

# Auxiliatura de Programación Web II (INF-122)

Auxiliar: Alex Franco Choque Vega

## Guía Completa de Git y GitHub

### Indice

<b>Introducción</b> .....	3
Control de Versiones .....	3
Concepto de Git .....	3
Concepto de GitHub .....	3
Conceptos Básicos de Control de Versiones .....	4
<b>Instalación y Configuración de Git</b> .....	4
Instalación en diferentes sistemas operativos .....	4
Windows .....	4
macOS .....	5
Linux (Ubuntu/Debian) .....	5
Linux (Fedora) .....	5
<b>Fundamentos de Git</b> .....	5
Iniciar un repositorio .....	5
Comandos básicos .....	6
Trabajando con Ramas (Branching) .....	7
Fusionando Cambios (Merging) .....	8
Fusión básica merge .....	8
Rebase vs Merge .....	8
<b>Resolución de Conflictos</b> .....	9
¿Qué causa los conflictos? .....	9
Proceso de resolución de conflictos .....	9
Herramientas para resolver conflictos .....	9
Prevenir conflictos .....	9
<b>Trabajando con Repositorios Remotos</b> .....	10
Concepto de repositorio remoto .....	10
Comandos para trabajar con remotos .....	10
<b>Introducción a GitHub</b> .....	11
¿Qué es GitHub? .....	11

Crear una cuenta y primer repositorio .....	11
Autenticación en GitHub.....	11
Flujos de Trabajo con GitHub.....	12
GitHub Flow .....	12
Gitflow .....	12
Trunk-Based Development .....	12
Pull Requests .....	13
Issues y Proyecto en GitHub .....	14
<b>GitHub Pages .....</b>	<b>15</b>
¿Qué es GitHub Pages?.....	15
GitHub Actions: Integración Continua .....	15
¿Qué son GitHub Actions? .....	15
Conceptos clave .....	15
<b>Buenas Prácticas .....</b>	<b>17</b>
<b>Recursos Adicionales .....</b>	<b>19</b>
Documentación oficial .....	19
Herramientas gráficas.....	19
<b>Ejercicios Prácticos .....</b>	<b>19</b>
<b>Conclusión .....</b>	<b>20</b>

# Introducción

## Control de Versiones

El control de versiones es un sistema que registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante.

Permite:

- Revertir archivos a estados anteriores
- Comparar cambios a lo largo del tiempo
- Ver quién modificó por última vez algo que podría estar causando problemas
- Identificar cuándo se introdujo un problema y por quién

## Concepto de Git

Git es un sistema de control de versiones distribuido, diseñado para manejar todo, desde proyectos pequeños hasta muy grandes, con velocidad y eficiencia. Git fue creado por Linus Torvalds en 2005 para el desarrollo del kernel de Linux y desde entonces se ha convertido en el sistema de control de versiones más utilizado en el mundo.

Características principales:

- Distribuido: Cada desarrollador tiene una copia completa del repositorio
- Rápido: La mayoría de las operaciones son locales
- Integridad: Todo en Git tiene un checksum antes de ser almacenado
- No destructivo: Es muy difícil perder información o datos

## Concepto de GitHub

GitHub es una plataforma de alojamiento basada en la web para repositorios Git. Añade funcionalidades de colaboración como:

- Interfaz web para visualizar repositorios
- Pull requests para revisión de código
- Issues para seguimiento de tareas y bugs
- Wikis para documentación
- Integración continua
- Y muchas otras características para facilitar el trabajo colaborativo

## Conceptos Básicos de Control de Versiones

### Tipos de Sistemas de Control de Versiones

1. **Sistemas Locales:** Copias simples de archivos en diferentes directorios.
2. **Sistemas Centralizados (CVS, Subversion):** Un servidor central contiene todos los archivos versionados.
3. **Sistemas Distribuidos (Git, Mercurial):** Cada cliente tiene una copia completa del repositorio.

