1. Práctica 1: Git (Fundamentos y Errores)

Esta práctica te introduce a los conceptos más básicos de Git, trabajando en una sola rama (`master`) y aprendiendo a gestionar los 3 .[es]tados"de Git y a corregir errores comunes.

1.1. Conceptos Clave

- **Directorio de trabajo (Working Directory):** Tus archivos y carpetas actuales.
- **Área de preparación (Staging Area):** Una "sala de espera"donde preparas los cambios que quieres guardar en el próximo çommit".
- **Repositorio (History):** El historial de todos los çommits"(instantáneas) guardados.

1.2. Fase 1: Preparación y Primera Confirmación

1. **Crear el proyecto:**
   - `mkdir PracticaGit`: Crea la carpeta.
   - `cd PracticaGit`: Entra en ella.
2. **Inicializar Git:**
   - `git init`: Inicia un repositorio de Git en la carpeta actual. Crea la carpeta oculta `.git`.
3. **Configurar tu identidad:**
   - `git config -global user.name "Tu Nombre"`: Establece tu nombre.
   - `git config -global user.email "tu@email.com"`: Establece tu email.
4. **Crear y Añadir tu primer archivo:**
   - `echo "Mi primer proyecto» README.txt`: Crea un archivo.
   - `git status`: Muestra el estado. El archivo aparece como Üntracked"(No rastreado), lo que significa que Git lo ve pero no le hace seguimiento.
   - `git add README.txt`: Añade el archivo al Staging Area.
   - `git status`: Ahora el archivo aparece como Çhanges to be committed"(Cambios listos para confirmar).
5. **Guardar en el historial:**
   - `git commit -m "Mensaje del commit"`: Guarda todo lo que estaba en el Staging Area como una nueva ïnstantánea.[en] el historial del repositorio.
6. **Revisar el historial:**
   - `git log -oneline`: Muestra un historial simple de todos los commits.

1.3. Fase 2: Modificaciones y Diferencias

1. **Modificar archivos:** Se modifica `README.txt` y se crea `documento.txt`.
2. **Ver diferencias (Diff):**
   - `git diff README.txt`: Compara tu Directorio de trabajo con el Staging Area. (En este punto, como no has hecho `git add`, también lo compara con el último commit).
   - `git add README.txt`: Preparas la nueva versión del README.
   - `git diff -staged`: Compara el Staging Area con el Repositorio (último commit).
   - Si modificas `README.txt` *otra vez* después de hacer `git add`, `git diff` te mostrará solo esa última línea que has añadido, ya que compara tu trabajo actual con lo que ya está en Staging.

1.4. Fase 3: Gestión de Errores (¡Muy importante!)

- **Escenario A: Descartar cambios en el Directorio de Trabajo**
  - Has modificado un archivo (ej. `README.txt`) pero no has hecho `git add`.
  - **Comando:** `git restore README.txt`
  - **Resultado:** El archivo vuelve a la versión del último commit. Tus cambios se pierden.
- **Escenario B: Sacar un archivo del Staging Area**
  - Has modificado `documento.txt` y has hecho `git add documento.txt` por error.
  - **Comando:** `git restore -staged documento.txt`
  - **Resultado:** El archivo sale del Staging Area, pero tus cambios se conservan en el Directorio de Trabajo.
- **Escenario C: Deshacer el último commit**
  - Has hecho un `git commit` por error.
  - **Comando:** `git reset -soft HEADÎ`
  - **Resultado:** El commit se deshace, pero todos los cambios de ese commit se mueven automáticamente al Staging Area. Es perfecto para corregir un mensaje de commit o añadir un archivo que olvidaste.
- **Escenario D: Recuperar un archivo borrado**
  - Has borrado un archivo sin querer (`rm documento.txt`).
  - **Comando:** `git restore documento.txt`

- **Resultado:** El archivo se recupera desde el último commit.

- **Escenario E: Ignorar archivos**
  - Creas un archivo que no quieres que Git rastree (ej. `log.txt`).
  - **Comando:** Crea un archivo llamado `.gitignore` y escribe el nombre de los archivos a ignorar dentro de él (ej. `log.txt`).
  - **Resultado:** `git status` dejará de mostrar `log.txt`.
  - **Importante:** Debes hacer `git add .gitignore` y `git commit` para guardar este archivo en el repositorio.

2. Práctica 2: Git (Ramas, Fusión y Rebase)

Esta práctica te enseña a trabajar con ramas (líneas de desarrollo paralelas), que es la base del trabajo en equipo.

