Resumen de Examen: Despliegue, Git y Docker

Resumido por Gemini

17 de noviembre de 2025

Índice

1. Introducción al Despliegue de Aplicaciones Web (Tema 01)

El **despliegue de aplicaciones web** es el proceso crítico de transferir una aplicación desde el entorno de desarrollo a un entorno de producción, haciéndola accesible a los usuarios finales.

1.1. Objetivos del Despliegue

- **Accesibilidad:** Asegurar que los usuarios puedan acceder a la aplicación.
- **Estabilidad:** Garantizar que la aplicación funcione de manera consistente.
- **Escalabilidad:** Facilitar el crecimiento de la aplicación según la demanda.
- **Seguridad:** Proteger la aplicación y los datos de amenazas.
- **Automatización:** Minimizar la intervención manual para reducir errores.

1.2. Tipos de Arquitecturas Web

- **Arquitectura Monolítica:** Todo el código de la aplicación (UI, lógica de negocio, acceso a datos) está en un único proyecto o servidor. Es simple de desarrollar inicialmente, pero difícil de escalar y mantener.
- **Arquitectura de Microservicios:** La aplicación se divide en múltiples servicios pequeños e independientes. Cada microservicio maneja una funcionalidad específica y puede ser desarrollado, desplegado y escalado de forma independiente. Aumenta la complejidad de gestión pero ofrece gran flexibilidad y resiliencia.
- **Arquitectura Serverless:** Las operaciones del servidor son gestionadas completamente por un proveedor de nube (FaaS - Function as a Service). Los desarrolladores solo se preocupan por el código de la función. Ofrece escalabilidad automática y pago por uso.

1.3. Integración y Despliegue Continuo (CI/CD)

Son prácticas DevOps para automatizar y mejorar el ciclo de vida del desarrollo.

- **Integración Continua (CI):** Integrar cambios de código frecuentemente en un repositorio central. Cada integración dispara una **construcción (build)** y la ejecución de **pruebas automatizadas**. Su objetivo es detectar errores tempranamante.
- **Entrega Continua (Continuous Delivery):** Extiende la CI. Asegura que el código que pasa las pruebas automatizadas esté siempre en un estado desplegable. El despliegue a producción suele ser un paso manual.
- **Despliegue Continuo (Continuous Deployment):** Va un paso más allá. **Cada cambio que pasa todas las pruebas se despliega automáticamente a producción**.

2. Arquitectura Web y Fundamentos del Despliegue (Tema 03)

2.1. Introducción al Desarrollo Web

El despliegue web (deployment) es el proceso de poner aplicaciones a disposición de los usuarios en un entorno de producción. Sus objetivos clave son la **accesibilidad**, **estabilidad**, **escalabilidad** y **seguridad**. Un buen despliegue acelera el **Time-to-Market** (tiempo de lanzamiento), permite la **iteración rápida**, la **automatización** y requiere **documentación**.

El código se ejecuta en dos lugares:

- **Lado del Cliente (Frontend):** Se ejecuta en el navegador del usuario. Usa HTML, CSS y JavaScript para la interfaz e interactividad.
- **Lado del Servidor (Backend):** Se ejecuta en el servidor. Usa lenguajes como PHP, Java, Python, etc., para la lógica de negocio y el acceso a bases de datos.

2.2. Componentes de una Web

- **Frontend:** Es la parte visible con la que el usuario interactúa (diseño, maquetación).
- **Backend:** Es la lógica que se ejecuta en el servidor (acceso a BBDD, administración).
- **Backend Universal (Agnóstico):** Es la tendencia actual donde el backend expone su funcionalidad a través de **APIs** (Application Programming Interfaces). Esto permite que un único backend sirva a múltiples clientes (aplicaciones web, móviles, de escritorio, etc.).

La diferencia entre **Página Web** y **Aplicación Web** es clave:

- **Página Web:** Un documento. Puede ser **estática** (contenido fijo, HTML/CSS/JS, no necesita servidor web para verse localmente) o **dinámica** (contenido variable, requiere un servidor).
- **Aplicación Web:** Un software complejo que proporciona un **servicio** al usuario (ej. un email, una tienda). Siempre requiere un servidor.