### Ventajas de los Sistemas Distribuidos

- No dependencia de un servidor central
- Trabajo offline
- Rapidez en la mayoría de operaciones
- Mayor flexibilidad en los flujos de trabajo

### Git (manera que trabaja)

Git almacena y maneja la información de forma diferente a otros sistemas:

- **Instantáneas vs. Diferencias:** Git guarda "instantáneas" completas del estado de los archivos, no solo las diferencias.
- **Estado de los archivos:** Los archivos en Git pueden estar en tres estados principales:
  - o *Confirmado (committed)*: Datos almacenados de manera segura en la base de datos local.
  - o *Modificado (modified)*: Archivo que ha cambiado pero no se ha confirmado a la base de datos.
  - o *Preparado (staged)*: Archivo modificado marcado para ir en la próxima confirmación.

## Instalación y Configuración de Git

### Instalación en diferentes sistemas operativos

#### Windows

1. Descargar el instalador desde [git-scm.com](https://git-scm.com)
2. Ejecutar el instalador y seguir las instrucciones
3. Opcional: Instalar GitHub Desktop para una interfaz gráfica

## macOS

1. Opción 1: Instalar mediante Homebrew: `brew install git`
2. Opción 2: Descargar el instalador desde [git-scm.com](http://git-scm.com)

## Linux (Ubuntu/Debian)

```
sudo apt-get update  
sudo apt-get install git
```

## Linux (Fedora)

```
sudo dnf install git
```

## Configuración inicial

Después de instalar Git, es importante configurar tu identidad:

```
git config --global user.name "JuanitoPerez"  
git config --global user.email "juanito1@ejemplo.com"
```

## Configuración del editor

```
git config --global core.editor "code --wait" # Para Visual Studio Code
```

## Ver configuración actual

```
git config --list
```

## Ayuda en Git

```
git help <comando>  
git <comando> --help
```

# Fundamentos de Git

## Iniciar un repositorio

Hay dos formas de obtener un repositorio Git:

1. **Inicializar un nuevo repositorio:**

```
mkdir mi-proyecto  
cd mi-proyecto  
git init
```

2. **Clonar un repositorio existente:**

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

## El ciclo de vida de los archivos

Los archivos en Git pasan por diferentes estados:

1. **Untracked**: Archivos que Git no conoce.
2. **Tracked**: Archivos que Git está siguiendo, que pueden estar:
  - **Unmodified**: Sin cambios desde la última confirmación.
  - **Modified**: Con cambios desde la última confirmación.
  - **Staged**: Preparados para la próxima confirmación.

## Comandos básicos

### Revisar el estado del repositorio

```
git status
```

### Añadir archivos al área de preparación (staging)

```
git add archivo.txt      # Añadir un archivo específico  
git add .                # Añadir todos los archivos modificados  
git add *.js            # Añadir todos los archivos .js
```

### Confirmar cambios (commit)

```
git commit -m "Mensaje descriptivo del cambio"
```

### Ver el historial de commits

```
git log                  # Ver historial completo  
git log --oneline        # Ver historial resumido  
git log --graph          # Ver historial con gráfico de ramas
```

### Ignorar archivos (.gitignore)

Para evitar que Git siga archivos innecesarios (como archivos de configuración local, archivos temporales, etc.), se utiliza un archivo **.gitignore**:

```
# Crear archivo .gitignore  
touch .gitignore  
  
# Ejemplo de contenido  
node_modules/  
*.log  
.DS_Store  
.env
```

## Deshacer cambios

**Descartar cambios en archivos modificados (no staged):**

```
git checkout -- archivo.txt  
git restore archivo.txt
```

**Deshacer un archivo del área de staging:**

```
git reset HEAD archivo.txt  
git restore --staged archivo.txt
```

**Modificar el último commit:**

```
git commit --amend -m "Nuevo mensaje"
```

**Revertir un commit (creando un nuevo commit):**

```
git revert <hash-del-commit>
```

**Deshacer varios commits (cuidado, reescribe la historia):**

```
git reset --soft HEAD~3  # Mantiene los cambios en staging  
git reset --mixed HEAD~3 # Default, mantiene los cambios pero no en staging  
git reset --hard HEAD~3  # Elimina los cambios !!!!!
```

