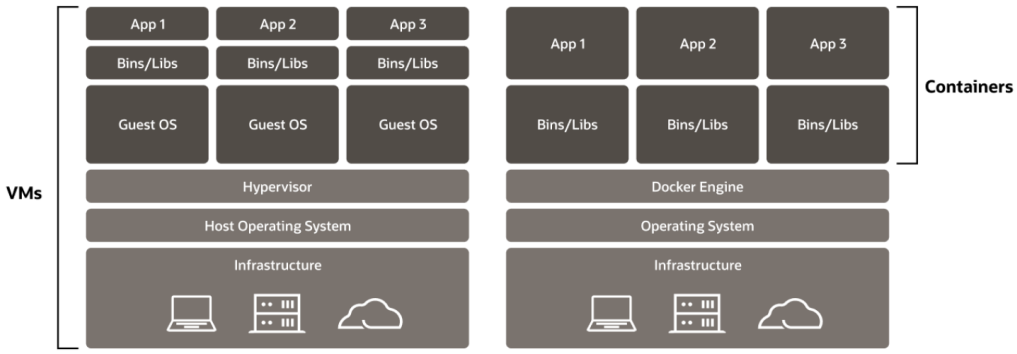
TEMA 4 – CONTENEDORES

1. Introducción (Docker)



Imagina que necesitas ejecutar dos aplicaciones en un equipo:

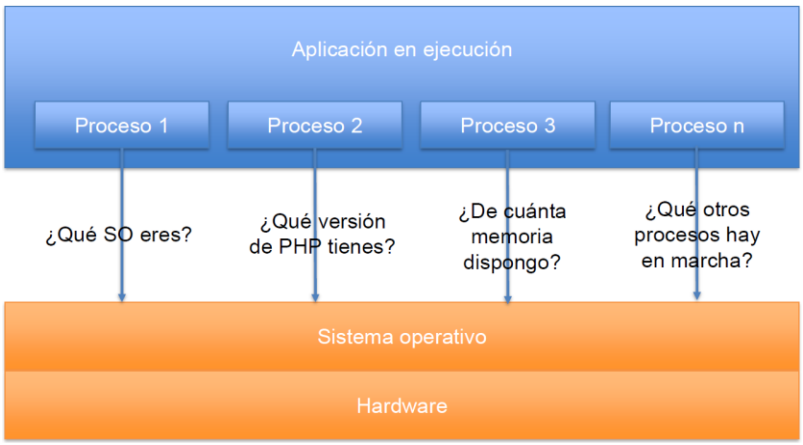
- Aplicación 1: Requiere la última versión de {PHP/Java/...}.
- Aplicación 2: Requiere una versión antigua de PHP {PHP/Java/...}, con características incompatibles con la última versión.

Posible solución: **Máquinas virtuales** separadas para cada aplicación, aparte del sistema operativo de la máquina huésped.

Posible problema: **Ineficiencia** (sistemas operativos separados, reserva de HW, etc.).

Idea de la "**containerización**": Utilizar solo un sistema operativo.

¿Cómo funciona Docker? En una aplicación normal (sin Docker)



Contenedor → La idea es interceptar esta comunicación entre los procesos de la aplicación y el sistema operativo y dar las respuestas que la aplicación espera para funcionar correctamente. La aplicación que necesita PHP v5 recibirá esa respuesta y la que necesita PHP v8 recibirá esa respuesta. **Se aíslan los entornos de ejecución, pero utilizando un solo S.O.**

¿Cómo funciona Docker?

Cada contenedor también tiene recursos reservados (memoria, usuarios, red, volúmenes...) para poder ejecutar una aplicación.

Este concepto es similar al de la máquina virtual, pero la diferencia está en que la manera de aislar estos recursos entre contenedores es **mucho más eficiente** que con una máquina virtual.

