Práctica 3.1: Despliegue de una API Restful con Jakarta (Java) mediante Maven en WildFly

1. Introducción

Aún a día de hoy muchísimos proyectos que siguen utilizando Java 8 por cuestiones legacy y de dificultad de migración. La última versión de JDK, la 24, aún no está ampliamente extendida. Estas dos razones provocan que Java 17 se esté convirtiendo en una alternativa bastante atractiva y plausible ya que proporciona un buen equilibrio entre compatibilidad y funcionalidades del lenguaje.

2. REST

REST (Representational State Transfer) es un estilo arquitectónico para diseñar *servicios web*. Se basa en la idea de que los recursos de un sistema (por ejemplo, datos o servicios) deben ser identificados mediante URLs, y los métodos HTTP estándar (GET, POST, PUT, DELETE) deben usarse para interactuar con esos recursos. A diferencia de otros protocolos como SOAP, REST es ligero, simple y fácil de usar.

Los principios clave de REST incluyen:

- ***Stateless* (sin estado)**: Cada solicitud HTTP es independiente y contiene toda la información necesaria para ser procesada. El servidor no mantiene el estado entre peticiones.
- ***Client-Server*:** La separación de las responsabilidades del cliente y el servidor. El cliente se encarga de la interfaz y el servidor de la lógica y almacenamiento de datos.
- ***Cacheable***: Las respuestas a las solicitudes deben ser etiquetadas como *cacheable* o no *cacheable*.
- ***Uniform Interface* (interfaz uniforme)**: Definir un conjunto común de reglas para interactuar con los recursos de manera consistente (por ejemplo, mediante las operaciones estándar de HTTP).
- ***Layered System* (sistema en capas)**: Los sistemas pueden estar organizados en capas (por ejemplo, proxy, balanceadores de carga), sin que el cliente necesite saber en qué capa está interactuando.

2.1 ¿Cómo es REST en el ecosistema Java?

En el ecosistema Java, REST se implementa principalmente utilizando JAX-RS (*Java API for RESTful Web Services*), una especificación que forma parte de Java EE (ahora ***Jakarta EE***) y proporciona una serie de APIs y herramientas para crear servicios web RESTful.

JAX-RS permite a los desarrolladores definir recursos (entidades) que estarán disponibles a través de HTTP mediante anotaciones. Estas anotaciones mapean métodos de la clase Java a las operaciones HTTP (como @GET, @POST, @PUT, @DELETE), y facilitan la manipulación de datos en formatos como JSON o XML.

2.2 ¿Qué es Jakarta RESTful Web Services (Jakarta REST)?

Jakarta RESTful Web Services es la continuación de JAX-RS bajo la nueva iniciativa Jakarta EE, después de que Oracle cediera Java EE a la Fundación Eclipse. En esta transición, los paquetes Java EE fueron renombrados a Jakarta EE, y JAX-RS se renombró como *Jakarta RESTful Web Services*.

La especificación de Jakarta REST (anteriormente JAX-RS) proporciona un conjunto de APIs y herramientas para desarrollar servicios web RESTful en el ecosistema Jakarta EE. Con Jakarta REST, los desarrolladores pueden crear servicios que exponen datos y funcionalidad a través de HTTP, y que son accesibles desde cualquier cliente que soporte HTTP, como navegadores, aplicaciones móviles, o incluso otros servicios web.

2.3 WildFly

WildFly es un servidor de aplicaciones Java de código abierto, anteriormente conocido como *JBoss AS* (*Application Server*). Es parte del ecosistema Jakarta EE (anteriormente Java EE) y está diseñado para ejecutar aplicaciones Java empresariales, incluidas aplicaciones web, servicios RESTful, y microservicios.

WildFly es conocido por ser ligero, rápido y altamente configurable. Soporta varias tecnologías como Servlets, JSP, EJB, JPA, JMS, y CDI, permitiendo el desarrollo de aplicaciones empresariales robustas y escalables. Además, es altamente modular, lo que le permite cargar solo los componentes necesarios para una aplicación específica, optimizando el uso de recursos.