2.1. Fase 1: Setup y `commit -amend`

- Se repite la configuración inicial (crear repo, `git init`, `git add`, `git commit`).
- **Corregir errores locales (antes de `git add`):**
  - `git reset main.js` (o `git restore -staged main.js`): Saca el archivo del Staging Area, pero mantiene tus cambios.
  - `git checkout main.js` (o `git restore main.js`): Descarta todos tus cambios en el archivo y lo devuelve a la versión del último commit.
- **Enmendar un Commit (Corregir el último commit):**
  - Hiciste un commit, pero te das cuenta de que te faltó un cambio o el mensaje está mal.
  - Haces el cambio que faltaba (ej. modificar `main.js`).
  - `git add main.js`
  - `git commit -amend -m "Nuevo mensaje corregido"`
  - **Resultado:** Esto no crea un nuevo commit. En su lugar, "funde"tus nuevos cambios con el commit anterior y reemplaza el mensaje. ¡No hagas esto si ya has subido el commit a un repositorio remoto!

2.2. Fase 2: Ramas, Merge vs. Rebase

- **Crear Ramas (Branches):**
  - `git checkout -b develop`: Crea una nueva rama llamada `develop` y te cambia a ella.
  - Haces cambios, `git add .` y `git commit`. Ahora esos cambios solo existen en la rama `develop`.
- **Caso de "Hotfix"(Arreglo urgente):**
  - Estás en `develop` pero hay un error en `master`.
  - `git checkout master`: Vuelves a la rama principal.
  - `git checkout -b hotfix/fix-log`: Creas una rama desde `master` para el arreglo.
  - Arreglas el error, haces `git add` y `git commit`.

- `git checkout master`.
- `git merge hotfix/fix-log`: Integras los cambios del hotfix en `master`. Como `master` no había cambiado, hace un "Fast-forward"(simplemente mueve el puntero).
- `git branch -d hotfix/fix-log`: Borras la rama de hotfix, ya no se necesita.

- **Merge (Fusión):**
  - **Concepto:** Tienes cambios en la rama A y cambios en la rama B que han divergido (ambas han avanzado). Quieres unir los cambios de B dentro de A.
  - `git checkout A`
  - `git merge B`
  - **Resultado:** Se crea un nuevo commit especial llamado "Merge commit"que tiene dos "padres"(la punta de A y la punta de B). El historial se ve como una red o una cremallera.
  - **Conflicto:** Si A y B modificaron la *misma línea* del *mismo archivo*, Git no sabrá qué versión elegir y te dará un conflicto de merge.
  - **Resolución:** Debes abrir el archivo en conflicto (verás marcadores `«"< , ===== , »>`), editarlo manually para dejar la versión correcta, y luego hacer `git add <archivo>` y `git commit` para finalizar la fusión.
- **Rebase (Reorganización):**
  - **Concepto:** Estás en una rama `feature` que creaste desde `develop`. Mientras trabajabas, `develop` ha recibido nuevos commits. Tu rama `feature` se ha quedado .ªtrás".
  - `git checkout feature`
  - `git rebase develop`
  - **Resultado:** Git .ªrranca"tus commits de la rama `feature`, actualiza `feature` con la última versión de `develop`, y luego vuelve a aplicar tus commits, uno por uno, *encima* de `develop`.
  - **Ventaja:** El historial se mantiene lineal y limpio. Cuando integres `feature` en `develop` (`git checkout develop` y `git merge feature`), será un "fast-forward"limpio.

2.3. Fase 3: Herramientas Avanzadas

- **Cherry Pick (Seleccionar):**
  - **Situación:** La rama `feature/small-utility` tiene dos commits (C1 y C2). Tú solo quieres el C1 en `develop` ahora mismo, porque C2 no está listo.

- `git checkout develop`.
- Buscas el hash (ID) del commit C1 (ej. `84076ab`).
- `git cherry-pick 84076ab`
- **Resultado:** Git copia *solo* ese commit y lo aplica en `develop`.

- **Reset (Deshacer localmente):**
  - **Situación:** Has hecho un commit en tu rama local, pero es un error y quieres borrarlo.
  - `git reset -soft HEADÎ`: Borra el último commit, pero deja los cambios en el Staging Area.
  - `git reset -hard HEADÎ`: ¡Peligroso! Borra el último commit y descarta todos los cambios permanentemente.
- **Revert (Deshacer en público):**
  - **Situación:** Has hecho un commit con un error (`674689d`) y ya lo has subido al repositorio remoto (otros ya lo pueden tener). No puedes usar `reset`.
  - `git revert 674689d`
  - **Resultado:** Git crea un nuevo commit que es la "marcha atrás"del commit erróneo. El historial se mantiene intacto (se ve el error y su corrección), lo cual es seguro para el trabajo en equipo.

2.4. Fase 4: Limpieza

- `git branch -d feature/rama`: Borra una rama que ya ha sido fusionada.
- `git branch -D feature/rama`: (Con `-D` mayúscula) Fuerza el borrado de una rama, incluso si no ha sido fusionada.

3. Práctica 1: Docker (Imagen Personalizada)

Esta práctica te enseña a crear tu propia imagen de Docker (basada en MariaDB), inicializarla con datos y subirla a Docker Hub.

