1. INTRODUCCIÓN

En el desarrollo de sistemas y servicios tecnológicos del Ideam, Git se ha consolidado como el sistema de control de versiones más utilizado, permitiendo un seguimiento eficiente de los cambios en el código y facilitando la colaboración entre equipos de desarrollo. GitLab complementa esta funcionalidad proporcionando una plataforma integral de gestión de repositorios, integración y despliegue continuo, así como herramientas de seguridad y automatización.

Este documento tiene como propósito ofrecer una guía clara y estructurada sobre el uso de Git y GitLab en el contexto del Ideam, estableciendo las bases conceptuales, metodológicas y normativas para su aplicación eficiente en el desarrollo y mantenimiento de sistemas de información, servicios tecnológicos y plataformas institucionales.

2. OBJETIVO

Proporcionar una guía metodológica para el uso de Git y GitLab en el desarrollo y mantenimiento de sistemas y servicios tecnológicos del Ideam, con el fin de optimizar la gestión de versiones, mejorar la colaboración y garantizar la trazabilidad del código fuente en los proyectos tecnológicos de la entidad.

3. ALCANCE

Este documento está dirigido a desarrolladores, arquitectos de software y equipos técnicos del Ideam o terceros para gestionar repositorios de código fuente utilizando Git y GitLab. La guía abarca desde conceptos básicos hasta prácticas avanzadas, incluyendo la configuración, uso de ramas, estrategias de fusionado, integración y despliegue continuo (CI/CD), seguridad y gobernanza del código, alineados con las necesidades de los sistemas y servicios tecnológicos de la entidad.

4. DEFINICIONES

- **Git**: Sistema de control de versiones distribuido que permite gestionar y registrar cambios en el código fuente de un proyecto de manera eficiente.
- **GitLab**: Plataforma de gestión de repositorios que proporciona herramientas para integración y despliegue continuo, seguridad y colaboración.
- **Repositorio**: Espacio de almacenamiento donde se guardan los archivos de un proyecto junto con su historial de cambios.
- **Branch (Rama)**: Versión paralela del código dentro de un repositorio, utilizada para desarrollar nuevas funcionalidades sin afectar la versión principal.
- **Merge (Fusión)**: Integración de cambios desde una rama hacia otra, combinando las modificaciones realizadas.
- **CI/CD (Integración y Despliegue Continuo)**: Prácticas automatizadas para verificar, probar y desplegar cambios en el código de manera eficiente.

5. SIGLAS

- **CI/CD**: Continuous Integration / Continuous Deployment (Integración y Despliegue Continuo).
- **SSH**: Secure Shell.
- **IDE**: Integrated Development Environment (Entorno de Desarrollo Integrado).
- **API**: Application Programming Interface (Interfaz de Programación de Aplicaciones).
- **HTTP**: Hypertext Transfer Protocol.

6. MARCO NORMATIVO

- **Ley 1581 de 2012**: Por la cual se dictan disposiciones generales para la protección de datos personales, aplicable a plataformas como GitLab cuando manejan información sensible.
- **Resolución 746 de 2022**: por la cual se fortalece el Modelo de Seguridad y Privacidad de la Información y se definen lineamientos adicionales a los establecidos en la Resolución No. 500 de 2021**.**
- **Marco de Referencia de Arquitectura Empresarial del Estado Colombiano**: Establece lineamientos para la gestión de tecnología en entidades gubernamentales, promoviendo el uso de herramientas como Git y GitLab para el desarrollo seguro de software en el Ideam.

7. DESCRIPCIÓN METODOLÓGICA DEL TEMA A DESARROLLAR

La metodología presentada en este documento se basa en buenas prácticas recomendadas a nivel internacional y adaptadas al contexto del Ideam. Se estructura en los siguientes pasos:

- **Configuración de Git y GitLab**: Instalación y configuración inicial para el uso eficiente de ambas herramientas en el desarrollo de sistemas y servicios tecnológicos del Ideam.
- **Gestión de repositorios**: Creación, clonación y mantenimiento de repositorios para un desarrollo organizado y alineado con los requerimientos tecnológicos de la entidad.
- **Uso de ramas y estrategias de fusionado**: Aplicación de metodologías como Git Flow y Trunk-Based Development para una mejor gestión del código en los proyectos del Ideam.
- **Integración y despliegue continuo (CI/CD)**: Automatización de pruebas, verificaciones y despliegues para mejorar la calidad del software institucional.

- **Seguridad y gobernanza del código**: Implementación de buenas prácticas para la protección del código fuente y la gestión de accesos en GitLab, garantizando la seguridad de los sistemas del Ideam.

8. SISTEMAS DE CONTROL DE VERSIONES

Un sistema de control de versiones (VCS, por sus siglas en inglés) es una herramienta que permite gestionar los cambios en archivos y proyectos a lo largo del tiempo. Su principal función es registrar cada modificación realizada en un conjunto de archivos, permitiendo a los desarrolladores revertir cambios, comparar versiones anteriores y colaborar de manera eficiente en equipo.

Características principales:

- **Historial de cambios**: Guarda todas las modificaciones realizadas en los archivos del proyecto.
- **Colaboración**: Facilita el trabajo en equipo, permitiendo a varios desarrolladores trabajar en el mismo código sin sobrescribir cambios.
- **Ramas y fusiones (branching & merging)**: Permite crear versiones paralelas del proyecto para desarrollar nuevas funciones sin afectar la versión principal.
- **Seguridad y recuperación:** Permite restaurar versiones anteriores en caso de errores o pérdida de información.
- **Seguimiento de autoría:** Registra quién hizo qué cambios y cuándo.

Tipos de sistemas de control de versiones:

- **Locales:** Almacenan versiones de archivos en un solo equipo (Ejemplo: copias manuales con diferentes nombres).
- **Centralizados (CVCS):** Un servidor central gestiona el historial de cambios y los usuarios acceden a él (Ejemplo: SVN, Perforce).
- **Distribuidos (DVCS):** Cada usuario tiene una copia completa del historial, lo que permite trabajar sin conexión y realizar fusiones más eficientes (Ejemplo: Git, Mercurial).

El sistema de control de versiones más popular actualmente es Git, utilizado en plataformas como GitHub, GitLab y Bitbucket.

8.1. GIT

Es un sistema de control de versiones distribuido. Permite a los desarrolladores llevar un registro de los cambios en el código fuente a lo largo del tiempo. Con Git, los desarrolladores pueden colaborar en proyectos, trabajar en paralelo en diferentes ramas de desarrollo, fusionar cambios y realizar un seguimiento del historial de revisiones.

8.2. GITLAB

Por otro lado, es una plataforma de gestión del ciclo de vida del desarrollo de software (SDLC). Ofrece funciones más allá del control de versiones proporcionado por Git. GitLab incluye características como seguimiento de problemas, solicitudes de extracción (merge requests), integración continua (CI), entrega continua (CD), wikis de proyectos y más.

8.3. GITLAB EN IDEAM

La Oficina de Informática ha implementado GitLab como plataforma para la gestión de repositorios de código fuente, permitiendo optimizar el desarrollo y mantenimiento de sistemas y servicios tecnológicos del Ideam. Su adopción facilita la colaboración entre equipos, el control de versiones y la implementación de buenas prácticas en el ciclo de vida del desarrollo de software (SDLC).

GitLab está disponible en el siguiente enlace:
https://gitlab.ideam.gov.co/

Para garantizar una adecuada organización, los proyectos dentro de la plataforma deben estructurarse en grupos alineados con las dependencias del Ideam, y dentro de estos, en subgrupos que correspondan a proyectos específicos.

Si se requiere la creación de un nuevo repositorio para gestionar código fuente en GitLab, la solicitud debe realizarse a través de la Mesa de Servicio.

Para trabajar con los repositorios en entornos locales, es necesario instalar y utilizar Git, ya sea de manera independiente o mediante herramientas integradas en los entornos de desarrollo (IDE) de preferencia.

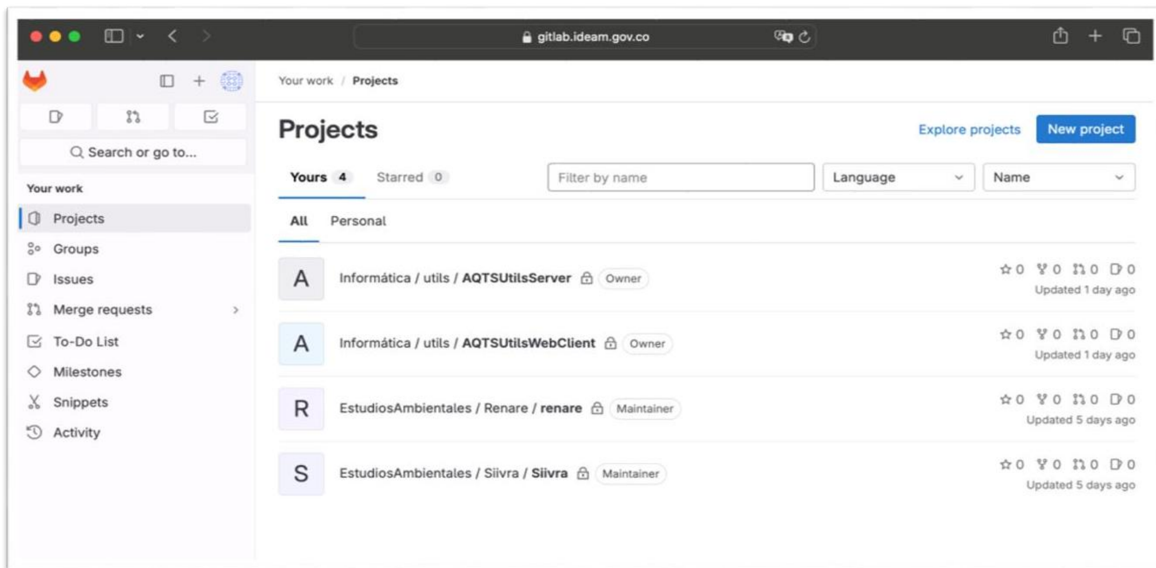

Imagen 1 Repositorio de proyectos

8.4. FUNCIONAMIENTO GIT – GITLAB

Como se ha mencionado, Git es un sistema de control de versiones que permite gestionar el historial de cambios de un proyecto, mientras que GitLab es una plataforma basada en Git que funciona como un servidor remoto para almacenar y gestionar repositorios de código fuente.