En resumen, WildFly es un servidor de aplicaciones Java eficiente y flexible que facilita el desarrollo y despliegue de aplicaciones Java empresariales y servicios web.

3. Instalación de Maven y WildFly

3.1 Instalación de Maven

Para instalar Maven tenemos varias opciones, podéis consultarlas aquí. La primera, recomendada, es mucho más sencilla y automatizada (establece todos los *paths* y variables de entorno), aunque con la segunda se podría conseguir un paquete más actualizado.

Ambos métodos vienen explicados aquí

Si optamos por seguir el primer método, el más sencillo, vemos que es tan simple como actualizar los repositorios:

```
sudo apt update
```

E instalar Maven:

```
sudo apt install maven
```

Para comprobar que todo ha ido correctamente, podemos ver la versión instalada de Maven:

```
mvn --v
```

3.2 Instalación del servidor de aplicaciones WildFly

1. Instalar Java Development KIt (JDK)

El kit de desarrollo de Java incluye la especificación del lenguaje Java y la especificación de la máquina virtual Java, así como proporciona la *Standard Edition* de la interfaz de programación de aplicaciones Java. Se trata de una dependencia vital para poder desarrollar aplicaciones Java.

Podemos elegir instalar;

- Instalar OpenJDK
- Instalar Java SE Development Kit

Para ello:

```
sudo apt update

sudo apt -y install default-jdk
```

A fecha de realización de estos apuntes, WildFly soporta Java 21, podéis confirmar vuestra versión instalada así:

```
sergio@Debian-DAW:~/rest-java17-wildfly$ java --version
openjdk 21.0.8 2025-07-15
OpenJDK Runtime Environment (build 21.0.8+9-Debian-1)
OpenJDK 64-Bit Server VM (build 21.0.8+9-Debian-1, mixed mode, sharing)
```

2. Descargar el *tarball* (archivo comprimido) de WildFly

Para bajarnos la última versión de WildFly disponible, utilizaremos las herramientas *curl* y *wget*. Si no las tuviéramos instaladas, primero lo haremos:

```
sudo apt install curl wget
```

Bajamos el archivo comprimido con la última versión de WildFly:

```
WILDFLY_RELEASE=$(curl -s
https://api.github.com/repos/wildfly/wildfly/releases/latest|grep tag_name|cut -d '"'
-f 4)
```

```
wget
https://github.com/Wildfly/wildfly/releases/download/${WILDFLY_RELEASE}/wildf
ly-${WILDFLY_RELEASE}.tar.gz
```

Descomprimimos:

```
tar xvf wildfly-${WILDFLY_RELEASE}.tar.gz
```

Movemos el contenido descomprimido al directorio /opt:

```
sudo mv wildfly-${WILDFLY_RELEASE} /opt/wildfly
```

3. Configurar WildFly como un servicio más de Systemd

Para facilitar la administración de WildFly, lo configuraremos con un servicio más del sistema. Para empezar añadimos el usuario de sistema ***wildfly***, que es el que se encargará del servicio:

```
sudo groupadd --system wildfly

sudo useradd -s /sbin/nologin --system -d /opt/wildfly  -g wildfly wildfly
```

Creamos el directorio que contendrá las configuraciones de WildFly:

```
sudo mkdir /etc/wildfly
```

Copiamos el archivo del servicio, así como los scripts de inicio al directorio `/opt/wildfly/docs/contrib/scripts/systemd/`:

```
sudo cp /opt/wildfly/docs/contrib/scripts/systemd/wildfly.conf /etc/wildfly/

sudo cp /opt/wildfly/docs/contrib/scripts/systemd/wildfly.service
/etc/systemd/system/

sudo cp /opt/wildfly/docs/contrib/scripts/systemd/launch.sh /opt/wildfly/bin/

sudo chmod +x /opt/wildfly/bin/launch.sh
```

Le damos los permisos adecuados al directorio /opt/wildfly:

```
sudo chown -R wildfly:wildfly /opt/wildfly
```

Recargamos los servicios de systemd (sistema que se encarga de manejar los servicios en nuestra Debian):

```
sudo systemctl daemon-reload
```

