

Módulo Extra: Contenedorización de Aplicaciones Java con Docker




Objetivo:

Aprender a contenerizar una aplicación Spring Boot y ejecutarla en un entorno aislado utilizando Docker.



Contenidos:

1. ¿Qué es Docker?

- **Docker** es una herramienta que te permite:
- Empaquetar tu aplicación con **todo lo que necesita para ejecutarse** (Java, dependencias, configuraciones).
- Crear un **contenedor**: un entorno aislado que funciona igual en cualquier parte (tu PC, un servidor, la nube).
-  “Si funciona en mi contenedor, funciona en todas partes.”



Módulo Extra: Contenedorización de Aplicaciones Java con Docker

¿Qué es un contenedor?

Un **contenedor** es como una pequeña "caja virtual" que:

- Contiene tu aplicación
- Se comporta igual en cualquier sistema operativo
- Es liviana, rápida y desechable (se puede detener y borrar sin miedo)

A diferencia de una máquina virtual, un contenedor **comparte el sistema operativo**, por eso es más liviano y rápido.

¿Qué necesito para usar Docker?

- Tener instalado Docker Desktop
- Crear dos archivos clave:
 - `Dockerfile` → instrucciones para construir tu app
 - `.dockerignore` → qué archivos evitar copiar (como `.git`, `target/`, etc.)



Módulo Extra: Contenedorización de Aplicaciones Java con Docker

¿Cómo funciona en tu proyecto Spring Boot?

- Tú desarrollas normalmente tu app
- Docker **compila** el .jar
- Docker lo **empaqueta con una versión de Java**
- Crea una **imagen Docker** con todo listo
- Luego puedes **correr tu app como contenedor**, sin importar en qué máquina estés

Instalación de Docker Desktop

1. Ve al sitio oficial: <https://www.docker.com/products/docker-desktop>
2. Una vez instalado, reinicia y abre Docker Desktop.
3. Verifica desde PowerShell o CMD:
`docker --version`