Al trabajar en un proyecto con Git en un entorno local, es fundamental sincronizar periódicamente los cambios con GitLab para respaldar el código y facilitar la colaboración. Durante el proceso de desarrollo, los equipos deberán subir y descargar archivos de la plataforma de manera continua.

Para realizar estas acciones, se emplean una serie de comandos y herramientas que permiten gestionar el código de manera eficiente, asegurando la trazabilidad y control de versiones a lo largo del desarrollo del software.

La plataforma GitLab del Ideam está disponible en el siguiente enlace:

https://gitlab.ideam.gov.co/

En las siguientes secciones, profundizaremos en el uso de GitLab y sus funcionalidades dentro del Ideam.

9. INSTALACIÓN GIT

Para utilizar Git en entornos de desarrollo locales, es necesario instalarlo en el sistema operativo correspondiente. Git está disponible para Windows, Linux y macOS, permitiendo a los desarrolladores gestionar repositorios de código fuente de manera eficiente.

En esta sección, se detallará el proceso de instalación de Git en Windows, incluyendo recomendaciones para optimizar su uso, como la instalación de Git Bash para una mejor gestión de comandos y la configuración de OpenSSL para asegurar la comunicación cifrada.

10. INSTALACIÓN GIT EN WINDOWS

Para instalar Git en su equipo, siga estos pasos detallados:

- **Acceda al sitio web oficial de Git** ingresando a la siguiente dirección: https://git-scm.com/
- En la página principal, ubique y haga clic en el botón **"Download"** o **"Descargar"**, el cual detectará automáticamente su sistema operativo.

- Si el sitio no selecciona automáticamente la versión correcta, asegúrese de elegir la opción correspondiente a su sistema operativo (**Windows, macOS o Linux**).
- Una vez iniciada la descarga, espere a que el archivo ejecutable se descargue completamente en su equipo antes de proceder con la instalación.

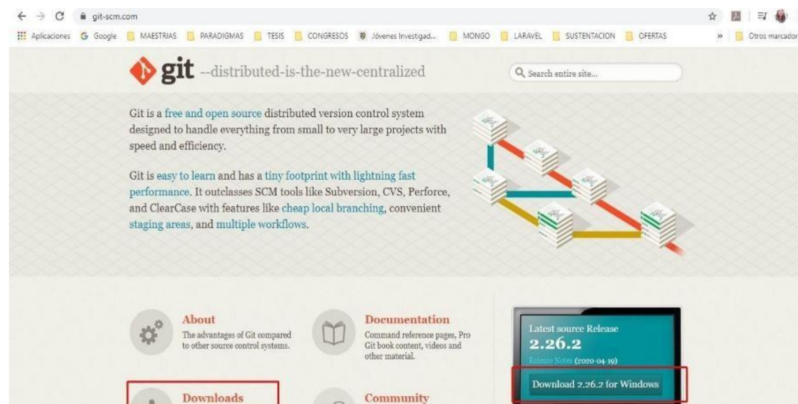

Imagen 2 Página oficial de descarga

Una vez completada la descarga del archivo de instalación de Git, proceda con la instalación en Windows siguiendo estos pasos detallados:

- **Ubique el archivo descargado** en su carpeta de descargas o en la ubicación donde haya sido guardado.
- **Haga doble clic en el archivo ejecutable** para iniciar el asistente de instalación.
- **Siga las instrucciones en pantalla**, seleccionando las opciones recomendadas según sus necesidades.

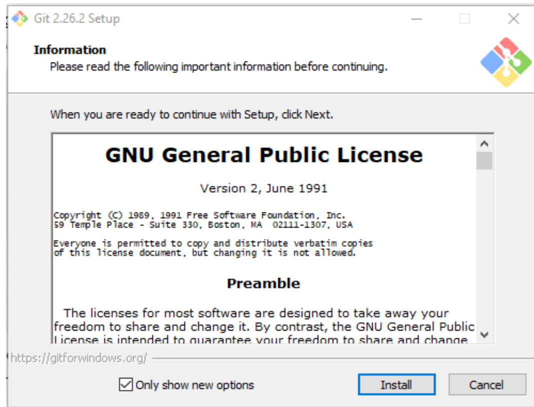

Imagen 3 Información de instalación

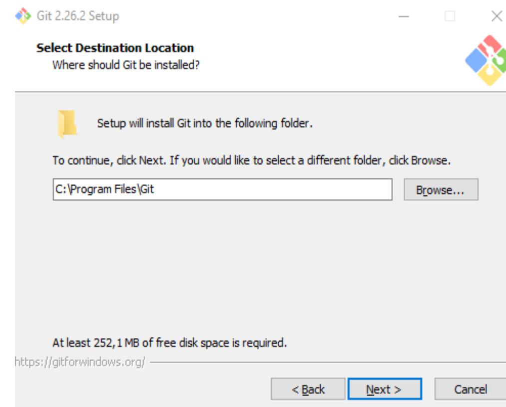

Imagen 4 Destino de instalación

Para facilitar el uso de comandos en la terminal en Windows, se recomienda instalar Git Bash, una herramienta que proporciona un entorno de línea de comandos similar al de Linux, permitiendo ejecutar comandos de Git de manera más eficiente. No obstante, su uso no es obligatorio, ya que también es posible utilizar Git a través de la consola de comandos de Windows (CMD) o PowerShell.

Además, para garantizar una comunicación segura al interactuar con repositorios remotos, Git utiliza SSL (Secure Sockets Layer). En sistemas Linux, esta configuración viene habilitada de forma predeterminada, pero en Windows es necesario activarla manualmente durante la instalación de Git. Se recomienda seleccionar la opción de OpenSSL en el asistente de instalación para asegurar una conexión cifrada y compatible con los servidores GitLab del Ideam.

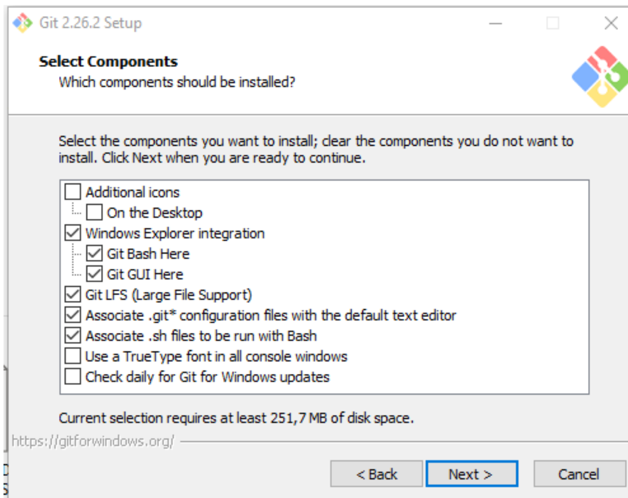

Imagen 5 Selección de componentes

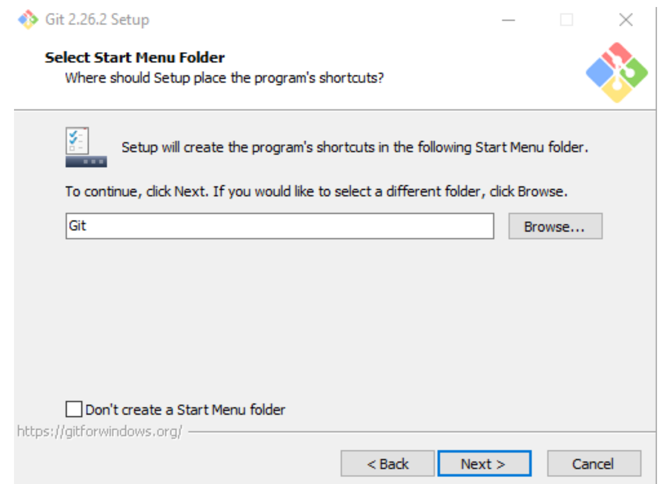

Imagen 6 Selección de carpeta

Dado que los sistemas operativos Windows y Linux manejan los saltos de línea de manera diferente, es recomendable configurar Git en Windows para que convierta automáticamente los finales de línea al estándar de Linux al enviar modificaciones al servidor. Esta opción se puede seleccionar durante la instalación de Git en Windows, asegurando así una mayor compatibilidad con los repositorios utilizados en entornos Linux.