## Trabajando con Ramas (Branching)

Concepto de ramas

Las ramas en Git son punteros ligeros que apuntan a un commit específico. Permiten:

- Desarrollo paralelo de características
- Experimentos aislados
- Organización del trabajo en equipo

Comandos para trabajar con ramas

**Ver ramas existentes**

```
git branch          # Listar ramas locales  
git branch -a      # Listar todas las ramas (locales y remotas)
```

**Crear una nueva rama**

```
git branch nueva-rama
```

**Cambiar a una rama**

```
git checkout nueva-rama  
git switch nueva-rama
```

**Crear y cambiar a una nueva rama (en un solo paso)**

```
git checkout -b nueva-caracteristica  
git switch -c nueva-caracteristica
```

## Eliminar ramas

```
git branch -d rama-finalizada      # Eliminar rama fusionada  
git branch -D rama-descartada      # Forzar eliminación de rama no fusionada
```

## Renombrar ramas

```
git branch -m nuevo-nombre        # Renombrar la rama actual  
git branch -m viejo-nombre nuevo-nombre # Renombrar otra rama
```

## Fusionando Cambios (Merging)

### Fusión básica merge

```
git checkout main                  # Cambiamos a la rama destino  
git merge feature-branch         # Fusionamos la rama de características en main
```

### Tipos de fusiones

1. **Fast-forward:** Cuando no hay commits divergentes.
2. **Merge commit:** Cuando hay commits divergentes, Git crea un commit de fusión.
3. **Rebase:** Reescribir la historia para obtener una línea de tiempo lineal.

## Rebase vs Merge

**Merge:** Preserva la historia exacta

```
git checkout main  
git merge feature
```

**Rebase:** Crea una historia lineal (más limpia)

```
git checkout feature  
git rebase main  
git checkout main  
git merge feature # Ahora será fast-forward
```

### Cuándo usar cada uno:

- **Merge:** Para ramas de características que se fusionan a la rama principal.
- **Rebase:** Para mantener tu rama de características actualizada con los cambios de la rama principal.

**Consejo:** Nunca rebase ramas públicas/compartidas.

# Resolución de Conflictos

## ¿Qué causa los conflictos?

Los conflictos ocurren cuando:

- Dos ramas modifican la misma línea de un archivo
- Una rama elimina un archivo mientras otra lo modifica

## Proceso de resolución de conflictos

1. Git indica qué archivos tienen conflictos
2. Abrir los archivos y buscar las marcas de conflicto:

```
<<<<< HEAD
Código de la rama actual
=====
Código de la rama que estamos fusionando
>>>>> nombre-de-rama
```

3. Editar el archivo para resolver el conflicto (eliminar marcadores y decidir qué código mantener)
4. Añadir los archivos resueltos con `git add`
5. Completar la fusión con `git commit`

## Herramientas para resolver conflictos

- Editor de texto con soporte Git (VS Code, Atom, etc.)
- Herramientas específicas:  
`git mergetool`
- Herramientas visuales como GitKraken, SourceTree, etc.

## Prevenir conflictos

- Comunicación clara en el equipo
- Ramas de corta duración
- Actualizaciones frecuentes desde la rama principal
- Dividir el trabajo para minimizar cambios en las mismas áreas

# Trabajando con Repositorios Remotos

## Concepto de repositorio remoto

Los repositorios remotos son versiones de tu proyecto alojadas en Internet o en alguna red. Permiten:

- Colaborar con otros desarrolladores
- Respaldar tu trabajo
- Compartir código

## Comandos para trabajar con remotos

### Ver repositorios remotos configurados

```
git remote -v
```

### Añadir un repositorio remoto

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

### Obtener cambios del remoto sin fusionar

```
git fetch origin
```

### Obtener cambios y fusionarlos (pull = fetch + merge)

```
git pull origin main
```

### Enviar cambios al remoto

```
git push origin main
```

### Eliminar una conexión remota

```
git remote remove nombre-remoto
```

## Trabajar con ramas remotas

### Crear una rama local que siga una rama remota

```
git checkout -b feature origin/feature
```

### Establecer seguimiento para una rama existente

```
git branch -u origin/feature
```

### Eliminar una rama remota

```
git push origin --delete rama-remota
```

# Introducción a GitHub

## ¿Qué es GitHub?

GitHub es una plataforma de alojamiento de código para el control de versiones y la colaboración. Permite:

