Estudiante: Jesús Ortiz Beriguete | 2019-8124

Carrera: Desarrollo de software

Asignatura: Programación III

Maestro: Kerlyn Tejada Belliard

Grupo: 1

Tema: Asignación Individual, Tarea 3 (5pts)

Desarrolla el siguiente Cuestionario

1. **¿Qué es Git?**

   Conceptos básicos

   Git es un sistema de control de versiones distribuido ampliamente utilizado para el seguimiento y administración de cambios en proyectos de software. Permite a los desarrolladores trabajar de manera colaborativa y controlar el historial de modificaciones en un repositorio local.

   GitHub, por otro lado, es una plataforma en línea basada en Git que proporciona un servicio de alojamiento de repositorios remotos y herramientas adicionales para el desarrollo de software en equipo. Además de ofrecer almacenamiento en la nube para repositorios Git, GitHub facilita la colaboración, el seguimiento de problemas, la gestión de proyectos y la integración continua.

Git es ampliamente utilizado en el desarrollo de software debido a su eficacia para el control de versiones y la colaboración en equipo. Algunos de los usos más comunes de Git son:

1- **Control de versiones:** Git permite realizar un seguimiento preciso de los cambios realizados en el código fuente, lo que facilita la identificación de quién hizo qué modificación y en qué momento. Esto es útil para revertir cambios no deseados, comparar versiones anteriores, fusionar ramas de desarrollo y gestionar conflictos.

2- **Colaboración en equipo:** Git permite que múltiples desarrolladores trabajen simultáneamente en un proyecto, ya sea en el mismo equipo o distribuidos geográficamente. Cada desarrollador puede clonar el repositorio, hacer cambios en su propia copia y luego fusionar esos cambios de manera segura con la rama principal.

3- **Ramificación y fusión:** Git facilita la creación de ramas de desarrollo independientes para experimentar, trabajar en nuevas características o solucionar problemas sin afectar la rama principal. Posteriormente, es posible fusionar esas ramas con la rama principal, incorporando los cambios realizados.

4- **Rastreo de Problemas y Compromisos (Commits):** Git permite vincular cambios de código a problemas específicos, proporcionando un seguimiento claro de las actualizaciones relacionadas con tareas específicas. Los commits capturan los cambios realizados en el código con mensajes descriptivos.

5- **Despliegue Continuo:** Git se integra fácilmente con flujos de trabajo de despliegue continuo, permitiendo la automatización del proceso de implementación y garantizando la entrega rápida y confiable de nuevas versiones del software.

La importancia de Git radica en varios aspectos:

Historial y auditoría: Git registra un historial completo de todos los

cambios realizados en un proyecto, lo que permite rastrear y auditar los cambios en el tiempo. Esto brinda transparencia y permite un seguimiento preciso del progreso del proyecto.

Reversión de cambios: Si algo sale mal o se necesita volver a una versión anterior, Git proporciona herramientas para revertir cambios de forma segura y rápida, sin perder el trabajo realizado.

Trabajo en equipo eficiente: Git permite que múltiples desarrolladores colaboren de manera efectiva, ya que pueden trabajar en paralelo en diferentes partes del proyecto y luego fusionar sus cambios de manera controlada.

Existen diferentes tipos de sistemas de control de versiones, y Git se clasifica como un sistema de control de versiones distribuido. A diferencia de los sistemas de control de versiones centralizados, donde se depende de un único repositorio central, Git permite que cada desarrollador tenga una copia completa del repositorio, lo que brinda mayor flexibilidad, redundancia y capacidad de trabajo offline

2. ¿Para que funciona el comando Git init?

- **Definición 1:**
  El comando git init en Git se utiliza para inicializar un nuevo repositorio en un directorio local. Cuando se ejecuta este comando, Git configura la carpeta oculta .git, que almacena la información de control de versiones, como el historial de cambios, configuraciones y referencias. Esto permite a los desarrolladores comenzar a realizar un seguimiento de los cambios en sus proyectos y aprovechar las capacidades de control de versiones de Git.

- **Definición 2:**
  Git init es un comando fundamental en Git que se utiliza para convertir un directorio existente en un repositorio Git. Al ejecutar este comando, se crea una estructura interna en el directorio llamada .git, que almacena metadatos y objetos necesarios para realizar el seguimiento de versiones. Este paso inicializa el control

de versiones en el proyecto, permitiendo a los desarrolladores realizar commits, ramificar el código y colaborar de manera efectiva.