1. **Estructura:**
   - Se crea una carpeta para el proyecto y un subdirectorio `sql`: `mkdir -p ~/db-personalizada`
2. **Script de Inicialización:**
   - Se crea un archivo `sql/init.sql`.
   - **Concepto Clave:** La imagen oficial de `mariadb` ejecutará automáticamente cualquier script `.sql` que encuentre en la carpeta `/docker-entrypoint-initdb.d/` la primera vez que se inicie el contenedor.
   - El script crea una base de datos `app_data` y una tabla `usuarios` con datos.
3. **Crear el `Dockerfile`:**
   - `FROM mariadb:10.5`: Le dice a Docker que use esta imagen como base.
   - `ENV ...`: Define las variables de entorno para la base de datos (contraseña de root, nombre de la BBDD, usuario y contraseña).
   - `COPY ./sql /docker-entrypoint-initdb.d/`: Copia tu carpeta `sql` local (que contiene `init.sql`) a la carpeta especial del contenedor.
   - `EXPOSE 3306`: Informa que el contenedor usa este puerto.
4. **Construir la Imagen:**
   - `docker build -t davgilfe/db-personalizada:v1 .`
   - `-t` se usa para "taggear"(etiquetar) la imagen con un nombre (en formato `usuario/repositorio:tag`).
   - El `.` al final le dice a Docker que el `Dockerfile` está en la carpeta actual.
5. **Probar la Imagen:**
   - `docker run -d -name prueba-db -p 3306:3306 davgilfe/db-personalizada:v1` (la práctica usa `3386:3386`, pero el estándar es `3306:3306`): Lanza el contenedor.
   - `docker exec -it prueba-db mariadb -u manager -psecret_manager app_data`: Entra a la base de datos *dentro* del contenedor.
   - `SHOW TABLES;` y `SELECT * FROM usuarios;`: Comprueba que la tabla y los datos existen (¡el script `init.sql` funcionó!).
6. **Publicar en Docker Hub:**
   - `docker login`: Inicia sesión en tu cuenta de Docker Hub.

- `docker push davgilfe/db-personalizada:v1` : Sube tu imagen para que esté disponible públicamente.

4. Práctica 2: Docker Compose (Multi-Contenedor)

Esta práctica te enseña a usar Docker Compose para gestionar una aplicación completa (WordPress + MariaDB) con un solo archivo de configuración.

4.1. Conceptos Clave

- **Servicios:** Son los contenedores que forman tu aplicación (ej. un servicio `web` y un servicio `db`).
- **Volúmenes:** Se usan para persistir los datos (guardar los datos de la BBDD aunque el contenedor se borre).
- **Redes:** Docker Compose crea una red interna para que los servicios se comuniquen entre sí usando sus nombres (DNS automático).

4.2. El Archivo `docker-compose.yml`

Este archivo YAML define toda tu aplicación:

- **Servicio `db` (MariaDB):**
  - `image: mariadb:10.5`: La imagen que usará.
  - `volumes: - data_mysql:/var/lib/mysql`: Conecta un volumen nombrado (`data_mysql`) a la carpeta de datos de MariaDB. Esto guarda tus datos de forma segura.
  - `environment: ...`: Define las contraseñas y el nombre de la base de datos (ej. `wordpress_db`).
- **Servicio `web` (WordPress):**
  - `image: wordpress:4.9.8`: La imagen de WordPress.
  - `depends_on: - db`: Importante: le dice a Compose que debe iniciar el servicio `db` *antes* que el servicio `web`.
  - `ports: - "8080:80"`: Mapea el puerto 8080 de tu PC al puerto 80 del contenedor (donde corre WordPress).
  - `environment: ...`:
    - `WORDPRESS_DB_HOST: db`: ¡Esta es la magia! WordPress se conecta al host "db", que es el nombre del otro servicio. Compose se encarga de que "db.ªpunte a la IP del contenedor de MariaDB.
    - El resto de variables (user, password, db_name) deben coincidir con las definidas en el servicio `db`.
- **Volúmenes (Nivel Superior):**

- `volumes: data_mysql:` : Aquí se declara formalmente el volumen nombrado que usa el servicio `db` .

4.3. Comandos de Ciclo de Vida

- `docker compose up -d` :
  - Lee el `docker-compose.yml` y crea/inicia todos los servicios, redes y volúmenes.
  - `-d` significa "detached", para que se ejecute en segundo plano.
- `docker compose ps` : Muestra el estado de los servicios (contenedores) de tu aplicación.
- `docker compose logs -f` : Muestra los logs de *todos* los servicios en tiempo real (muy útil para ver errores).
- `docker compose exec web /bin/bash` : Entra a la terminal del contenedor `web` para depurar.
- `docker compose down` :
  - Para y elimina los contenedores y la red.
  - Importante: NO borra los volúmenes nombrados (`data_mysql` se queda). Si vuelves a hacer `up` , tu base de datos seguirá ahí.
- `docker compose down -v` :
  - Hace lo mismo que `down` , pero SÍ borra los volúmenes (`-v`). Úsalo para empezar de cero.