Esto permite a los desarrolladores trabajar en paralelo sin interferir en el trabajo de los demás.

En resumen, Git es una herramienta fundamental para cualquier desarrollador que quiera trabajar de forma eficiente y colaborativa en un

proyecto de software o cualquier otro proyecto que requiera control de versiones.

Por qué git es una necesidad

Un repositorio de código es una necesidad para cualquier proyecto de software, ya sea pequeño o grande. Aquí hay algunas razones por las que un repositorio de código es esencial:

Control de versiones: Un repositorio de código te permite mantener un registro de todas las versiones del código fuente que has creado. Esto te permite rastrear cambios en el código y revertir a versiones anteriores si algo sale mal. También te permite comparar diferentes versiones del código para ver cómo ha cambiado a lo largo del tiempo.

Colaboración: Si trabajas en un proyecto con otros desarrolladores, un repositorio de código te permite colaborar de forma efectiva. Puedes trabajar en diferentes ramas del código fuente de forma aislada y luego fusionar tus cambios en una sola rama principal. Esto te permite trabajar de forma paralela sin interferir en el trabajo de los demás. Además, un repositorio de código te permite hacer comentarios y revisiones de código para ayudar a mejorar la calidad del proyecto.

Historial de cambios: Un repositorio de código te permite ver quién hizo qué cambio en el código fuente y cuándo lo hizo. Esto te permite tener una comprensión clara de cómo ha evolucionado el proyecto a lo largo del tiempo. También te permite responsabilizar a los desarrolladores por los cambios que han realizado y solucionar problemas si algo sale mal.

Backups: Un repositorio de código también te permite tener copias de seguridad de tu código fuente en caso de que algo salga mal. Si tienes un accidente con tu computadora o pierdes tus archivos, puedes simplemente clonar el repositorio de código en una nueva computadora y continuar trabajando.

En resumen, un repositorio de código es una necesidad para cualquier proyecto de software, ya que te permite mantener un registro de las versiones del código, colaborar de forma efectiva, tener un historial de cambios y hacer copias de seguridad de tu código fuente.

Instalación de git

Aquí hay una breve guía para la instalación y configuración de Git:

Descarga Git: Lo primero que debes hacer es descargar Git desde el sitio web oficial de Git: https://git-scm.com/downloads. Asegúrate de descargar la versión correcta para tu sistema operativo.

Instala Git: Una vez que hayas descargado Git, debes seguir las instrucciones de instalación en pantalla para instalar Git en tu sistema.

Configura tu nombre de usuario y correo electrónico: Después de instalar Git, debes configurar tu nombre de usuario y correo electrónico en Git. Puedes hacer esto utilizando los siguientes comandos en la línea de comandos de Git:

```
git config --global user.name "Tu nombre de usuario"
git config --global user.email "tu-correo-electronico@example.com"
```

Configura la línea de comando: Si deseas utilizar Git desde la línea de comandos, puedes configurar la línea de comando de Git utilizando el siguiente comando:

```
git config --global core.editor "tu-editor-preferido"
```

Verifica la instalación: Para verificar que Git se ha instalado correctamente y está configurado correctamente, puedes ejecutar el siguiente comando en la línea de comandos de Git:

```
git --version
```

Este comando debería mostrar la versión de Git que has instalado.

Con estos pasos, deberías estar listo para empezar a utilizar Git en tu proyecto. Recuerda que Git es una herramienta muy poderosa, y es importante que te tomes el tiempo de aprender cómo utilizarla correctamente antes de comenzar a trabajar en un proyecto real.

