**Introducción a Git y GitHub**

**Control de versiones:**

Un Sistema de Control de Versiones (VCS) se refiere a*l método* utilizado para guardar las versiones de un archivo para referencia futura.

De manera intuitiva muchas personas ya utilizan control de versiones en sus proyectos al renombrar las distintas versiones de un mismo archivo de varias formas como blogScript.js, blogScript\_v2.js, blogScript\_v3.js, blogScript\_final.js, blogScript\_definite\_final.js, etcétera. Pero esta forma de abordarlo es propenso a errores y inefectivo para proyectos grupales.

Además, con esta forma de abordarlo, rastrear qué cambió, quién lo cambió y porqué se cambió, es un esfuerzo tedioso. Esto resalta la importancia de un sistema de control de versiones confiable y colaborativo como Git.

**Repositorio local**

Un repositorio local es una copia del repositorio de código fuente que resides en tu propia máquina. Es donde haces tus cambios y pruebas antes de sincronizarlos con el repositorio remoto. En Git, el repositorio local contiene todo el historial de versiones y la configuración del proyecto.

**Repositorio remoto**

Un repositorio remoto es una versión del repositorio que reside en un servidor remoto, como GitHub, GitLab o Bitbucket. Este repositorio actúa como un punto central donde se almacenan y comparten los cambios del proyecto con otros colaboradores.

**Commit**

Un **commit** es una acción que guarda los cambios en el repositorio local. Cada commit tiene un mensaje descriptivo que explica el propósito del cambio. Los commits forman un historial de cambios que se puede consultar y revertir si es necesario.

**Push**

El comando **push** se utiliza para enviar los commits del repositorio local al repositorio remoto. Esto actualiza el repositorio remoto con los cambios realizados localmente y permite que otros colaboradores vean y accedan a esos cambios.

**Branch (rama)**

Una **branch** (o rama) en Git es una línea de desarrollo independiente del proyecto. Las ramas permiten trabajar en diferentes características o arreglos de errores sin afectar la rama principal del proyecto (generalmente llamada main o master). Cada rama tiene su propio historial de commits.

**Merge (combinar)**

El **merge** (o combinar) es el proceso de integrar los cambios de una rama en otra. Por ejemplo, una vez que una característica ha sido completada en una rama, puedes combinar esos cambios en la rama principal mediante un merge. Este proceso puede generar conflictos si los cambios en las ramas son incompatibles.

**Pull request**

Un **pull request** es una solicitud para revisar y fusionar cambios de una rama a otra en un repositorio remoto. Generalmente, se utiliza en plataformas como GitHub para que otros colaboradores revisen el código antes de integrarlo en la rama principal. Los pull requests facilitan la revisión y discusión del código.

**Fork (bifurcación)**

Un **fork** (o bifurcación) es una copia independiente de un repositorio que se crea para experimentar con cambios sin afectar el repositorio original. Los forks son comunes en proyectos de código abierto, permitiendo a los usuarios desarrollar características o corregir errores en su propia copia antes de contribuir de vuelta al proyecto original.

**Clone (clonar)**

El comando **clone** se usa para crear una copia completa de un repositorio remoto en tu máquina local. Este proceso descarga todos los archivos y el historial de commits del repositorio remoto a tu sistema local, permitiéndote trabajar en el proyecto de manera local.

**Conflictos de merge**

Los **conflictos de merge** ocurren cuando Git no puede combinar automáticamente dos ramas debido a cambios incompatibles en el mismo archivo o línea de código. Cuando esto sucede, Git marca el archivo en conflicto y el usuario debe resolver manualmente las diferencias antes de completar el merge.

**Gitignore**

Un archivo **.gitignore** es un archivo de configuración en un repositorio Git que especifica qué archivos o directorios deben ser ignorados por Git. Los archivos listados en .gitignore no serán rastreados ni incluidos en los commits, lo cual es útil para evitar incluir archivos temporales, de configuración local o archivos generados automáticamente.