Los contenedores utilizan unas **características del kernel de Linux**. Estas son dos de las principales:

- **namespaces**: Cada **proceso** tiene un **espacio propio e independiente** para interactuar con la CPU (p.ej. procesos en namespaces distintos pueden utilizar un mismo puerto, como el 80).
- **cgroups**: Permiten **poner límites a los recursos** disponibles (p.ej. RAM, recursos de red, usuarios, espacio en disco duro, etc.).

El contenedor utiliza el kernel de Linux… ¿y si el SO es Win/Mac?

Se instala una **ligera máquina virtual Linux** que se ejecuta en **Hyper -V (Win)** o en **HyperKit (Mac)**, hipervisores que incorporan estos SOs para ejecución de máquinas virtuales.

En **Win** también se utiliza **WSL2** (Windows Subsystem for Linux), que permite **ejecutar Linux de manera nativa en Windows**.

También existen **contenedores nativos de Windows**, que **utilizan directamente su kernel**.

2. Instalación y conceptos básicos

Demonio (daemon): Proceso principal de Docker. **Gestiona los objetos de Docker** (imágenes, contenedores, redes, volúmenes, etc.) y se **comunica con otros demonios Docker**.

Cliente: Herramienta para **interaccionar con el demonio**.

Registro: **Almacén de imágenes**. Destaca **Docker Hub** como **registro público**.

Imagen: **Plantilla** para **crear un contenedor Docker**. Puede estar basada en otras imágenes.

Contenedor: **Instancia en ejecución de una imagen**. Parte de la imagen y admite diferentes opciones de configuración.

Volumen: Permiten la **persistencia de datos** cuando el **contenedor no está en ejecución**. También se utilizan para **compartir datos entre contenedores**.

Servicio: Define cómo se debe ejecutar un contenedor y permite escalar el contenedor a múltiples instancias (réplicas) distribuidas en un clúster de Docker.

3. Trabajo con imágenes y contenedores

Es aconsejable aprender a manejarse con la terminal o línea de comandos, ya que en muchas ocasiones (por ejemplo, servidores basados en Linux) no se dispone de interfaz gráfica. No obstante, en Windows puedes utilizar Docker Desktop.

4. Persistencia y volúmenes

Los **contenedores** son **instancias en ejecución de una imagen**. Por definición, cuando terminan su ejecución, todos los datos que no estuvieran en la imagen se perderán.

Los **volúmenes** son el **mecanismo** que emplea Docker para **almacenar datos de forma persistente**, de forma que estén disponible en futuras ejecuciones del contenedor.

En este esquema hay **3 formas en que el contenedor gestiona los datos**:

- **bind mount** (montaje de enlace).
- **volume** (volumen).
- **tmpfs mount** (montaje tmpfs).

Volume vs Bind mount

	Volumen	Bind mount
Gestión	Docker	Usuario
Ubicación	Ruta de Docker en sistema ficheros	Cualquier ruta en el equipo host
Portabilidad	Mas portable	Menos portable
Desempeño	++	+
Seguridad	Mayor seguridad	Menor seguridad
Configuración	No requiere rutas absolutas del host	Rutas absolutas del host
Uso compartido	Sencillo	Riegos

¿Cuándo usar volúmenes o montaje de directorio?

Volúmenes: Cuando el **contenedor necesita** una **base de datos**.

Montaje de directorio: Cuando tenemos el **código de una aplicación** (web o de otro tipo) en la **máquina host** y queremos que se **ejecute en el contenedor**.

NOTA: No es recomendable acceder a los volúmenes que cree Docker, especialmente cuando el contenedor esté en ejecución, ya que es el propio Docker quien los gestiona.

5. Gestión de múltiples contenedores

Docker permite gestionar contenedores de manera individual, pero es habitual que el trabajo de desarrollo requiera el uso de diferentes aplicaciones y servicios. Por ejemplo, para **desarrollar una aplicación web** puede ser necesario el **servidor web** (p.ej. Apache), una **base de datos** (p.ej. MariaDB) y una **aplicación de gestión de base de datos** (p.ej. phpMyAdmin). Esto supone gestionar varios contenedores por separado.