2.3. Arquitecturas Web

El modelo fundamental es la **Arquitectura Cliente-Servidor**.

- **Ventajas:** Centralización del control, escalabilidad, portabilidad (independiente del SO cliente) y fácil mantenimiento (encapsulación).
- **Evolución:** Web 1.0 (estática) $\rightarrow$ Web 2.0 (social, contenido de usuario) $\rightarrow$ Web 3.0 (semántica, IA).

2.3.1. Modelos de Arquitectura Software

- **Monolítica:** Todos los componentes en un solo bloque. Fácil de iniciar, pero difícil de escalar y mantener.
- **De Capas:** Divide la app en capas lógicas (presentación, lógica de negocio, acceso a datos). Mejora la modularidad.

- **Microservicios:** Divide la app en servicios muy pequeños y autónomos. Cada uno se puede desarrollar, desplegar y escalar de forma independiente. (Ej. Netflix). Aumenta la flexibilidad pero también la complejidad de gestión.
- **Serverless:** El proveedor de nube gestiona toda la infraestructura. El desarrollador solo escribe funciones. Ofrece escalabilidad automática y pago por uso.
- **SOA (Arquitectura Orientada a Servicios):** Componentes como servicios independientes que se comunican. Promueve la interoperabilidad.
- **EDA (Arquitectura Orientada a Eventos):** Los componentes se comunican mediante eventos (emiten y se suscriben). Permite un gran desacoplamiento.

2.3.2. Modelo-Vista-Controlador (MVC)

Es un patrón de arquitectura que separa la lógica:

- **Modelo:** Representa los datos y la lógica de negocio (interactúa con la BBDD).
- **Controlador:** Responde a las acciones del usuario, pide datos al Modelo y los pasa a la Vista.
- **Vista:** Presenta los datos al usuario (la UI).

En el MVC tradicional del lado del servidor, cada petición del cliente implica un refresco de toda la página, lo que no es reactivo".

2.3.3. Patrones de Diseño (SOLID)

Principios para crear software robusto y mantenible:

- **S**RP (Principio de Responsabilidad Única).
- **O**CP (Principio Abierto/Cerrado).
- **L**SP (Principio de Sustitución de Liskov).
- **I**SP (Principio de Segregación de Interfaces).
- **D**IP (Principio de Inversión de Dependencias).

2.4. El Protocolo HTTP y HTTPS

HTTP (HyperText Transfer Protocol) es la base de la comunicación web. Es un protocolo **no orientado a la conexión** y **sin estado** (stateless), lo que significa que cada petición es independiente (las sesiones y cookies se usan para simular estado).

- **Flujo:** El cliente envía una **Petición HTTP** (Método, Ruta, Cabeceras) y el servidor devuelve una **Respuesta HTTP** (Código de Estado, Cabeceras, Cuerpo).
- **Cabeceras (Headers):** Metadatos. Ej. 'Accept' (qué formato acepta el cliente), 'Content-Type' (qué formato envía el servidor), 'Host' (dominio), 'Cache-Control' (reglas de caché).
- **Métodos/Verbos HTTP:** Definen la acción deseada.
  - **GET:** Obtener un recurso.
  - **POST:** Enviar datos para crear un nuevo recurso (datos en el cuerpo).
  - **PUT:** Actualizar/reemplazar un recurso existente.

    - **DELETE:** Borrar un recurso.
    - **HEAD:** Pide solo las cabeceras de un GET (para verificar existencia).
- **Códigos de Estado HTTP:** Indican el resultado de la petición.
    - **2xx (Éxito):** '200 OK' (Correcto).
    - **3xx (Redirección):** El recurso se ha movido.
    - **4xx (Error del Cliente):** '404 Not Found' (No encontrado), '403 Forbidden' (Prohibido).
    - **5xx (Error del Servidor):** El servidor falló.
- **HTTPS (Secure):** Es HTTP con cifrado. Usa **SSL/TLS** y **certificados digitales** emitidos por una **Autoridad de Certificación (AC)** para garantizar la privacidad y autenticidad.