**Comparación entre Git y GitHub**

**Diferencia 1: Git vs. GitHub — Función principal**

**Git** es un sistema de control de versiones distribuido que registra las distintas versiones de un archivo (o conjunto de archivos). Le permite a los usuarios acceder, comparar, actualizar, y distribuir cualquiera de las versiones registradas en cualquier momento.

Sin embargo **GitHub** principalmente es una plataforma de alojamiento para albergar tus repositorios Git en la web. Esto permite a los usuarios mantener sus repositorios remotos privados o abiertos para esfuerzos colaborativos.

**Diferencia 2: Git vs. GitHub — Plataforma de operación**

Los usuarios instalan y ejecutan Git en sus equipos locales. Esto significa que la mayoría de las operaciones de Git se pueden lograr sin una conexión a internet.

Sin embarg GitHub es un servicio basado en la web que opera solamente en línea. Esto significa que necesitas estar conectado para hacer cualquier cosa en GitHub.

**Diferencia 3: Git vs. GitHub — Creadores**

Linus Torvalds comenzó Git en Abril del 2005.

Chris Wanstrath, P. J. Hyett, Tom Preston-Werner, y Scott Chacon fundaron GitHub.com en Febrero 2008.

**Diferencia 4: Git vs. GitHub — Mantenedores**

En Julio 2005, Linus Torvalds entregó el mantenimiento de Git a Junio C. Hamano — quien ha sido el principal mantenedor desde entonces.

En Octubre 2018, Microsoft compró GitHub.

**Diferencia 5: Git vs. GitHub — Competidores**

Algunas alternativas populares para Git son Mercurial, Team Foundation Version Control (TFVC), Perforce Helix Core, Apache Subversion, y IBM Rational ClearCase.

**Instalación de Git**

**En Windows**

1. **Descargar el Instalador:**
   * Ve a la [página de descargas de Git](https://git-scm.com/downloads).
   * Descarga el instalador para Windows (.exe).
2. **Ejecutar el Instalador:**
   * Haz doble clic en el archivo descargado para iniciar el proceso de instalación.
3. **Configuración del Instalador:**
   * **Pantalla de bienvenida**: Haz clic en "Next".
   * **Selección de componentes**: Acepta los componentes predeterminados y haz clic en "Next".
   * **Selección del editor de texto predeterminado**: Puedes elegir el editor que prefieras (por defecto es Vim). Haz clic en "Next".
   * **Configuración de la variable de entorno PATH**: Selecciona la opción "Git from the command line and also from 3rd-party software" para agregar Git a la variable PATH. Haz clic en "Next".
   * **Configuración de la terminal**: Elige "Use Windows' default console window" o "Use MinTTY" según prefieras. Haz clic en "Next".
   * **Configuración de line endings**: Puedes seleccionar "Checkout Windows-style, commit Unix-style line endings" para compatibilidad con Windows. Haz clic en "Next".
   * **Configuración de la configuración de Git**: Acepta las opciones predeterminadas y haz clic en "Next".
   * **Instalar**: Haz clic en "Install" y espera a que se complete la instalación. Luego, haz clic en "Finish".
4. **Verificar la Instalación:**
   * Abre el "Símbolo del sistema" o "Git Bash" y escribe git --version. Deberías ver la versión de Git instalada.

**En macOS**

1. **Instalar usando Homebrew (recomendado):**
   * Abre la aplicación **Terminal**.
   * Si no tienes Homebrew instalado, instálalo con el siguiente comando:

https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

* + Una vez instalado Homebrew, ejecuta:

brew install git

1. **Instalar desde el sitio web:**
   * Alternativamente, puedes descargar el instalador para macOS desde la [página de descargas de Git](https://git-scm.com/downloads) y seguir las instrucciones del asistente de instalación.
2. **Verificar la Instalación:**
   * Abre la **Terminal** y escribe git --version. Deberías ver la versión de Git instalada.