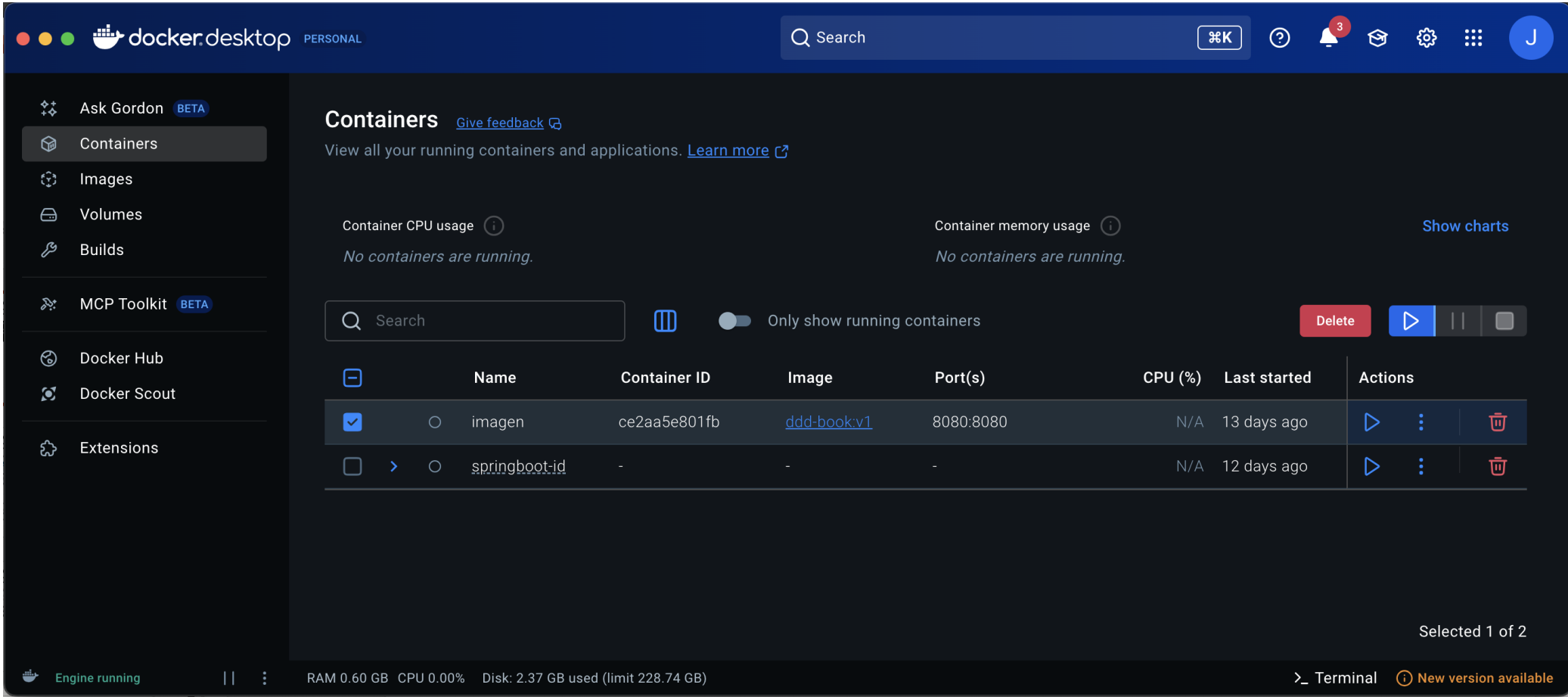


Módulo Extra: Contenedorización de Aplicaciones Java con Docker

```
[(base) desarrollo@MacBook-Pro-de-Ingrid ~ % docker --version  
Docker version 28.3.0, build 38b7060  
(base) desarrollo@MacBook-Pro-de-Ingrid ~ %
```



Módulo Extra: Contenedorización de Aplicaciones Java con Docker



The screenshot shows the Docker Desktop interface. The left sidebar contains navigation options: Ask Gordon (BETA), Containers (selected), Images, Volumes, Builds, MCP Toolkit (BETA), Docker Hub, Docker Scout, and Extensions. The main panel is titled 'Containers' and includes a search bar, a toggle for 'Only show running containers', and a 'Delete' button. Below this is a table of running containers.

	Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
<input checked="" type="checkbox"/>	imagen	ce2aa5e801fb	ddd-book.v1	8080:8080	N/A	13 days ago	
<input type="checkbox"/>	springboot-id	-	-	-	N/A	12 days ago	

At the bottom of the interface, a status bar shows 'Engine running', system resources (RAM 0.60 GB, CPU 0.00%, Disk: 2.37 GB used), and a 'Terminal' button. A 'New version available' notification is also present.



Módulo Extra: Contenedorización de Aplicaciones Java con Docker



¿Cómo empaquetar tu aplicación Spring Boot con un Dockerfile?



Estructura esperada del proyecto (como la que ya tienes):

```
springboot-security-demo-jwt/  
├─ src/  
├─ target/  
├─ pom.xml  
└─ Dockerfile ← este es el archivo que vamos a crear
```