- **Definición 3:**
  Cuando se ejecuta git init, se está dando inicio a un nuevo repositorio de control de versiones. Este comando configura el entorno de trabajo para que Git comience a rastrear y gestionar los cambios en los archivos del proyecto. La creación de la carpeta .git dentro del directorio del proyecto establece las bases para funciones como el historial de cambios, ramificación y fusión, lo que permite a los desarrolladores trabajar de manera organizada y colaborativa en el desarrollo de software.

  Uso del comando "git init":

  Inicializar un Repositorio **($ git init)**: Este comando se ejecuta dentro del directorio del proyecto que se desea convertir en un repositorio Git. Crea una nueva carpeta oculta llamada .git, que contiene la estructura interna de Git y comienza a realizar un seguimiento de los cambios en ese directorio.

  Ejemplos de Uso:

  1- Nuevo Proyecto:
  $ mkdir nuevo_proyecto
  $ cd nuevo_proyecto
  $ git init

  Con este comando iniciamos un nuevo repositorio Git en el directorio 'nuevo_proyecto´.

  2- Convertir un proyecto existente:
  $ cd proyecto_existente
  $ git init

  Con esto iniciamos un repositorio Git en un proyecto existente

que aún no está bajo control de versiones.

CASOS EN LOS QUE NO SE DEBE UTILIZAR ´Git Init´:

1- **Dentro de Otro Repositorio Git:** No se debe ejecutar git init dentro de un subdirectorio que ya esté dentro de otro repositorio Git. Cada repositorio Git debe tener su propio directorio principal.

 # No se recomienda ejecutar git init aquí
$ cd proyecto_existente/subdirectorio
$ git init

Ya que esto puede provocar conflictos y comportamientos inesperados si se intenta anidar repositorios Git.

2- **Cuando se Desea Clonar un Repositorio Existente:** Si se desea trabajar con un repositorio Git existente y no crear uno desde cero, se debe utilizar el comando "git clone" en lugar de "git init".

# Utilizar git clone para clonar un repositorio existente
$ git clone url_del_repositorio

Utilizamos este comando, ya que git clone descargará una copia del repositorio existente y configurará automáticamente el seguimiento remoto.

3- **Cuando se Desea Inicializar un Proyecto con un Origen Remoto:** Si se planea trabajar con un repositorio remoto desde el principio, es más conveniente utilizar git clone con la URL del repositorio remoto en lugar de git init.

# Utilizar git clone para inicializar un proyecto con un origen remoto
$ git clone url_del_repositorio_remoto

Ya que de esta manera se puede configurar automáticamente un seguimiento remoto y descargará los archivos del repositorio remoto.

3. ¿Qué es una rama?

Rama en Git:

En el sistema de control de versiones Git, una rama es una línea de desarrollo independiente que representa una serie de cambios en el código fuente. Las ramas permiten la evolución paralela del software, ya que los desarrolladores pueden trabajar en diferentes aspectos del proyecto simultáneamente sin interferir entre sí. Al utilizar ramas, se mejora la organización del desarrollo y se facilita la incorporación de nuevas características o correcciones de errores de manera ordenada.

Funciones de una rama

1- **Desarrollo Paralelo:** Las ramas facilitan el desarrollo paralelo, permitiendo a los desarrolladores trabajar en distintas características o problemas simultáneamente sin interferir entre sí. Cada rama representa un hilo independiente de desarrollo, lo que facilita la colaboración entre desarrolladores en un proyecto sin interferencias directas entre sus cambios.

   Los desarrolladores pueden crear nuevas ramas para abordar funcionalidades específicas sin afectar la rama principal. Esto fomenta un flujo de trabajo eficiente y distribuido, donde diferentes equipos o desarrolladores individuales pueden avanzar en sus respectivas tareas sin conflictos inmediatos.

2- **Experimentación y Pruebas:** Las ramas proporcionan un entorno aislado para la experimentación y pruebas sin comprometer la estabilidad de la rama principal. Los desarrolladores pueden probar nuevas ideas, implementar cambios arriesgados o explorar soluciones alternativas en una rama separada sin afectar el código base.

Al crear una rama específica para experimentos, los desarrolladores pueden realizar cambios sin el temor de afectar el estado funcional del proyecto principal. Esto fomenta la innovación y la mejora continua al permitir que los desarrolladores prueben enfoques diferentes antes de integrar cambios en la rama principal.