Para verificar que Git se ha instalado correctamente en su sistema, puede hacerlo de la siguiente manera:

Desde el menú de inicio de Windows:

- Abra el menú de inicio y busque **Git Bash** o **Git CMD**.
- Si estas aplicaciones aparecen en la lista, significa que Git se ha instalado correctamente.

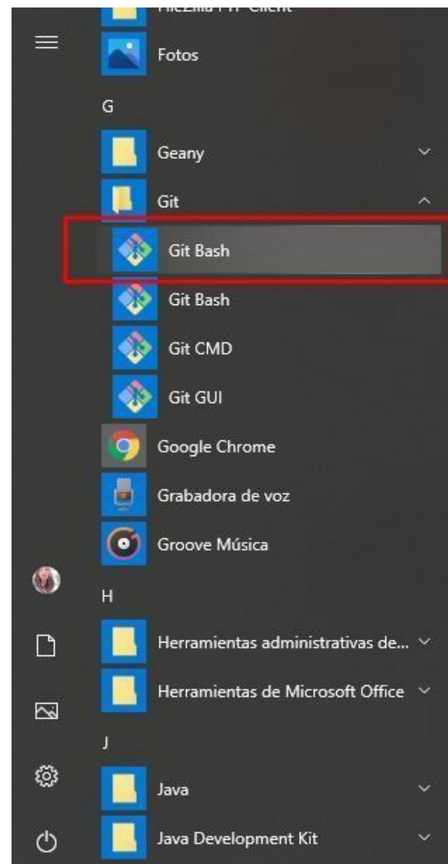

Imagen 7 Búsqueda de Git Bash en menú Windows

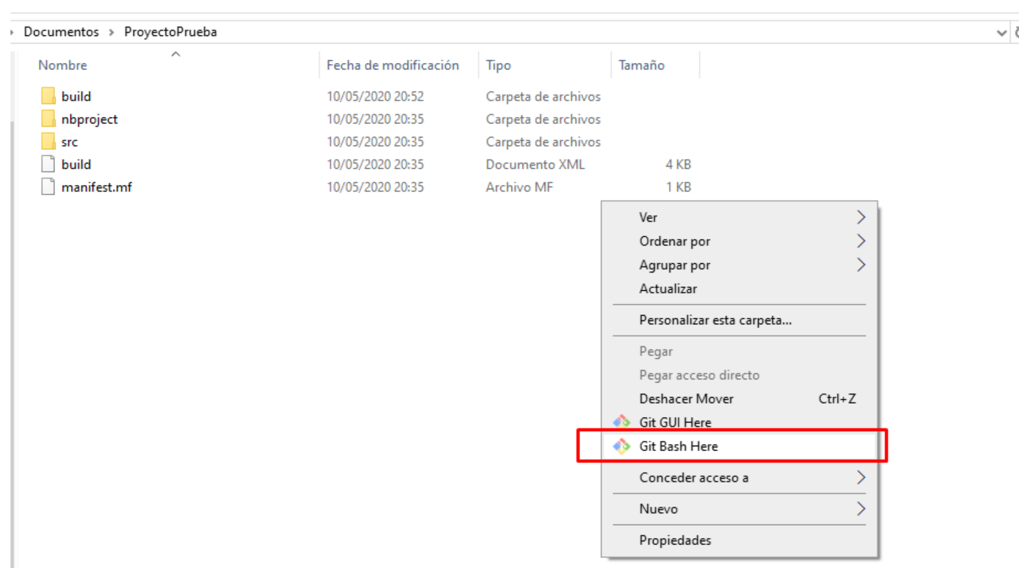

Imagen 8 verificación

11. CONFIGURACIÓN DE GIT

Para configurar Git en su equipo, es necesario realizar algunas configuraciones iniciales a través de la terminal **Git Bash**. A continuación, se describen los pasos detallados para establecer la configuración básica y preparar el entorno de trabajo.

11.1. VERIFICACIÓN DE LA INSTALACIÓN

Antes de configurar Git, es recomendable asegurarse de que la instalación se realizó correctamente. Para ello, abra **Git Bash** y ejecute el siguiente comando:

- git –versión (Este comando devolverá la versión de Git instalada en el sistema)

```
marce@DESKTOP-JV1GF7E MINGW64 ~
$ git --version
git version 2.26.2.windows.1
```

Imagen 9 Verificación de instalación Git Bash

11.2. CONFIGURACIÓN DEL USUARIO

Para poder registrar correctamente los cambios en los repositorios, es necesario configurar el **nombre de usuario** y el **correo electrónico**, que serán utilizados en cada confirmación (**commit**).

Usamos ahora > **git config** para configurar el nombre de usuario y el correo electrónico

Ejecute los siguientes comandos en la terminal de Git Bash, reemplazando "Su Nombre" y "su.correo@institucional.com" con sus datos:

git config --global user.name "Su Nombre"

git config --global user.email "su.correo@institucional.com"

```
marce@DESKTOP-JV1GF7E MINGW64 ~
$ git config --global user.email marcelaguerrero1396@gmail.com

marce@DESKTOP-JV1GF7E MINGW64 ~
$ git config --global user.name "Marcela Guerrero"
```

Imagen 10 Configuración de usuario

Estos datos quedarán almacenados en la configuración global de Git y se usarán en todos los repositorios en su sistema.

Una vez configurado podremos clonar o inicializar un proyecto para trabajar en el usando clone o init.

11.3. EXPLORACIÓN DE LA CONFIGURACIÓN DE GIT

Para verificar la configuración actual de Git, puede ejecutar el siguiente comando:

> git config –list

Si desea conocer el origen de cada configuración almacenada, utilice:

> git config --list --show-origin

11.4. CLONACIÓN E INICIALIZACIÓN DE REPOSITORIOS EN GIT

11.4.1. GIT CLONE

Clonación de un Repositorio Existente.

Para trabajar con un proyecto alojado en **GitLab**, primero debe asegurarse de que el repositorio ya ha sido creado dentro de la plataforma de la entidad. Una vez disponible, puede clonarlo en su equipo utilizando los comandos que GitLab proporciona en la interfaz del repositorio.

Pasos para clonar un repositorio:

- **Acceda a GitLab** y ubique el repositorio del proyecto que desea clonar.
- En la interfaz del repositorio, busque la opción **Clone** y seleccione el método de clonación:
  - **HTTPS:** Recomendado para la mayoría de los usuarios.
  - **SSH:** Requiere configuración previa de claves SSH.
- Copie la URL del repositorio haciendo clic en el botón **Copy URL**.
- Abra **Git Bash** o una terminal y ejecute el siguiente comando, reemplazando rutadesuproyecto con la URL copiada:

  > git clone rutadesuproyecto

  Ejemplo:

  > git clone https://gitlab.ideam.gov.co/informatica/bigdata/exporter.git

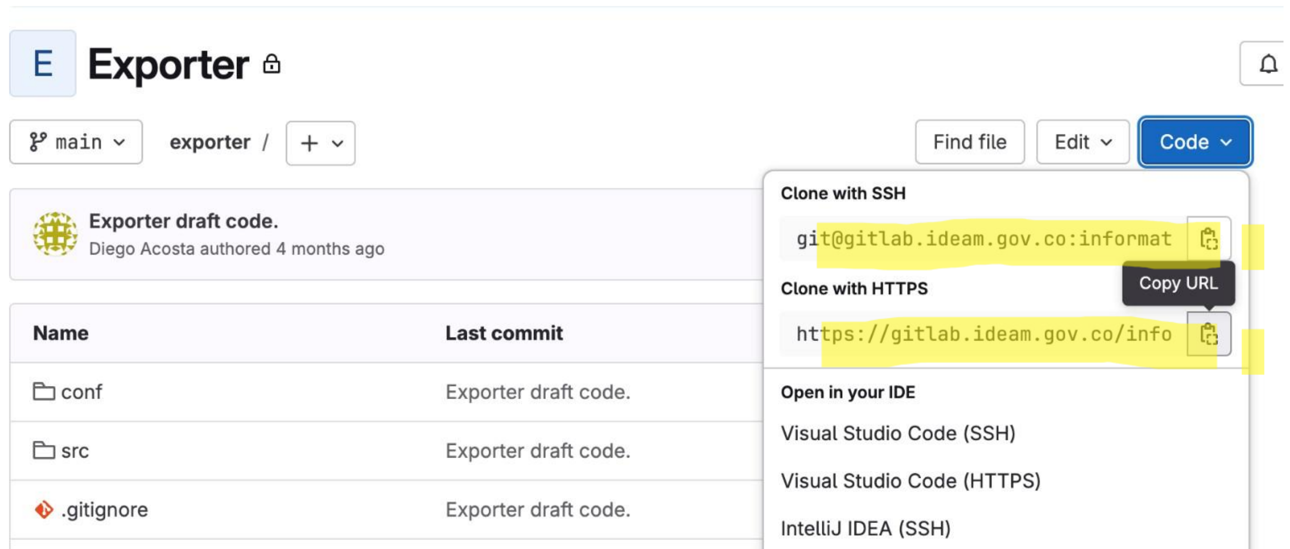

Imagen 11 Clonación de código

Una vez finalizada la clonación, encontrará en su equipo una copia local del repositorio, donde podrá ver los archivos del proyecto y comenzar a trabajar en ellos.

