Mujeres Digitales 2025

GIT / GITHUB



AHK Deutsch-Kolumbianische Industrie- und Handelskammer
Cámara de Industria y Comercio Colombo-Alemana

Fomentado por:
Ministerio Federal de Economía y Energía
en virtud de una decisión del Bundestag alemán

ORGANIZACIÓN KEY



Git/GitHub

1. Control de versiones con Git y GitHub pág 3
   - 1.1. Fundamentos del control de versiones
   - 1.2. ¿Qué es el control de versiones? pág 4
   - 1.3. Ventajas del control de versiones en proyectos de desarrollo web
   - 1.4. Diferencia entre control de versiones local y remoto pág 5
2. Instalación y configuración de Git pág 6
   - 2.1. Instalación en distintos sistemas operativos
   - 2.2. Configuración inicial (nombre de usuario, correo) pág 7
   - 2.3. Comandos básicos de configuración (git config) pág 8
3. Repositorios: creación y estructura
   - 3.1. ¿Qué es un repositorio Git? pág 9
   - 3.2. Inicializar un repositorio local (git init)
   - 3.3. Clonar un repositorio remoto (git clone) pág 10
   - 3.4. Estructura típica de un proyecto en Git
4. Ciclo de trabajo con Git (local) pág 11
   - 4.1. git add, git commit, git status
   - 4.2. git log, git diff pág 13
   - 4.3. Buenas prácticas al hacer commits pág 14
   - 4.4. Deshacer cambios: git restore, git reset, git checkout pág 15
5. GitHub: control de versiones en la nube pág 18
   - 5.1. ¿Qué es GitHub y para qué se usa?
   - 5.2. Crear una cuenta y un repositorio en GitHub
   - 5.3. Subir un repositorio local a GitHub (git remote, git push) pág 19
   - 5.4. Sincronizar cambios: git pull, git fetch pág 20
6. Trabajo con ramas (branches) pág 21
   - 6.1. ¿Qué es una rama y por qué usarla?
   - 6.2. Crear, cambiar, fusionar y eliminar ramas (git branch, git checkout, git merge)
   - 6.3. Resolución de conflictos de merge pág 22
   - 6.4. Flujos comunes de trabajo: feature branch, main-dev pág 25
7. Colaboración en proyectos con GitHub pág 26
   - 7.1. Forks y Pull Requests



7.2. Revisiones de código y comentarios pág 27
7.3. Issues, etiquetas y gestión de tareas pág 28
7.4. Asignación de roles y permisos pág 29
8. Archivos esenciales en un proyecto
8.1. README.md: estructura y contenido profesional pág 30
8.2. .gitignore: qué es y cómo usarlo pág 33
8.3. Licencias de software: breve introducción pág 34
8.4. Archivos comunes en proyectos de desarrollo web pág 35
9. Buenas prácticas y flujos recomendados pág 38
9.1. Convenciones de commits y nombres de ramas
9.2. Flujos de trabajo colaborativos:
Git Flow vs. GitHub Flow pág 39
9.3. Estructura de carpetas y documentación clara pág 41
9.4. Automatización y control de archivos generados pág 42



1. Control de versiones con Git y GitHub

En el mundo del desarrollo web, como en cualquier disciplina basada en programación y trabajo colaborativo, los proyectos nunca se construyen de una sola vez ni sin errores. Una aplicación o un sitio web evoluciona constantemente: se crean nuevas funcionalidades, se ajusta el diseño, se corrigen bugs, se optimiza el rendimiento y se prueban diferentes enfoques. En ese proceso dinámico, es muy fácil perder el control sobre qué cambió, quién lo cambió, cuándo lo hizo o incluso romper algo que antes funcionaba correctamente.

Para enfrentar ese reto, Git se ha convertido en el estándar más utilizado a nivel mundial para el control de versiones, mientras que GitHub es la plataforma en la nube que facilita centralizar, compartir y colaborar en proyectos con otros desarrolladores.

Aprender a utilizar Git y GitHub no solo es recomendable, es esencial. Es el equivalente a tener un sistema que guarda el historial de tu proyecto, permite retroceder en el tiempo si algo falla y hace posible que un equipo de desarrolladores trabaje en conjunto sin generar caos.

1.1 Fundamentos del control de versiones

Imagina que estás creando una aplicación web o diseñando el frontend de un sitio. Cada día haces cambios: ajustas estilos en el CSS, agregas nuevas funcionalidades en JavaScript, optimizas el backend o corriges errores en el código. Pero un día, algo deja de funcionar como esperabas. O quizá quieres volver a una versión anterior que sabías que estaba bien. ¿Cómo retrocedes sin perder todo el trabajo reciente? ¿Cómo compartes tu proyecto con un equipo en el que varios desarrolladores trabajan al mismo tiempo sin pisarse los cambios?

Aquí es donde entra el control de versiones: una herramienta que te permite guardar el historial completo de tu proyecto, identificar quién hizo qué cambios y colaborar con otros de forma ordenada y segura.

Analogía:
Piensa en el control de versiones como un Google Docs para código. Cada cambio que haces queda registrado, puedes volver atrás si lo necesitas, y varios desarrolladores pueden trabajar en el mismo proyecto al mismo tiempo sin que reine el caos.



1.2 ¿Qué es el control de versiones?

El control de versiones (o *version control* en inglés) es un sistema que registra los cambios realizados a un archivo o conjunto de archivos a lo largo del tiempo.

Esto te permite:

- Ver qué cambió.
- Quién lo cambió.
- Cuándo lo cambió.
- Recuperar versiones anteriores si algo sale mal.

Ejemplo:
Supongamos que tienes este archivo llamado app.js:

```
// Versión inicial
const nombre = "Ana";
console.log("Hola " + nombre);
```

Días después, haces un cambio:

```
// Versión modificada
function saludar(nombre) {
  console.log(`Hola ${nombre}`);
}

saludar("Ana");
saludar("Pedro");
```

Con un sistema de control de versiones, puedes guardar ambas versiones, documentar por qué hiciste ese cambio, y volver fácilmente a cualquiera de las dos si es necesario.

1.3 Ventajas del control de versiones en proyectos de desarrollo web

El control de versiones es esencial en desarrollo web por varias razones:

- Historial completo del proyecto
  Puedes volver a una versión anterior del código. Por ejemplo, si un cambio en tu archivo app.js rompe la funcionalidad del sitio, puedes revisar qué se modificó y deshacerlo fácilmente.
- Trabajo colaborativo ordenado



Permite que varios desarrolladores trabajen al mismo tiempo en el mismo proyecto sin sobrescribir los cambios de otros. Git maneja esto mediante ramas (branches) y fusiones (merges), que exploraremos más adelante.

- Rastreabilidad de cambios
  Cada cambio en el código lleva un mensaje que explica qué se hizo y por qué. Esto es vital en proyectos web, donde una simple modificación en el CSS, HTML o JavaScript puede cambiar la experiencia del usuario.
- Reproducibilidad y consistencia
  En desarrollo web es común mover un proyecto desde un entorno local al servidor de producción. Con control de versiones, puedes asegurar que todos los entornos (desarrollo, pruebas y producción) usen exactamente el mismo código, evitando errores por versiones distintas.

Analogía:
Piensa en tu proyecto como un documento en progreso. El control de versiones te permite:

- Guardar cada capítulo por separado.
- Ver qué editor hizo qué corrección.
- Volver a una versión anterior si una edición no funciona.
- Comparar cómo era un párrafo antes y después.

1.4 Diferencia entre control de versiones local y remoto

Control de versiones local

Cuando usas Git de forma local, todo el historial del proyecto se guarda en tu computadora personal. Esto es útil si estás trabajando solo o aún no estás conectado a internet.

- Puedes seguir registrando versiones.
- Puedes crear ramas, hacer pruebas, deshacer errores.
- Todo permanece en tu máquina hasta que lo "subas" o compartas.

Control de versiones remoto

Cuando conectas tu repositorio local a una plataforma como GitHub, puedes:

- Guardar una copia en la nube.
- Trabajar en equipo desde distintas ubicaciones.
- Hacer revisiones de código, sugerencias y colaboraciones.
- Tener respaldo en caso de que pierdas tu computador.