Docker Compose → Herramienta que **simplifica la definición y administración de aplicaciones multicontenedor**. Incorporada en Docker Desktop (Win).

Ventajas:

- ✓ Definición de contenedores, redes y volúmenes en un único archivo YAML (.yaml /.yml).
- ✓ Disminución de redundancias y comandos repetitivos.
- ✓ Definir entornos de desarrollo coherentes para trabajo en equipo.
- ✓ Definir dependencias entre servicios, asegurando el orden adecuado.
- ✓ Definir redes personalizadas para que los servicios se comuniquen.
- ✓ Definir volúmenes compartidos.

Nota sobre las rutas del servidor Apache en Linux

El **servidor Apache** puede utilizar por defecto **dos rutas distintas para alojar las aplicaciones web**: **/var/www/html** o **/usr/local/apache2/ htdocs**.

En **distribuciones Ubuntu o Debian** la ruta es habitualmente **/var/www/html**. En la **imagen php:8.2** -apache la ruta es habitualmente **/usr/local/apache2/ htdocs**, aunque algunas imágenes pueden estar basadas en distribuciones Ubuntu o Debian y entonces utilizar como ruta **/var/www/html**.

Docker Swarm y Kubernetes

La **gestión de múltiples contenedores** también recibe el nombre de **orquestación**. **Docker Compose** es una **herramienta de orquestación limitada a un solo host**, lo que hace que esté indicada para desarrollo, pruebas y producción a pequeña escala.

Por otro lado, **Docker Swarm (Docker)** y **Kubernetes (Google)** permiten **orquestar muchos contenedores en clústeres de múltiples nodos**, asegurando alta disponibilidad en entornos más complejos y de mayor escala.

Kubernetes es más complejo, pero ofrece **más funciones y flexibilidad**. Es la elección adecuada para **despliegues en producción a gran escala**.

6. Imágenes personalizadas

Una imagen personalizada permite **definir (y desplegar) un entorno específico para nuestra aplicación**. Las instrucciones para su creación deben estar en un fichero **Dockerfile**.

FROM → Indica la **imagen base a partir de la cual se construirá la nueva imagen**.

RUN → **Ejecuta comandos en la imagen** durante la construcción.

COPY o ADD → **Copia archivos o directorios desde el host a la imagen**.

WORKDIR → **Establece el directorio de trabajo dentro de la imagen**.

CMD → Define el **comando que se ejecutará cuando se inicie un contenedor** de la imagen.

EXPOSE → Declara el **puerto** en el que la **aplicación escucha**.

ENV → **Define variables de entorno dentro del contenedor**.

7. Conclusiones

Aislamiento y portabilidad: Docker permite **empaquetar aplicaciones con sus dependencias y versiones específicas**, así como desplegarlas en cualquier sistema con Docker instalado. Se pueden utilizar imágenes de terceros o construir las propias (Dockerfile).

Eficiencia en el uso de recursos: **Consume menos recursos que una máquina virtual**.

Simplificación del despliegue: **Docker Compose** permite **definir y gestionar múltiples contenedores** como un solo servicio y en **un solo fichero YAML**, facilitando así su despliegue.

Dependencias: **Docker Compose** permite definir el **orden de inicio de los contenedores**, para asegurar que las dependencias entre ellos estén bien gestionadas.

TEMA 5 – INTEGRACION Y DESPLIEGUE CONTINUO

Iteraciones

En cada iteración se integran los **cambios que van realizando los desarrolladores** y el técnico de operaciones puede ir revisando y dando feedback sobre los resultados del testeo y el despliegue a (pre)producción (staging).

DevOps: **Combinar elementos típicos de desarrollo y de operaciones**.

Componentes clave en integración y despliegue continuo:

- ✓ **Repositorios de código con control de versiones**: Git y GitHub/GitLab
- ✓ **Pruebas automatizadas**: PHPUnit (PHP), JUnit (Java), Mocha (Javascript), etc.
- ✓ **Herramientas de automatización**: Jenkins, Travis CI, GitHub Actions, GitLab CI/CD
- ✓ **Entornos de prueba y producción**: Servidores en la nube (AWS, Azure, etc.).