11.4.2. GIT INIT

Creación de un Nuevo Repositorio

Si desea iniciar un proyecto desde cero sin clonar un repositorio existente, puede crear un nuevo repositorio local utilizando el comando **git init**.

Pasos para inicializar un repositorio Git:

- **Ubíquese en la carpeta** donde desea crear el repositorio. Puede navegar hasta la ubicación deseada con el siguiente comando:
  - cd carpeta-mi-proyecto
- **Ejecute el comando**
  - git init:
- Git creará un repositorio vacío en la carpeta indicada, permitiéndole comenzar a realizar seguimiento a los archivos del proyecto.
- **Importante:** Al crear un repositorio con git init, este no estará vinculado a ningún repositorio remoto. Para conectarlo posteriormente a un servidor GitLab, deberá configurar la conexión manualmente utilizando el comando git remote add origin URL-del-repositorio.

12. COMANDOS PARA LA GESTIÓN DE REPOSITORIOS REMOTOS

A continuación, se presentan los comandos esenciales para gestionar la conexión entre un repositorio **Git local** y un **repositorio remoto**, permitiendo sincronizar cambios con plataformas como **GitLab** o **GitHub**.

- **Conectar el Repositorio Local con un Repositorio Remoto**

  Antes de comenzar a trabajar con un repositorio remoto, es necesario establecer una conexión con la dirección del repositorio en la plataforma correspondiente. Para ello, utilice el siguiente comando:

git remote add origin https://github.com/example/my-project.git

Nota: La URL debe ser reemplazada por la dirección del repositorio remoto real.

- **Configurar Conexión Mediante SSH**

Para evitar tener que ingresar manualmente las credenciales (usuario y contraseña) cada vez que se realice una acción sobre el repositorio remoto, se recomienda configurar una conexión segura mediante **SSH**. Para ello, agregue la URL SSH del repositorio con el siguiente comando:

git remote add origin git@github.com:example/my-project.git

Importante: Para utilizar SSH, debe generar y configurar una **clave pública y privada** en su equipo y agregarla en la configuración de la cuenta en GitLab.

- **Ver la URL del Repositorio Remoto**

Para verificar la dirección del repositorio remoto con el que está conectado el repositorio local, use el siguiente comando:

git remote -v

Este comando mostrará una lista con los enlaces de **fetch** (descarga) y **push** (envío) del repositorio remoto.

- **Modificar la URL del Repositorio Remoto**

Si por algún motivo necesita cambiar la dirección del repositorio remoto (por ejemplo, para cambiar de HTTPS a SSH), puede actualizar la URL con el siguiente comando:

git remote set-url origin git@github.com:example/my-project.git

Este comando reemplazará la URL anterior con la nueva especificada.

Recomendación: Si está trabajando en un entorno corporativo, asegúrese de utilizar el protocolo recomendado por la entidad (HTTPS o SSH) para conectar su repositorio local con GitLab.

13. GIT STATUS Y ESTADOS DE LOS ARCHIVOS

El comando **git status** permite conocer el estado actual del repositorio en la máquina local. Este comando muestra información sobre:

- Archivos que han cambiado.
- Archivos listos para ser confirmados (**staged**).
- Archivos no rastreados por Git (**untracked**).
- Sincronización con el repositorio remoto.

13.1. GIT STATUS

Cuando ejecutamos **git status**, el resultado dependerá del estado actual de los archivos en el repositorio. A continuación, se describen los posibles estados:

- **Sin cambios pendientes:**

Si no hay modificaciones en los archivos del repositorio, Git mostrará un mensaje indicando que **no hay cambios para confirmar**.

```
On branch main
Your branch is up to date with 'origin/main'.
nothing to commit, working tree clean
```

Imagen 12 Estado de cambios pendientes

- **Archivos modificados pero no añadidos al área de preparación:**

Si se han editado archivos pero no se han agregado al área de preparación, git status mostrará los archivos como **modificados** (**Modified**).

```
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified: archivo.txt
```

Imagen 13 Estado de ediciones

- **Archivos listos para confirmar (staged):**

Si los archivos han sido agregados al área de preparación con git add, estarán **listos para confirmarse** en el repositorio local (**Staged**).

```
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   nuevo_archivo.txt
```

Imagen 14 Estado de archivos para confirmar

- **Archivos no rastreados:**

Si se han creado nuevos archivos que Git aún no ha agregado al control de versiones, estos aparecerán como **Untracked** (No rastreados). Para incluirlos en Git, primero se debe ejecutar git add.

```
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        nuevo_archivo.txt
```

Imagen 15 Estado de archivos no rastreados

13.2. ESTADOS DE LOS ARCHIVOS EN GIT Y SU FLUJO

Entre el repositorio local de Git y el repositorio remoto en **GitLab**, los archivos pueden pasar por varios estados dentro del flujo de trabajo de Git:

- **Untracked (No rastreado):**
  - El archivo es nuevo y Git aún no lo está siguiendo.

  - Para rastrear este archivo, es necesario agregarlo con git add.
- **Modified (Modificado):**
  - El archivo ya existe en el repositorio, pero ha sido editado desde la última confirmación.
  - Git detecta los cambios, pero aún no han sido preparados para confirmación.
- **Staged (En área de preparación):**
  - El archivo ha sido agregado al área de preparación con git add.
  - Está listo para ser confirmado con git commit.
- **Committed (Confirmado):**
  - Los cambios han sido guardados en el historial local del repositorio con git commit.
  - Aún no han sido enviados al repositorio remoto.
- **Pushed (Sincronizado con GitLab):**
  - Los cambios han sido enviados al repositorio remoto en **GitLab** con git push.

La siguiente gráfica ilustra el flujo de estados de los archivos en Git:

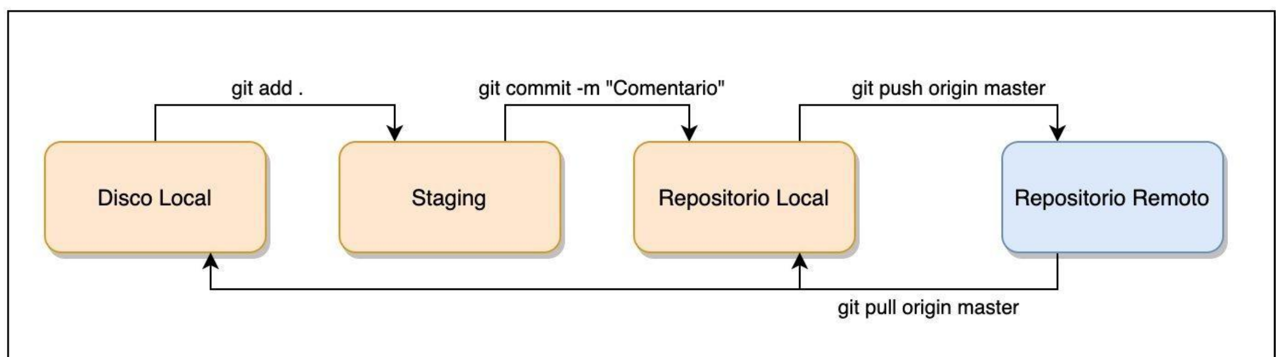

Imagen 16 Flujo de estados

Este flujo permite gestionar el versionamiento de los archivos de manera organizada y segura, asegurando que los cambios sean rastreados y sincronizados correctamente con el repositorio remoto.

14. COMANDOS BASICOS PARA LA GESTIÓN DEL PROYECTO EN EL REPOSITORIO LOCAL

Para gestionar los archivos dentro del repositorio local de Git, se utilizan los siguientes comandos básicos. Estos permiten agregar, confirmar y preparar cambios antes de enviarlos al repositorio remoto.

14.1. GIT ADD AGREGAR ARCHIVOS AL ÁREA DE PREPARACIÓN

El comando **git add** indica a Git qué archivos deben ser rastreados y enviados al área de preparación (**Staging**), antes de ser confirmados en el repositorio local.

Agregar archivos específicos: Si se desea agregar archivos específicos al área de preparación, se debe ejecutar:

git add index.html index.css index.js

Agregar todos los archivos modificados en el directorio actual: Para agregar todos los archivos que han sido modificados dentro del directorio de trabajo, se puede utilizar:

git add .

14.2. GIT COMMIT CONFIRMAR LOS CAMBIOS

Una vez que los archivos han sido agregados al área de preparación, se deben confirmar para guardarlos en el historial de Git.

Realizar una confirmación con un mensaje descriptivo: El comando git commit guarda los cambios en el repositorio local junto con un mensaje que describe los cambios realizados:

git commit -m "este es un mensaje descriptivo del cambio"

15. SINCRONIZACIÓN CON REPOSITORIO REMOTO

Una vez que los cambios en el repositorio local han sido confirmados (**committed**), el siguiente paso es subirlos al repositorio remoto en GitLab.

Para hacerlo, es fundamental comprender **hacia qué rama remota se enviarán los cambios**.

15.1. ESTRUCTURA DEL REPOSITORIO EN GIT

Git organiza los cambios en una estructura similar a un árbol:

- **Commits**: Son los nodos del historial de cambios del proyecto.
- **Ramas (Branches)**: Son líneas de desarrollo independientes que pueden bifurcarse y fusionarse.
- **HEAD**: Es un puntero especial que indica en qué commit y en qué rama se está trabajando actualmente.