Comparación

Aspecto	Local	Remoto (GitHub, GitLab, etc.)
Ubicación del repositorio	Tu computadora	Servidor en la nube
Accesibilidad	Solo tú lo ves	Puede ser público o compartido
Ideal para...	Probar cosas antes de subir	Trabajos colaborativos y respaldo

Ejemplo:

```
# Crear repositorio local
git init

# Luego puedes conectar tu repositorio a GitHub
git remote add origin https://github.com/tu_usuario/mi_proyecto.git
git push -u origin main
```

Así, tu repositorio local y el de la nube quedan sincronizados.

2. Instalación y configuración de Git

Antes de comenzar a trabajar con Git, necesitamos asegurarnos de que esté correctamente instalado y configurado en nuestro equipo. Esta fase inicial puede compararse con preparar una cocina antes de empezar a cocinar: necesitas los utensilios (Git instalado), debes presentarte (decir quién eres: tu nombre y correo) y elegir tu espacio de trabajo (el editor de texto donde escribirás tus instrucciones). Esta sección cubrirá todo lo necesario para dejar tu entorno listo para comenzar a usar Git en proyectos reales.

2.1 Instalación en distintos sistemas operativos

Cuando hablamos de instalar Git, estamos diciendo: “quiero esta herramienta lista para usar en mi máquina”. Dependiendo del sistema operativo que utilices (Windows, macOS o Linux), el proceso puede variar ligeramente, pero el resultado será el mismo: tendrás acceso a la consola de Git y podrás empezar a registrar tu trabajo.

Windows

Si estás en Windows, el proceso de instalación es muy amigable. Git proporciona un instalador gráfico que puedes descargar desde su página oficial.

- Ve al sitio oficial de Git: https://git-scm.com
- Haz clic en el botón "Download for Windows".



- Se descargará un instalador .exe. Ejecútalo y sigue estos pasos:
    - Acepta la licencia.
    - Selecciona el editor predeterminado (puedes elegir Visual Studio Code).
    - Deja las demás opciones por defecto, salvo que tengas necesidades específicas.
    - En la parte de "Adjusting your PATH environment", selecciona: "Git from the command line and also from 3rd-party software".
- Una vez instalado, abre la terminal de Git llamada Git Bash (se instala junto con Git).
- Verifica la instalación con: git –version
- Deberías ver algo como: git version 2.41.0.windows.1

macOS

En macOS, muchas veces Git ya viene instalado. Puedes verificarlo rápidamente abriendo tu aplicación de Terminal y escribiendo:

- Abre la aplicación Terminal (puedes buscarla con Spotlight).
- Escribe: git –version
- Si Git no está instalado, macOS te ofrecerá instalar las herramientas de desarrollo Xcode. Acepta, y macOS se encargará de instalar Git automáticamente o sigue las instrucciones.
- Si quieres instalar Git manualmente con Homebrew (si ya lo tienes): brew install git
- Verifica con: git –version

Linux (Debian/Ubuntu)

En Linux, especialmente distribuciones como Ubuntu, puedes instalar Git desde la terminal con este comando:

- Abre la terminal y ejecuta:
- sudo apt update
- sudo apt install git
- Verifica que esté instalado: git –version

2.2 Configuración inicial (nombre de usuario, correo)

Una vez que Git está instalado, hay algo que debemos hacer de inmediato: decirle quién eres. Esto puede sonar extraño, pero es muy importante. Git lleva un registro exacto de qué persona hizo cada cambio en un proyecto, y para eso necesita tu nombre y correo.



Configurar tu nombre de usuario
Abre tu terminal o Git Bash y escribe: *git config --global user.name "Tu Nombre"*

Configurar tu correo electrónico
Abre tu terminal o Git Bash y escribe:
git config --global user.email tucorreo@ejemplo.com

Esto le dice a Git: "cada vez que yo haga algo en este computador, quiero que me identifiques con este nombre y correo".

Importante: Usa el mismo correo que tienes registrado en GitHub para que tus contribuciones se reflejen correctamente en tu perfil.

2.3 Comandos básicos de configuración (git config)

La herramienta git config es como el panel de control de Git. Nos permite ver qué configuraciones hemos hecho, cambiarlas, o agregar nuevas.

Ver todas tus configuraciones
git config –list

Este comando te mostrará una lista con todo lo que Git sabe de ti hasta ahora, incluyendo tu nombre, correo, editor y otras opciones.

Cambiar algo que ya configuraste
¿Cometiste un error escribiendo tu correo? No pasa nada. Solo vuelve a usar el mismo comando con los nuevos datos:
git config --global user.email nuevo@correo.com

Diferencias entre --global, --local y --system
Git tiene distintos niveles de configuración:

- --global: Se aplica a todos los proyectos en tu computadora.
- --local: Solo se aplica al proyecto actual (cuando ya estás dentro de una carpeta con Git inicializado).
- --system: Se aplica a todos los usuarios del computador (rara vez se necesita).

3. Repositorios: creación y estructura

Hasta ahora, ya tienes Git instalado y configurado. Ahora viene el corazón del asunto: los repositorios. Un repositorio es donde Git guarda todo. Es como una caja mágica



donde no solo se almacenan tus archivos, sino también su historia completa: quién los cambió, cuándo, por qué, y cómo eran antes. No es solo una carpeta: es una carpeta con memoria.

En esta sección vamos a aprender qué es exactamente un repositorio Git, cómo crear uno desde cero, cómo conectarte a uno que ya existe en línea, y qué aspecto tiene por dentro. A partir de aquí, todo tu trabajo con Git girará en torno a estas "cajas de control de versiones".

3.1 ¿Qué es un repositorio Git?

Imagina que estás escribiendo un libro, y cada vez que haces un cambio —cambias un capítulo, corriges una coma, agregas una nota— alguien toma una foto de esa versión y la guarda con una etiqueta: *"Versión 1.2, corregida el 3 de mayo por Sofía"*. Eso, en esencia, es lo que hace un repositorio Git.

Un **repositorio Git** es un espacio de trabajo (una carpeta) que está bajo vigilancia de Git. Git vigila los archivos que le indiques, y cada vez que tú lo decidas, él guarda un registro de cómo estaban en ese momento. Eso se llama hacer un **commit**.

Hay dos tipos de repositorios que usaremos:

- **Local:** Vive en tu computador. Es donde escribes código, pruebas cosas, y haces commits.
- **Remoto:** Vive en la nube (por ejemplo, en GitHub). Es donde subes tu trabajo para compartirlo, respaldarlo o colaborar con otras personas.

3.2 Inicializar un repositorio local (git init)

Crear un repositorio desde cero es como abrir un nuevo cuaderno y decirle a Git: "por favor, empieza a registrar todo lo que pase aquí".

Paso a paso para crear tu primer repositorio local:

- **Crea una carpeta para tu proyecto (si no existe aún):**

  mkdir mi-proyecto
  cd mi-proyecto

- Inicializa Git en esa carpeta: *git init*
  Al hacer esto, Git crea una carpeta oculta llamada .git. Esta carpeta contiene toda la información del repositorio: historial, configuraciones, ramas, etc.
- No borres nunca la carpeta .git Es como el alma del proyecto.



- Verifica que Git esté funcionando: *git status*
  Deberías ver un mensaje como:
  On branch main
  No commits yet
  nothing to commit (create/copy files and use "git add" to track)

Esto significa que el repositorio está vacío, pero listo para empezar.

3.3 Clonar un repositorio remoto (git clone)

¿Y si alguien ya tiene un cuaderno escrito y tú quieres tu propia copia para leer, modificar o aportar? En Git, eso se llama **clonar** un repositorio.

¿Qué es clonar?

Clonar significa: "descargar una copia completa de un repositorio que está en internet (por ejemplo en GitHub) a tu propio computador, incluyendo todo su historial".

Paso a paso para clonar un repositorio:

- Ve a GitHub, busca el repositorio que quieres clonar y copia la **URL del repositorio** (puede ser HTTPS o SSH, normalmente algo como https://github.com/usuario/repositorio.git).
- Abre tu terminal y escribe:
  git clone https://github.com/usuario/repositorio.git
- Git descargará todos los archivos y los pondrá en una carpeta con el nombre del repositorio.
- Entra a esa carpeta: cd repositorio
- Listo, ahora tienes una copia completa con todo el historial.

Consejo: puedes clonar cualquier repositorio público para aprender cómo otros programan. ¡Es una excelente forma de estudiar!