- Alojar repositorios Git
- Colaborar con otros desarrolladores
- Seguir proyectos open source
- Gestionar proyectos con herramientas integradas

## Crear una cuenta y primer repositorio

1. Registrarse en [GitHub](#)
2. Crear un nuevo repositorio desde la interfaz web
3. Seguir las instrucciones para añadir un repositorio existente o crear uno desde cero

## Autenticación en GitHub

### Usando HTTPS con token de acceso personal

1. Generar un token en GitHub: Settings > Developer settings > Personal access tokens
2. Usar el token como contraseña al hacer git push/pull

### Usando SSH

1. Generar par de claves SSH:

```
ssh-keygen -t ed25519 -C "juanito2@ejemplo.com"
```

2. Añadir la clave pública a GitHub: Settings > SSH and GPG keys
3. Configurar Git para usar SSH:

```
git remote set-url origin git@github.com:usuario/repositorio.git
```

## La interfaz de GitHub

- **Código:** Explorador de archivos y código
- **Issues:** Sistema de seguimiento de problemas y tareas
- **Pull Requests:** Solicitud de cambios para revisión
- **Actions:** Automatización de flujos de trabajo (CI/CD)
- **Projects:** Gestión de proyectos con tableros Kanban
- **Wiki:** Documentación del proyecto

- **Insights:** Estadísticas y análisis del repositorio
- **Settings:** Configuración del repositorio

## Flujos de Trabajo con GitHub

### GitHub Flow

Un flujo de trabajo ligero basado en ramas:

1. Crear una rama desde `main`
2. Realizar cambios y commits
3. Abrir un Pull Request
4. Discutir y revisar los cambios
5. Desplegar para probar (opcional)
6. Fusionar a `main`

### Gitflow

Un modelo de ramificación más complejo:

- `main`: Código de producción estable
- `develop`: Rama de integración para características
- `feature/*`: Ramas para nuevas características
- `release/*`: Ramas para preparar releases
- `hotfix/*`: Ramas para correcciones urgentes

### Trunk-Based Development

Un enfoque donde los desarrolladores colaboran en una única rama:

1. Pull de los cambios recientes
2. Desarrollo en pequeños incrementos
3. Prueba local exhaustiva
4. Commit y push frecuente (al menos diario)
5. CI/CD automatizado

### Elegir el flujo adecuado

- **GitHub Flow:** Equipos pequeños, despliegue continuo
- **Gitflow:** Proyectos grandes, releases planificadas

- **Trunk-Based:** Equipos experimentados, CI/CD fuerte

## Pull Requests

¿Qué es un Pull Request?

Un Pull Request (PR) es una propuesta de cambios que un colaborador quiere hacer a un repositorio. Facilita:

- Revisión de código
- Discusión sobre los cambios
- Pruebas automatizadas
- Documentación de decisiones

### Crear un Pull Request

1. Realizar cambios en una rama
2. Subir la rama al repositorio remoto:  

```
git push origin mi-rama
```
3. En GitHub, ir al repositorio y hacer clic en "Compare & pull request"
4. Completar la descripción del PR y enviar

### Anatomía de un buen Pull Request

- **Título:** Conciso y descriptivo
- **Descripción:** Explicar qué, por qué y cómo
- **Referencias:** Links a issues relacionados
- **Capturas de pantalla:** Para cambios visuales
- **Criterios de aceptación:** Qué debe cumplir el PR

### Revisión de código

- Examinar la lógica, estilo y calidad del código
- Hacer comentarios específicos y constructivos
- Sugerir cambios concretos
- Aprobar o solicitar cambios

### Fusionar un Pull Request

Opciones de fusión:

1. **Create a merge commit:** Crea un commit de fusión (preserva historia)
2. **Squash and merge:** Combina todos los commits en uno (historia limpia)
3. **Rebase and merge:** Aplica los commits uno a uno (historia lineal)

## Issues y Proyecto en GitHub

### Trabajando con Issues

Los Issues son la forma de rastrear tareas, mejoras y bugs en GitHub.