2.5. Servicio Web y Comunicación con APIs

Un **Servicio Web** es una **API** (Application Programming Interface) que permite a las aplicaciones comunicarse entre sí a través de la web, usando HTTP y formatos como JSON o XML.

La diferencia clave: una **página web dinámica** devuelve HTML para un usuario, mientras que un **servicio web (API)** devuelve datos (JSON/XML) para otra aplicación.

2.5.1. Protocolos y Estilos de API

- **REST:** Estilo arquitectónico que usa verbos HTTP (GET, POST, etc.) para operaciones CRUD sobre recursos. Es simple, escalable y sin estado. Puede sufrir de .ºver-fetching"(traer datos de más) o ünder-fetching"(necesitar múltiples peticiones).
- **GraphQL:** Lenguaje de consulta para APIs. El cliente pide **exactamente** los datos que necesita en una sola petición POST. Muy eficiente y flexible.
- **gRPC:** Framework de RPC (Llamada a Procedimiento Remoto) de alto rendimiento. Usa HTTP/2 y Protocol Buffers (binario). Es extremadamente rápido, ideal para comunicación interna entre microservicios.
- **WebSocket:** Protocolo que permite comunicación **bidireccional y en tiempo real** sobre una conexión persistente. Ideal para chats, juegos, notificaciones en vivo.
- **SOAP:** Protocolo formal más antiguo, basado en XML. Es verboso pero muy estandarizado (usado en sistemas empresariales/legado).

2.5.2. JWT (JSON Web Token)

Es un estándar para crear **tokens de acceso** usados para autenticación y autorización. Es un objeto JSON compacto y firmado digitalmente que consta de tres partes:

1. **Header:** Tipo de token y algoritmo de firma.
2. **Payload:** Los çlaims.º datos (ej. ID de usuario, roles, fecha de expiración).
3. **Signature:** La firma, que garantiza que el token no ha sido alterado.

2.5.3. Arquitectura Netflix

Netflix usa **microservicios** desplegados en la nube (**AWS**). Usa su propia CDN (Open Connect).

- **Frontend:** React.js.
- **Backend:** Principalmente **Java** para la lógica de microservicios y **Python** para análisis de datos y Machine Learning (recomendaciones).
- **Bases de Datos: Cassandra (NoSQL)** para datos masivos como historial de visualización y **MySQL (SQL)** para datos transaccionales como facturación.

2.6. Funcionamiento de una Web Dinámica

- **Páginas Estáticas:** Contenido fijo (HTML/CSS/JS). El servidor solo envía los archivos. Son rápidas y simples.
- **Páginas Dinámicas:** Contenido generado en el servidor "sobre la marcha". Requieren un lenguaje de servidor (PHP, Python) y una BBDD.

Flujo de una Petición Dinámica:

1. El cliente (navegador) solicita una página.
2. El servidor web (ej. Apache) recibe la petición.
3. El servidor web pasa la petición a un **módulo de ejecución** (ej. el intérprete de PHP, o un servidor de aplicaciones como Tomcat).
4. El módulo ejecuta el script (ej. código PHP), que puede consultar una **base de datos**.
5. El script genera el **contenido HTML final**.
6. El servidor web envía esta página HTML resultante al cliente.

2.7. Funcionamiento y Configuración de Servidores

2.7.1. Servidores Web (Apache y Nginx)

Un **Servidor Web** es un programa que espera peticiones de clientes y responde con recursos (páginas, imágenes).