3.4 Estructura típica de un proyecto en Git

Después de inicializar o clonar un repositorio, es útil entender qué hay dentro y por qué. Aunque cada proyecto es diferente, muchos comparten una estructura común. Vamos a ver un ejemplo:

```
mi-proyecto/
├── .git/          # Carpeta oculta que guarda todo el historial de Git
└── README.md      # Archivo que explica de qué trata el proyecto
```



```
├── index.js/        # Archivo principal con el código en JavaScript
└── .gitignore       # Archivo que indica qué no debe registrar Git
```

Archivos clave:

- README.md: El cartel de bienvenida del proyecto. Explica qué es, cómo se usa, y cómo se instala. ¡Nunca debe faltar!
- .gitignore: Lista los archivos o carpetas que no quieres que Git controle (por ejemplo: archivos temporales, contraseñas, etc.).

¿Y la carpeta .git?

Está oculta, pero es fundamental. Si la borras, Git olvida todo lo que sabía de tu proyecto. Si alguna vez necesitas "borrar la memoria" de Git, puedes eliminarla… pero hazlo con cuidado.

4. Ciclo de trabajo con Git (local)

Una vez tienes tu repositorio listo y ya sabes cómo crearlo o clonarlo, es momento de empezar a trabajar realmente con Git. Aquí es donde comienza la magia del seguimiento de cambios, y donde Git empieza a registrar tu historia como desarrollador o analista.

Pero… ¿cómo le digo a Git que hice un cambio? ¿Cómo lo guardo? ¿Cómo veo qué ha pasado antes? Y, si me equivoco, ¿puedo retroceder?

Estas preguntas son exactamente lo que vamos a resolver en esta sección. Aquí aprenderás cómo Git sigue tu trabajo en tres pasos muy claros: añadir, registrar, revisar. Verás también cómo volver atrás si algo sale mal (y créeme, a todos nos pasa).

4.1 git add, git commit, git status

¿Cómo funciona el flujo de trabajo básico?

Cuando trabajas en un proyecto con Git, el flujo más común se puede imaginar como una especie de "cinta transportadora":

- Cambias archivos (trabajas normalmente).
- Le dices a Git: "Quiero guardar esto" → git add.
- Git prepara esos archivos para ser registrados.
- Luego le dices: "Guarda estos cambios" → git commit.



- Git toma una “foto” del estado de los archivos y lo guarda con una etiqueta.

Paso a paso:

- git status: Verifica el estado actual
  Antes de hacer cualquier cosa, es buena práctica preguntarle a Git: “¿qué está pasando?”. Para eso sirve git status.
  Git te dirá si hay archivos modificados, nuevos archivos sin seguimiento, o si todo está limpio. Un ejemplo típico:

  git status

  On branch main
  Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  modified: script.js

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

- git add: Agrega cambios al área de staging
  Cuando modificas archivos, Git no los guarda automáticamente. Primero debes decirle cuáles quieres incluir en la próxima "foto". Esto se llama staging.
  git add index.js
  Esto no guarda todavía el cambio, solo lo pone en la lista de espera para ser registrado.

  Analogía: Imagina que estás preparando una maleta para un viaje. git add es como colocar ropa sobre la cama: todavía no la guardas, pero ya sabes qué vas a empacar.

- git commit: Guarda los cambios
  Cuando ya tienes todo listo, haces un commit. Es como tomar una foto de cómo están tus archivos en ese momento.
  git commit -m "Corrige el bug de lógica en calculadora JS"
  El mensaje entre comillas debe explicar brevemente qué hiciste. Así, tú y tu equipo podrán entender qué pasó en cada punto de la historia.

  Consejo: Escribe mensajes de commit claros y descriptivos. Piensa que no son solo para ti ahora, sino para ti (o tu equipo) dentro de unas semanas cuando necesiten entender qué cambió.



- git remote: Conecta tu repositorio local con el remoto
  Ahora elegimos el destino de nuestro repositorio local, indicando hacia dónde queremos que se suba nuestro proyecto
  git remote add origin https://github.com/tuusuario/mi-repo.git
- git push: Sube tu trabajo al repositorio remoto
  Haz de cuenta que con esto damos la orden de enviar nuestro proyecto
  git push -u origin main

4.2 git log, git diff

Ahora que ya hiciste varios commits, ¿cómo ves lo que hiciste en el pasado? ¿Y si quieres ver qué cambió exactamente entre dos momentos?

Aquí entran dos comandos muy útiles:

git log: historial de commits

Este comando te muestra todos los commits que has hecho, en orden cronológico.

git log
La salida típica será algo como:

commit 4f3a25...
Author: Tu Nombre <tuemail@example.com>
Date: Mon May 3 10:00:00 2025 -0500

Corrige error en script de validación

commit e7b191...
Author: Tu Nombre <tuemail@example.com>
Date: Sun May 2 15:30:00 2025 -0500

Agrega script de controladores

Consejo: Para ver un log más compacto: git log –oneline

git diff: compara versiones

¿No estás segura de qué cambió entre dos commits o desde el último commit? Usa git diff.

git diff



Esto mostrará las diferencias línea por línea entre tu versión actual y la última registrada. Las líneas con - son las eliminadas, y las + son las agregadas.

4.3 Buenas prácticas al hacer commits

En Git, los commits son tu diario de desarrollo. Y como todo buen diario, entre más claro escribas, mejor te entenderás en el futuro.

Aquí van algunas buenas prácticas:

- Haz commits pequeños y frecuentes
  Guarda cada avance significativo. Es mejor tener 10 commits claros que 1 enorme e ilegible.
- Usa mensajes claros y concisos
  Un buen mensaje responde: ¿qué hiciste y por qué? Piensa el mensaje como el título de una noticia. Claro, directo y sin adornos innecesarios.
  Incorrecta:
  git commit -m "cambios"
  Correcta:
  git commit -m "Agrega validación para de inicio de sesión de usuario"

  ¿Cuántos caracteres puede tener un mensaje de commit?
  Cuando haces un git commit -m "mensaje aquí", estás escribiendo el mensaje corto del commit. Este mensaje debe ser claro y conciso, y por convención:
  - La primera línea (el mensaje corto) no debe exceder los 50 caracteres.
  - Después puedes dejar una línea en blanco y escribir un mensaje largo (descripción) que puede extenderse más (normalmente hasta 72 caracteres por línea), pero eso se hace cuando no usas -m, sino que entras al editor por defecto de Git.

  Ejemplo correcto y completo (al usar el editor por defecto):
  "Agrega función para validar datos del formulario

  Se añade una función en JavaScript que revisa si los campos
  nombre y correo tienen contenido antes de enviar el formulario"

  Si usas git commit -m, quédate con un resumen corto, directo al punto:
  git commit -m "Agrega validación de datos en formulario"
- Evita guardar archivos temporales o resultados
  Usa .gitignore para excluir archivos generados automáticamente (como .log, .DS_Store, o node_modules cuando trabajes con dependencias instaladas con



npm install o yarn install -body-parser, cors en Express o rxjs, reflect-metada en NestJS-).

4.4 Deshacer cambios: git restore, git reset, git checkout

¿Y si te equivocaste? ¿Hay forma de volver atrás?
Git tiene varias herramientas para deshacer cosas, pero debes usarlas con cuidado. Vamos a ver las más comunes.

- git restore: deshace cambios en un archivo antes del commit
  Digamos que cambiaste un archivo pero te arrepientes. Puedes volver al estado anterior con:
  git restore archivo.js
- git reset: saca archivos del área de staging
  ¿Agregaste algo con git add pero no querías hacerlo aún?
  git reset archivo.js
  Esto no borra los cambios, solo los quita del staging.
- git checkout: volver a una versión anterior (más avanzado)
  Si quieres volver a un commit anterior:
  git checkout <ID-del-commit>
  Esto te deja en un estado de solo lectura. Si quieres trabajar desde ahí, debes crear una rama.
  Esta herramienta es poderosa pero delicada. Usa checkout con precaución y mejor con ayuda si estás empezando.

¿Cómo saber el ID de un commit para usar con git checkout?

Cada commit en Git tiene un identificador único, llamado hash o commit ID. Es una secuencia larga como esta: e7b191a1cb2e94a3f92d2ff2cdd74829aaba6e5e

Como mencionamos anteriormente, para poder ver el ID de un commit, puedes usar git log o git log –online obteniendo como resultado:

e7b191a (HEAD -> main) Agrega validación en carga de formulario
4f3a25f Corrige error en conexión con base de datos

Para volver a ese commit con git checkout, puedes usar los primeros caracteres del hash: git checkout e7b191a

Git permite usar solo los primeros 6-8 caracteres si son suficientes para identificar de forma única el commit.