Iniciamos el servicio WildFly y lo habilitamos para que se inicie automáticamente al arrancar la máquina:

```
sudo systemctl start wildfly

sudo systemctl enable wildfly
```

Comprobamos que se etá ejecutando:

```
sergio@Debian-DAW:~$ sudo systemctl status wildfly
● wildfly.service - The WildFly Application Server
     Loaded: loaded (/etc/systemd/system/wildfly.service; enabled; preset: enabled)
     Active: active (running) since Thu 2025-09-11 17:47:44 CEST; 53s ago
 Invocation: 31187a59a6d3439fa091deab89c935a7
   Main PID: 3729 (launch.sh)
      Tasks: 56 (limit: 4619)
     Memory: 244.1M (peak: 253.9M)
        CPU: 13.296s
     CGroup: /system.slice/wildfly.service
             ├─3729 /bin/bash /opt/wildfly/bin/launch.sh standalone standalone.xml 0.0.0.0
             ├─3730 /bin/sh /opt/wildfly/bin/standalone.sh -c standalone.xml -b 0.0.0.0
             └─3851 java "-D[Standalone]" "-Djdk.serialFilter=maxbytes=10485760;maxdepth=128;maxa

Sep 11 17:47:44 Debian-DAW systemd[1]: Started wildfly.service - The WildFly Application Server.
```

4. Crear usuarios de WildFly

WildFly tiene medidas de seguridad activadas para las interfaces de administración. Por este motivo necesitamos crear un usuario que sea capaz de acceder a la consola de administración de forma remota o usando el CLI.

WildFly nos proporciona un archivo para administrar los usuarios, así que en primer lugar y para crear un usuario:

```
sudo /opt/wildfly/bin/add-user.sh
```

Cuando nos pregunte qué tipo de usuario queremos crear:

```
What type of user do you wish to add?

a) Management User (mgmt-users.properties)

b) Application User (application-users.properties)

(a):  a
```

Le dais el nombre de vuestro usuario (vuestro nombre, no el mío):

```
Enter the details of the new user to add.

Using realm 'ManagementRealm' as discovered from the existing property files.

Username : sergio
```

Y le dais una contraseña:

```
Password recommendations are listed below. To modify these restrictions edit the add-
user.properties configuration file.

The password should be different from the username

The password should not be one of the following restricted values {root, admin,
administrator}

The password should contain at least 8 characters, 1 alphabetic character(s), 1
digit(s), 1 non-alphanumeric symbol(s)

Password : <Vuestro Password>

Re-enter Password :  <Confirmar vuestro Password>
```

Y finalizáis la creación:

```
What groups do you want this user to belong to? (Please enter a comma separated list,
or leave blank for none)[  ]: <Enter>

About to add user 'sergio' for realm 'ManagementRealm'

Is this correct yes/no? yes

Added user 'sergio' to file '/opt/wildfly/standalone/configuration/mgmt-
users.properties'

Added user 'sergio' to file '/opt/wildfly/domain/configuration/mgmt-users.properties'

Added user 'sergio' with groups  to file '/opt/wildfly/standalone/configuration/mgmt-
groups.properties'

Added user 'sergio' with groups  to file '/opt/wildfly/domain/configuration/mgmt-
groups.properties'

Is this new user going to be used for one AS process to connect to another AS
process?

e.g. for a slave host controller connecting to the master or for a Remoting
connection for server to server EJB calls.

yes/no? yes
```

```
To represent the user add the following to the server-identities definition
```

5. Accediendo a la consola de administración de WildFly

Para comprobar que todo está correcto, vamos a cerciorarnos de que podemos acceder a la consola de administración.