15.2. VERIFICAR LA RAMA EN LA QUE SE ESTA TRABAJANDO

Antes de sincronizar el código con el repositorio remoto, es recomendable verificar en qué rama se encuentra el trabajo actual.

Para comprobar la rama activa, se usa: git branch

Si se desea visualizar más detalles, incluyendo la relación con el repositorio remoto, se puede ejecutar: git status

15.3. GIT PUSH ENVIAR LOS CAMBIOS AL REPOSITORIO REMOTO

Para sincronizar los cambios con la rama remota correspondiente, se utiliza el siguiente comando: git push origin nombre-de-la-rama

Ejemplo:

> git push origin main

Nota:

- **origin** es el nombre del repositorio remoto por defecto.
- **main** es la rama a la que se subirán los cambios.

Si es la primera vez que se envían cambios a una nueva rama, se puede usar:

git push --set-upstream origin nombre-de-la-rama

Este comando establece la relación entre la rama local y la remota.

16. WORKFLOW DE GITFLOW

GitFlow es una estrategia de ramificación utilizada en entornos de desarrollo colaborativo para organizar el flujo de trabajo, facilitando la gestión de versiones, correcciones y nuevas funcionalidades.

16.1. VENTAJAS DE GITFLOW

- **Separación clara** entre desarrollo y producción mediante las ramas develop y main.
- **Facilita el trabajo en equipo** permitiendo que cada funcionalidad se desarrolle en una rama independiente (feature/*).
- **Soporta versiones y mantenimiento** con ramas release/* y hotfix/* para control de versiones y corrección de errores.
- **Estructura bien definida** y de propósito claro.
- **Compatible con integración continua (CI/CD)** para automatización de despliegues y pruebas.

16.2. RAMAS PRINCIPALES EN GITFLOW

- **main (o master) – Rama de Producción**
  - Contiene el código estable y listo para producción.
  - No se modifica directamente sin revisión de cambios.
- **develop – Rama de Desarrollo**
  - Contiene el código en fase de pruebas antes de ser liberado a producción.
  - Actúa como rama intermediaria entre feature y main.

16.3. RAMAS COMMUNMENTE USADAS EN GITFLOW

- **feature/* – Ramas de Funcionalidad**
  - Se crean para desarrollar nuevas funcionalidades sin afectar main o develop.
  - Una vez terminadas, se fusionan en develop.
  - Generalmente, se eliminan tras la integración.

  Ejemplo de creación de una rama de funcionalidad:

  ```
  git checkout -b feature/nueva-funcionalidad develop
  ```

- **hotfix/* – Ramas de Corrección Rápida**
  - Se crean para solucionar errores críticos en producción.
  - Se originan desde main, y una vez solucionado el problema, se fusionan tanto en main como en develop.

  Ejemplo de corrección rápida:

  ```
  git checkout -b hotfix/correccion-bug main
  ```

- **release/* – Ramas de Preparación para Producción**
  - Se utilizan para estabilizar una versión antes de fusionarla con main.

- Permiten realizar pruebas y correcciones antes del lanzamiento.

  Ejemplo de creación de una rama de lanzamiento:

  git checkout -b release/v1.0 develop

16.4. RAMAS AUXILIARES

Aunque en **GitFlow** se sugiere por experiencia el uso de ramas determinadas la estrategia puede variar dependiendo de las necesidades de cada equipo de desarrollo, de la estrategia se espera que permita suplir las necesidades del proyecto y del equipo de trabajo, sin embargo, es importante que también se tengan en cuenta situaciones como por ejemplo el crecimiento desmedido del repositorio a la hora de mantener branches ya integrados.

Por lo generar y para las todas las ramas, las sugeridas, las principales y otras que puedan ser creadas se recomienda seguir una nomenclatura común en todo el proyecto como mínimo, a la hora de nombrar las ramas es importante aplicar esta nomenclatura, por ejemplo: el uso de minusculas en el nombre de la rama, separar las palabras que componen la rama con guiones al piso o guiones normales. (ejemplo: feature/nuevo-cambio), igualmente si la rama corresponde a una ajuste o solicitud realizada por herramientas de atención o gestión como lo podrían ser la mesa de servicio o jira, asana, redmine etc., puede incluirse el identificador de por ejemplo el número de ticket o el identificador de la historia de usuario dentro del nombre de la rama, (ejemplo: fix/4321-correccion-acceso-usuarios)

17. FLUJO DE TRABAJO DE GITFLOW

- Desarrolladores crean una rama feature/* para trabajar en una nueva funcionalidad.
- Una vez terminada, la integran en develop para pruebas y validaciones.

- Cuando la versión está lista, se crea una rama release/* para revisión final.
- Tras pasar las pruebas, la **release/*** se fusiona en **main** y en **develop.**
- Si surge un error crítico en producción, se crea una rama hotfix/* desde main para su corrección inmediata.

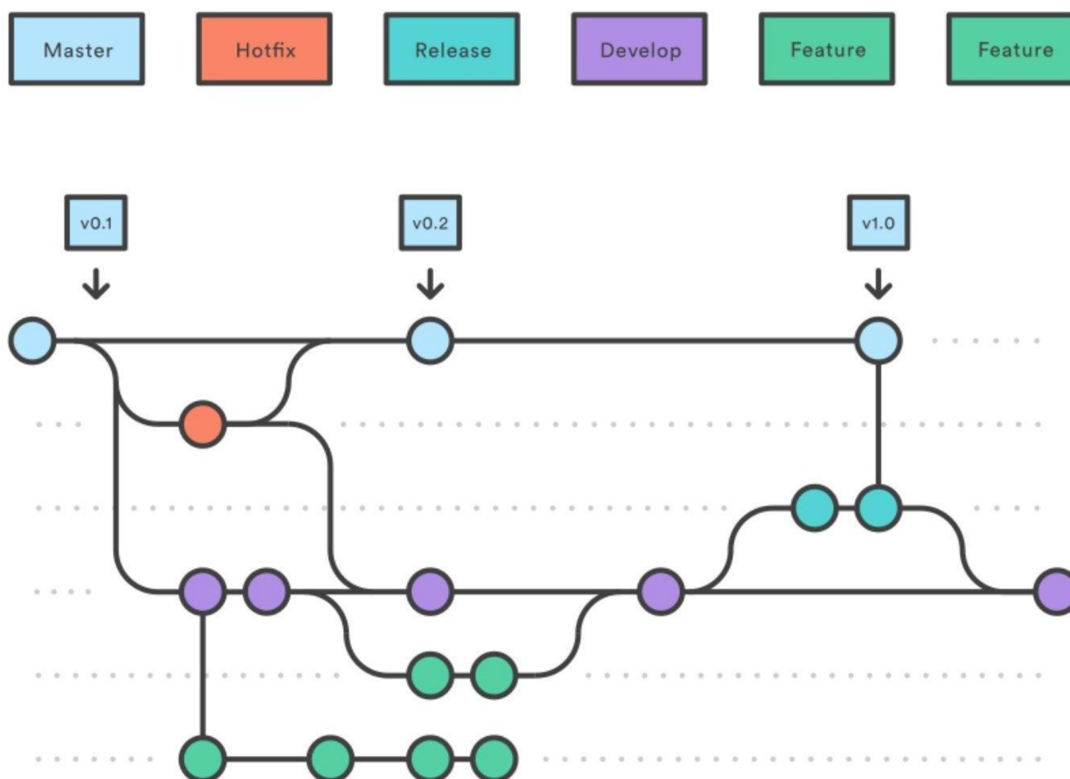

Imagen 17 Flujo de trabajo GitFlow

18. COMANDOS PARA LA GESTIÓN DEL WORKFLOW DE GIT

18.1. GIT PUSH

Sube los cambios desde el repositorio local al **remoto** después de haber hecho commit.

> git push origin <RamaRemota>

18.2. GIT PULL

Descarga y fusiona los cambios del repositorio remoto con el repositorio local.

> git pull origin <RamaRemota>

18.3. GIT CHECKOUT

Cambia a una rama específica o a un commit en particular.

> git checkout <branch>

> git checkout <commit>

19. COMANDOS PARA LA GESTIÓN DE RAMAS (BRANCHES)

19.1. GIT BRANCH

- Crear una nueva rama:
  - git branch <my_branch>

- Listar todas las ramas del proyecto:
  - git branch

  (La rama actual se resaltará en verde o con un * dependiendo del terminal.)

- Cambiar la rama de trabajo
  - git ckeckout <my_branch>

- Fusionar cambios de la rama 'my_branch' dentro de la rama 'master'
    - git ckeckout master
    - git merge <my_branch>

- Descargar los últimos cambios de master e incluirlos en la rama 'my_branch'
    - git ckeckout <my_branch>
    - git merge master

20. ESTRATEGIA BRANCHING RECOMENDADA

Si bien **GitFlow** es una estrategia ideal para proyectos grandes con múltiples desarrolladores, en proyectos mucho más pequeños o con menos cambios, **se recomienda como mínimo el uso de las ramas main y develop**.

Dependiendo del tamaño del proyecto hay ramas que se pueden considerar a la hora de trabajar, por existen enfoques en los que se crear ramas **fix /*** para encargarse de los ajustes a los errores encontrados sobre ambientes de desarrollo.

También en proyectos grandes que cuentan con cientos de branches se hace obvio la necesidad de suprimir las ramas usadas por los desarrolladores una vez se realiza el merge o pull request, esto con la finalidad de e vitar el crecimiento descontrolado de las ramas del repositorio.

21. PULL REQUEST (PR) Y REVISION DE CODIGO (CODE REVIEW)

El proceso de solicitud de fusión de ramas, conocido como Merge Request (MR) en GitLab o Pull Request (PR) en GitHub y Bitbucket, es una práctica esencial en los flujos de trabajo colaborativos con Git. Este mecanismo permite garantizar

la calidad del código, reducir errores antes de su despliegue en producción y mantener un historial de cambios organizado y comprensible.

En los sistemas de control de versiones, los MR y PR son solicitudes que permiten integrar cambios desde una rama de desarrollo hacia otra, generalmente develop o main. Antes de completar la fusión, se realiza una revisión de código (Code Review), un proceso estructurado en el que otros desarrolladores evalúan las modificaciones para asegurar que cumplen con los estándares de calidad, buenas prácticas y requisitos del proyecto.

21.1 ¿QUÉ ES UN MERGE REQUEST O PULL REQUEST?

Un Merge Request (MR) en GitLab o un Pull Request (PR) en GitHub y Bitbucket es una solicitud formal que un desarrollador crea para fusionar los cambios de su rama con otra dentro del repositorio del proyecto.

Para ello, el desarrollador debe:

- Subir los cambios al repositorio remoto en una rama específica.
- Abrir un Merge Request (MR) en GitLab:
  - Acceder al repositorio en GitLab.
  - Ir a Merge Requests y hacer clic en New Merge Request.

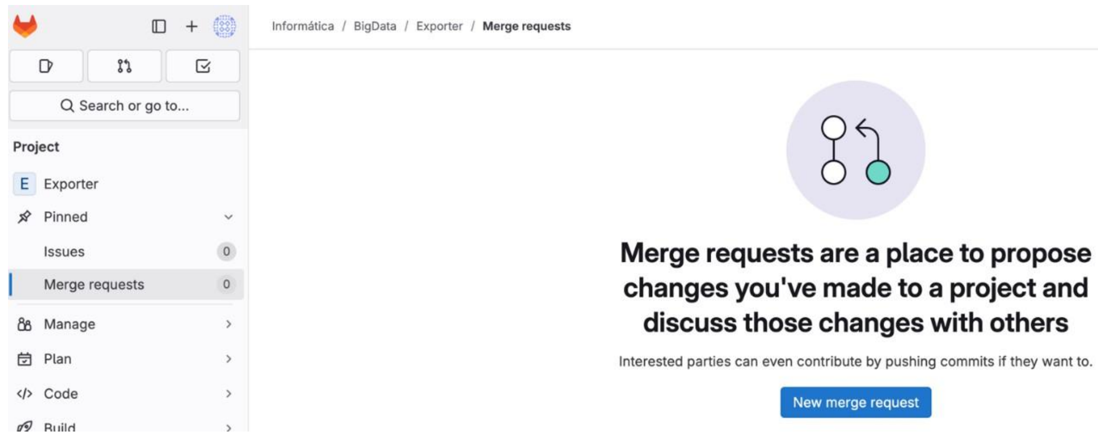

Imagen 18 Solicitud de cambios de las ramas

Una vez abierto un Merge Request (MR) en GitLab, es necesario seleccionar las ramas involucradas en la fusión:

- Rama de origen: corresponde a la rama donde el desarrollador ha realizado sus cambios. Generalmente, se trata de una rama específica creada para una nueva funcionalidad, corrección de errores o mejora.

- Rama de destino: es la rama en la que se integrarán los cambios. En la mayoría de los casos, esta suele ser develop si se trata de una fase de desarrollo, o main cuando se busca implementar cambios en producción.

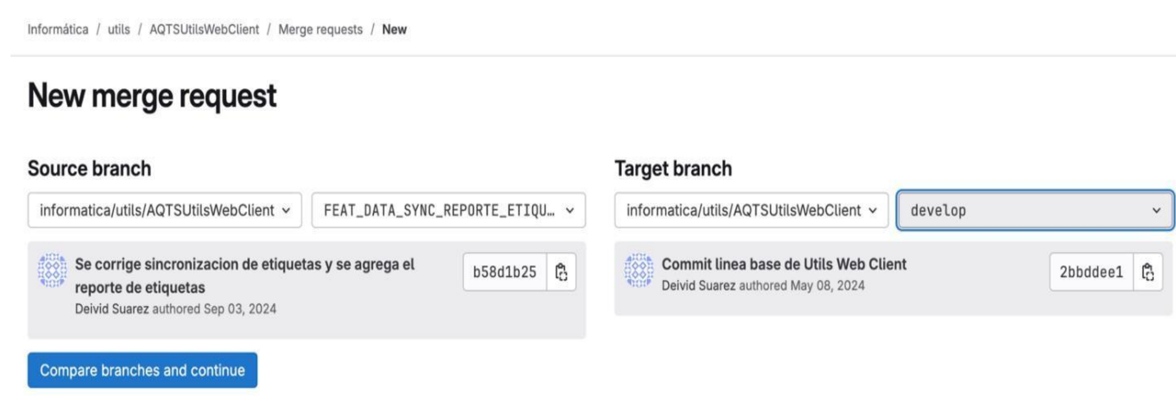

Imagen 19 Selección de ramas

Después de seleccionar las ramas y hacer clic en "Compare branches and continue", GitLab mostrará un formulario donde se debe proporcionar información clave sobre el Merge Request (MR).

Entre los campos a completar se incluyen:

- Título: una descripción breve y clara del propósito del MR.
- Descripción: detalles sobre los cambios realizados, incluyendo el contexto, propósito y cualquier información relevante para la revisión.
- Asignado: la persona responsable de dar seguimiento al MR.
- Revisor(es): desarrolladores encargados de revisar el código antes de su fusión.
- Milestone (hito): si el MR está relacionado con una entrega o fase específica del proyecto.
- Etiquetas: categorías o identificadores que faciliten la organización y el seguimiento del MR.

21.1 RECOMENDACIONES PARA LA APROBACIÓN DE UN PULL REQUEST

El flujo de aprobación indica de debe ser otro miembro del equipo o una persona diferente quien realice la aprobación de la integración que solicita el desarrollador, la persona que realice esta aprobación debe realizar el proceso de code review. En ocasiones por el tamaño de los equipos o las personas responsables no hay más usuarios técnicos a cargo diferentes al desarrollador para realizarlo, en estos casos conviene desde antes plantear una estrategia de revisión que involucre a terceros o en caso de no ser posible por diferentes situaciones implementar una plantilla y sistemas de análisis de código como sonar que mejoren la calidad de los commits realizados por el desarrollador.

New merge request

From FEAT_DATA_SYNC_REPORTE_ETIQUETAS into develop Change branches

Title (required)

Feat data sync reporte etiquetas

Mark as draft
Drafts cannot be merged until marked ready.

Description

Preview

Describe the goal of the changes and what reviewers should be aware of.

Switch to rich text editing

Add description templates to help your contributors to communicate effectively!

Assignee

Unassigned — Assign to me

Reviewer

Unassigned

Milestone

Select milestone

Labels

Select label

Merge can start

Anytime
Requires that merge checks pass.

Merge options

- [x] Delete source branch when merge request is accepted.
- [ ] Squash commits when merge request is accepted.

Create merge request — Cancel

Imagen 20 Formulario solicitud de fusión

Una vez completados estos campos, se debe hacer clic en "Crear Merge Request" para enviar la solicitud y dar inicio a la revisión de código (Code Review). Durante esta etapa, los revisores podrán analizar los cambios, proponer mejoras y aprobar la fusión cuando el código cumpla con los estándares establecidos.

22. REVISIÓN DE CODIGO (CODE REVIEW)

La Revisión de Código (Code Review) es el proceso de examinar el código fuente de un desarrollador antes de fusionarlo en la rama principal (develop o main). Su objetivo es garantizar la calidad del código, mejorar su mantenimiento y prevenir errores antes de su despliegue.

Este proceso puede ser realizado por otros desarrolladores del equipo o mediante herramientas automatizadas que analizan el código en busca de problemas.

22.1. BENEFICIOS DE LA REVISIÓN DE CÓDIGO

- Detectar errores antes de llegar a producción.
- Mejorar la calidad y mantenibilidad del código.
- Fomentar buenas prácticas de programación.
- Compartir conocimiento entre los desarrolladores.
- Reducir costos al evitar errores en etapas avanzadas.

22.2. CÓMO HACER UN CODE REVIEW

Para hacer una revisión de código efectiva, sigue este flujo de trabajo:

Paso	Descripción
Entender el contexto	Leer la descripción del PR/MR antes de revisar.
Revisar la implementación	Verificar la lógica, eficiencia y seguridad del código.
Comprobar estilo y convenciones	Asegurar que el código sigue las guías de estilo.
Verificar pruebas	Asegurar que hay pruebas suficientes para los cambios.

Probar manualmente	Ejecutar y validar el código en un entorno real.
Proporcionar feedback constructivo	Hacer comentarios claros, útiles y respetuosos

Tabla 1 flujo de trabajo

22.3. ¿QUÉ REVISAR?

- Se debe asegurar que el código cumpla estándares mínimos (Por ejemplo, forma de nombrar las variables, mensajes de error, mejores prácticas, etc.)
- Encontrar bugs (Tener más de un par de ojos puede ayudar a prevenir bugs, memory leaks, problemas de seguridad, etc.).
- Compartir conocimiento (Los desarrolladores pueden compartir conocimiento con otros lo que los ayuda a mejorar el código y aprender nuevos temas).
- Verificar que el código sea comprensible.
- Verificar que el código haga lo que se supone debe hacer.
- Diseño, ¿el código propuesto se integrará bien con el resto de los componentes?, ¿está en el componente correcto?
- ¿Los test diseñados cubren el código realizado?
- ¿Los comentarios son entendibles y claros?
- ¿Es el código más complejo de lo que necesita ser?

https://google.github.io/eng-practices/review/reviewer/

23. RECOMENDACIONES EN LA GESTIÓN DE COMMITS

- Múltiples commits pequeños de un feature en lugar de un solo commit gigantesco

- Apoyarse en lideres de desarrollo o compañeros para realizar la revisión de código (code review) y así mejorar la calidad del código y realizar detección temprana de errores.
- Use los verbos imperativos en la descripción del cambio realizado, por ejemplo, add para archivos agregados, change para cambios en archivos existentes, remove para archivos que se eliminan.
- Usar prefijos en los commits para diferenciar los tipos de commit como fix, feat, docs, etc. Puede ser útil también agregar la identificación del requerimiento o historia de usuario.
- No use punto final ni puntos suspensivos en los mensajes de commit.
- Puede apoyarse en herramientas para realizar los commit, usar la consola o simplemente usar la herramienta que suele venir integrada en su IDE.

24. COMANDOS PARA DESHACER CAMBIOS

Los comandos usados para deshacer cambios deben usarse con cuidado y saber que se está realizando, ya que puedes perder avances realizados, antes de ejecutar estos comandos hay que asegurarse de revisar la documentación y comprender de forma clara el cambio que se realizara sobre el repositorio y las ramas.

24.1. GIT RESET

Reset se utiliza para deshacer cambios en distintas áreas del repositorio moviendo el puntero de la rama actual (HEAD) a otro commit. Se aplican cuando ya el commit ha sido creado, este comando nos permite regresar a una versión o commit anterior sin la posibilidad de RECUPERAR.

Regresa el repositorio al commit indicado, conservando los cambios en staging y disco duro permitiendo aplicarlos luego por medio de un commit

> git reset --soft <id commit>

Regresa el repositorio al commit indicado, eliminando los cambios realizados posterior a este, tanto en staging como en disco duro. SIGNIFICA PELIGRO.

> git reset --hard <id commit>

Se saca el archivo de staging pero los cambios efectuados siguen estando en el archivo local.

> git reset HEAD <nombre archivo>

24.1.1. ¿PARA QUÉ SE USA?

El comando git reset se utiliza en Git para deshacer cambios en distintas áreas del repositorio:

- El historial de commits
- El área de preparación (staging area)
- El directorio de trabajo (working directory)

24.2. GIT REVERT

Revert se usa para registrar nuevos commits y revertir el efecto de commits anteriores, crea un nuevo commit que deshace los cambios de un commit

especifico y crea (si los cambios lo permiten) un nuevo commit, conservando los cambios del historial realizados.

> git revert abc1234

Esto crea un nuevo commit que deshace lo que hizo el commit abc1234, esta es la forma segura y recomendada de deshacer cambios en proyectos compartidos. Puede ser usado para deshacer los últimos 3 commits, uno por uno con un nuevo commit para cada uno, por ejemplo con el siguiente comando:

> git revert HEAD~2..HEAD

24.2.1. ¿PARA QUÉ SE USA?

- Deshacer un commit específico, pero sin borrar el historial.

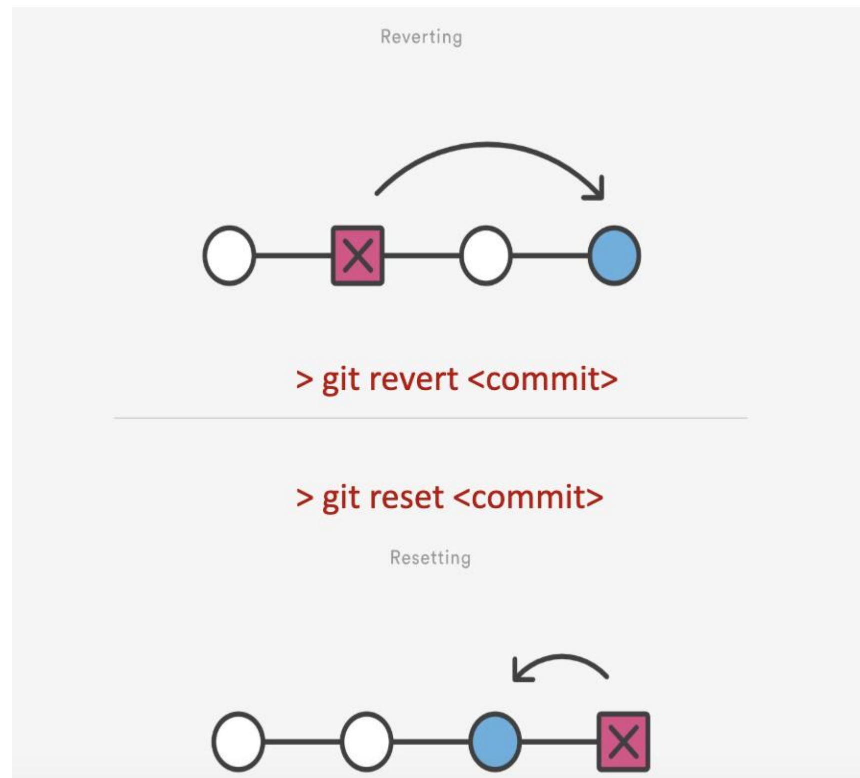

Imagen 21 Deshacer o revertir cambios

24.3. GIT RESTORE

El comando git restore se introdujo en Git 2.23 para facilitar y simplificar tareas comunes relacionadas con deshacer cambios en los archivos del proyecto.

Es una alternativa moderna a algunos usos de git checkout y git reset, y su objetivo es hacer más claro qué estás restaurando y desde dónde.

24.3.1. ¿PARA QUÉ SE USA?

- Deshacer cambios en archivos del working directory (archivos que editaste pero aún no agregaste al staging).

- Quitar archivos del área de staging (los que agregaste con git add, pero aún no hiciste commit).

Ejemplo: Si cambiaste un archivo, pero aún no hiciste commit y quieres devolverlo a su estado anterior

> git restore archivo.txt

24.4. COMANDOS DE GIT RESET, REVERT, RESTORE

Comando	**¿Qué hace?**
git reset --soft	Mueve el puntero de la rama al commit anterior, conserva los cambios en staging
git reset --mixed	Igual que --soft, pero quita los cambios del staging
git reset --hard	Vuelve a un commit anterior y borra todos los cambios no confirmados
git revert <commit>	Crea un nuevo commit que invierte los cambios de otro commit
git restore <archivo>	Restaura un archivo al estado del último commit (descarta cambios locales)
git restore --staged	Quita archivos del staging area (deshace git add)

Tabla 2 Comandos

25. COMANDOS DE GIT STASH

El comando git stash almacena temporalmente (o guarda en un stash) los cambios que hayas efectuado en el código en el que estás trabajando para que puedas trabajar en otra cosa y, más tarde, regresar y volver a aplicar los cambios más tarde. Guardar los cambios en stashes resulta práctico si tienes que cambiar

rápidamente de contexto y ponerte con otra cosa, pero estás en medio de un cambio en el código y no lo tienes todo listo para confirmar los cambios.

Si se están haciendo cambios en una rama que no es la correcta o se necesita cambiar de rama y no perder los cambios que se están haciendo y esos cambios aun no merecen hacer el commit se puede usar stage que guarda los cambios de forma temporal para luego usarlos en una rama o crear una rama nueva con estos cambios.

crear una rama nueva con estos cambios.

```
> git stash
```

Muestra los stash guardados.

```
> git stash list
```

Colocar el trabajo del stash en la rama actual.

```
> git stash pop
```

Crea una nueva rama con los cambios que están en el stash.

```
> git stash branch my-new-branch
```

Borrar el trabajo guardado en el stash.

```
> git stash drop
```

25.1. ¿POR QUÉ USAR GIT STASH?

Imagina esto:

- Estás trabajando en una funcionalidad.

- De repente, necesitas cambiar de rama para atender una urgencia (como corregir un bug).
- Pero no puedes cambiar de rama porque tienes cambios sin confirmar.

En lugar de hacer un commit incompleto o perder tus cambios, puedes hacer:

> git stash

- Git guarda esos cambios
- Tu árbol de trabajo queda limpio
- Puedes cambiar de rama
- Y luego recuperar los cambios con git stash pop

25.2. COMANDOS ÚTILES DE GIT STASH

Comando	¿Qué hace?
git stash	Guarda cambios no confirmados y limpia el working directory
git stash push -m "mensaje"	Guarda con un nombre personalizado
git stash list	Lista todos los stashes guardados
git stash show	Muestra los archivos afectados por el último stash
git stash show -p	Muestra el diff del stash
git stash pop	Restaura los cambios y elimina el stash
git stash apply	Restaura los cambios pero **no elimina** el stash
git stash drop	Elimina un stash específico
git stash clear	Borra **todos** los stashes guardados

Tabla 3 Comandos útiles

26. RECOMENDACIONES PARA EVITAR MALAS PRÁCTICAS EN GIT

Git es una herramienta potente para la manipulación y gestión del historial de código fuente por sus grandes capacidades ofrece muchas herramientas que deben usarse con inteligencia o buen conocimiento, si no se convierten en malas prácticas, evitar el uso de malas prácticas es esencial para no llegar a situaciones donde se vea comprometida la integridad del repositorio o del código almacenado, así mismo existen comportamientos que debemos evitar al usar las herramientas que nos ofrece GIT, como también podemos pensar en complementar estas herramientas software adicional que mejoren la calidad y la gestión de nuestro código.

26.1. COMANDOS PELIGROSOS

Comando	¿Qué hace?	¿Recomendación?
git reset --hard	Borra commits y cambios sin recuperación	Solo en local y con precaución
git push --force	Reescribe el historial remoto	Usar solo si trabajas solo o lo acuerdas con el equipo
git cherry-pick	Trae commits individuales de otras ramas	Úsalo con cuidado para no duplicar historial
git stash	Guarda cambios temporalmente	Úsalo, pero no lo olvides ahí por semanas

Tabla 4 Comandos peligrosos

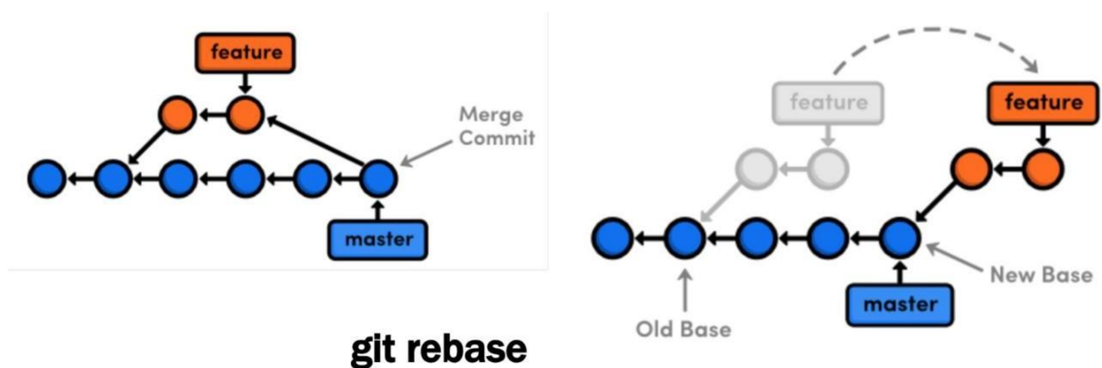

Imagen 22 Git rebase

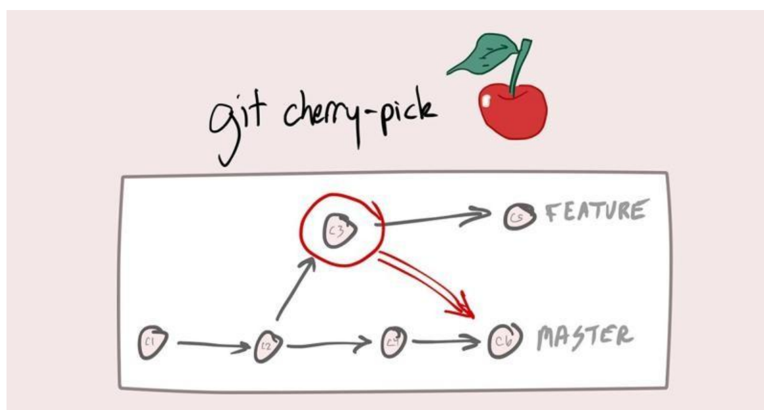

Imagen 23 Git cherry-pick

26.2. COMMITS

Buenas prácticas	Malas prácticas
Haz commits pequeños y enfocados	Hacer un solo commit enorme con muchos cambios mezclados
Escribe mensajes claros y descriptivos	Usar mensajes genéricos como “arreglos”, “cambios”, “final”
Sigue una convención (feat:, fix: etc.)	No explicar qué hace el commit o dejarlo vacío

Realiza commits frecuentemente	Esperar hasta el final y hacer un solo mega commit
Incluye pruebas si aplica	Subir código sin verificar que funciona o sin test

Tabla 5 Commits

26.3. USO DE RAMAS

Buenas prácticas	Malas prácticas
Usa ramas para nuevas funcionalidades	Trabajar directamente en main o develop
Nombra ramas de forma clara (feature/, bugfix/)	Nombres confusos o sin contexto (ramanueva)
Mantén las ramas actualizadas con main	Hacer merges grandes y difíciles después de semanas
Elimina ramas cuando ya no se usen	Dejar ramas viejas ocupando espacio innecesario

Tabla 6 Uso de Ramas

26.4. MERGING Y MERGE REQUESTS

Buenas prácticas	Malas prácticas
Usa Merge Request (MR) para todo	Hacer git merge directo en main sin revisión
Revisa el código de tus compañeros (Code Review)	Aceptar MR sin leer o sin probar los cambios
Resuelve conflictos antes de hacer merge	Dejar conflictos sin resolver o forzar el merge
Usa squash si hay muchos commits sucios	Merge de commits innecesarios o poco claros

Escribe una descripción clara en el MR	Enviar MR vacíos o sin explicar los cambios

Tabla 7 Merging y Merge

26.5. LIMPIEZA Y ORGANIZACIÓN DEL REPOSITORIO

Buenas prácticas	Malas prácticas
Elimina ramas locales/remotas que ya no se usan	Acumular ramas viejas sin propósito
Evita subir archivos grandes o temporales	Subir .log, .tmp, o carpetas node_modules/
Usa .gitignore para evitar archivos innecesarios	Subir archivos personales, binarios o de build
Usa tags para marcar versiones estables	No documentar ni versionar releases

Tabla 8 Limpieza y Organización Repositorios

27. CREACIÓN DEL ARCHIVO .GITIGNORE

El archivo .gitignore se usa para decirle a Git qué archivos o carpetas debe ignorar, es decir, no incluirlos en el control de versiones.
Esto es muy útil para evitar subir archivos temporales, personales, sensibles o innecesarios al repositorio.

Es importante porque:

- Mantiene el repositorio limpio y profesional
- Evita subir archivos grandes o que cambian constantemente (como logs o compilados)
- Protege archivos sensibles (como claves o configuraciones locales)
- Ahorra espacio y mejora el rendimiento del repositorio

Algunos tipos de archivos que suelen ser ignorados:

Tipo de archivo	Ejemplo
Archivos de sistema	.DS_Store, Thumbs.db
Archivos de entorno local	.env, .vscode/settings.json
Archivos temporales	*.log, *.tmp, ~archivo.txt
Archivos compilados o binarios	*.class, *.exe, dist/, build/
Dependencias	node_modules/, vendor/

Tabla 9 Tipos de archivos

27.1. ¿COMO CREARLO?

Puedes crearlo manualmente o por ejemplo usar alguna de las siguientes webs:

- GitHub mantiene una colección de .gitignore listos para diferentes tecnologías: https://github.com/github/gitignore

- Y también puedes usar este sitio: https://www.toptal.com/developers/gitignore

28. DOCUMENTOS RELACIONADOS EN EL SGI

GTI-P001 procedimiento de soporte de mesa de servicio ti

GTI-P005 procedimiento construcción o mantenimiento evolutivo de software

GTI-P003 procedimiento gestión de cambios

29. BIBLIOGRAFÍA

- Chacon, S., & Straub, B. (2014). *Pro Git* (2.ª ed.). Apress. https://git-scm.com/book/en/v2
- Git. (s.f.). *Git documentation*. https://git-scm.com/doc
- GitLab. (s.f.). *GitLab documentation*. https://docs.gitlab.com
- GitHub, Inc. (s.f.). *gitignore templates*. https://github.com/github/gitignore
- Toptal. (s.f.). *Gitignore generator*. https://www.toptal.com/developers/gitignore
- Google. (s.f.). *Google engineering practices: Code review developer guide*. https://google.github.io/eng-practices/review/reviewer/
- Congreso de Colombia. (2012). *Ley 1581 de 2012: Por la cual se dictan disposiciones generales para la protección de datos personales*. https://www.funcionpublica.gov.co/documents/418537/1135246/Ley_1581_2012.pdf
- Ministerio de Tecnologías de la Información y las Comunicaciones – MinTIC. (s.f.). *Marco de referencia de arquitectura empresarial del Estado colombiano*. https://www.mintic.gov.co/portal/inicio/Micrositios/Marco-de-referencia/
- Ministerio de Tecnologías de la Información y las Comunicaciones – MinTIC. (2022). *Resolución 746 de 2022: Por la cual se fortalecen los lineamientos del Modelo de Seguridad y Privacidad de la Información*. https://www.mintic.gov.co/

30. CONTROL DE CAMBIOS

VERSIÓN	FECHA	DESCRIPCIÓN
1.0	30/04/2025	Documento inicial