8. Java y Spring Framework

8.1. Introducción y objetivos

Una de las tecnologías que se pueden encontrar en el lado del servidor o *back-end* es la arquitectura orientada a microservicios, un tipo de sistema distribuido. Para implementar este tipo de arquitectura es posible utilizar múltiples tecnologías.

En nuestro caso, utilizaremos el lenguaje Java y un *framework* desarrollado para él, Spring, que nos ayudará con esta tarea. Dada la complejidad de este tipo de arquitecturas, nos será imposible abarcar en un solo tema todos los componentes de esta, pero sí que nos detendremos en aspectos muy importantes, como la funcionalidad de Service Discovery a través de Eureka o el uso de servidores perimetrales con Spring Cloud Gateway.

Comenzaremos haciendo una breve introducción al lenguaje de programación Java, mencionando sus principales características. En este tema no se hará una explicación detallada de este lenguaje de programación. Continuaremos explicando qué es Spring y cuáles son sus características principales. Estudiaremos algunos módulos concretos de este *framework,* como son *boot* o *data.* Finalmente, aprenderemos a crear nuestras propias imágenes Docker de los microservicios construidos para facilitar un despliegue en máquinas remotas.

Puedes hacer uso del IDE que consideres. Trabajaremos con las siguientes versiones:

* Java 17.
* Spring 6.
* Spring Boot 3.

<https://github.com/>

El código de apoyo que usaremos en este tema es el siguiente:

* [Servidor de registro Eureka](https://github.com/UnirCs/back-end-eureka).
* [Servidor perimetral Cloud Gateway](https://github.com/UnirCs/back-end-cloud-gateway).
* [Aplicación de inventario – Productos](https://github.com/UnirCs/back-end-inventory-products).
* [Aplicación de inventario – Órdenes](https://github.com/UnirCs/back-end-inventory-orders).

No obstante, es probable que durante las sesiones de clase se liberen o se muestren más repositorios.

No dejes de lado la propia documentación de [Spring Framework](https://docs.spring.io/spring-framework/reference/6.1-SNAPSHOT/index.html).

8.2. El lenguaje Java

Java es a la vez un lenguaje de programación y el nombre de una plataforma informática que fue comercializada por primera vez en 1995 por Sun Microsystems. Posteriormente, esta empresa fue adquirida por Oracle en 2010, por lo que es esta empresa la que actualmente ostenta la propiedad del lenguaje.

Hoy en día es uno de los lenguajes más usados en el desarrollo de aplicaciones, especialmente en la industria del software y en el sector empresarial. Esto se debe, principalmente, a su potencia (similar a la de lenguajes como C y C++) y a su simplicidad, ya que oculta características poco usadas y confusas de los lenguajes de los que deriva.

Se trata de un lenguaje de programación orientado a objetos que proporciona las librerías necesarias para construir aplicaciones y sistemas distribuidos.

Una de sus principales ventajas es su máquina virtual, que es donde se ejecutan todos los programas escritos en este lenguaje. Gracias a este JRE (*Java runtime environment*) podemos ejecutar cualquier aplicación escrita en Java en toda máquina que tenga instalado el componente. Además, soporta la programación multihilo con librerías especialmente pensadas para ello. Como desventaja (aunque esto se ha ido corrigiendo en versiones más recientes) queda que Java es un lenguaje fuertemente tipado y su escritura puede ser algo más tediosa que en otros lenguajes de programación, como JavaScript.

Dada su amplia utilidad, es muy posible que ya conozcas este lenguaje o que en alguna ocasión hayas tenido que programar con él. Si no es el caso, te recomendamos estudiarlo. En la sección A fondo, dentro del recurso *Documentación sobre Java,* dispones de un enlace a la documentación oficial de Oracle para entender sus características y sintaxis.

En este documento no hablaremos en detalle de este lenguaje de programación. Si no lo conoces, te recomendamos seguir las siguientes lecciones magistrales en las que se explican los conceptos básicos de Java que necesitaremos saber para poder empezar a utilizar el framework Spring.

8.3. Spring Framework

Spring es un *framework* de código abierto destinado al desarrollo de aplicaciones empresariales en Java. Antes de que existiera este *framework*, Java ofrecía el modelo de Enterprise Java Beans, que en algunas ocasiones resultaba tedioso y pesado. Spring tiene características muy propias, como lo es la famosa inyección de dependencias y el uso de POJO (*plain old Java object*) como objetos de negocio.

Este *framework* ha supuesto un cambio radical respecto al estándar que existía para este lenguaje. Las aplicaciones desarrolladas con él son ligeras y, gracias a su flexibilidad, han ido apareciendo cada vez más módulos que lo han convertido en todo un *stack* de tecnologías para todas las capas de una aplicación.

Una de las ventajas de Spring se encuentra en el despliegue de la aplicación. En lugar de generar un archivo WAR con la aplicación web y desplegarlo en un servidor web, Spring incluye en el propio JAR de la aplicación un servidor web que se ejecutará cuando ejecutemos el JAR y nuestra aplicación correrá dentro de ese servidor embebido (estilo Tomcat).

El *framework* de Spring está formado por varios módulos, tal y como podemos ver en la Figura 1:

A diagram of a software framework

Description automatically generated

Figura 1. Módulos principales de Spring. Fuente: Spring Team, 2022.

A continuación, detallaremos algunos de ellos:

A screenshot of a phone

Description automatically generated

Tabla 1. Módulos de Spring. Fuente: elaboración propia.