Para ser capaces de ejecutar los scripts de WildFly desde nuestra sesión actual, añadimos el directorio `/opt/wildfly/bin/` a nuestro PATH:

```
cat >> ~/.bashrc <<EOF

export WILDFLY_BIN="/opt/wildfly/bin/"

export PATH=\$PATH:\$WILDFLY_BIN

EOF
```

Y para no tener que reiniciar nuestra sesión actual SSH, aplicamos los cambios en la sesión actual así:

```
source ~/.bashrc
```

Y validamos que nos podemos conectar sin problemas:

```
admin@ip-172-31-89-217:~$ jboss-cli.sh --connect

Authenticating against security realm: ManagementRealm

Username: sergio

Password:

[standalone@localhost:9990 /] exit
```

6. Accediendo a la consola de administración desde la interfaz web

Debemos modificar el archivo `/opt/wildfly/bin/launch.sh` y dejarlo así:

```
#!/bin/bash


if [ "x$wildfly_HOME" = "x" ]; then

    wildfly_HOME="/opt/wildfly"

fi


if [[ "$1" == "domain" ]]; then

    $wildfly_HOME/bin/domain.sh -c $2 -b $3
```

```
else
    $wildfly_HOME/bin/standalone.sh -c $2 -b $3 -bmanagement=0.0.0.0
fi
```

Reiniciamos el servicio tras el cambio:

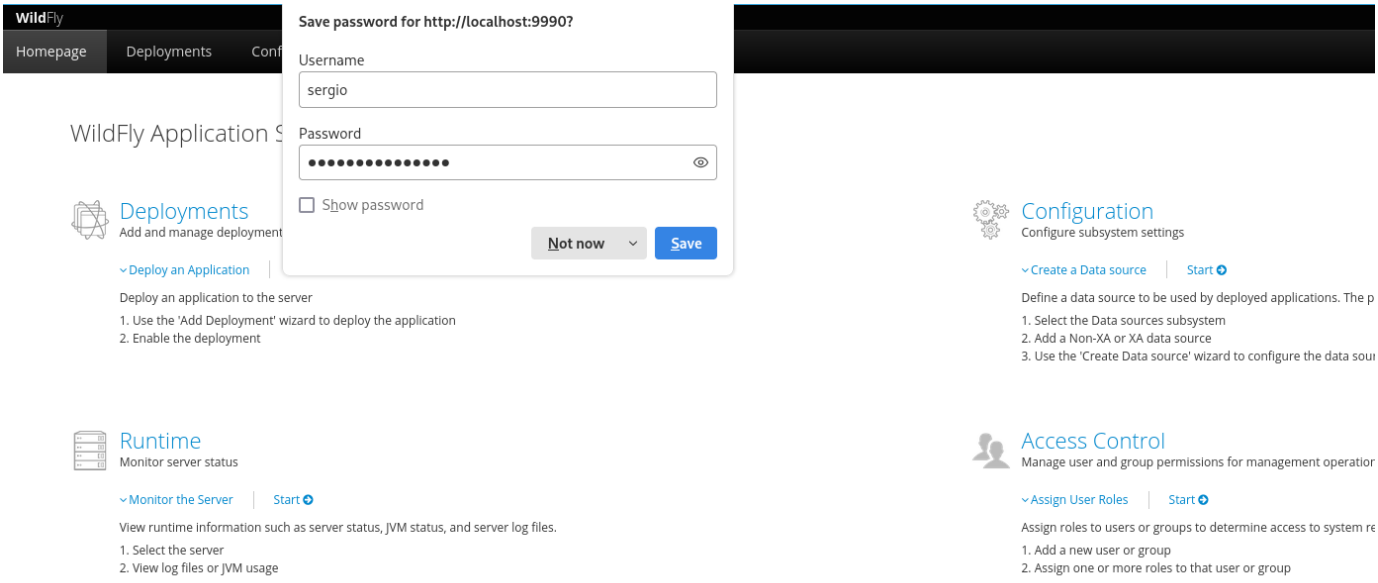
```
sudo systemctl restart wildfly
```

Comprobamos su estado:

```
systemctl status wildfly
```

Y tras esto, debemos ser capaces de acceder a nuestra interfaz web de administración en la URL `http://maqVIR:9990` y podremos autenticarnos con el usuario creado anteriormente.

Recuerda que en el grupo de seguridad deberás habilitar el acceso al puerto 9990 para no tener problemas.