Comandos básicos de git

Aquí hay una lista de algunos de los comandos básicos de Git que te ayudarán a comenzar:

git init: Este comando inicializa un nuevo repositorio de Git en tu proyecto. Debes ejecutar este comando en la carpeta raíz de tu proyecto.

git add: Este comando agrega los archivos modificados o nuevos a la "zona de preparación" (o "área de stage") para que puedan ser incluidos en el siguiente commit. Por ejemplo, para agregar todos los archivos modificados o nuevos, puedes ejecutar el siguiente comando:

```
git add.
```

git commit: Este comando crea un nuevo commit en el repositorio de Git que contiene los archivos agregados a la zona de preparación. Debes proporcionar un mensaje de commit para describir los cambios que se están realizando. Por ejemplo, para crear un nuevo commit con un mensaje, puedes ejecutar el siguiente comando:

```
git commit -m "Mensaje de commit"
```

git status: Este comando muestra el estado actual del repositorio de Git, incluyendo los archivos que han sido modificados, agregados o eliminados.

git log: Este comando muestra el historial de commits del repositorio de Git, incluyendo quién realizó el commit, cuándo se realizó y el mensaje de commit.

git pull: Este comando descarga los cambios más recientes del repositorio remoto y los fusiona con tu rama local.

git push: Este comando envía tus cambios locales al repositorio remoto. Por ejemplo, para enviar los cambios de tu rama local "main" al repositorio remoto "origin", puedes ejecutar el siguiente comando:

```
git push origin main
```

git branch: Este comando muestra las ramas del repositorio de Git. También puedes crear nuevas ramas con este comando.

git checkout: Este comando te permite cambiar entre ramas del repositorio de Git. Por ejemplo, para cambiar a la rama "develop", puedes ejecutar el siguiente comando:

```
git checkout develop
```

Estos son solo algunos de los comandos básicos de Git. Hay muchos otros comandos más avanzados que puedes utilizar a medida que te familiarices más con Git.

Commits y restauración de archivos

Aquí te explico sobre commits y restauración de archivos:

Commits:

Un commit en Git es un registro de los cambios que has realizado en tu código. Cada commit tiene un mensaje asociado que describe los cambios que se realizaron. Al crear un commit, se guarda una instantánea de los cambios realizados en ese momento en el código.

Para crear un commit, primero necesitas agregar los archivos que deseas incluir en el commit utilizando el comando git add. Luego, puedes crear el commit utilizando el comando git commit. Por ejemplo:

```
git add archivo1 archivo2

git commit -m "Descripción del commit"
```

Restauración de archivos:

Si has cometido un error y necesitas restaurar un archivo a una versión anterior, Git puede ayudarte a hacerlo. Puedes hacer esto utilizando el comando git checkout. Este comando te permite cambiar a una versión anterior de un archivo o rama.

Para restaurar un archivo a una versión anterior, debes saber el hash del commit que contiene la versión que deseas restaurar. Puedes encontrar el hash del commit utilizando el comando git log. Luego, puedes utilizar el comando git checkout para restaurar el archivo a la versión anterior. Por ejemplo:

```
git log

git checkout <hash-del-commit> archivo1
```

Este comando cambiará el archivo "archivo1" a la versión almacenada en el commit especificado por <hash-del-commit>. Asegúrate de guardar el archivo restaurado antes de realizar más cambios.

Recuerda que Git es una herramienta muy poderosa, y es importante que te tomes el tiempo de aprender cómo utilizarla correctamente antes de comenzar a trabajar en un proyecto real. También es importante hacer commits con frecuencia para mantener un historial claro y detallado de los cambios en tu código.

Ignorar archivos

Es muy común tener archivos en un proyecto que no se deben incluir en el control de versiones de Git. Por ejemplo, archivos de configuración con información sensible como contraseñas, claves de API, entre otros. Para evitar que Git los agregue accidentalmente, podemos usar el archivo ".gitignore".

El archivo ".gitignore" es un archivo de texto simple que contiene una lista de patrones de archivos o directorios que Git debe ignorar. Para crear un archivo ".gitignore" para tu proyecto, puedes usar un editor de texto y agregar los patrones que deseas ignorar.

Aquí hay algunos ejemplos de patrones que puedes agregar a tu archivo ".gitignore":

Ignorar todos los archivos con una extensión específica:

```
*.log
```

Este patrón ignorará todos los archivos con la extensión ".log".