- **Apache:** Servidor web modular, de código abierto.
- **Directivas (Configuración):** Son las reglas.
  - `DocumentRoot`: Directorio raíz de los archivos web (ej. `/var/www/html/`).
  - `Listen`: Puerto de escucha (ej. `80`).
  - `ServerName`: Dominio del sitio (ej. `www.ejemplo.com`).
- **Virtual Hosts:** Permite que un solo servidor Apache aloje múltiples sitios web. Pueden ser **basados en nombre** (múltiples dominios en una IP) o **basados en IP** (una IP por sitio).

2.7.2. Servidores de Aplicaciones (Tomcat)

Es un software que va más allá de un servidor web, proporcionando un entorno para ejecutar lógica de negocio, balanceo de carga, etc..

- **Apache Tomcat:** Es un servidor de aplicaciones específico para Java; funciona como un **contenedor de Servlets y JSP**. Requiere tener el **JDK** (Java Development Kit) instalado.

2.7.3. Gestores de Bases de Datos

Software para almacenar y gestionar datos.

- **Relacionales (SQL):** MySQL, MariaDB, PostgreSQL, SQL Server.
- **NoSQL (No Relacionales):** MongoDB (orientado a documentos).

2.8. Seguridad y Monitorización

- **Seguridad:** Incluye **Autenticación** (verificar identidad) y **Control de Acceso** (qué puede hacer el usuario).
  - **Autenticación Basic:** Envía usuario/contraseña (codificado en Base64) en cada petición. Es simple pero inseguro sin HTTPS.
  - **Esquema Digest:** Usa hashes para mayor seguridad que Basic.
- **Monitorización (Logs):** Archivos de registro esenciales para el mantenimiento.
  - **CLF (Common Log Format):** Un formato estándar para los logs de acceso.
  - **Rotación de Logs:** Proceso automático (ej. con `logrotate`) para comprimir, archivar y eliminar logs antiguos para que no consuman todo el disco.

3. Control de Versiones con Git y GitHub

Git es un software de **control de versiones distribuido (DVCS)** diseñado por Linus Torvalds. A diferencia de los sistemas centralizados, cada desarrollador tiene una copia local completa del historial del proyecto.

3.1. Conceptos Clave de Git

- **Repositorio (Repository):** La base de datos que almacena todo el historial de cambios del proyecto.
- **Commit (Confirmación):** Una ïnstantánea"del proyecto en un momento específico. Cada commit tiene un hash SHA-1 único.
- **Rama (Branch):** Una línea de desarrollo independiente. Es un puntero a un commit. La rama principal suele llamarse `main` o `master`.
- **Área de Preparación (Staging Area / Index):** Una zona intermedia donde se seleccionan los cambios que se incluirán en el próximo commit.
- **Directorio de Trabajo (Working Directory):** La copia local de los archivos del proyecto donde se realizan las modificaciones.
- **HEAD:** Un puntero que apunta al último commit de la rama en la que te encuentras actualmente.

3.2. Ciclo de Vida de los Archivos

- **Modificado (Modified):** El archivo ha cambiado, pero no está en el staging area.
- **Preparado (Staged):** El archivo modificado ha sido añadido al staging area (listo para el commit).
- **Confirmado (Committed):** El cambio está guardado de forma segura en el repositorio local.

3.3. Comandos Esenciales de Git

3.3.1. Configuración e Inicialización

- `git config --global user.name "Tu Nombre"`: Configura tu nombre.
- `git config --global user.email "tu@email.com"`: Configura tu email.
- `git init`: Inicializa un nuevo repositorio Git en el directorio actual.
- `git clone <URL>`: Descarga (clona) un repositorio remoto.

3.3.2. Gestión de Cambios

- `git status`: Muestra el estado del directorio de trabajo y del staging area.
- `git add <archivo>`: Añade un archivo al staging area.
- `git add .`: Añade todos los cambios actuales al staging area.
- `git commit -m "Mensaje descriptivo"`: Guarda los cambios del staging area en un nuevo commit.

- `git diff`: Muestra las diferencias entre el trabajo actual y el staging area.
- `git diff --staged`: Muestra las diferencias entre el staging area y el último commit.