2. Integración continua

Práctica de desarrollo de software consistente en **integrar frecuentemente los cambios en el código a un repositorio común** (varias veces al día).

El objetivo es **detectar de forma automatizada y resolver de manera rápida** los posibles **errores** que haya en el código.

Pilares de la integración continua:

- **Control del código fuente** (control de versiones): Entorno colaborativo (trabajo en ramas, merging, resolución de conflictos, etiquetado de versiones, etc.).
- **Compilación (build) automática**: Compilar el código, gestionar dependencias y empaquetar el ejecutable (artefacto binario → carpetas "bin", "dist", "target").
- **Testeo automático**: Pruebas unitarias, de integración, funcionales, de eficiencia, etc.
- **Seguridad**: Identificar vulnerabilidades, como contraseñas débiles, configuraciones inadecuadas o dependencias obsoletas.

Compilación (build) automática

PHP es **interpretado** directamente por un **servidor web** (Apache/ Nginx) a través de un **módulo de PHP** o a través de la **línea de comandos** (CLI).

Por su parte, **Javascript** es **interpretado por el navegador**.

Compilación (build) automática en lenguajes interpretados → **Bundle (paquete)**

PHP → Existe la opción de **empaquetar toda la aplicación** en un único **fichero .phar** (el servidor Apache/Nginx debe estar configurado para ejecutar estos ficheros).

Hay que asegurarse que en el fichero quedan incluidas también las dependencias necesarias (lo puedes hacer mediante Composer).

Javascript → También **permite crear un bundle de la aplicación** mediante herramientas como **webpack, parcel o rollup**, utilizando además **npm o yarn** para la **gestión de las dependencias** del proyecto. El framework **Angular** utiliza **webpack**.

NOTA: El "**bundling" no es tan común en PHP**. Sin embargo, en **Javascript** es **aconsejable**, sobre todo en web frontend, ya que el código Javascript se descarga al cliente y esta estrategia supone menos solicitudes HTTP y una carga más eficiente.

Testeo

Utilizar herramientas y scripts para ejecutar pruebas automáticas del software.

Tipos de **pruebas automatizadas**:

- **Pruebas unitarias**: Verifican **unidades individuales de código**, como funciones o métodos.
- **Pruebas de integración**: Evalúan cómo **interactúan** diferentes **módulos** del software.
- **Pruebas funcionales**: Aseguran que las **funciones** del software cumplan con **requisitos especificados**.
- **Pruebas de sistema**: Comprueban el **sistema** en su totalidad para verificar su **comportamiento** en un **entorno completo**.

Testeo - Pruebas unitarias con PHPUnit

PHPUnit es un **framework de testeo** para realizar **pruebas unitarias en PHP**.

Su funcionamiento se basa en el uso de **aserciones (assertions)**, que son **métodos para verificar** que el **comportamiento del código es el esperado**. Una aserción **compara el resultado de la ejecución del código** con el **resultado esperado**. Si coinciden, toma el valor "true" y la prueba pasa. En caso contrario, el valor es "false" y no pasa la prueba.

Consideraciones de las pruebas unitarias:

- Solo se prueban **métodos públicos de cada clase**.
- **No** se debe hacer **uso de las dependencias de la clase bajo prueba**.
- El test **no debe implementar lógica de negocio** (if... else, for, etc.).
- La **estructura de un test unitario** debe ser:
  - **Preparación de los datos** de entrada.
  - **Ejecución** del test.
  - **Comprobación del test** (assertion), con solo una comprobación por test.

3. Despliegue continuo

Es el conjunto de **prácticas** que **permiten** el **proceso de entrega de cambios en el código**, desde la fase de integración hasta el entorno de producción, de manera **automática**.

Es una forma de asegurarse que nuestra aplicación está siempre a punto de pasar a producción, de manera rápida y fiable, ya sea una versión final o una versión de prueba.