Ignorar un archivo específico:

```
archivo-secreto.txt
```

Este patrón ignorará el archivo "archivo-secreto.txt".

Ignorar un directorio completo:

```
mi-carpeta-secreta/
```

Este patrón ignorará todo el contenido de la carpeta "mi-carpeta-secreta/".

Ignorar archivos con un nombre específico en un directorio específico:

```
mi-carpeta-secreta/*.log
```

Este patrón ignorará todos los archivos con la extensión ".log" en la carpeta "mi-carpeta-secreta/".

Una vez que hayas creado tu archivo ".gitignore", debes agregarlo a tu repositorio de Git. Para hacer esto, simplemente ejecuta el siguiente comando:

```
git add .gitignore

git commit -m "Agregar archivo .gitignore"
```

De esta manera, los archivos que agregaste en el archivo ".gitignore" se ignorarán en Git y no se incluirán en tus commits.

Ramas, uniones, conflictos y tags en Git:

Ramas:

Las ramas en Git te permiten crear líneas de desarrollo separadas en tu proyecto, lo que te permite trabajar en diferentes características o solucionar diferentes problemas de manera independiente. Las ramas en Git son muy ligeras, lo que significa que cambiar de una rama a otra es muy rápido.

Para crear una nueva rama, puedes usar el comando git branch seguido del nombre de la nueva rama. Por ejemplo, para crear una nueva rama llamada "mi-rama", puedes ejecutar el siguiente comando:

```
git branch mi-rama
```

Una vez que hayas creado la nueva rama, puedes cambiar a ella utilizando el comando git checkout. Por ejemplo:

```
git checkout mi-rama
```

Uniones:

Una vez que hayas trabajado en una rama separada y estés listo para combinarla con la rama principal, puedes hacer una unión. Las uniones en Git pueden ser muy simples o pueden requerir resolución de conflictos.

Para unir una rama en Git, primero necesitas cambiar a la rama en la que deseas unir la rama actual. Luego, puedes utilizar el comando git merge seguido del nombre de la rama que deseas unir. Por ejemplo:

```
git checkout main

git merge mi-rama
```

Conflictos:

Si has realizado cambios en diferentes partes de un archivo en diferentes ramas, Git puede tener dificultades para combinar las ramas. Esto se conoce como un conflicto de unión. Cuando se produce un conflicto de unión, Git te notificará en qué archivos se ha producido el conflicto y debes resolver el conflicto manualmente.

Para resolver un conflicto, debes editar manualmente el archivo con el conflicto y decidir qué cambios se deben mantener. Una vez que hayas resuelto el conflicto, debes agregar los cambios al índice y luego crear un nuevo commit. Por ejemplo:

```
# Editar archivo para resolver el conflicto
```

```
git add archivo-modificado

git commit -m "Solucionar conflicto de unión"
```

Tags:

Los tags en Git te permiten crear un marcador en un punto específico en la historia de tu proyecto. Los tags son útiles para marcar versiones específicas de tu proyecto y para marcar hitos importantes.

Para crear un tag en Git, puedes utilizar el comando git tag seguido del nombre del tag y del hash del commit al que deseas hacer referencia. Por ejemplo:

```
git tag v1.0 <hash-del-commit>
```

También puedes crear un tag en Git en el commit actual utilizando el comando git tag con solo el nombre del tag. Por ejemplo:

```
git tag v1.0
```

Para ver una lista de todos los tags en tu proyecto, puedes utilizar el comando git tag. Para ver la información detallada de un tag en particular, puedes utilizar el comando git show seguido del nombre del tag. Por ejemplo:

```
git show v1.0
```

Recuerda que Git es una herramienta muy poderosa y es importante que te tomes el tiempo de aprender cómo utilizarla correctamente antes de comenzar a trabajar en un proyecto real. También es importante hacer commits con frecuencia para mantener un historial claro y detallado de los cambios en tu código.