3.3.3. Historial y Deshacer Cambios

- `git log`: Muestra el historial de commits.
- `git revert <hash_commit>`: Crea un **nuevo commit** que deshace los cambios de un commit anterior. Es seguro para historial público.
- `git reset --hard <hash_commit>`: **(Peligroso)** Mueve el HEAD a un commit anterior, descartando todos los cambios posteriores en el directorio de trabajo y el historial. **Nunca usar en historial público**.
- `git rm --cached <archivo>`: Saca un archivo del staging area (deja de rastrearlo) pero lo mantiene en el disco.

3.3.4. Ramificación y Fusión

- `git branch <nombre_rama>`: Crea una nueva rama.
- `git checkout <nombre_rama>`: Cambia a una rama existente.
- `git checkout -b <nombre_rama>`: Crea una nueva rama y cambia a ella.
- `git merge <rama_a_fusionar>`: Integra los cambios de otra rama en la rama actual. Puede crear un "merge commit".
- `git rebase <rama_base>`: **(Avanzado)** Re-escribe la historia. Mueve los commits de tu rama actual .^encima"de la rama base, creando un historial lineal. **Regla de Oro: Nunca hagas rebase en ramas públicas/compartidas**.

3.3.5. Trabajo con Remotos (GitHub)

- `git remote add origin <URL>`: Conecta tu repositorio local a un remoto (llamado 'origin').
- `git push origin <rama>`: Sube tus commits locales al repositorio remoto.
- `git fetch origin`: Descarga los cambios del remoto, pero **no los fusiona** con tu trabajo local.
- `git pull origin <rama>`: Descarga los cambios del remoto y **automáticamente los fusiona** (equivale a fetch + merge).

3.3.6. Ignorar Archivos

- **.gitignore:** Un archivo de texto que le dice a Git qué archivos o directorios debe ignorar (ej. `node_modules/`, `*.log`, `/dist/`).

3.4. GitHub para Colaboración

- **Fork (Bifurcación):** Una copia personal de un repositorio de otro usuario en tu propia cuenta de GitHub. Te permite experimentar sin afectar el proyecto original.

- **Pull Request (PR) (Solicitud de Incorporación):** El mecanismo para proponer cambios. Abres una PR para pedirle al dueño del repositorio original que revise tus cambios (en tu fork o en tu rama) y los fusione (merge) en la rama principal. Es la base de la revisión de código.

3.5. Flujos de Trabajo (Workflows)

- **GitHub Flow:** Simple. `master`/`main` está siempre desplegable. Creas ramas (ej. `feature-login`) para cualquier cambio. Abres una PR para discutir y revisar. Una vez aprobada, se fusiona en `master` y se despliega.
- **GitFlow:** Complejo. Usa dos ramas permanentes: `master` (producción estable) y `develop` (integración de características). Usa ramas temporales para `feature-*`, `release-*` (preparar lanzamiento) y `hotfix-*` (arreglos urgentes en producción).

3.6. Documentación de Código

- **Markdown (.md):** Lenguaje de marcado ligero para archivos `README.md`. Usa `#` para títulos, `*` o `-` para listas, `**negrita**`, `*cursiva*`, `` `código en línea` ``, y ``` ``bloque de código``` ```.
- **JavaDoc (Java):** Comentarios `/** ... */` que generan documentación HTML. Usa etiquetas clave:
  - `@author`, `@version`
  - `@param <nombre><descripción>`: Describe un parámetro.
  - `@return <descripción>`: Describe el valor de retorno.
  - `@throws <excepción><descripción>`: Documenta una excepción.
  - `@see <referencia>`: Enlace a otra clase/método.
- **XMLdoc (C#):** Comentarios `/// ...` que usan XML.
  - `<summary>...</summary>`: Descripción breve.
  - `<remarks>...</remarks>`: Descripción larga.
  - `<param name="nombre»...</param>`: Describe parámetro.
  - `<returns>...</returns>`: Describe retorno.
  - `<exception cref="tipo»...</exception>`: Documenta excepción.