### Crear un Issue efectivo

- Título claro y descriptivo
- Descripción detallada del problema o tarea
- Pasos para reproducir (para bugs)
- Comportamiento esperado vs actual
- Capturas de pantalla o logs relevantes
- Etiquetas, asignados y milestone

### Organización con etiquetas

- **bug**: Algo no funciona como se espera
- **enhancement**: Nuevas características o mejoras
- **documentation**: Mejoras en la documentación
- **good first issue**: Bueno para nuevos contribuyentes
- **help wanted**: Se necesita ayuda extra

### Tableros de proyecto

GitHub Projects permite organizar el trabajo en tableros Kanban:

1. **Crear un proyecto**: En la pestaña Projects del repositorio
2. **Definir columnas**: Por ejemplo: To Do, In Progress, Review, Done
3. **Añadir issues**: Arrastrar issues a las columnas
4. **Automatizar transiciones**: Configurar movimientos automáticos

### Vincular Pull Requests con Issues

Para cerrar automáticamente issues al fusionar PRs:

- Usar palabras clave en los mensajes de commit o PR:
  - o "Fixes #42"
  - o "Closes #15"
  - o "Resolves #7"

# GitHub Pages

## ¿Qué es GitHub Pages?

GitHub Pages es un servicio de hosting gratuito que toma archivos HTML, CSS y JavaScript directamente desde un repositorio en GitHub y publica un sitio web.

### Configurar GitHub Pages

1. Ir a Settings > Pages en el repositorio
2. Seleccionar la rama fuente (normalmente `main` o `gh-pages`)
3. Elegir la carpeta (root o `/docs`)
4. Guardar para activar la publicación

### Opciones comunes

- **Sitio personal:** `username.github.io`
- **Sitio de proyecto:** `username.github.io/repository`
- **Dominio personalizado:** Configurable en settings

### Generadores de sitios estáticos

GitHub Pages funciona bien con generadores como:

- Jekyll (integrado)
- Hugo
- Gatsby
- VuePress
- Docusaurus

## GitHub Actions: Integración Continua

### ¿Qué son GitHub Actions?

GitHub Actions es una plataforma de CI/CD (Integración Continua/Entrega Continua) que permite automatizar flujos de trabajo directamente desde GitHub.

### Conceptos clave

- **Workflow:** Un proceso automatizado configurable
- **Event:** Actividad que dispara un workflow (push, PR, etc.)
- **Job:** Conjunto de pasos que se ejecutan en un runner
- **Step:** Tarea individual que puede ejecutar comandos

- **Action:** Aplicación reutilizable para simplificar tareas comunes

Ejemplo básico: Workflow para tests

```
# .github/workflows/test.yml
name: Run Tests

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3

        - name: Set up Node.js
          uses: actions/setup-node@v3
          with:
            node-version: "16"

        - name: Install dependencies
          run: npm ci

        - name: Run tests
          run: npm test
```

## Casos de uso comunes

- Ejecutar pruebas automáticas
- Verificar calidad de código (linting)
- Construir y publicar paquetes
- Desplegar aplicaciones
- Notificar sobre cambios en el repositorio

# Buenas Prácticas

## Mensajes de commit significativos

- Usar formato: `tipo(alcance): mensaje corto`
  - o Tipos: feat, fix, docs, style, refactor, test, chore
  - o Ejemplo: `feat(auth): añadir autenticación con Google`
- Primera línea: menos de 50 caracteres, resumen conciso
- Cuerpo (opcional): explicación detallada del cambio

## Organización del trabajo

- Ramas pequeñas y enfocadas
- Commits frecuentes y pequeños
- Pull requests manejables (<400 líneas)
- Mantener actualizada la rama de trabajo