- **Reducción de riesgos**: Cambios pequeños y frecuentes.
- **Mejora la calidad del software**: Pruebas automatizadas.
- **Acelera el paso a la fase de producción**: Comercialización.
- **Facilita el feedback** de los usuarios.

Entrega continua → A diferencia del despliegue continuo, en la entrega continua, el **despliegue no se realiza de manera automática** (por ejemplo, cada vez que se hace un push a GitHub), sino que se debe **iniciar manualmente**.

4. Servidores de automatización

Los servidores de automatización son **plataformas** que **facilitan la automatización de tareas repetitivas** relacionadas con los **procesos de integración continua** (CI) y **despliegue/entrega continuos** (CD).

Se basan en **scripts predefinidos** que cubren la **integración del código fuente**, la realización de **pruebas automáticas** y el **despliegue en diferentes entornos**.

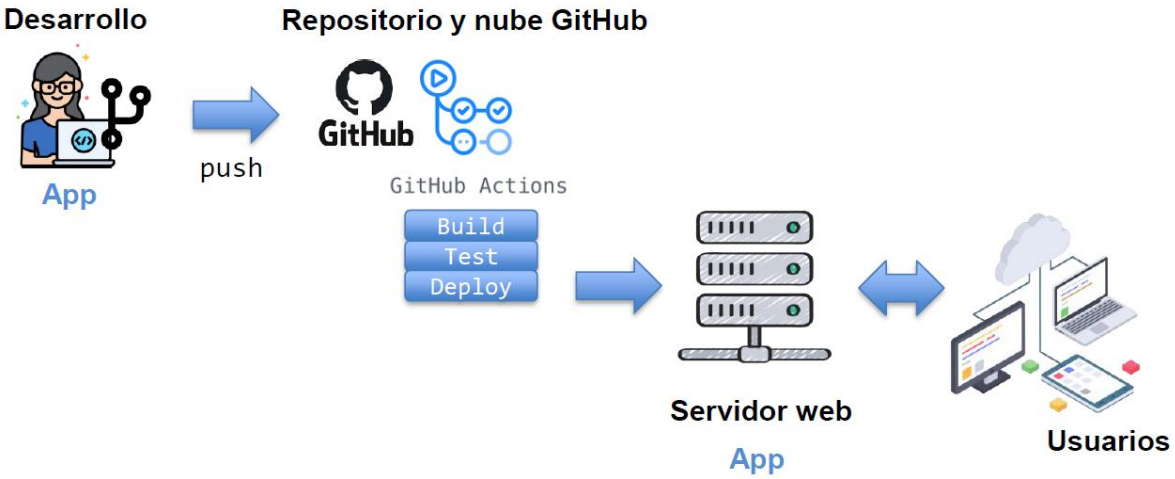
Ejemplos:

- Jenkins
- GitHub Actions
- GitLab CI/CD
- CircleCI
- Travis CI

	Jenkins	GitHub Actions
Infraestructura	Gestionada por **usuario**	Gestionada por GitHub
Configuración	UI y Jenkinsfile (**Groovy**)	Archivos **YAML**
Integración	Multiplataforma	Nativa con GitHub
Escalabilidad	Gestión del usuario	Gestión automática
Extensibilidad	Amplio soporte de plugins	Marketplace de acciones
Coste	Infraestructura del usuario	Gratis (repositorio publico)
Curva de aprendizaje	Alta	Media

GitHub Actions

GitHub Actions es una plataforma de CI/CD que permite **ejecutar flujos de trabajo** (compilación, testeo, despliegue, etc.) **cuando sucede algún evento** en el repositorio de GitHub.



5. Conclusiones

La integración y despliegue continuos (CI/CD) permiten **optimizar la entrega de valor en la cadena de desarrollo y producción** de software, automatizando flujos de trabajo que incluyen tareas de desarrollo y operaciones.