**Creación y Gestión de Repositorios en GitHub**

**Opciones Disponibles al Crear un Repositorio**

**En Git**

* **git init**: Inicializa un nuevo repositorio local.
* **git init --bare**: Crea un repositorio bare (sin área de trabajo).

**En GitHub**

1. **Nombre del Repositorio**: Especifica un nombre único.
2. **Descripción**: Opcional para explicar el proyecto.
3. **Visibilidad**: Público o privado.
4. **Inicializar con**:
   * **README**: Archivo inicial con información del proyecto.
   * **.gitignore**: Para excluir archivos específicos.
   * **Licencia**: Para definir términos de uso.
5. **Plantillas**: Comienza con una estructura predefinida.

**Gestión de un Repositorio**

**Colaboradores**

* **Añadir**: Desde "Settings" > "Collaborators and teams" en GitHub.
* **Permisos**: Read, Write, Admin.

**Configuración**

* **General**: Ajustes básicos en "Settings".
* **Branches**: Protege ramas en "Settings" > "Branches".
* **GitHub Actions**: Configura flujos de trabajo automáticos en "Actions".
* **Issues y Proyectos**: Administra en "Issues" y "Projects".
* **Seguridad**: Configura alertas y análisis en "Security".
* **Webhooks**: Configura integraciones en "Webhooks".

**Flujo de Trabajo en GitHub**

1. **Crear Branches (Ramas)**:
   * **Propósito**: Trabajar en características o correcciones sin afectar la rama principal.
   * **Ejemplo**: Crear una rama feature/search para añadir una nueva funcionalidad.

git checkout -b feature/search

1. **Commits**:
   * **Propósito**: Registrar cambios en el repositorio local.
   * **Ejemplo**: Hacer un commit después de añadir la funcionalidad.

git add .

git commit -m "Add search functionality"

1. **Push**:
   * **Propósito**: Enviar cambios al repositorio remoto en GitHub.
   * **Ejemplo**: Hacer un push de la rama feature/search.

git push origin feature/search

1. **Pull Request (PR)**:
   * **Propósito**: Solicitar revisión e integración de cambios.
   * **Ejemplo**: Crear un PR para fusionar feature/search en main.
2. **Merge**:
   * **Propósito**: Integrar cambios de una rama en otra después de la revisión.
   * **Ejemplo**: Hacer merge de feature/search en main después de la aprobación del PR.

**Casos de Uso y Ejemplos Prácticos**

[**Crear y probar en Node.js**](https://docs.github.com/es/enterprise-cloud@latest/actions/use-cases-and-examples/building-and-testing/building-and-testing-nodejs)

Puedes crear un flujo de trabajo de integración continua (CI) para construir y probar tu proyecto Node.js.

[**Compilar y probar PowerShell**](https://docs.github.com/es/enterprise-cloud@latest/actions/use-cases-and-examples/building-and-testing/building-and-testing-powershell)

Puedes crear un flujo de trabajo de integración continua (IC) para compilar y probar tu proyecto de PowerShell.

**Crear y Probar en Python**

Puedes crear un flujo de trabajo de integración continua (CI) para construir y probar tu proyecto de Python.

**Crear y probar en Ruby**

Puedes crear un flujo de trabajo de integración continua (CI) para crear y probar tu proyecto de Ruby.