3- **Aislamiento de Cambios:** Cada rama en Git actúa como un entorno aislado, lo que significa que los cambios realizados en una rama no afectan directamente a otras ramas hasta que se fusionan. Este aislamiento facilita la gestión de cambios, reduciendo la posibilidad de conflictos y simplificando la identificación de la fuente de errores.

   Los desarrolladores pueden trabajar con confianza en sus propias ramas, sabiendo que los cambios no impactarán inmediatamente otras partes del proyecto. Esto simplifica el proceso de revisión de código, pruebas y mantenimiento general al garantizar que los cambios se integren de manera controlada y planificada.

   Ejemplos de Uso:

   1- **Desarrollo de Nuevas Funcionalidades:**

      $ git branch nueva_funcionalidad
      $ git checkout nueva_funcionalidad
      # Realizar cambios y commits en la nueva rama
      $ git checkout master **# Regresar a la rama principal**
      $ git merge nueva_funcionalidad **# Fusionar los cambios en la rama principal**

   En este ejemplo, se crea una nueva rama llamada "nueva_funcionalidad" para desarrollar una característica específica. Después de completar el desarrollo, los cambios se fusionan de nuevo en la rama principal.

   2- **Corrección de Errores (Bug Fixes):**

      $ git branch correccion_bug

```
$ git checkout correccion_bug
# Corregir el error y realizar commits en la nueva rama
$ git checkout master  # Regresar a la rama principal
$ git merge correccion_bug  # Fusionar los cambios en la rama principal
```

En este caso, se crea una rama específica ("correccion_bug") para abordar un error crítico. Una vez que la corrección se ha realizado y probado, se fusionan los cambios de vuelta en la rama principal.

CASOS EN LOS QUE NO UTILIZAR RAMA:

1- **Proyectos Muy Pequeños:** En proyectos pequeños o scripts simples, donde la complejidad de trabajar con múltiples ramas no proporciona beneficios significativos, puede ser innecesario utilizar ramas.

2- **Proyectos de Una Sola Persona:** En proyectos donde solo una persona está trabajando y no hay necesidad inmediata de desarrollo paralelo, el uso de ramas puede ser opcional. Un único flujo de desarrollo puede ser suficiente.

3- **Desarrollo de Hotfixes Urgentes:** En situaciones críticas que requieren correcciones inmediatas y urgentes, algunos equipos pueden optar por realizar cambios directamente en la rama principal para una solución más rápida, en lugar de crear una rama separada.

Recomendaciones:

1- **Proyectos más Grandes y Colaborativos:** Se recomienda utilizar ramas en proyectos más grandes y colaborativos para mantener un flujo de trabajo organizado y facilitar la gestión de cambios. Esto mejora la coordinación entre desarrolladores y equipos.

2- **Nombrar Ramas de Manera Descriptiva:** Es una buena

práctica nombrar las ramas de manera descriptiva (por ejemplo, "nueva_funcionalidad" o "correccion_bug") para comprender fácilmente su propósito y facilitar la colaboración y revisión de código.

4. ¿Como saber es que rama estoy?

Para verificar en qué rama te encuentras en un repositorio Git, puedes utilizar el siguiente comando en tu terminal:

$ git branch

Este comando te mostrará una lista de todas las ramas en tu repositorio, y la rama actual estará resaltada con un asterisco (*). El nombre de la rama actual se encuentra junto al asterisco.

Si prefieres obtener solo el nombre de la rama actual de manera más concisa, puedes usar el siguiente comando:

$ git branch --show-current

Este comando te mostrará únicamente el nombre de la rama actual sin la lista completa de todas las ramas.

Entonces:

git branch: Muestra todas las ramas y resalta la rama actual con un asterisco.

git branch --show-current: Muestra solo el nombre de la rama actual.

5. ¿Quién creo git?

Git fue concebido por Linus Torvalds, el creador del kernel de Linux, en 2005. La motivación detrás de la creación de Git surgió de la necesidad de encontrar un sistema de control de versiones distribuido eficaz para supervisar el desarrollo colaborativo del kernel de Linux.

Previo a la adopción de Git, el proyecto del kernel de Linux empleaba BitKeeper como su sistema de control de versiones. Sin embargo, en 2005, la relación entre la comunidad de desarrollo de Linux y la empresa que gestionaba BitKeeper se volvió problemática. La situación empeoró cuando la licencia gratuita que permitía el uso de BitKeeper para el desarrollo del kernel de Linux fue retirada.

