Resumen de Examen: Despliegue, Git y Docker

Resumido por Gemini

17 de noviembre de 2025

Índice

1. 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.

1.1. 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.

1.2. 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.

1.3. Comandos Esenciales de Git

1.3.1. 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.

1.3.2. 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.

1.3.3. 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.

1.3.4. 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**.

1.3.5. 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).

1.3.6. Ignorar Archivos

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

1.4. 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.

1.5. 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).

1.6. 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.

2. 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.

2.1. Conceptos Fundamentales

2.1.1. 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.

2.1.2. 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.

2.2. Imágenes y Dockerfile

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

2.2.1. 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).

2.2.2. 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.

2.2.3. 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).

2.3. 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.

2.4. 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`.

2.5. 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**.

2.6. 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.

2.7. 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.

2.8. 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.