4. Configuración de la carpeta de proyecto para Maven

En teoría, si desarrollarais una aplicación de 0, deberíais crearos una carpeta con el nombre que queráis y que contendrá todos los archivos y directorios necesarios para la aplicación y para poder desplegarla con Maven. Una vez creada la carpeta, dentro de ella y como ya sabemos de la parte teórica, deber ir el archivo de configuración pom.xml. No obstante, este módulo se centra en el proceso de despliegue y no en el de desarrollo.

Así las cosas, supondremos que la aplicación que hemos desarrollado es la que podéis encontrar en este link:

https://github.com/improvess/rest-java17-wildfly

Deberéis clonaros el repositorio y ubicar el pom.xml en el lugar correcto dentro del directorio.

Dentro del POM debemos establecer la dependencia de Jakarta API tal que así:

```
<dependencies>


    <dependency>
        <groupId>jakarta.platform</groupId>
        <artifactId>jakarta.jakartaee-api</artifactId>
        <version>10.0.0</version>
        <scope>provided</scope>
    </dependency>


</dependencies>
```

Por defecto, Maven compila los fuentes con Java 1.8 así que debemos decirle explícitamente en el pom.xml qué versión de Java debe utilizar:

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>17</java.version>
    <maven.compiler.source>${java.version}</maven.compiler.source>
    <maven.compiler.target>${java.version}</maven.compiler.target>
    <failOnMissingWebXml>false</failOnMissingWebXml>
</properties>
```

Si usamos JDK 17 con versiones antiguas del plugin de compilación de Maven nos dará un error como este:

```
Fatal error compiling: error: release version 17 not supported
```

Pero podemos solventarlo fácilmente indicando en el pom.xml que se utilice la última versión de dicho plugin:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.12.1</version>
    <configuration>
        <release>${java.version}</release>
    </configuration>
</plugin>
```

En este mismo sentido, para hacer un *build* del *war* de una aplicación, debemos indicar una versión adecuada del plugin encargado de ello:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-war-plugin</artifactId>
    <version>3.4.0</version>
</plugin>
```

Y, por último, configuramos el despliegue y retirada de una aplicación en WildFly usando Maven:

```
org.wildfly.plugins wildfly-maven-plugin 4.2.2.Final
```

Quedando el pom.xml definitivo tal que así:

```
<project  xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">


    <modelVersion>4.0.0</modelVersion>


    <groupId>com.mycompany.myproject</groupId>
    <artifactId>modulename.backend</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>war</packaging>
```

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>17</java.version>
    <maven.compiler.source>${java.version}</maven.compiler.source>
    <maven.compiler.target>${java.version}</maven.compiler.target>
    <failOnMissingWebXml>false</failOnMissingWebXml>
</properties>

<dependencies>
    <dependency>
        <groupId>jakarta.platform</groupId>
        <artifactId>jakarta.jakartaee-api</artifactId>
        <version>10.0.0</version>
        <scope>provided</scope>
    </dependency>
</dependencies>

<build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.12.1</version>
            <configuration>
                <release>${java.version}</release>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
            <version>3.4.0</version>
        </plugin>
        <plugin>
            <groupId>org.Wildfly.plugins</groupId>
            <artifactId>Wildfly-maven-plugin</artifactId>
            <version>4.2.2.Final</version>
        </plugin>
```

```
        </plugins>
    </build>
</project>
```

5. Breve explicación del código

Definición de la aplicación REST con una clase de Java:

```
package com.mycompany.myproject.module;


import jakarta.ws.rs.ApplicationPath;
import jakarta.ws.rs.core.Application;


@ApplicationPath("/api")
public class RestApplication extends Application {
  // NOP
```