Stash y rebase

Aquí te explico sobre stash y rebase en Git:

```
Stash:
```

A veces, mientras estás trabajando en una rama en particular, puede que necesites cambiar a otra rama para realizar una tarea urgente, pero no estás listo para hacer un commit con los cambios que has realizado en la rama actual. En estos casos, puedes usar el comando git stash para guardar los cambios sin realizar un commit y aplicarlos más tarde.

El comando git stash toma todos los cambios que aún no se han confirmado y los guarda en una pila temporal. Luego, puedes cambiar a otra rama y realizar otras tareas. Cuando estés listo para volver a la rama original, puedes aplicar los cambios que guardaste usando el comando git stash apply. Por ejemplo:

```
# Guardar los cambios sin realizar un commit

git stash

# Cambiar a otra rama

git checkout otra-rama

# Realizar otras tareas

# Volver a la rama original

git checkout rama-original

# Aplicar los cambios guardados

git stash apply
```

Rebase:

El comando rebase en Git te permite cambiar la historia de la rama actual. Con el rebase, puedes mover los commits de una rama a otra, eliminar commits, fusionar commits y editar los mensajes de commit.

El rebase se utiliza a menudo en lugar de la unión (merge) para mantener un historial de commits más limpio y fácil de entender. En lugar de crear un nuevo commit de unión, el rebase mueve los cambios a la rama de destino y se asegura de que se apliquen en el orden correcto.

Para realizar un rebase, primero debes cambiar a la rama que deseas reorganizar y luego ejecutar el comando git rebase seguido del nombre de la rama que contiene los cambios que deseas aplicar. Por ejemplo:

```
# Cambiar a la rama que deseas reorganizar

git checkout mi-rama

# Realizar el rebase

git rebase rama-destino
```

Durante el proceso de rebase, Git puede encontrar conflictos que deben resolverse manualmente. Una vez que se hayan resuelto los conflictos, debes agregar los cambios al índice y continuar con el rebase utilizando el comando git rebase --continue.

Es importante tener en cuenta que el rebase cambia la historia de la rama, por lo que debes tener cuidado al realizar un rebase en una rama que ya ha sido compartida con otros usuarios. Si no estás seguro de si debes usar el rebase o la unión, es mejor discutirlo con tu equipo de desarrollo.

Aquí te explico los fundamentos de GitHub:

¿Qué es GitHub?

GitHub es una plataforma de alojamiento de repositorios de código que permite a los desarrolladores trabajar juntos en proyectos de software. GitHub se basa en Git, un sistema de control de versiones distribuido, que permite a los desarrolladores trabajar juntos en proyectos de software de manera eficiente.

Además de alojar repositorios de código, GitHub ofrece una variedad de herramientas para colaborar en proyectos, incluyendo seguimiento de problemas (issue tracking), pull requests (solicitudes de cambios), wikis y herramientas de integración continua.

¿Por qué usar GitHub?

GitHub es una herramienta muy popular entre los desarrolladores porque ofrece varios beneficios, entre ellos:

Alojamiento gratuito de repositorios de código abierto.

Herramientas integradas para colaborar en proyectos de software.

Facilidad de uso y una interfaz intuitiva.

Integración con otras herramientas de desarrollo, como Travis CI, CircleCI y Heroku.

Creación de un repositorio en GitHub

Para crear un repositorio en GitHub, sigue estos pasos:

Inicia sesión en GitHub y haz clic en el botón "New repository" (nuevo repositorio) en la página principal.

Elige un nombre para tu repositorio y selecciona si deseas que sea público o privado.

Agrega una descripción y selecciona si deseas inicializar el repositorio con un archivo README.

Haz clic en el botón "Create repository" (crear repositorio) para crear tu nuevo repositorio.

Trabajando con repositorios en GitHub

Una vez que hayas creado un repositorio en GitHub, puedes trabajar en él usando la interfaz web o usando Git en tu máquina local. Puedes clonar el repositorio en tu máquina local, hacer cambios y luego enviar esos cambios de vuelta al repositorio en GitHub usando los comandos de Git, como git add, git commit y git push.