Módulo Extra: Contenedorización de Aplicaciones Java con Docker

✓ Paso 1: Crea el archivo Dockerfile

En la raíz del proyecto, crea un archivo llamado Dockerfile (sin extensión) con el siguiente contenido:

```
# Usa una imagen base con Java
FROM openjdk:17-jdk-slim

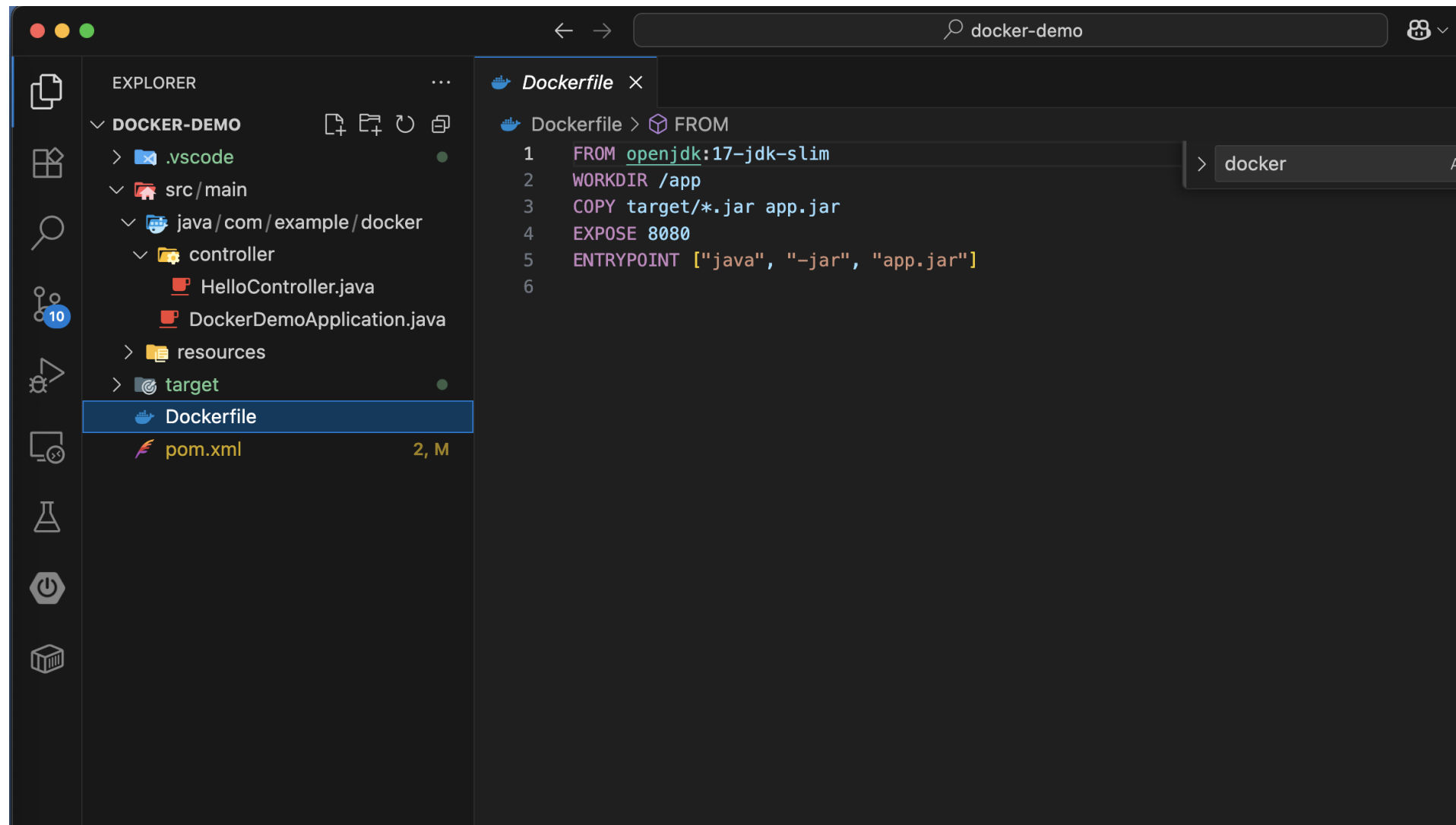
# Crea un directorio en el contenedor
WORKDIR /app

# Copia el archivo JAR generado por Spring Boot
COPY target/*.jar app.jar

# Expone el puerto 8080 (o el que uses)
EXPOSE 8080

# Comando para ejecutar la app
ENTRYPOINT ["java", "-jar", "app.jar"]
```





Módulo Extra: Contenedorización de Aplicaciones Java con Docker

✓ Paso 2: Compilar tu proyecto y generar el .jar

Abre terminal y desde la raíz del proyecto ejecuta:

```
./mvnw clean package -DskipTests
```

Esto generará algo como:

```
target/springboot-security-demo-jwt-0.0.1-SNAPSHOT.jar
```



Módulo Extra: Contenedorización de Aplicaciones Java con Docker

✓ Paso 3: Construir la imagen Docker

Ejecuta en terminal desde el mismo directorio:

```
docker build -t jwt-demo:v1 .
```

✓ Verás algo como: Successfully tagged jwt-demo:v1

✓ Paso 4: Ejecutar el contenedor

```
docker run -p 8080:8080 jwt-demo:v1
```

Si todo está bien, verás logs de Spring Boot y luego:

```
Tomcat started on port(s): 8080 (http)
```



Módulo Extra: Contenedorización de Aplicaciones Java con Docker

✓ Paso 5: Probar con Postman

Igual que antes:

- POST <http://localhost:8080/api/auth/login> → devuelve el token.
- GET <http://localhost:8080/api/private/data> → usa el token como Authorization: Bearer



Módulo Extra: Contenedorización de Aplicaciones Java con Docker

✅ ¿Qué hace Docker por ti en este proyecto?

1. **Compila tu aplicación Spring Boot y genera un .jar** con Maven (usando `mvn clean package`).

2. **Lo empaqueta dentro de una imagen Docker**, que es como una “caja” que incluye:

- El .jar compilado
- Una máquina virtual de Java (JDK o JRE)
- Un pequeño sistema operativo base (como Alpine o Debian)

🧳 **Resultado: una imagen portátil lista para ejecutar**

Con esa imagen puedes:

- Llevarla a **otra máquina** (Mac, Windows, Linux... no importa).
- Subirla a **DockerHub** y ejecutarla desde cualquier lugar.
- Desplegarla en **servidores, nubes o clusters Kubernetes**.
- Ejecutarla con un simple:

```
docker run -p 8080:8080 tu-nombre-de-imagen
```



Módulo Extra: Contenedorización de Aplicaciones Java con Docker

¿Y si no tuvieras Docker?