Esto básicamente establece /api como la raíz de la API RESTful. Para definir un servicio REST:

```
package com.mycompany.myproject.module.services;
//Muchos imports…


@Path("myservice")
public class MyService {


 @GET
 @Path("/hello")
 public Response sayHello(@Context HttpServletRequest request) {
  Response response = Response.ok("hello!").build();
  return response;
 }


 @GET
 @Path("/pojo/list")
 @Produces({ MediaType.APPLICATION_JSON })
 public List<Pojo> getAll() {
  return Arrays.asList(new Pojo(1, "LALALA"), new Pojo(2, "LElele"));
```

```
}

@GET
@Path("/pojo/find/{id}")
@Produces({ MediaType.APPLICATION_JSON })
public Pojo find(@PathParam("id") Integer id) {
 return new Pojo(1, UUID.randomUUID().toString());
}

@POST
@Path("/pojo/new")
@Consumes({ MediaType.APPLICATION_JSON })
public Response create(Pojo pojo) {
 System.out.println("Creating new Pojo: " + pojo);
 return Response.status(201).build();
}

@PUT
@Path("/pojo/update")
@Consumes({ MediaType.APPLICATION_JSON })
public Response update(Pojo pojo) {
 System.out.println("Updating the Pojo: " + pojo);
 return Response.status(204).build();
}

@DELETE
@Path("/pojo/remove")
public Response delete(@QueryParam("id") Integer id) {
 System.out.println("Removing pojo with id: " + id);
 return Response.status(204).build();
}

}
```

En esencia, se están definiendo seis *endpoints* y los hace corresponder con los métodos HTTP, GET, PUT, POST y DELETE.

6. Despliegue

6.1 Build

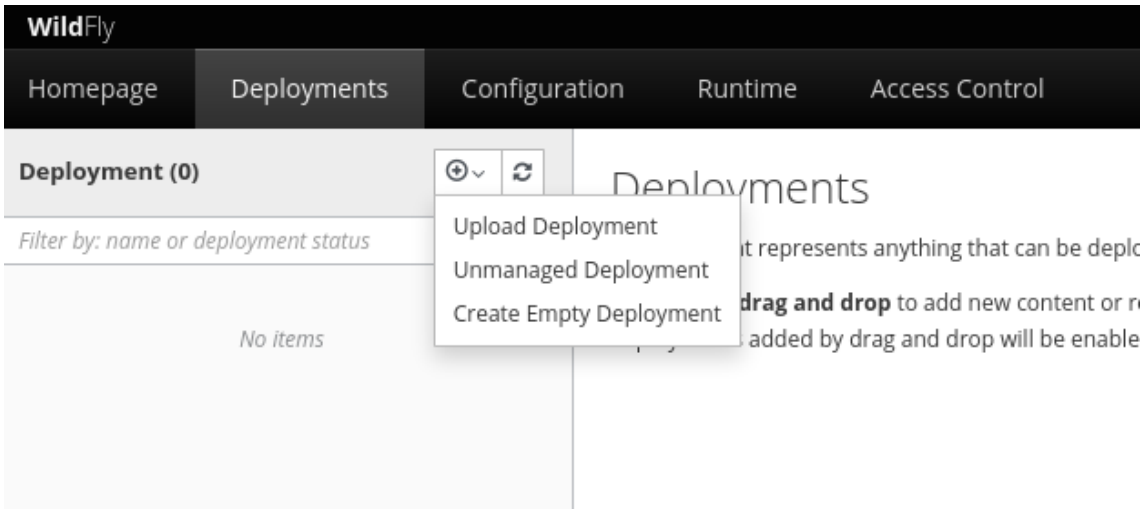
Para decirle a Maven que haga el *build* del archivo *war*, usaremos el siguiente comando, para limpiar restos de compilaciones anteriores y realizar la nueva de forma "limpia":

```
sudo mvn clean package
```

Esto generará el paquete **`target/module.backend-0.0.1-snapshot.war.`** Veamos ahora cómo desplegarlo en WildFly.

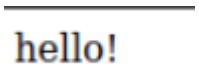
6.2 Despliegue en WildFly mediante interfaz gráfica

Para el despliegue mediante interfaz gráfica, deberemos iniciar sesión en dichar interfaz: http:IP-VM:9990 e ir a “Deployments → Upload Deployment



Aquí subiremos el archivo .war almacenado en el directorio /target del proyecto. Para confirmar que la aplicación está corriendo podemos hacer una llamada al *endpoint* **hello** mediante nuestro navegador:

```
http://IP-VM:8080/myproject/module/backend/api/myservice/hello
```