También puedes colaborar en proyectos de otros usuarios de GitHub haciendo fork (bifurcación) de su repositorio y luego creando una pull request (solicitud de cambios) para que puedan revisar tus cambios y fusionarlos con su repositorio.

Seguimiento de problemas y pull requests en GitHub

GitHub también ofrece herramientas para realizar un seguimiento de los problemas (bug tracking) y las solicitudes de cambios (pull requests) en tus proyectos de software. Puedes crear un nuevo problema o pull request en la página de tu repositorio y luego asignarlo a un miembro de tu equipo o etiquetarlo para un seguimiento más fácil.

Los pull requests permiten a los usuarios colaborar en proyectos de software al permitirles enviar cambios al repositorio original y solicitar que se fusionen con la rama principal. Una vez que se ha creado un pull request, los miembros del equipo pueden revisar los cambios y aprobarlos

o solicitar cambios adicionales antes de fusionarlos con el repositorio principal.

Repositorios remotos y push, pull

Los repositorios remotos son una parte fundamental de GitHub. Los repositorios remotos son versiones de tu proyecto alojados en la plataforma de GitHub en lugar de en tu máquina local. Esto permite que tú y otros colaboradores trabajen juntos en un proyecto, compartiendo los cambios y actualizaciones en tiempo real.

A continuación, te explicaré cómo trabajar con repositorios remotos en GitHub:

Clonar un repositorio remoto

El primer paso para trabajar con un repositorio remoto es clonarlo en tu máquina local. Para clonar un repositorio remoto, haz lo siguiente:

Copia la URL del repositorio remoto que deseas clonar.

Abre una terminal y navega al directorio donde deseas clonar el repositorio.

Ejecuta el comando git clone seguido de la URL del repositorio remoto:

```
git clone https://github.com/usuario/nombre-repositorio.git
```

Push y Pull

Una vez que has clonado un repositorio remoto, puedes hacer cambios en tu máquina local y enviarlos de vuelta al repositorio remoto en GitHub mediante el comando git push. Esto sincronizará tus cambios con el repositorio remoto para que otros colaboradores puedan verlos y trabajar con ellos.

Si otros colaboradores han hecho cambios en el repositorio remoto mientras estabas trabajando en tu máquina local, puedes sincronizar tu copia local con el repositorio remoto mediante el comando git pull. Esto descargará los cambios más recientes en tu máquina local y actualizará tu copia del proyecto.

Crear un nuevo repositorio remoto

Si deseas crear un nuevo repositorio remoto en GitHub, sigue estos pasos:

Inicia sesión en GitHub y haz clic en el botón "New repository" (nuevo repositorio) en la página principal.

Elige un nombre para tu repositorio y selecciona si deseas que sea público o privado.

Agrega una descripción y selecciona si deseas inicializar el repositorio con un archivo README.

Haz clic en el botón "Create repository" (crear repositorio) para crear tu nuevo repositorio.

Añadir un repositorio remoto existente

Si deseas agregar un repositorio remoto existente a tu proyecto, utiliza el comando git remote add. Esto agrega un nuevo repositorio remoto a tu proyecto y te permite hacer push y pull a ese repositorio.

Por ejemplo, si deseas agregar el repositorio remoto origin con la URL https://github.com/usuario/nombre-repositorio.git, puedes utilizar el siguiente comando:

```
git remote add origin https://github.com/usuario/nombre-repositorio.git
```

Estos son los conceptos básicos para trabajar con repositorios remotos en GitHub. Recuerda que Git y GitHub ofrecen una gran cantidad de herramientas y funcionalidades para colaborar en proyectos de software y rastrear cambios en tu proyecto.

Fetch y pull

Fetch y pull son dos comandos diferentes en Git que se utilizan para obtener actualizaciones de un repositorio remoto.