Una captura de pantalla de una computadora

Descripción generada automáticamente

**Desafío Adicional**

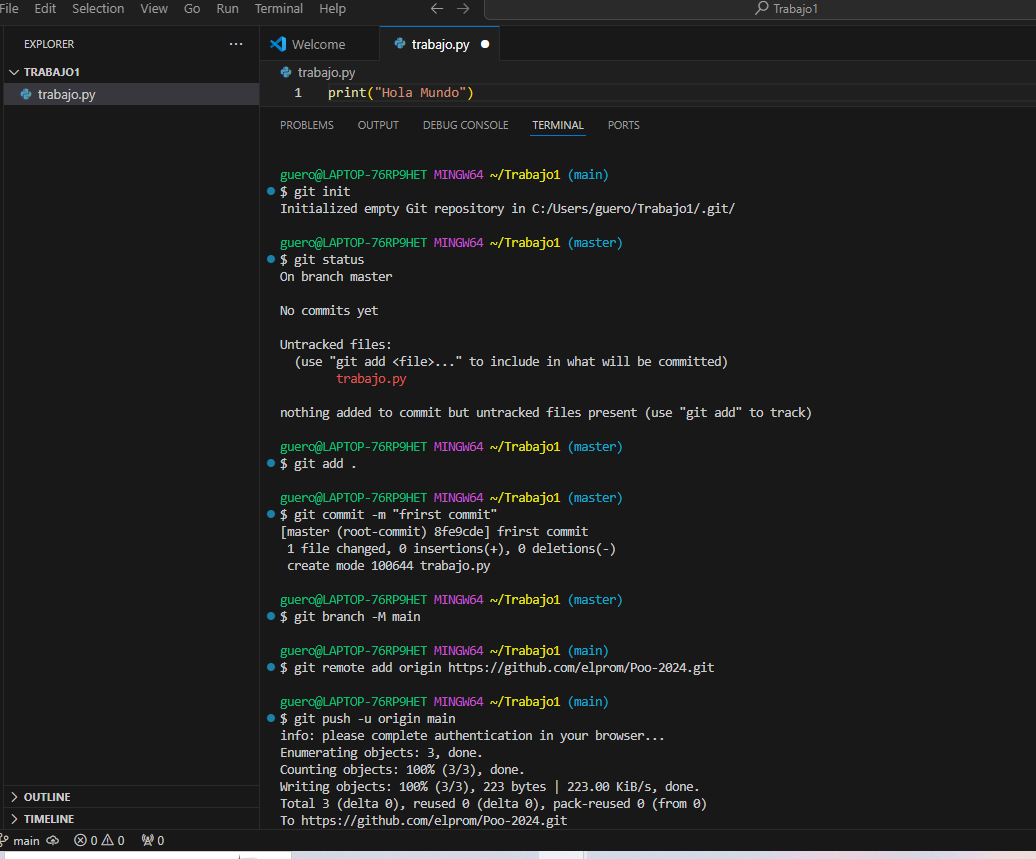
**GitHub Actions**

¿Cómo se utiliza?

1. **Crear un Workflow:**
   * Los workflows se definen en archivos YAML ubicados en .github/workflows/ en el repositorio.
2. **Configurar Eventos**:
   * Define cuándo debe ejecutarse el workflow (push, pull\_request, schedule, etc.).
3. **Definir Jobs y Steps**:
   * Los jobs son tareas que se ejecutan en paralelo o secuencialmente.
   * Los steps dentro de los jobs son comandos o acciones que se ejecutan.

**Beneficios en Gestión de Proyectos y Colaboración**

1. **Automatización**:
   * **CI/CD**: Automatiza la integración continua y el despliegue continuo, ejecutando pruebas y despliegues automáticamente en cada commit o pull request.
   * **Tareas Repetitivas**: Automatiza tareas repetitivas como la compilación, pruebas, o análisis de código.
2. **Consistencia**:
   * **Entornos Reproducibles**: Garantiza que el código se construya y se pruebe en un entorno limpio y consistente, reduciendo errores por diferencias en entornos locales.
3. **Colaboración Eficiente**:
   * **Revisiones Automáticas**: Las pruebas y verificaciones automáticas ayudan a identificar problemas antes de la revisión de código, haciendo que el proceso de revisión sea más eficiente.
   * **Transparencia**: Los resultados de los workflows son visibles en la pestaña "Actions" del repositorio, facilitando la supervisión del estado del proyecto.

Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente