DESARROLLO DE APLICACIONES WEB · TEMA 1

1.1 Control de versiones con Git

Resumen y esquema de contenidos: Git, GitHub, flujo de ramas, Pull Requests y Code Reviews.

1 ¿Qué es un sistema de control de versiones?

Un **almacén en la nube para equipos de desarrollo**: guarda el código fuente y **todo su historial** de cambios.

- Conserva el historial completo de versiones anteriores.
- Documenta cada cambio: quién, qué, cuándo y por qué.
- Permite **revertir** el proyecto a un estado anterior.
- Permite crear **ramas** (líneas de desarrollo paralelas) y **forks** (copias independientes).
- Evita y ayuda a resolver **conflictos** cuando dos personas tocan el mismo código.

> **Repositorio local vs. remoto**
> Se trabaja siempre sobre el **repositorio local**. La sincronización con el **repositorio remoto** es siempre manual y explícita (nunca automática, a diferencia de Google Drive o Dropbox).

Git es, hoy en día, el sistema de control de versiones dominante en la industria (CVS, Subversion, Mercurial y Bazaar han quedado relegados a proyectos heredados).

2 Git básico: puesta en marcha

Qué necesitas

- Un **cliente Git** en tu ordenador (git-scm.com), o el integrado en tu editor (VS Code, PhpStorm...).
- Una cuenta en un **servidor Git remoto**: **GitHub** o **GitLab** son los más habituales.

Crear un repositorio nuevo

```
$ git init                              # crea el repo local
$ git branch -M main                    # fuerza el nombre de rama "main"
$ git remote add origin <URI>           # conecta con el remoto
$ git push -u origin main
```

> **⚠ Autenticación: ya no vale usuario + contraseña**
> Desde 2021, GitHub y GitLab exigen un **token de acceso personal (PAT)** o una **clave SSH** para autenticarte al hacer `push`. La contraseña de tu cuenta web ya no funciona por línea de comandos.

Configuración inicial (una sola vez por ordenador)

```
$ git config --global user.name "Mi-nombre"
$ git config --global user.email "mi-email@ejemplo.com"
$ git config --list                    # comprobar
```

El archivo .gitignore

Lista de todo lo que **no** debe subirse al repositorio: archivos de configuración con contraseñas o claves (`.env`), datos de prueba, y carpetas de dependencias regenerables como `vendor/` (Laravel → `composer install`) o `node_modules/` (→ `npm install`). Plantillas por lenguaje en github.com/github/gitignore.

3 Flujo de trabajo cotidiano

1	2	3	4
Desarrollar con normalidad	`git add` a la Staging Area	`git commit` al repo local	`git pull` y `git push`

```
Workspace  →  git add  →  Staging Area  →  git commit  →  Local Repo (HEAD)
Local Repo (HEAD)  ⇄  git pull / git push  ⇄  Remote Repo
```

> **Staging Area**
> "Pista de despegue": aquí decides qué cambios están listos para pasar a un commit. No todo lo que escribes se sube automáticamente.

> **Commit**
> Empaqueta lo que hay en la Staging Area con un mensaje descriptivo obligatorio. Aún **no** llega al remoto.

```
$ git add archivo1 archivo2      # o: git add *
$ git commit -m "Mensaje claro y breve"
$ git pull                       # antes de subir, por si hay cambios ajenos
$ git push
$ git status                     # ver en qué estado está todo
```

Buen mensaje de commit: "Arreglo el fallo del id de usuario al actualizar la foto de perfil". Mal mensaje: "asdf", "cambios varios". Convención recomendada: Conventional Commits (`feat:`, `fix:`, `docs:`...).

4 Deshacer, comparar y ramificar

Deshacer cambios

Situación	Comando
Cambios sin commit, quiero descartarlos	`git reset --hard`
Ver historial de commits	`git log --oneline`
Deshacer un commit ya subido (sin reescribir historial)	`git revert id-del-commit`
Ver el código de un commit antiguo (temporalmente)	`git checkout id-del-commit`
Volver al último commit de `main`	`git checkout main`

Git ≥ 2.23 introduce `git switch` (cambiar de rama) y `git restore` (descartar cambios en archivos) como alternativas más claras al "todoterreno" `git checkout`.