4. Virtualización con Contenedores (Docker)

Docker es una plataforma de código abierto para desarrollar, lanzar y ejecutar aplicaciones en **contenedores**. Permite empaquetar una aplicación y sus dependencias en un entorno aislado.

4.1. Conceptos Fundamentales

4.1.1. Contenedores vs. Máquinas Virtuales (MV)

- **Máquinas Virtuales (MV):** Virtualizan el **hardware**. Cada MV necesita un sistema operativo (SO) completo (Guest OS) sobre un Hipervisor. Son pesadas y lentas para arrancar.
- **Contenedores:** Virtualizan el **sistema operativo**. Comparten el kernel del SO anfitrión (Host OS) y solo contienen la aplicación y sus dependencias (bibliotecas, etc.). Son ligeros, rápidos y portables.

4.1.2. Arquitectura de Docker

- **Docker Engine (Motor):** La aplicación cliente-servidor.
  - **Docker Daemon (dockerd):** El servidor. Un proceso en segundo plano que gestiona los objetos de Docker (imágenes, contenedores, redes, volúmenes).
  - **API REST:** La interfaz que usan los programas para hablar con el Daemon.
  - **Docker Client (CLI):** La terminal (ej. `docker run...`). Es la interfaz de usuario que habla con la API REST.
- **Imágenes (Images):** Plantillas de **solo lectura** e inmutables. Contienen las instrucciones para crear un contenedor (ej. un Ubuntu con Apache). Se componen de capas.
- **Contenedores (Containers):** Una **instancia ejecutable** y modificable de una imagen. Es la imagen en funcionamiento.
- **Registro (Registry):** Un lugar para almacenar y compartir imágenes. **Docker Hub** es el registro público por defecto.

4.2. Imágenes y Dockerfile

Un **Dockerfile** es un archivo de texto plano (la receta") que define las instrucciones necesarias para **construir** una imagen de Docker.

4.2.1. Comando de Construcción

- `docker build -t mi-app:v1 .`
- `-t` (tag): Asigna un nombre y etiqueta (tag) a la imagen.
- `.`: Indica el **contexto de construcción** (el directorio actual, donde está el Dockerfile).

4.2.2. Instrucciones Fundamentales del Dockerfile

- `FROM <imagen>:<tag>`: **(Obligatoria)** Define la imagen base. Se recomienda evitar `:latest`.
- `RUN <comando>`: Ejecuta comandos **durante la construcción** de la imagen (ej. `RUN apt-get update`). Cada `RUN` crea una nueva capa.

- `COPY <origen><destino>`: Copia archivos/directorios desde el contexto (host) al sistema de archivos de la imagen. **Se prefiere `COPY` sobre `ADD`.**
- `ADD <origen><destino>`: Similar a `COPY`, pero puede manejar URLs y descomprimir archivos `.tar`.
- `WORKDIR <ruta>`: Establece el directorio de trabajo por defecto para las siguientes instrucciones (`RUN`, `CMD`, `COPY`).
- `EXPOSE <puerto>`: **Documenta** el puerto que el contenedor expone. **No publica el puerto**, solo es informativo.
- `CMD [.ejecutable", "param1"]`: Define el comando por defecto que se ejecutará **cuando el contenedor se inicie**. Solo puede haber un `CMD`. Si el usuario especifica un comando en `docker run`, este `CMD` se ignora.
- `ENTRYPOINT [.ejecutable", "param1"]`: Convierte la imagen en un ejecutable. El comando **siempre se ejecuta**. Lo que el usuario ponga en `docker run` se añade como argumento al `ENTRYPOINT`.
- `ENV <key>=<value>`: Establece variables de entorno.
- `VOLUME <ruta>`: Crea un punto de montaje para almacenamiento persistente.
- `USER <nombre>`: Especifica el usuario con el que se ejecutará el contenedor (para evitar usar **root**).
- `LABEL <key>=«value>"`: Añade metadatos a la imagen (ej. `maintainer`).
- `HEALTHCHECK`: Define un comando para verificar el estado de salud del contenedor.