Ante este escenario, Linus Torvalds tomó la decisión de crear su propio sistema de control de versiones distribuido. Su objetivo era desarrollar una herramienta que satisficiera las necesidades específicas del proyecto del kernel de Linux. Git fue diseñado con un enfoque distribuido, eficiente y rápido, incorporando un conjunto de características que lo hacían especialmente adecuado para un proyecto de código abierto a gran escala, con numerosos colaboradores distribuidos geográficamente.

En resumen, Linus Torvalds desarrolló Git como una solución específica para las demandas del desarrollo del kernel de Linux. La eficiencia, rapidez y capacidad de gestionar proyectos colaborativos llevaron a Git a convertirse en una herramienta esencial en la comunidad de desarrollo de software en general.

6. ¿Cuáles son los comandos más esenciales de Git?

Función	Descripción
git init	**Inicializa un nuevo repositorio Git.**

Función	Descripción
git clone <url_del_repositorio>	**Clona un repositorio existente en un nuevo directorio.**
git add <nombre_del_archivo>	**Agrega cambios al área de preparación (staging).**
git commit -m "Mensaje del commit"	**Registra los cambios en el repositorio.**
git status	**Muestra el estado de los archivos en el directorio de**
	trabajo y el área de preparación.
git log	**Muestra el historial de commits.**
git branch	**Lista, crea o borra ramas.**
git checkout <nombre_de_rama>	**Cambia entre ramas o versiones específicas de archivos.**
git merge <nombre_de_rama>	**Fusiona cambios de una rama a otra.**
git pull	**Recupera cambios del repositorio remoto y fusiona en la**
	rama local.
git push <nombre_remoto> <nombre_de_rama>	**Envía cambios locales a un repositorio remoto.**
git remote -v	**Lista los repositorios remotos.**
git fetch	**Recupera cambios del repositorio remoto sin fusionar.**
git diff	**Muestra las diferencias entre los cambios no**
	confirmados y el último commit.
git reset <nombre_de_archivo>	**Deshace cambios locales.**

Función	Descripción
git remote add <nombre_remoto> <url_del_repositorio>	**Agrega un nuevo repositorio remoto.**
git remote remove <nombre_remoto>	**Elimina un repositorio remoto.**
git pull --rebase	**Recupera cambios del repositorio remoto y aplica rebase en lugar de merge.**
git log --oneline	**Muestra el historial de commits en una línea por commit.**
git tag <nombre_del_tag>	**Crea un nuevo tag para el commit actual.**
git diff HEAD	**Muestra las diferencias entre el directorio de trabajo y el último commit**
git branch -d <nombre_de_rama>	**Borra una rama después de fusionar sus cambios.**
git stash	**Guarda temporalmente los cambios locales no comprometidos**
git submodule add <url_del_submódulo>	**Agrega un submódulo a tu repositorio.**
git clean -n	**Muestra los archivos no rastreados que serán eliminados con git clean.**
git cherry-pick <commit_hash>	**Aplica un commit específico a la rama actual.**
git revert <commit_hash>	**Crea un nuevo commit que deshace los cambios de un commit anterior.**
git remote show <nombre_remoto>	**Muestra información detallada sobre un repositorio remoto**
git log --graph --oneline --all	**Muestra el historial de commits con un gráfico ASCII.**
git bisect start	**Inicia una búsqueda binaria para encontrar el commit que introdujo un problema**

7. ¿Qué es Git Flow?

Git Flow es un conjunto de reglas y prácticas para el uso de Git, un sistema de control de versiones distribuido. Fue propuesto por Vincent Driessen en 2010 como una forma de estructurar y organizar el flujo de trabajo en proyectos de desarrollo de software. Aquí hay una descripción general de Git Flow:

Estructura de Git Flow:

1. **Branches Principales:**

    - Master: Representa la rama principal del código en producción.
    - Develop: Es la rama donde se integran todas las características antes de pasar a master

2. **Branches de Soporte:**

    - feature: Ramas para desarrollar nuevas características.
    - release: Ramas destinadas a preparar una nueva versión de producción.
    - hotfix: Ramas para corregir rápidamente errores en producción.

Importancia y Ventajas:

1- **Organización del Flujo de Trabajo:** Git Flow proporciona una estructura clara y definida para el desarrollo de software, facilitando la colaboración y la gestión del código.

2- **Control de Versiones:** Facilita el control de versiones, con ramas específicas para nuevas características, versiones de release y correcciones de errores.