Resolver un conflicto de merge

Git marca en el archivo las dos versiones en conflicto. Hay que editarlas a mano, elegir qué líneas quedan, borrar las marcas `<<<<<<<` / `=======` / `>>>>>>>`, y hacer `git add` + `git commit`.

Ramas

```
$ git branch nombre-rama         # crear rama
$ git switch nombre-rama         # cambiar a ella
$ git switch -c nombre-rama      # crear y cambiar en un paso
$ git switch main
$ git pull
$ git merge nombre-rama          # fusionar con main
```

Una rama permite desarrollar una funcionalidad sin romper `main`. Fusionar ramas "a pelo" en local, sin revisión, es arriesgado en un equipo real → de ahí el **Pull Request**.

5 Flujo de ramas: GitHub Flow

Estrategia sencilla y muy extendida (frente a Git Flow completo, con `develop` / `release` / `hotfix`). Cuatro reglas:

1. `main` siempre desplegable; nadie trabaja directo sobre ella
2. Rama nueva desde `main` por cada tarea: `feature/...`, `fix/...`
3. Se trabaja y se sube (`push`) a esa rama con normalidad
4. Se abre un **Pull Request** para fusionar con `main`

Idea clave: ningún cambio llega a `main` sin pasar antes por una revisión.

6 Pull Requests

Petición formal (en GitHub/GitLab, no un comando de Git) para fusionar una rama con otra, normalmente `main`. Permite:

- Ver el **diff** completo: líneas añadidas y eliminadas, archivo a archivo.
- Añadir **comentarios** generales o sobre líneas concretas.
- Lanzar **comprobaciones automáticas** (tests, linters) antes de poder fusionar.
- Actualizarse solo al subir nuevos commits corrigiendo la revisión.
- Exigir **aprobación** (idealmente de alguien distinto al autor) antes de fusionar.

Flujo típico en GitHub

```
$ git push -u origin nombre-de-tu-rama
```

→ GitHub ofrece crear el Pull Request → título + descripción claros → asignar revisores → esperar comentarios.

7 Code Reviews

La revisión de código, normalmente dentro de un Pull Request, persigue:

- **Detectar errores** que el autor no ve por estar metido en el problema.
- **Mantener coherencia**: mismas convenciones de nombres y estilo en todo el proyecto.
- **Compartir conocimiento** entre el equipo.
- **Detectar problemas de diseño**: duplicación, funciones sobrecargadas, nombres confusos, falta de tests.

> **Buenas prácticas**
> - Revisa el **código**, nunca a la persona.
> - PRs **pequeños y frecuentes** (se revisan mejor que uno enorme).
> - Comentarios concretos; si puedes, sugiere el cambio directamente.
> - Distingue lo **bloqueante** de lo que es una mejora para otro PR.

8 Chuleta de comandos esenciales

Comando	Para qué sirve
`git init`	Crear un repositorio local
`git clone <URI>`	Copiar un repositorio remoto ya existente
`git status`	Ver qué ha cambiado y en qué estado está todo
`git add <archivo>`	Pasar cambios a la Staging Area
`git commit -m "..."`	Registrar los cambios en el historial local
`git push`	Subir commits al repositorio remoto
`git pull`	Bajar los cambios del repositorio remoto
`git log --oneline`	Ver el historial de commits, resumido
`git branch`	Listar / crear ramas
`git switch <rama>`	Cambiar de rama
`git merge <rama>`	Fusionar una rama con la actual
`git revert <id>`	Deshacer un commit creando uno nuevo

9 Para saber más

- Documentación oficial de Git: git-scm.com/docs
- Tutoriales de Atlassian (aplicables a cualquier servidor Git): atlassian.com/es/git/tutorials
- GitHub Flow explicado por GitHub: docs.github.com/es/.../github-flow
- Plantillas de `.gitignore`: github.com/github/gitignore