4.2.3. Optimización de Imágenes (Buenas Prácticas)

- **Build Multi-Stage:** Usar varias sentencias `FROM` en un Dockerfile (ej. 'FROM ... AS builder'). Una etapa "builderçompila el código y una etapa final runner"(ligera) solo copia los binarios compilados. Reduce drásticamente el tamaño.
- **Minimizar Capas:** Encadenar comandos `RUN` con `&&` y
  .
- **Limpieza:** Eliminar caché y archivos innecesarios en la **misma capa** `RUN` (ej. `rm -rf /var/lib/apt/lists/*`).
- **Optimizar Caché:** Ordenar las instrucciones. Poner las que cambian menos (ej. instalación de dependencias) **antes** que las que cambian más (ej. `COPY . .`). Copiar `package.json` e instalar, y *luego* copiar el resto del código.
- **Usar .dockerignore:** Excluir archivos innecesarios (`.git`, `node_modules`, `dist/`) del contexto de build.
- **Usar Imágenes Base Ligeras:** Preferir `alpine` (ej. `node:7-alpine`) o `distroless`.
- **Un Contenedor, Un Proceso:** Cada contenedor debe hacer una sola cosa (ej. un contenedor para la web, otro para la BBDD).

4.3. Manejo de Contenedores (Comandos CLI)

- `docker run [OPTIONS] <imagen>[COMMAND]`: Crea y arranca un contenedor.

    - `-d` (detached): Modo separado (segundo plano).
    - `-it` (interactive + TTY): Modo interactivo (para shells).
    - `-p <puerto_host>:<puerto_contenedor>`: Publica (mapea) un puerto.
    - `-e <VAR>=<VALOR>`: Pasa variables de entorno.
    - `--name <nombre>`: Asigna un nombre legible.
    - `--rm`: Borra el contenedor automáticamente cuando se detiene.
    - `-v <origen>:<destino>`: Monta un volumen o bind mount.
    - `--mount type=...,source=...,target=...`: Sintaxis explícita para volúmenes/bind mounts.
    - `--net` o `--network <red>`: Conecta el contenedor a una red.
- `docker ps`: Lista contenedores **en ejecución**.
- `docker ps -a`: Lista **todos** los contenedores (incluidos detenidos).
- `docker stop <id_o_nombre>`: Detiene un contenedor.
- `docker start <id_o_nombre>`: Inicia un contenedor detenido.
- `docker rm <id_o_nombre>`: **Elimina** un contenedor (debe estar detenido).
- `docker exec -it <id_o_nombre>/bin/bash`: Ejecuta un comando (ej. una shell) en un contenedor **que ya está en ejecución**.
- `docker logs <id_o_nombre>`: Muestra la salida (logs) de un contenedor.
- `docker images` o `docker image ls`: Lista las imágenes locales.
- `docker rmi <id_o_nombre_imagen>` o `docker image rm ...`: **Elimina** una imagen.

4.4. Persistencia de Datos

Los contenedores son **efímeros**: sus datos internos se pierden cuando se eliminan. Para persistir datos, se usan:

- **Volúmenes (Volumes): (Opción preferida)**. Son áreas de almacenamiento gestionadas **por Docker**. Se crean con `docker volume create` o se dejan que Docker los cree. Sobreviven a la eliminación del contenedor. Ideal para bases de datos.
    - Uso: `-v nombre-volumen:/ruta/en/contenedor`.
    - Uso (mount): `--mount type=volume,source=mi-vol,target=/app-data`.
- **Bind Mounts: (Ideal para desarrollo)**. Enlaza un directorio o archivo **del host** dentro del contenedor. Los cambios en el host se reflejan instantáneamente en el contenedor.
    - Uso: `-v /ruta/en/host:/ruta/en/contenedor`.
    - Ejemplo (directorio actual): `-v "$PWD":/app`.