## Documentación

- README.md completo y actualizado
- Comentarios en el código para explicar "por qué" no "qué"
- Mensajes de commit informativos
- Wiki para documentación extensa

## Seguridad

- No compartir credenciales en el código
- Usar variables de entorno para secretos
- Revisar dependencias regularmente
- Configurar protección de ramas

## Trucos y Consejos Avanzados

### Alias útiles

```
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.st status
git config --global alias.unstage 'reset HEAD --'
git config --global alias.last 'log -1 HEAD'
```

## Git stash

Guardar cambios temporalmente:

```
git stash          # Guardar cambios
git stash list    # Ver stashes guardados
git stash apply   # Aplicar último stash
git stash apply stash@{2} # Aplicar stash específico
git stash drop stash@{0} # Eliminar stash
git stash pop     # Aplicar y eliminar último stash
```

## Git cherry-pick

Aplicar commits específicos a la rama actual:

```
git cherry-pick <hash-commit>
```

## Git bisect

Encontrar commit que introdujo un bug:

```
git bisect start      # Iniciar búsqueda
git bisect bad        # Marcar commit actual como malo
git bisect good <commit-hash> # Marcar commit conocido como bueno
# Git bisect irá navegando automáticamente
git bisect good       # Si el commit actual está bien
git bisect bad         # Si el commit actual tiene el bug
git bisect reset      # Terminar la búsqueda
```

## Hooks de Git

Scripts que se ejecutan en ciertos eventos:

- pre-commit: Antes de crear un commit
- post-commit: Despues de crear un commit
- pre-push: Antes de enviar cambios al remoto

Ejemplo (verificar tests antes de commit):

```
# .git/hooks/pre-commit
#!/bin/sh
npm test

# Si los tests fallan, el commit se cancela
if [ $? -ne 0 ]; then
  echo "Tests failed. Commit aborted."
  exit 1
fi
```

## Recursos Adicionales

### Documentación oficial

- [Git Documentation](#)
- [GitHub Docs](#)
- [Pro Git Book](#)

### Herramientas gráficas

- [GitHub Desktop](#)
- [GitKraken](#)
- [SourceTree](#)
- [Git Extensions](#)

## Ejercicios Prácticos

### Ejercicio 1: Configuración inicial

1. Instala Git en tu computadora
2. Configura tu nombre de usuario y email
3. Genera una clave SSH y agrégala a tu cuenta de GitHub

### Ejercicio 2: Tu primer repositorio

1. Crea un repositorio local
2. Añade algunos archivos y haz commits
3. Crea un repositorio en GitHub
4. Conecta tu repositorio local con el remoto
5. Sube tus cambios a GitHub

### Ejercicio 3: Trabajo con ramas

1. Crea una rama `feature`
2. Haz algunos cambios y commits en esa rama
3. Vuelve a la rama `main` y haz otros cambios
4. Fusiona la rama `feature` en `main`
5. Resuelve cualquier conflicto que aparezca

## Ejercicio 4: Colaboración en GitHub

1. Forma tu grupo para el proyecto
2. Haz fork del repositorio de tu compañero
3. Clona tu fork y crea una rama
4. Realiza algunos cambios y súbelos
5. Crea un Pull Request
6. Tu compañero debe revisar y fusionar el PR

## Ejercicio 5: GitHub Actions

1. Añade un archivo de configuración de GitHub Actions
2. Configura un workflow simple (por ejemplo, verificar sintaxis)
3. Haz push para activar el workflow
4. Verifica los resultados en la interfaz de GitHub

## Conclusión

Git y GitHub son herramientas fundamentales en el desarrollo de software moderno. Dominarlas te permitirá:

- Trabajar de manera más eficiente
- Colaborar efectivamente con otros desarrolladores
- Mantener un historial completo de tu código
- Implementar flujos de trabajo profesionales

Recuerda que la práctica es clave para dominar Git, usalo en todos tus proyectos personales y más en los colaborativos para desarrollar habilidad y confianza con estas herramientas poderosas.

Hecho con  por Franco