3- **Entregas Estables**: Permite tener versiones estables y en desarrollo por separado, lo que es crucial para entornos de producción.

4- **Colaboración Eficiente:** Favorece la colaboración entre desarrolladores al proporcionar un flujo de trabajo común.

Para qué Sirve:

1- **Desarrollo Estructurado:** Facilita el desarrollo ordenado de nuevas características, correcciones y versiones.

2- **Control de Versiones:** Ayuda a gestionar eficientemente las versiones del software, permitiendo la integración de características y correcciones de errores de manera controlada.

Ejemplos de Uso:

1- **Iniciar una Nueva Funcionalidad:**

   git flow feature start nueva-funcionalidad

2- **Preparar una Nueva Versión de Producción:**

   git flow release start 1.0.0

3- **Corregir un Error en Producción:**

   git flow hotfix start correction-error

CASOS DONDE NO UTILIZAR GIT FLOW:

1- **Proyectos Pequeños o Individuales:** Para proyectos pequeños o proyectos personales, la complejidad de Git Flow puede ser excesiva.

2- **Metodologías Ágiles:** En entornos ágiles donde se prefieren despliegues continuos y entregas frecuentes, Git Flow podría ser demasiado rígido.

3- **Equipos no Familiarizados con Git Flow:** Si un equipo no está acostumbrado o no encuentra útil la estructura proporcionada por Git Flow, podría ser contraproducente.

8. ¿Qué es trunk-based development?

El Trunk-Based Development (TBD) es una metodología de desarrollo de software que se caracteriza por mantener una única rama principal, a menudo llamada "trunk" o "master", como la rama activa y principal del repositorio de código. En lugar de utilizar ramas a largo plazo para el desarrollo de nuevas características, el TBD aboga por integrar cambios directamente en la rama principal tan pronto como estén listos. Este enfoque se basa en la premisa de que la rama principal siempre debe estar en un estado funcional y listo para ser desplegada en producción.

Características del Trunk-Based Development:

1- **Rama Principal Única:** Utiliza una única rama principal (trunk o master) para el desarrollo continuo.

2- **Integración Continua:** Favorece la integración continua, donde los cambios se fusionan con la rama principal tan pronto como estén listos.

3- **Entregas Continuas:** Se orienta hacia entregas más frecuentes y pequeñas, lo que facilita la identificación temprana de problemas.

Importancia y Ventajas

1- **Retroalimentación Rápida:** Permite obtener retroalimentación rápida sobre el impacto de los cambios al integrar continuamente en la rama principal.

2- **Reducir Conflictos de Integración:** Al integrar con frecuencia, se minimizan los conflictos de integración, ya que los cambios son más

pequeños y manejables.

3- **Mayor Visibilidad:** Todos los cambios están directamente en la rama principal, lo que proporciona una visibilidad clara del estado del proyecto.

4- **Entregas Más Rápidas:** Al eliminar el proceso de ramificación extenso, las entregas pueden realizarse más rápidamente.

Ejemplos de Uso:

1- **Commit y Fusiones Continuas:** Los desarrolladores envían sus cambios al repositorio central y se fusionan con la rama principal después de una revisión y pruebas.

2- **Despliegue Continuo:** La rama principal se considera siempre lista para ser desplegada en producción, fomentando un enfoque de despliegue continuo.

3- **Pruebas Automatizadas:** Uso extensivo de pruebas automatizadas para garantizar la calidad y estabilidad del código antes de la integración.

Casos donde no Utilizar Trunk-Based Development:

- **Proyectos Altamente Inestables:** Si el código base está experimentando una alta inestabilidad, es posible que sea más apropiado utilizar un enfoque más estructurado como Git Flow.

- **Equipos No Preparados para Integración Continua:** Si un equipo no tiene prácticas de prueba automatizadas sólidas o no está listo para adoptar la integración continua, el Trunk-Based Development puede ser desafiante.

- **Proyectos con Requisitos Regulatorios Específicos:** En algunos casos, proyectos con requisitos regulatorios estrictos pueden necesitar

procesos más controlados, lo que podría no ser totalmente compatible con el modelo de Trunk-Based Development.

2-Desarrolle un ejercicio práctico en Azure Devops o GitHub con las siguientes Características

Entregas:
Link de GitHub donde se desarrollen las siguientes actividades:
-Crear un proyecto.
-Utilizar la técnica Git Flow en su proyecto.
-Proyecto funcional.