4.5. Redes (Networking)

- **bridge (puente):** La red por defecto (llamada `bridge`). Los contenedores están en una red interna privada y aislada. Se necesita `-p` para exponer puertos. La resolución DNS por nombre no funciona por defecto.
- **host:** El contenedor comparte la interfaz de red del host. No hay aislamiento de red.
- **Redes Definidas por Usuario:** `docker network create mired`. **(Recomendado)**. Los contenedores conectados a la misma red de usuario pueden comunicarse entre sí usando sus **nombres como DNS**.

4.6. Docker Hub (Registro)

- `docker pull <imagen>:<tag>`: Descarga una imagen.
- `docker login`: Inicia sesión en Docker Hub (u otro registro).
- `docker tag <imagen_local><usuario>/<repo>:<tag>`: Re-etiqueta una imagen local con el formato requerido para subirla.
- `docker push <usuario>/<repo>:<tag>`: Sube la imagen etiquetada al registro.
- **Builds Automatizados:** Docker Hub puede enlazarse a GitHub/Bitbucket y construir la imagen automáticamente cuando hay un `push` al repositorio, si hay un Dockerfile.

4.7. Docker Compose (Orquestación Local)

Herramienta para definir y ejecutar aplicaciones **multi-contenedor** (ej. WordPress + MariaDB) en un solo host.

- **docker-compose.yml:** Archivo YAML que define la aplicación.
- **Estructura Clave:**
  - `version: '3'`: Versión del formato.
  - `services:`: Define los contenedores (servicios).
  - `volumes:`: Declara volúmenes nombrados.
  - `networks:`: Declara redes de usuario.
- **Definición de un Servicio (ej. 'web'):**
  - `image: <nombre>`: Usa una imagen de Hub (ej. `wordpress:4.9.8`).
  - `build: .`: Construye desde un Dockerfile en el directorio actual.
  - `container_name: <nombre>`: Asigna un nombre fijo al contenedor.
  - `ports: ["8080:80"]`: Mapeo de puertos.
  - `volumes: ["./wordpress:/var/www/html", "data:/var/lib/mysql"]`: Monta un bind mount y un volumen.
  - `environment: ["VAR=VALOR"]`: Variables de entorno.
  - `depends_on: ["db"]`: Define el orden de inicio (espera a que 'db' inicie).
- **Comandos Principales:**

- `docker-compose up -d`: (Re)crea e inicia todos los servicios en segundo plano.
- `docker-compose stop`: Detiene los contenedores.
- `docker-compose down`: Detiene y **elimina** los contenedores y redes.
- `docker-compose down -v`: Igual, pero **también elimina los volúmenes**.
- `docker-compose ps`: Lista los servicios en ejecución.
- `docker-compose logs -f`: Muestra los logs en tiempo real.
- `docker-compose exec <servicio><cmd>`: Ejecuta un comando en un servicio.

4.8. Docker Swarm (Orquestación Distribuida)

Herramienta **nativa** de Docker para orquestar contenedores en un **clúster** de múltiples máquinas (nodos).

- **Arquitectura:** Maestro-Esclavo.
  - **Manager (Maestro):** Gestiona el clúster y delega tareas.
  - **Worker (Esclavo):** Ejecuta los contenedores (tareas).
- **Configuración:**
  - `docker swarm init`: Inicializa el clúster en el nodo Manager.
  - `docker swarm join --token ...`: Une un nodo Worker al clúster.
- **Servicios (Services):** Define las tareas a ejecutar.
  - **Replicados:** (Defecto) Ejecuta un número definido de réplicas (escalabilidad).
  - **Globales:** Ejecuta exactamente una tarea en **cada** nodo (ej. para monitoreo).
- **Stacks:** La forma de desplegar aplicaciones multi-contenedor en Swarm, usando un archivo `docker-compose.yml`.
  - Comando: `docker stack deploy -c docker-compose.yml mi_app`.
  - Se añade la sección `deploy:` al YAML para definir `replicas:`.
- **Redes Overlay:** Redes virtuales que permiten la comunicación transparente entre contenedores situados en diferentes hosts (nodos) del clúster.