Las **partes fundamentales** en CI/CD son:

- ✓ **Repositorios de código** con control de versiones (Git/GitHub)
- ✓ **Compilación** (build) **automática** (bundles /Maven/ Make)
- ✓ Implementación de **pruebas automatizadas** (PHPUnit)
- ✓ Uso de **herramientas de automatización** (GitHub Actions)
- ✓ Uso de **entornos de desarrollo, pruebas y producción** (local/ MVs/AWS)

DESPLIEGUE DE APLICACIONES WEB

PREGUNTAS TEORIA

Que comparten los contenedores entre si → Kernel del sistema operativo.

Que tecnología del kernel de Linuys permite aislar procesos → Namespaces.

Para que sirven principalmente los cgroups → Limitar recursos.

En Windows, Docker normalmente se apoya en → WSL2.

Un contenedor es → Una instancia en ejecución de una imagen.

Que componente gestiona imágenes, redes y volúmenes → Docker Daemon.

Docker Hub es → Un registro de imágenes.

Cual NO es un objeto gestionado por Docker → Hipervisores.

Que contenedor lista contenedores en ejecución → Docker ps.

Que comando elimina una imagen → Docker rmi.

Que hace Docker system prune -a → Borra todo lo no usado.

Que opción de Docker run publica un puerto → Docker -p.

Que hace Docker exec -it<contenedor>/bin/sh → Abre una Shell interactive dentro del contenedor.

Que opción de Docker run asigna variables de entorno → Docker -e.

Por defecto, al detener un contendor → Los datos se pierden si no hay volumen.

Donde se almacena normalmente los volúmenes gestionados por Docker → En /var/lib/Docker/volumenes.

Un bind mount → Un mapeo directo a una carpeta del host.

Cuando es recomendable usar volúmenes → Bases de datos.

Que comando crea un volumen → Docker volumen créate.

Que comando borra volúmenes sin usar → Docker volumen prune.

Docker compose usa archivos en formato → YAML.

Para que sirve depens_on en Compose → Definir orden de arranque.

Que comando levanta servicios en segundo plano → Docker compose up -d.

Que comando detiene y elimina servicios de Compose → Docker compose down.

En Compose, cada service equivale a → Un contenedor.

Docker Compose es adecuado sobre todo para → Desarrollo y producción pequeña.

Que instrucción define la imagen base en un Dockerfile → FROM.

Que comando construye una imagen → Docker créate.

Que instrucción copia archivos al contenedor durante la construcción → COPY.

DESPLIEGUE DE APLICACIONES WEB

Para subir una imagen a Docker Hub, el orden correcto es → Build → tag → push.

Que caracteriza principalmente el enfoque Agile → Desarrollo iterativo e incremental.

Que problema pretende evitar con CI/CD → Riesgo acumulado por entregas tardías.

Ventaja clave de trabajar por iteraciones → Permite feedback temprano de operaciones.

Que es un "pipeline" en CI/CD → Una cadena automatizada de fases.

DevOps implica principalmente → Unificar desarrollo y operaciones.

La integración continua consiste en → Integrar cambios frecuentemente en un repositorio común.

El objetivo principal de la integración continua es → Detectar errores de forma temprana.

Cual NO es un pilar de la integración continua → Despliegue manual obligatorio.

El control de versión permite principalmente → Gestionar ramas, merges y conflictos.

En CI, la "build automática" sirve para → Compilar, gestionar dependencias y generar artefactos.

En un lenguaje compilado (Java, C#), la build implica → Generar un binario o artefacto.

En un lenguaje interpretado (PHP, JS), la build suele implicar → Generar un bundle o paquete.

Que tipo de pruebas verifica funciones o métodos individuales → Pruebas unitarias.

Las pruebas de integración verifican → La integración entre módulos.

Las pruebas funcionales buscan comprobar → El software cumple los requisitos.

Las pruebas de sistemas evalúan → Un sistema completo.

El despliegue continuo significa → Los cambios llegan a producción automáticamente.

Una ventaja del despliegue continuo → Menos riesgos con cambios pequeños.

Cual es un servidor de automatización CI/CD → Jenkins.

Una ventaja de GitHub Actions frente a Jenkis → Integración nativa con GitHub.