El comando fetch descarga los cambios desde el repositorio remoto pero no los integra automáticamente con tu copia local del repositorio. En cambio, los cambios descargados se almacenan en una rama separada llamada origin/<nombre-de-la-rama>.

```
git fetch origin
```

El comando pull realiza dos acciones en una sola operación. En primer lugar, descarga los cambios del repositorio remoto como lo hace fetch. En segundo lugar, integra automáticamente los cambios descargados en la rama actual en tu copia local del repositorio.

```
git pull origin <nombre-de-la-rama>
```

En resumen, fetch y pull descargan cambios desde el repositorio remoto. La diferencia es que fetch solo descarga los cambios y los almacena en una rama separada, mientras que pull descarga los cambios y los integra automáticamente en tu copia local del repositorio.

Es importante tener en cuenta que si se tiene una rama local y una rama remota con el mismo nombre, Git intentará fusionarlas automáticamente al usar pull, lo que puede generar conflictos. En cambio, fetch te permite examinar los cambios descargados antes de integrarlos manualmente usando el comando merge. Por lo tanto, es importante entender la diferencia entre estos dos comandos y usar el que sea más apropiado para tu flujo de trabajo.ch vs pull

Clonando un repositorio

Clonar un repositorio es el proceso de crear una copia local de un repositorio remoto en tu computadora. Esto te permite trabajar con los archivos del repositorio en tu propia máquina y hacer cambios sin afectar directamente al repositorio remoto.

Para clonar un repositorio en Git, sigue estos pasos:

Abre la terminal en tu computadora y navega hasta el directorio donde deseas guardar la copia del repositorio.

Ejecuta el siguiente comando para clonar el repositorio:

```
git clone URL-del-repositorio
```

Reemplaza URL-del-repositorio con la dirección del repositorio que deseas clonar. Puedes encontrar la dirección en el sitio web de GitHub, en la página del repositorio, en el botón verde "Code".

Presiona "Enter" para ejecutar el comando. Git comenzará a descargar el repositorio remoto y crear una copia local en tu computadora. Esto puede llevar unos segundos o varios minutos, dependiendo del tamaño del repositorio y la velocidad de tu conexión a Internet.

Una vez que la descarga haya finalizado, deberías ver una carpeta con el nombre del repositorio en tu directorio de trabajo. Ahora puedes trabajar en los archivos del repositorio y hacer cambios locales sin afectar al repositorio remoto.

Markdown

Markdown es un lenguaje de marcado ligero que se utiliza para formatear texto de manera simple y rápida. Fue creado por John Gruber en 2004, con el objetivo de proporcionar una sintaxis fácil de leer y escribir para crear documentos en la web.

El lenguaje Markdown es fácil de aprender y se utiliza comúnmente en la documentación de proyectos, foros, blogs, wikis y otros lugares donde se necesita formatear el texto. Se puede utilizar para crear títulos, listas, enlaces, imágenes, código, y otros elementos de formato comunes.

La sintaxis de Markdown es muy intuitiva y consiste en escribir texto plano utilizando ciertos caracteres especiales. Por ejemplo, para crear un encabezado, simplemente escribe # seguido del texto del encabezado:

```
# Este es un encabezado de nivel 1

## Este es un encabezado de nivel 2

### Este es un encabezado de nivel 3
```

Markdown es ampliamente compatible y se puede convertir a HTML u otros formatos de texto en línea o mediante herramientas de conversión. Muchas plataformas de blogging y sitios web populares, como GitHub, Stack Overflow y Reddit, admiten Markdown como formato de entrada para crear publicaciones y comentarios de manera rápida y sencilla.

Administrando pull request

Un pull request es una solicitud de cambios en un proyecto alojado en una plataforma de control de versiones como GitHub. Cuando alguien desea contribuir con un proyecto, crea un pull request que propone cambios específicos en el código fuente del proyecto. El dueño del proyecto puede revisar el pull request y decidir si desea fusionar los cambios en el proyecto principal.