Tendrías que:

- Tener Java instalado en cada servidor
- Copiar el .jar manualmente
- Configurar puertos, entorno, etc.

Con Docker, **todo eso está resuelto dentro de la imagen**



Módulo Complementario: Desarrollo y despliegue de servicios Java en WildFly

Objetivo:

Comprender la estructura de aplicaciones Java EE para WildFly, desplegar microservicios en este servidor de aplicaciones, y comparar su enfoque con Spring Boot.

Contenidos Teóricos:

1. ¿Qué es WildFly?

- **Servidor de aplicaciones** de código abierto, antes conocido como JBoss AS.
- Soporta la especificación **Jakarta EE (antes Java EE)**.
- Ideal para aplicaciones empresariales en entornos gestionados.
- Soporta **JAX-RS, CDI, JPA, EJB, JMS**, entre otros.



Módulo Complementario: Desarrollo y despliegue de servicios Java en WildFly

2. Instalación de WildFly

Paso a paso:

- Descargar desde: <https://www.wildfly.org/downloads/>
- Descomprimir:
`unzip wildfly-XX.X.Final.zip`
- Iniciar servidor:
`./bin/standalone.sh`
- Acceder a consola web: <http://localhost:9990>



Módulo Complementario: Desarrollo y despliegue de servicios Java en WildFly

3. Estructura de una Aplicación Java EE para WildFly

Un proyecto puede generarse como .war (Web Application Archive). Ejemplo de estructura:

```
miapp/
├─ src/
│   └─ main/java/
│       └─ com/miempresa/servicios/
│           └─ SaludoResource.java
├─ src/main/webapp/WEB-INF/
│   └─ web.xml (opcional en versiones modernas)
└─ pom.xml
```



Módulo Complementario: Desarrollo y despliegue de servicios Java en WildFly

◆ 2. Arquitectura general

Componente	Propósito	Similar en Spring Boot
 JAX-RS	Desarrollo de servicios REST	@RestController, Spring Web
 JPA	Persistencia con ORM	Spring Data JPA
 EJB (opcional)	Lógica empresarial modular	Beans de servicio
 CDI	Inyección de dependencias estándar	Spring DI (@Autowired)
 WAR	Paquete estándar para despliegue	.jar ejecutable en Spring Boot
 Standalone/Domain	Modos de operación del servidor	N/A (Spring Boot siempre standalone)



Módulo Complementario: Desarrollo y despliegue de servicios Java en WildFly






◆ 3. Comparación: Spring Boot vs. WildFly (Similitudes)

Característica común	WildFly	Spring Boot
Ejecutan apps Java	✓	✓
Permiten crear APIs REST (JAX-RS / @RestController)	✓	✓
Usan anotaciones, inyección de dependencias, etc.	✓	✓
Pueden generar .war o .jar para desplegar	✓	✓
Pueden correr en servidores o contenedores Docker	✓	✓



Módulo Complementario: Desarrollo y despliegue de servicios Java en WildFly

◆ 3. Comparación: Spring Boot vs. WildFly (Diferencias clave)

Aspecto	WildFly (Java EE/Jakarta EE)	Spring Boot
 Arquitectura	Tradicional, basada en un servidor de aplicaciones externo (como WildFly)	Todo en un solo proceso ejecutable (.jar standalone)
 Modelo de despliegue	Empaquetas .war y lo despliegas en el servidor (como Tomcat, WildFly, etc.)	El servidor (Tomcat o Jetty) viene embebido en el .jar
 Complejidad	Más verboso, usa más configuración XML (aunque ha mejorado)	Más liviano, basado en convenciones y menos configuración
 Arranque	Necesitas iniciar WildFly y desplegar ahí	Se ejecuta directamente: <code>java -jar app.jar</code>
 Enfoque	Java EE/Jakarta EE (especificaciones estándar)	Spring Framework (ecosistema propio de herramientas)



Módulo Complementario: Desarrollo y despliegue de servicios Java en WildFly

◆ 4. ¿Cuál usar y cuándo?

Quiero...	Entonces usa...
Seguir estándares Java EE y trabajar con arquitecturas clásicas (JPA, EJB, CDI, etc.)	WildFly / Java EE / Jakarta EE
Arrancar rápido, usar menos configuración y trabajar en microservicios modernos	Spring Boot



Módulo Complementario: Desarrollo y despliegue de servicios Java en WildFly

Ejemplo:

<https://github.com/Joselota/Relatorias/tree/main/CursoJavaAvanzado/wildfly-demo>