6.3 Despliegue en WildFly

Ya tenemos WildFly corriendo desde el principio de este proceso. Para realizar el despliegue, basta con meter el siguiente comando:

```
sudo mvn wildfly::deploy
```

Os pedirá vuestro usuario/contraseña y si todo ha ido bien, veréis algo como esto:

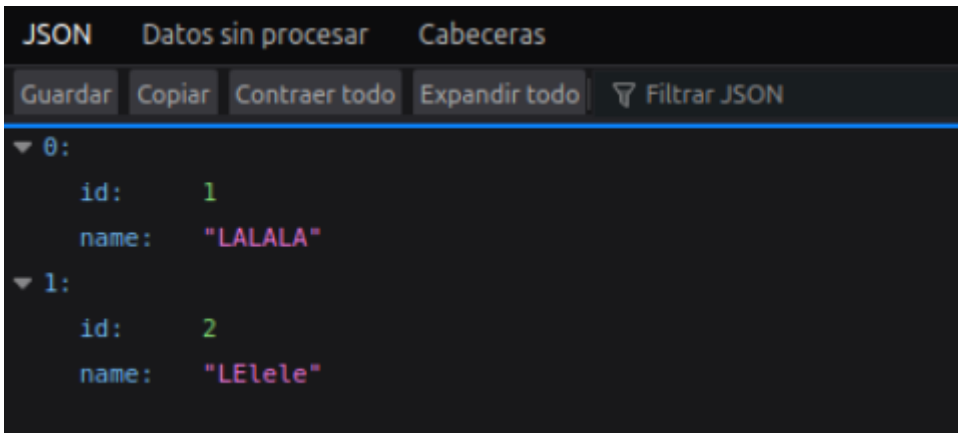
```
[INFO] --- wildfly-maven-plugin:4.2.2.Final:deploy (default-cli) @ modulename.backend ---
[INFO] JBoss Threads version 2.4.0.Final
[INFO] JBoss Remoting version 5.0.27.Final
[INFO] XNIO version 3.8.9.Final
[INFO] XNIO NIO Implementation Version 3.8.9.Final
[INFO] ELY00001: WildFly Elytron version 2.2.1.Final
Username:despliegue2S
Password:
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  17.612 s
[INFO] Finished at: 2024-11-09T11:48:47Z
[INFO] ------------------------------------------------------------------------
```

Para confirmar que la aplicación está corriendo podemos hacer una llamada al *endpoint* **hello** mediante nuestro navegador:

`http://IP-VM:8080/myproject/module/backend/api/myservice/hello`

hello!

También podemos comprobar el *endpoint* `/pojo/list`:



Que nos devuelve una respuesta JSON, correctamente renderizada por el navegador.

Comprobad que ocurre lo mismo con */pojo/find.*

Usando curl para hacer peticiones

Ejecuta el siguiente comando para poder ver a tiempo real el log de Wildfly:

```
tail -f /opt/wildfly/standalone/log/server.log
```

Y realiza las siguientes peticiones en el terminal de tu instancia, de forma que simulamos diferentes llamadas a los *endpoints* de la API:

- Crear nueva entrada (pon tu nombre, no el que viene en este comando):

```
curl -d '{"id":"2023", "name":"Despliegue"}' -H "Content-Type: application/json" -X POST http://localhost:8080/myproject/module/backend/api/myservice/pojo/new
```

- Actualizar una entrada (pon tu nombre, no el mío):

```
curl -d '{"id":"55", "name":"Sergio"}' -H "Content-Type: application/json" -X PUT http://localhost:8080/myproject/module/backend/api/myservice/pojo/update
```

- Eliminar una entrada:

```
curl -X DELETE http://localhost:8080/myproject/module/backend/api/myservice/pojo/remove?id=3
```

TAREA

1. Muestra las entradas de los logs que se corresponden con estas peticiones

2. Explica las partes del pom.xml

3. ¿Por qué al hacer el mvn wildfly::deploy automáticamente despliega el war en el servidor?

En el vídeo se debe mostrar cómo se despliega el proyecto tanto por la interfaz gráfica como por comandos, así como también que funcionan los endpoints.