Aquí hay algunos pasos clave para administrar pull requests en GitHub:

Revisión del pull request: Una vez que alguien crea un pull request, el dueño del proyecto o el mantenedor puede revisarlo para ver si los cambios propuestos son útiles y seguros para el proyecto. Es importante verificar que los cambios cumplan con los estándares de codificación y que no introduzcan errores en el código.

Comentarios y discusión: Si el dueño del proyecto tiene preguntas o comentarios sobre el pull request, puede dejar comentarios en la página de la solicitud para discutir los cambios propuestos. Esta discusión es útil para asegurarse de que todas las partes involucradas comprendan los cambios y las implicaciones.

Pruebas de integración: Antes de fusionar los cambios, el dueño del proyecto puede realizar pruebas de integración para asegurarse de que los cambios propuestos no rompan otras partes del proyecto. Por ejemplo, se pueden ejecutar pruebas automatizadas para verificar que el código nuevo no provoque errores o interrupciones en el proyecto.

Fusionar o cerrar la solicitud: Si el dueño del proyecto está satisfecho con los cambios propuestos, puede fusionar el pull request para incorporar los cambios en el proyecto principal. Si el pull request no se considera útil o no cumple con los estándares del proyecto, el dueño del proyecto puede cerrar la solicitud.

En resumen, la administración de pull requests en GitHub es un proceso importante para garantizar que los cambios propuestos en un proyecto sean seguros y útiles. La revisión cuidadosa, la discusión, las pruebas de integración y la fusión selectiva son todas partes importantes de este proceso.

Flujos de trabajo con github

GitHub es una plataforma de alojamiento de código y control de versiones basada en Git, que permite a los desarrolladores colaborar en proyectos de software de manera eficiente. A continuación, se describen algunos flujos de trabajo comunes que se utilizan en GitHub:

Fork y pull request: Este es un flujo de trabajo común para proyectos de código abierto. Los contribuyentes comienzan haciendo un fork del proyecto original, lo que crea una copia del proyecto en su cuenta de GitHub. Luego hacen los cambios en su propio fork y crean un pull request para solicitar que los cambios se fusionen en el proyecto original.

Branch y pull request: Este flujo de trabajo implica que los desarrolladores creen una rama de trabajo separada para realizar cambios en el código. Los cambios se realizan en la rama, y luego se crea un pull request para solicitar que los cambios se fusionen en la rama principal del proyecto.

Gitflow: Este flujo de trabajo es una metodología de ramificación que utiliza varias ramas de Git para desarrollar nuevas funciones y realizar mantenimiento en un proyecto. El flujo de trabajo Gitflow implica la creación de dos ramas principales: una rama "develop" y una rama "master". Las nuevas características se desarrollan en ramas separadas y se fusionan en la rama "develop". Cuando se realiza un lanzamiento, la rama "develop" se fusiona en la rama "master".

Continuous Integration (CI): La integración continua es una práctica que implica la integración frecuente de cambios de código en un proyecto. Los desarrolladores crean ramas separadas y crean pull requests para solicitar la fusión de cambios. Los sistemas de integración continua se utilizan para compilar, probar y validar los cambios antes de fusionarlos en el proyecto principal. GitHub Actions es una de las herramientas populares para la integración continua en GitHub.

En resumen, hay varios flujos de trabajo que se pueden utilizar en GitHub, dependiendo del proyecto y de cómo se desee colaborar en el código. La elección del flujo de trabajo correcto puede ayudar a garantizar que los cambios se realicen de manera eficiente y se mantengan en el tiempo.

Referencias

[1] Sitio Oficial
https://github.com/

[2] Comandos
https://www.freecodecamp.org/news/10-important-git-commands-that-every-developer-should-know/

[3] Git en el Servidor
https://git-scm.com/book/es/v2/Git-en-el-Servidor-Los-Protocolos

[4] Git en entornos distribuidos
https://git-scm.com/book/es/v2/Git-en-entornos-distribuidos-Flujos-de-trabajo-distribuidos