¿Qué es un hash?

Primero, es importante entender qué es un **hash**. En términos simples, un **hash** es una función matemática que toma una entrada (como el contenido de un archivo o el historial de un commit) y lo transforma en una secuencia fija de caracteres. En el caso de Git, este **hash** es único para cada commit y es una representación compacta del estado del proyecto en un momento específico.

Imagina que tienes un libro. Si aplicas una función de **hash** al contenido de ese libro, obtienes una **huella digital** única del contenido del libro. Si incluso cambias una sola palabra del libro, la huella digital cambiará por completo.

El **hash** que Git genera para cada commit es como una **huella digital del commit**, de modo que no importa cuántos commits haya en tu proyecto, cada uno tendrá un hash único.

¿Cómo está construido el hash de un commit en Git?

Un commit en Git genera un **hash SHA-1** (Security Hash Algorithm - 1), que es una cadena de 40 caracteres alfanuméricos. El **SHA-1** toma el contenido del commit (como los cambios realizados, los archivos involucrados, el autor, la fecha y la referencia al commit anterior) y lo convierte en una cadena única.

Ejemplo de un commit ID:
e7b191a1cb2e94a3f92d2ff2cdd74829aaba6e5e

¿Qué significa cada parte de un commit ID?
Cada uno de los caracteres de un commit ID no tiene un **significado específico** por sí mismo. Sin embargo, lo que importa es que este conjunto único de caracteres es el identificador que Git usa internamente para rastrear ese commit en el repositorio.

- **El hash completo (40 caracteres)**: Representa de manera única ese commit, en función de su contenido. Aun cuando el commit sea muy grande o involucre cambios complejos, el hash siempre será único.
- **Primeros caracteres**: Los primeros 7-10 caracteres suelen ser suficientes para identificar un commit de forma única en el repositorio, pero Git utiliza todo el hash completo para asegurar la unicidad.
- **El resto del hash**: Git asegura que no haya dos commits con el mismo hash. Incluso si dos commits son muy similares, cualquier pequeña diferencia, como el orden de los archivos o una sola línea de código, cambiará el hash completamente.



Analogía:
Imagina que el hash es como un **código de barras** para un producto. Si tienes muchos productos similares, aunque se vean parecidos, cada uno tiene un código de barras único. Si cambias alguna característica del producto, el código de barras cambia por completo.

¿Qué información se usa para generar el hash?

El hash se genera a partir de diversos elementos del commit, entre los cuales se incluyen:

- **El contenido de los archivos modificados**.
- **El autor del commit** (nombre y correo electrónico).
- **La fecha y hora del commit**.
- **El mensaje de commit**.
- **El commit anterior** (padre del commit), que ayuda a formar una cadena de commits.

Git toma todos estos detalles y les aplica una función de hash, generando la cadena de 40 caracteres que **es única** para ese commit específico.

¿Por qué es importante el hash en Git?

El hash es crucial porque:

- **Identificación única**: Cada commit tiene un hash único que permite a Git rastrear y gestionar el historial del proyecto de manera eficiente.
- **Integridad**: Si alguien intenta cambiar el contenido de un commit, el hash cambiará, lo que significa que Git detectará cualquier alteración. Esto proporciona una seguridad robusta para el historial de tu proyecto.
- **Referencias eficientes**: Puedes usar el hash para referenciar un commit en particular sin necesidad de recordar todos los detalles del mismo. Esto es muy útil para hacer cosas como revertir a un commit anterior.

Recapitulación de comandos clave

Acción	Comando
Ver estado actual	git status
Agregar cambios al staging	git add archivo.js
Registrar cambios	git commit -m "mensaje"
Apuntar al repositorio en la nube	git remote
Enviar o empujar el repositorio	git push



Ver historial	git log
Ver diferencias	git diff
Deshacer cambios no guardados	git restore archivo.js
Quitar archivo del staging	git reset archivo.js
Ir a una versión anterior	git checkout <id>

5. GitHub: control de versiones en la nube

En las secciones anteriores hemos aprendido cómo usar Git en nuestro computador local para llevar un control preciso de los cambios en nuestros archivos. Pero, ¿qué ocurre cuando queremos colaborar con otras personas o simplemente tener una copia de seguridad de nuestros proyectos en línea? Aquí es donde entra GitHub. GitHub no es Git, sino un servicio web que usa Git como tecnología base y nos permite llevar nuestro control de versiones a la nube.

5.1 ¿Qué es GitHub y para qué se usa?

GitHub es una plataforma basada en la web que permite almacenar repositorios Git de manera remota. Fue creada pensando en la colaboración y el trabajo en equipo, aunque también es usada por personas que trabajan solas para tener su código respaldado y accesible desde cualquier lugar.

¿Para qué se usa GitHub?

- Para guardar tus repositorios en la nube.
- Para colaborar con otras personas en un mismo proyecto.
- Para crear y gestionar documentación con archivos como README.md.
- Para manejar versiones públicas o privadas de tu código.
- Para usar funcionalidades avanzadas como Issues, Pull Requests, Actions y más.

5.2 Crear una cuenta y un repositorio en GitHub

Antes de poder trabajar con GitHub necesitas una cuenta. Puedes crearla de forma gratuita:

- Ve a https://github.com.
- Haz clic en Sign up.
- Completa el formulario con tu correo electrónico, nombre de usuario y contraseña.
- Verifica tu correo y ¡listo! Ya tienes una cuenta de GitHub.



Crear un repositorio

Un repositorio en GitHub es como una carpeta online donde vas a guardar tu proyecto y todo su historial.

- Inicia sesión en tu cuenta.
- Haz clic en el botón + en la esquina superior derecha y selecciona New repository.
- Asigna un nombre a tu repositorio (por ejemplo, mi-primer-repo).
- Puedes añadir una descripción si lo deseas.
- Decide si quieres que el repositorio sea público (todo el mundo lo puede ver) o privado (solo tú o quienes tú invites).
- No marques ninguna casilla adicional por ahora (como Initialize with a README), ya que vamos a subir un repositorio desde nuestro equipo.
- Haz clic en Create repository.

GitHub te llevará a una pantalla que te da varias opciones para subir tu proyecto. Vamos a usar la opción de "push an existing repository from the command line".

5.3 Subir un repositorio local a GitHub (git remote, git push)

Una vez que ya tienes tu repositorio creado localmente (por ejemplo, con git init) y has hecho algunos commits, es momento de conectarlo con GitHub.

Paso a paso para subir un proyecto local a GitHub

Supongamos que tienes esta estructura en tu computadora:

```
mi-proyecto/
├── index.js
└── README.md
```

Ya iniciaste el repositorio con git init y hiciste tu primer commit con:

- *git add .*
- *git commit -m "Primer commit"*

Ahora sigamos estos pasos:

- git remote: Agrega la URL de GitHub como repositorio remoto
  git remote add origin https://github.com/tu-usuario/mi-primer-repo.git
  origin es el nombre que le damos a ese repositorio remoto. Puedes llamarlo como quieras, pero "origin" es el estándar.
- git push: Empuja (sube) tu rama principal al repositorio remoto.



git push -u origin main
El parámetro **-u** indica que en adelante, cuando hagas git push, Git sabrá que debe enviar los cambios a origin main.

¿Qué pasa si tu rama principal se llama master?

Algunos repositorios aún usan master como nombre de la rama principal. Puedes verificarlo con: *git branch*
Y si fuera el caso, entonces usa: *git push -u origin master*

5.4 Sincronizar cambios: git pull, git fetch

Una vez que estás trabajando con GitHub, es fundamental mantener sincronizados los cambios que ocurren en la nube con tu máquina local.

git pull

Este comando sirve para **descargar los cambios desde GitHub y aplicarlos directamente** a tu repositorio local. *git pull origin main*
Esto es como decir: "Tráeme todo lo nuevo de la nube y actualiza mi copia local".

git fetch

Este comando también **descarga los cambios del repositorio remoto**, pero **no los aplica automáticamente** a tu copia local. *git fetch origin*
Luego puedes ver las diferencias y decidir qué cambios integrar.

Diferencia entre pull y fetch

Un paralelismo sería así:

- **git fetch**: ir al correo, revisar la bandeja de entrada, pero **no abrir los mensajes** todavía.
- **git pull**: ir al correo, **abrir los mensajes** y aplicar lo que dicen.

Usualmente, si estás trabajando sola, git pull es más práctico. Si trabajas en equipo y quieres revisar qué cambió antes de integrarlo, git fetch es más seguro.



6. Trabajo con ramas (branches)

Cuando trabajamos en un proyecto de software, no siempre es buena idea modificar directamente el trabajo principal (la rama principal, llamada main o master). Sería como hacer cambios directamente sobre el documento final de una tesis sin pasar antes por un borrador. Para evitar errores, confusiones o sobrescribir trabajo importante, Git nos da una herramienta fundamental: las **ramas**.

Podemos imaginar una rama como una línea paralela de tiempo en la que podemos trabajar sin afectar la principal. Como si estuviéramos haciendo una copia de nuestro proyecto para experimentar o desarrollar una nueva funcionalidad, sin poner en peligro lo que ya funciona.

6.1 ¿Qué es una rama y por qué usarla?

Una **rama** (branch) en Git es una versión paralela de tu código. Es como tomar un desvío en una autopista: puedes probar cosas nuevas en ese camino, y luego decidir si regresas al principal o integras lo que hiciste.

¿Por qué usar ramas?

- Para **crear nuevas funcionalidades** sin afectar el código principal.
- Para **experimentar** o hacer pruebas.
- Para **trabajar en equipo**, permitiendo que cada persona trabaje en su propia copia.
- Para **mantener versiones separadas**, como por ejemplo una versión de desarrollo y una versión estable.

Ejemplo:
Supongamos que estamos trabajando en un sitio web y queremos agregar una sección de contacto. En lugar de tocar directamente el código principal, creamos una rama: *git branch contacto*

Luego nos movemos a esa rama para trabajar: *git checkout contacto*

Allí podemos hacer cambios, probar cosas, y cuando estemos seguros de que todo funciona, podemos fusionarlo con la rama principal (main).

6.2 Crear, cambiar, fusionar y eliminar ramas (git branch, git checkout, git merge)

- Crear una nueva rama
  git branch nombre-de-la-rama



Por ejemplo: *git branch formulario-contacto*
Esto crea una nueva rama que parte del estado actual, pero no te mueve a ella todavía.

- Cambiar de rama
  git checkout nombre-de-la-rama
  O, a partir de la versión Git 2.23, también puedes usar: *git switch nombre-de-la-rama*
  switch es más intuitivo para usuarios nuevos, pero checkout sigue siendo muy usado.
- Crear y cambiar en un solo paso
  git checkout -b nombre-de-la-rama
  Esto combina los dos pasos anteriores: crear y moverte.
  git checkout -b formulario-contacto
  Ver todas las ramas: *git branch*
  La rama actual estará marcada con un asterisco *.
- Fusionar ramas (merge)
  Una vez que terminaste de trabajar en una rama secundaria, puedes fusionarla con la principal:
  - Primero, muévete a la rama principal: *git checkout main*
  - Luego fusiona: *git merge formulario-contacto*

  Esto traerá todos los cambios hechos en la rama formulario-contacto hacia main.
- Eliminar una rama
  Una vez fusionada, puedes eliminar la rama secundaria: git branch -d nombre-de-la-rama
  Por ejemplo: *git branch -d formulario-contacto*
  Si no fusionaste y aún así quieres eliminarla, usa -D (mayúscula).

6.3 Resolución de conflictos de merge

Cuando se trabaja con Git, los conflictos de *merge* (fusión) son inevitables en proyectos colaborativos o incluso personales si se manejan varias ramas. Un conflicto ocurre cuando Git no puede decidir automáticamente cómo fusionar los cambios hechos en dos ramas porque ambas modificaron la misma parte del mismo archivo de forma diferente.

Para que esto tenga más sentido en el contexto del desarrollo web, imaginemos que tú y otra persona están trabajando en el mismo script llamado validacion.js.



Escenario: trabajo colaborativo en desarrollo web

Imagina que el archivo validacion.js contiene lo siguiente en la rama main:

```
function limpiarEdad(edad) {
  return isNaN(edad) ? 0 : edad;
}
```

Ahora, tú creas una nueva rama para trabajar en una mejora:

git checkout -b validacion-avanzada

En esta rama, decides mejorar la validación asegurando que la edad no sea negativa:

```
function limpiarEdad(edad) {
  const numero = Number(edad);
  return isNaN(numero) || numero < 0 ? 0 : numero;
}
```

Mientras tanto, alguien más, trabajando en main, cambia esa misma función para añadir un límite superior razonable (ejemplo: 120 años):

```
function limpiarEdad(edad) {
  const numero = Number(edad);
  return isNaN(numero) || numero > 120 ? 0 : numero;
}
```

Cuando intentas fusionar tu rama validacion-avanzada hacia main, Git te mostrará un conflicto:

Auto-merging validación.js
CONFLICT (content): Merge conflict in validación.js
Automatic merge failed; fix conflicts and then commit the result.

¿Qué significa esto?

Git está diciéndote: *"No sé cuál versión debería conservar: la que está en main o la que viene desde tu rama. Tú decides."*

Al abrir el archivo validación.js, verás algo como esto:



```
<<<<<<< HEAD
function limpiarEdad(edad) {
  const numero = Number(edad);
  return isNaN(numero) || numero > 120 ? 0 : numero;
}
=======
function limpiarEdad(edad) {
  const numero = Number(edad);
  return isNaN(numero) || numero < 0 ? 0 : numero;
}
>>>>>>> validacion-avanzada
```

Esto se interpreta así:

- Lo que está entre <<<<<<< HEAD y ======= es lo que existe actualmente en la rama en la que estás (main).
- Lo que está entre ======= y >>>>>>> validacion-avanzada es lo que viene desde la rama que estás intentando fusionar (validacion-avanzada).

¿Cómo lo resuelves?

Ahora tú decides qué parte conservar, o si combinas lo mejor de ambas. Una forma de integrarlo podría ser así:

```
function limpiarEdad(edad) {
  const numero = Number(edad);
  return isNaN(numero) || numero < 0 || numero > 120 ? 0 : numero;
}
```

Después de hacer el cambio manualmente, guardas el archivo, y luego indicas a Git que el conflicto fue resuelto:

```
git add validación.js
git commit
```

Ese commit registra que resolviste el conflicto.

Consejo para identificar dónde hay conflictos

Puedes usar un editor de código como VS Code que resalta los conflictos con colores y botones como "*Aceptar cambios entrantes*", "*Aceptar cambios actuales*" o "*Aceptar ambos*" para elegir entre las versiones.



Además, si quieres listar rápidamente en qué archivos hay conflictos, puedes usar: git status

Git te dirá exactamente qué archivos necesitan intervención.

Analogía
Piensa en un conflicto como si dos personas están llenando la misma celda en una hoja de cálculo, cada una con valores distintos. Cuando fusionas sus hojas, Excel te pregunta: *"¿Cuál valor debería dejar aquí?"* Git hace exactamente eso, pero con tu código.

6.4 Flujos comunes de trabajo: feature branch, main-dev

En equipos profesionales (y también como buena práctica personal), se siguen ciertos flujos o modelos de trabajo que organizan el uso de ramas. Uno de los más usados es el modelo de ramas por características (feature branch).

Feature branch
Cada nueva funcionalidad se desarrolla en una rama independiente. Por ejemplo:

git checkout -b feature/login
git checkout -b feature/contact-form

Esto mantiene el main limpio y libre de cambios experimentales.

Rama de desarrollo (dev) y rama principal (main)
Otro modelo muy extendido es tener dos ramas permanentes:

- main: siempre contiene la versión estable y lista para producción.
- dev: contiene la versión en desarrollo, que puede tener cambios que aún están en pruebas.

Los pasos suelen ser:

- Trabajar sobre dev o una rama derivada (feature/registro).
- Fusionar la rama de funcionalidad en dev.
- Cuando se completa una versión, se fusiona dev en main.

Este flujo minimiza los riesgos y permite controlar mejor qué se publica.



7. Colaboración en proyectos con GitHub

Hasta este punto hemos trabajado con Git y GitHub desde una perspectiva más personal: controlando versiones de nuestros archivos localmente, subiendo cambios a la nube y manteniendo la organización del proyecto. Pero Git y GitHub brillan verdaderamente cuando trabajamos en equipo.

La colaboración en GitHub no solo consiste en compartir un repositorio con otras personas, sino también en establecer una dinámica clara: cómo proponer cambios, cómo revisarlos, cómo comunicarse dentro del código y cómo gestionar tareas. Esta sección te guiará a través de los elementos fundamentales para trabajar de forma ordenada y profesional con otras personas en un mismo repositorio.

Imagina que GitHub es como una pizarra colaborativa en línea donde todos pueden escribir, pero hay reglas: unos solo pueden sugerir cambios, otros deben aprobarlos, y hay un sistema para registrar quién hace qué y cuándo. Vamos a descubrir cómo funciona.

7.1 Forks y Pull Requests

Cuando hablamos de *forks* y *pull requests*, estamos hablando del corazón del trabajo colaborativo entre personas que no pertenecen al mismo equipo del repositorio original. Esta es la forma recomendada (y muchas veces obligatoria) de contribuir a proyectos de código abierto.

¿Qué es un *fork*?

Un *fork* es una copia completa de un repositorio público en tu propia cuenta de GitHub. Imagina que estás en una biblioteca y encuentras un libro de ejercicios muy útil. No puedes escribir en el original, pero sí puedes sacar fotocopias para resolver los ejercicios por tu cuenta. Eso es un *fork*.

Para hacer un *fork*, simplemente:

- Ve al repositorio público (por ejemplo: https://github.com/desarrollo-web/proyecto-calculadora).
- Haz clic en el botón “Fork” en la parte superior derecha.
- GitHub copiará todo el proyecto a tu propia cuenta.

Ahora tienes una versión independiente, en la que puedes hacer cambios libremente.



¿Qué es un *pull request*?

Una vez has trabajado en tu *fork* y consideras que tus cambios pueden mejorar el repositorio original, puedes proponer que integren tus aportes. Esa propuesta se llama un **pull request** (abreviado como PR).

"Hola, hice estos cambios en mi copia. ¿Les gustaría que los agregáramos al original?"

Para hacer un pull request:

- Ve a tu repositorio *forkeado*.
- Haz clic en "Contribute" → "Open pull request".
- Escribe un mensaje claro explicando tus cambios.
- Envía el PR.

El equipo del repositorio original puede revisar tus cambios, dejar comentarios, sugerir mejoras, o directamente aprobarlo y fusionarlo.

7.2 Revisiones de código y comentarios

Cuando alguien abre un *pull request*, comienza una etapa crucial: **la revisión de código**.

Aquí, otras personas pueden:

- Ver exactamente qué líneas se modificaron.
- Dejar comentarios directamente sobre el código.
- Aprobar los cambios o pedir ajustes antes de fusionarlos.

Analogía:
Piensa en esto como corregir un ensayo que escribió tu compañero. Usas un bolígrafo rojo para señalar errores ortográficos, dar sugerencias o felicitarlo por una idea clara. En GitHub, se hace igual, pero en el código.

Ejemplo:

```
// Código original
function validarEmail(email) {
  return email.includes("@");
}

// Comentario en el PR:
@rev-usuario: ¿Qué pasa si alguien escribe "@" sin nada más? Tal vez deberías usar una expresión regular para validar mejor el formato del correo.
```



Este tipo de revisión hace que los proyectos sean más robustos, menos propensos a errores y más fáciles de mantener.

7.3 Issues, etiquetas y gestión de tareas

Los *issues* (problemas o tareas) son una de las herramientas más valiosas en GitHub para **gestionar el trabajo pendiente, errores, mejoras o ideas**.

¿Qué es un *issue*?

Un *issue* es simplemente una entrada escrita, como una nota o ticket, que describe algo que hay que hacer. Por ejemplo:

- "El formulario no valida correctamente los correos electrónicos."
- "Optimizar el tiempo de respuesta del endpoint /login."
- “Agregar documentación para el archivo validacion.js.”

Cada *issue* puede tener:

- Un título y descripción detallada.
- Comentarios de diferentes usuarios.
- Etiquetas (por ejemplo: bug, mejora, documentación).
- Asignaciones (personas responsables).
- Milestones (etapas del proyecto en las que se agrupan tareas).

Ejemplo:

Título: Validación incorrecta de correos electrónicos

Descripción:
Actualmente la función `validarEmail` en `validacion.js` solo revisa si el texto contiene `"@"`. Esto permite correos inválidos como `"@"` o `"usuario@"`.

Propuesta:
Mejorar la validación usando una expresión regular básica que verifique el formato `usuario@correo.com`.

***Etiquetas:** `bug`, `validación`*

***Asignado a:** @dev-junior*

Esto no solo ayuda a organizar el trabajo, sino que permite que el equipo vea el progreso de manera clara.



7.4 Asignación de roles y permisos

En proyectos colaborativos, no todas las personas tienen los mismos privilegios. GitHub permite definir **roles** para garantizar que el acceso esté alineado con las responsabilidades de cada persona.

Roles más comunes en GitHub

- **Owner (Propietario):** Control total sobre el repositorio.
- **Admin:** Puede cambiar configuraciones, gestionar colaboradores.
- **Write:** Puede hacer *push* directamente a ramas.
- **Read:** Solo puede ver el código, abrir *issues* o *pull requests*.
- **Triage:** Puede gestionar *issues* y *pull requests*, pero no escribir código.
- **Maintainer (en organizaciones):** Encargado de revisar contribuciones externas.

Asignar bien estos roles evita errores como que alguien borre por accidente la rama principal o fusione un PR sin revisar.

¿Cómo asignar un rol?

- Ve a la pestaña Settings del repositorio.
- En Collaborators, añade el usuario.
- Asigna el rol según lo que necesite hacer.

8. Archivos esenciales en un proyecto

Imagina que alguien encuentra tu proyecto en internet. ¿Cómo sabría qué hace tu código? ¿Cómo sabría ejecutarlo? ¿Cómo evitaría romperlo sin querer? ¿Y cómo podría colaborar contigo de manera efectiva?

La respuesta a estas preguntas no está solamente en el código, sino en **los archivos auxiliares que organizan, explican y protegen tu trabajo**. Son como los letreros, manuales y reglas de convivencia de una casa bien ordenada.

En esta sección vamos a conocer cuatro tipos de archivos clave que deberían estar presentes en todo proyecto serio, especialmente en proyectos de desarrollo web:

- Un archivo que explique el propósito del proyecto (README.md)
- Un archivo que diga qué **no debe subirse** a GitHub (.gitignore)
- Un archivo que defina **la licencia** de uso (como MIT, GPL, etc.)
- Y un archivo que indique qué paquetes o dependencias necesita el proyecto (package.json)



8.1 README.md: estructura y contenido profesional

El archivo README.md es lo primero que las personas verán al ingresar a tu repositorio en GitHub. Es como el cartel de bienvenida, el manual y la carta de presentación **todo en uno**. Usualmente está escrito en un formato llamado Markdown, que permite usar títulos, listas, negritas, enlaces, imágenes y más, sin ser tan complejo como HTML.

¿Para qué sirve?

- Explica **qué hace** el proyecto.
- Indica **cómo instalarlo y usarlo**.
- Sirve de guía para que otros puedan **colaborar contigo**.
- Es una oportunidad para mostrar tu trabajo de forma clara y profesional.

Estructura recomendada para proyectos de desarrollo web (este ejemplo es académico)

Aquí tienes una plantilla con comentarios explicativos:

```
# Proyecto de Desarrollo Web: [Nombre del Proyecto]
## Información del Estudiante
- **Nombre:** [Tu Nombre]
- **Curso:** [Tu Curso]
- **Institución:** [Tu Institución]
- **Fecha:** [Fecha de realización]

---
## Objetivo
Explica brevemente qué se quiere lograr con este proyecto de desarrollo web.
*Ejemplo: Construir una aplicación web que permita a los usuarios registrarse, iniciar sesión y publicar artículos en un blog sencillo.*
---

## Estructura del Repositorio
/Proyecto_Web/
├── public/                # Archivos estáticos (imágenes, estilos, scripts de cliente)
├── src/                   # Código fuente principal
│   ├── index.js           # Punto de entrada del servidor
│   ├── routes/            # Definición de rutas
│   ├── controllers/       # Lógica de negocio
│   ├── models/            # Modelos y conexión a base de datos
│   └── views/             # Plantillas HTML/EJS/Handlebars
```




```
├── tests/              # Pruebas unitarias y de integración
├── capturas/           # Imágenes del funcionamiento de la app
├── package.json        # Dependencias y scripts de npm
├── .gitignore          # Archivos que no deben subirse al repositorio
└── README.md           # Este archivo
```

Instalación y Uso

1. Clonar el repositorio:
   ```bash
   git clone https://github.com/usuario/proyecto-academico.git
   cd proyecto-academico

2. Instalas dependencias:
npm install

3. Configurar variables de entorno:
PORT=3000
MONGO_URI=mongodb://localhost:27017/gestion_academica

4. Iniciar el servidor:
npm start
---

## Proceso de Desarrollo

### 1. Configuración inicial
- Creación del proyecto con `npm init`.
- Instalación de dependencias (`express`, `dotenv`, etc.).

### 2. Implementación del backend
- Definición de rutas REST (GET, POST, PUT, DELETE).
- Conexión a la base de datos.
- Validación de datos de entrada.

### 3. Desarrollo del frontend
- Maquetación en HTML y CSS.
- Interactividad con JavaScript en el navegador.
- Consumo de API con `fetch`.
   ```



4. Pruebas y despliegue
- Pruebas unitarias básicas con `jest` o `mocha`.
- Despliegue en plataformas como Heroku, Vercel o Netlify.

Resultados y Aprendizajes Clave
- Ejemplo de conclusiones:
* - El proyecto permitió reforzar conocimientos en manejo de rutas y controladores.*
* - Se logró implementar un CRUD completo con conexión a base de datos.*
* - Se comprendió el flujo básico cliente-servidor en una aplicación web.*

Archivos Entregados

Archivo	*Descripción*
`src/index.js`	*Código principal del servidor*
`src/routes/*.js`	*Definición de las rutas de la aplicación*
`src/views/*.ejs`	*Plantillas del frontend*
`capturas/*.png`	*Evidencias del funcionamiento del sistema*
`package.json`	*Lista de dependencias y scripts de ejecución*

Requisitos (si aplica)
- Node.js 18+
- Navegador web actualizado
- Dependencias: `express`, `dotenv`, `mongoose` o `mysql2` (según BD elegida)

Contacto
Para dudas o comentarios: [tu_correo@ejemplo.com]

8.2 .gitignore: Qué es y cómo usarlo

Cuando trabajamos en proyectos, es muy común tener archivos **que no queremos compartir** con otras personas: pueden ser archivos grandes, datos confidenciales, o simplemente archivos temporales que tu sistema genera automáticamente.

El archivo .gitignore sirve justamente para eso: decirle a Git qué archivos o carpetas **no debe tener en cuenta**. Es como si le dijeras a tu asistente: "No toques estos papeles, no los metas en la carpeta del proyecto". Con eso, evitas subir cosas innecesarias o peligrosas a tu repositorio.



Ejemplo:
Supongamos que estás trabajando en un proyecto web sencillo y tienes los siguientes archivos:

```
.
├── index.html
├── src/
│   └── validación.js
├── public/
│   └── style.css
├── node_modules/
├── .gitignore
└── package.json
```

En este proyecto hay dependencias en node_modules/, archivos generados por builds y quizá un .env con credenciales.

No quieres subir esos archivos al repositorio. Para eso, puedes crear un archivo llamado .gitignore con este contenido:

```
# Dependencias
node_modules/

# Archivos del sistema
.DS_Store
Thumbs.db

# Logs
*.log

# Variables de entorno (si usas)
.env
```

Paso a paso:

- Crear el archivo
  En la raíz del proyecto, crea un archivo de texto llamado .gitignore. El punto al inicio es importante.
- Agregar patrones de archivos o carpetas a ignorar
  Puedes usar comodines como *.log para ignorar todos los archivos .log, o escribir nombres específicos como datos_sensibles.js.



- Guardar y listo
  Git ignorará todo lo que coincida con lo que escribiste.
  git add .gitignore
  git commit -m "Agregar .gitignore"

Importante: si ya agregaste un archivo al repositorio y luego lo pones en .gitignore, Git no lo eliminará automáticamente. Tienes que eliminarlo manualmente con git rm --cached nombre_del_archivo.

8.3 Licencias de software: breve introducción

Cuando subes tu proyecto a GitHub, lo estás haciendo público. Pero público no significa que cualquiera pueda hacer lo que quiera con él. Para eso existe el concepto de licencias.

Una licencia de software te permite decidir cómo otros pueden usar tu trabajo: si pueden modificarlo, venderlo, copiarlo, citarlo, mezclarlo, etc.

¿Qué es un archivo LICENSE?

Es un archivo de texto (sin extensión) donde especificas bajo qué condiciones legales permites que otros usen tu código. GitHub tiene incluso un botón para elegir una cuando creas un repositorio.

Licencia	¿Se puede usar comercialmente?	¿Se puede modificar?	¿Debe reconocer al autor?	¿Debe mantener misma licencia?
MIT	Aplica	Aplica	Aplica	N/A
GPL	Aplica	Aplica	Aplica	Aplica
Apache 2.0	Aplica	Aplica	Aplica	N/A
CC BY-NC	N/A (No Comercial)	Aplica	Aplica	N/A

Para proyectos de desarrollo web académicos o personales, la más común es la MIT, porque permite a otros reutilizar el código sin complicaciones, siempre que te den crédito.



Ejemplo:
MIT License

Copyright (c) 2025 Nombre del Autor

Permission is hereby granted, free of charge, to any person obtaining a copy of this software [...] (se continúa con el texto legal completo)

Puedes generarlo automáticamente desde https://choosealicense.com/

8.4 Archivos comunes en proyectos de desarrollo web

En cualquier proyecto de desarrollo web (ya sea frontend, backend o fullstack), es fundamental dejar claro **qué dependencias, librerías y configuraciones** requiere el sistema para funcionar.

Así otra persona (o incluso tú misma en el futuro) pueda clonar el repositorio y ejecutar el proyecto sin preocuparse por instalar manualmente cada librería.

Archivos de dependencias

Cada ecosistema tiene su archivo para registrar las librerías y versiones exactas:

- JavaScript/TypeScript: package.json

```
{
  "name": "proyecto-web",
  "version": "1.0.0",
  "scripts": {
    "start": "node index.js"
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

  - Para crearlo automáticamente: npm init -y
  - Para instalar las dependencias en otra máquina: npm install
  - Para ejecutar (si tienes definido un script “start”): npm start

- Python: requirements.txt

```
Flask==2.3.2
```



pandas==2.1.0
numpy==1.25.0

- Para crearlo automáticamente: pip freeze > requirements.txt
- Para instalar las dependencias en otra máquina:
  pip install -r requirements.txt
- Para ejecutar el proyecto: python app.py

- Java: pom.xml (Maven) o build.gradle (Gradle)

  Ejemplo de pom.xml

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.ejemplo</groupId>
  <artifactId>proyecto-web</artifactId>
  <version>1.0.0</version>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
      <version>3.1.0</version>
    </dependency>
  </dependencies>
</project>
```

  - Para crearlo automáticamente: mvn archetype:generate
  - Para instalar las dependencias en otra máquina: mvn install
  - Para ejecutar (ejemplo con Spring Boot): mvn spring-boot:run

- PHP: composer.json

```
{
 "name": "usuario/proyecto-web",
 "require": {
  "slim/slim": "^4.10"
 },
 "scripts": {
   "start": "php -S localhost:8000 -t public"
 }
```



}

- Para crearlo automáticamente: composer init
- Para instalar las dependencias en otra máquina: composer install
- Para ejecutar (servidor local): php -S localhost:8000 -t public

- Ruby: Gemfile

  source "https://rubygems.org"

  gem "sinatra"
  gem "puma"

  - Para crearlo: Simplemente crear un archivo Gemfile y escribir las gemas (paquetes) necesarias.
  - Para instalar las dependencias en otra máquina: bundle install
  - Para ejecutar (ejemplo con Sinatra): ruby app.rb

Aunque cambie el formato, todos cumplen la misma función: gestionar dependencias y asegurar que el proyecto sea reproducible.

Archivos de configuración más comunes.

- .env: Guarda variables sensibles como credenciales, claves API o configuraciones locales. Nunca debe subirse al repositorio, se incluye en .gitignore.
- Dockerfile: Permite empaquetar la aplicación en un contenedor, asegurando que se ejecute igual en cualquier máquina.

Archivos de compilación o transpilación (dependen de la tecnología usada)

- tsconfig.json (TypeScript): Define cómo el compilador de TypeScript transforma el código en JavaScript.
- babel.config.json (JavaScript moderno): Permite usar funcionalidades nuevas de JavaScript en navegadores antiguos o entornos que no las soportan.
- webpack.config.js o equivalentes (para empaquetado de frontend): Indica cómo se deben empaquetar, optimizar y servir los archivos del proyecto (HTML, CSS, JS, imágenes).

Archivos de automatización

- Makefile: Facilita ejecutar comandos comunes como make build o make run sin tener que recordarlos todos.



- Configuraciones de CI/CD: por ejemplo .github/workflows/ci.yml en GitHub Actions, automatizan pruebas, despliegues y verificación de código cuando se suben cambios al repositorio.

9. Buenas prácticas y flujos recomendados

Hasta este punto ya tienes una sólida base sobre Git, GitHub y cómo trabajar con control de versiones. Pero, como en todo en la vida, no se trata solo de saber cómo usar una herramienta, sino también de cómo usarla bien.

Las buenas prácticas en Git no solo hacen que tu proyecto funcione mejor, sino que también permiten que otras personas —tus compañeros, tu docente, un reclutador o tú misma en el futuro— puedan entender fácilmente lo que hiciste y por qué lo hiciste. Esta sección te ayudará a cultivar esos hábitos que hacen que tu trabajo destaque por su claridad, orden y profesionalismo.

9.1 Convenciones de commits y nombres de ramas

Imagina que estás revisando un historial de cambios y te encuentras con mensajes como:

- "cosas"
- "arreglo"
- "cambio final final V3"

A todos nos ha pasado, pero cuando estás en un equipo o estás tratando de entender un código que escribiste hace 2 meses, estos mensajes no ayudan. Por eso, seguir una convención clara para los mensajes de commit y los nombres de ramas puede marcar la diferencia entre un caos y un proyecto saludable.

Convención de mensajes de commit

Una práctica común es estructurar los mensajes con un formato tipo:
<tipo>: <descripción corta>
Donde <tipo> puede ser:

- feat: se agregó una funcionalidad nueva.
- fix: se solucionó un error.
- docs: cambios en la documentación.
- style: formato, espacios, punto y coma, etc. (sin cambios en el código).
- refactor: refactorización del código (sin agregar ni eliminar funcionalidad).
- test: se agregaron o corrigieron pruebas.



- chore: tareas menores que no afectan la lógica del código.

Ejemplos:

- feat: agregar autenticación con JWT en el backend
- fix: corregir bug en validación del formulario de registro
- docs: actualizar README con instrucciones de instalación
- style: aplicar prettier al código del frontend

Convención para nombres de ramas

Una rama debe decirte para qué fue creada, sin ser demasiado extensa. Algunas convenciones comunes: <tipo>/<nombre-descriptivo>

- feature/ para nuevas funcionalidades
- bugfix/ para errores detectados en desarrollo
- hotfix/ para errores críticos en producción
- refactor/ para reestructurar código

Ejemplos:

- feature/login-con-google
- bugfix/error-css-navbar
- hotfix/pago-duplicado
- refactor/optimizar-controladores

9.2 Flujos de trabajo colaborativos: Git Flow vs. GitHub Flow

¿Qué es un flujo de trabajo en Git?

Es una estrategia organizada que te dice cómo deberías usar ramas, commits y merges en un equipo. No es lo mismo trabajar solo que con 4 personas. Y no es lo mismo entregar un trabajo práctico que mantener una aplicación en producción.

Vamos a conocer dos de los más populares: Git Flow y GitHub Flow.

Git Flow: el clásico robusto

- Ideal para proyectos más grandes o con múltiples versiones.
  Versiones estables en producción (ej: la web pública).
- Una rama de desarrollo constante (ej: donde se integran nuevas secciones, APIs o features).



- Cambios urgentes que deben aplicarse rápido.

Estructura típica:

- main: contiene el código listo para producción.
- develop: donde se integran las nuevas funcionalidades antes de pasar a producción.
- feature/*: ramas para trabajar en nuevas funciones.
- release/*: preparaciones para versiones específicas.
- hotfix/*: arreglos urgentes que se aplican directamente sobre main.

Ejemplo:
git checkout develop
git checkout -b feature/navbar-responsive

- Ventaja: Mantiene orden cuando hay varios equipos trabajando en frontend, backend y APIs al mismo tiempo.
- Desventaja: Puede sentirse burocrático para páginas o proyectos chicos.

GitHub Flow: más simple y directo

Ideal cuando trabajas en proyectos web donde los cambios deben salir rápido

- Landing pages.
- Sitios pequeños.
- Proyectos con integración continua (CI/CD) donde cada push puede desplegarse en el hosting (ej: Vercel, Netlify, GitHub Pages).

Pasos:

- Se trabaja directamente desde main o una rama pequeña.
- Se crea una rama feature/lo-que-sea.
- Se trabaja allí.
- Se hace un Pull Request.
- Se revisa.
- Se hace el merge a main y despliegue automático (ej: Netlify o Vercel lo publican al instante).

Más liviano, menos burocrático, pero exige buena comunicación.

- Ventaja: Perfecto para sitios donde se necesita iterar rápido (ej: marketing, landing pages, e-commerce).



- **Desventaja**: No tan bueno si manejas muchas versiones o si tienes un backend y frontend complejos en paralelo.

Comparación

Aspecto	Git Flow	GitHub Flow
Complejidad	Alta (más ramas)	Baja (flujo simple)
Ideal para	Software en producción	Proyectos colaborativos
Control de versiones	Más estructurado	Más flexible

9.3 Estructura de carpetas y documentación clara

En desarrollo web es muy fácil terminar con un proyecto desordenado: imágenes sueltas, estilos mezclados con scripts y archivos de configuración perdidos. Una buena estructura permite que cualquier persona entienda rápidamente tu proyecto.

Estructura de proyecto recomendada

```
mi-proyecto-web/
├── public/          # Archivos públicos (favicon, index.html base, imágenes estáticas)
├── src/             # Código fuente de la aplicación
│   ├── assets/      # Recursos: imágenes, íconos, fuentes
│   ├── components/  # Componentes reutilizables
│   ├── pages/       # Páginas principales (Home, Login, Dashboard, etc.)
│   ├── services/    # Conexión con APIs o lógica de backend
│   ├── styles/      # Archivos CSS/SCSS globales o módulos
│   └── utils/       # Funciones auxiliares
├── tests/           # Pruebas unitarias o de integración
├── docs/            # Documentación adicional
├── .env             # Variables de entorno (API keys, configuración local)
├── .gitignore       # Archivos/carpetas a ignorar (node_modules/, build/, .env, etc.)
├── package.json     # Dependencias y scripts
└── README.md        # Guía principal del proyecto
```

Consejos

- Separa siempre lo público (public/) de lo que compilas (src/).
- Usa README.md para explicar cómo instalar y ejecutar el proyecto.
- No subas dependencias ni configuraciones sensibles (node_modules/, .env).
- Documenta lo necesario: un comentario corto y claro es mejor que nada.



9.4 Automatización y control de archivos generados

En proyectos de desarrollo web, algunos archivos no deberían versionarse en Git porque cambian constantemente o se generan automáticamente durante la compilación. Guardarlos solo hace más pesado y confuso el historial del repositorio.

Archivos y carpetas comunes a ignorar:

- node_modules/ (dependencias instaladas con npm o yarn).
- dist/, build/ (archivos de compilación o empaquetado).
- coverage/ (resultados de pruebas automáticas).
- .env (variables de entorno con credenciales o configuraciones locales).
- *.local.json u otros archivos de configuración personal.

Buenas prácticas:

- Usa siempre un .gitignore para excluir estos archivos del repositorio.
- Versiona solo el código fuente y la configuración necesaria para reconstruir el proyecto.
- Para assets grandes (imágenes pesadas, videos, PDFs), utiliza un servicio externo (CDN, S3, Firebase, Cloudinary) o, si es necesario, Git LFS.

Analogía:
Piensa en tu repositorio como el manual de construcción de una casa.

- Guardas los planos (código fuente).
- Guardas la lista de materiales (package.json, configuraciones).
- Pero no guardas la pila de escombros que sobra cada vez que trabajas (builds, cachés).
- Tampoco guardas tu libreta con contraseñas del WiFi (variables de entorno).