La versión inicial de Spring se publicó en marzo de 2004. En nuestros ejemplos y desarrollos utilizaremos una de sus versiones estables más recientes, la versión 6.

A continuación, explicaremos uno de los conceptos clave de Spring: la inyección de dependencias. Esta característica es una de las principales señas de identidad de este *framework* y facilita mucho el trabajo a los programadores a la hora de configurar su aplicación y reutilizar piezas comunes de código.

El *framework* gira en torno a esta funcionalidad, denominada también inversión de control. Para trabajar con este *framework* es fundamental conocer esta funcionalidad. Esta funcionalidad hace uso del contenedor de *beans* de Spring (*spring container*), que se encarga de crear los objetos, configurarlos y controlar su ciclo de vida. Algunas funcionalidades de este contenedor se pueden modificar mediante configuración XML o programáticamente, pero su comportamiento por defecto es más que suficiente para la mayoría de las aplicaciones. Gracias al uso del contenedor, Spring creará un contexto de aplicación que tendrá registrada toda la información de los *beans* que se están utilizando. Pero ¿qué es un *bean*?

Un *bean* es un objeto Java gestionado por el *framework* Spring.

Es decir, los *beans,* como casi todo en Java, son objetos, pero estos se encuentran gestionados por el *core* de Spring y, por tanto, se alojan dentro de su contenedor. Los *beans* de Spring se pueden declarar mediante XML o mediante POJO, aunque es preferible esta última opción, ya que siempre es más farragoso trabajar con ficheros XML. Más tarde, en otros puntos del código, declararemos *beans* que referenciarán a otros definidos previamente.

Siguiendo el patrón de la inversión de control, un componente tiene que declarar las dependencias que tiene, pero no se encarga de obtenerlas. Esa parte corresponde a Spring que evaluará (en *runtime*) las definiciones de los *beans* en busca de dependencias y, si las encuentra, tratará de satisfacerlas, inyectándolas (como objetos Java) en aquellos sitios donde se necesiten. Gracias a este patrón se reduce, en gran medida, el acoplamiento entre los diferentes componentes de una aplicación.

La inyección de dependencias en Spring puede darse a nivel de atributo, mediante constructores e incluso a través de interfaces.

Otro punto importante para tener en cuenta es el *scope* (alcance) de los *beans* de Spring. Por defecto, en Spring hay siete tipos de *scope* diferentes, cada uno de ellos aplicable a situaciones concretas. Son los siguientes:

* *Singleton*. Este es el *scope* por defecto en Spring. Si no especificamos claramente lo contrario, el contenedor de Spring creará una única instancia compartida del objeto Java al que hace referencia el *bean*. Es muy importante tener en cuenta que, como la instancia que se crea es compartida, puede ser usada por distintos componentes si se encuentra marcada como dependencia de estos. Para evitar condiciones de carrear u otros problemas de índole similar, se recomienda siempre que los *beans* con este *scope* no tengan estado *(stateless),* es decir, que sus atributos nunca se vean modificados. Todos los *beans* de este tipo son creados y configurados por defecto en el arranque de la aplicación. Este comportamiento se puede modificar utilizando la anotación *lazy* en la definición del *bean.*
* *Prototype.* Este *scope* le indicará al contenedor de Spring que debe crear cada vez una nueva instancia del objeto Java al que hace referencia el *bean* cada vez que sea solicitado como dependencia. Al contrario que en el caso anterior, se recomienda utilizar este *scope* cuando el cliente tenga estado (*stateful*), ya que de lo contrario se estaría generando una cantidad innecesaria de objetos.
* *Request.* Solo tiene sentido utilizarlo en aplicaciones web. El contenedor de Spring creará una nueva instancia del objeto Java referenciado cada vez que reciba una petición HTTP.
* *Session.* Solo tiene sentido utilizarlo en aplicaciones web. El contenedor de Spring creará una nueva instancia del objeto Java referenciado por cada sesión HTTP.
* *Application.*
* *Websocket.*

Los últimos dos tipos de *scope* no son muy utilizados y es por ello por lo que no nos detendremos a estudiarlos.

No obstante, si lo deseas, puedes obtener más información sobre los diferentes tipos de *scope* en la sección A fondo, *Scopes de Spring*.

8.4. Spring boot

Spring boot (versión 3) es un módulo muy importante de Spring que facilita enormemente la creación de aplicaciones *stand-alone* y autoconfiguradas, lo que reduce en gran medida el coste de ponerlas en producción.

Algunas de sus características principales son:

* Crear aplicaciones Spring *stand-alone.*
* Embeber un servidor Tomcat, Jetty o Undertow en el JAR de la aplicación construida, de esta forma se elimina la necesidad de desplegar un archivo WAR en un servidor web.
* Configurar automáticamente librerías de Spring y de terceros, siempre que esto sea posible (cuando la librería del tercero lo permite).
* Dotar a la aplicación de un conjunto de características necesarias para un despliegue productivo,como por ejemplo métricas (*logs*) y *health checks*.
* Eliminar la dependencia de ficheros de configuración XML.

Gracias a Spring boot podemos reducir la configuración de algunas librerías de terceros muy utilizadas a la mínima expresión y desplegar nuestra aplicación de forma sencilla (tanto en un entorno local como en uno productivo). Es llamativo que este módulo integre un servidor Tomcat de manera automática.

Tradicionalmente, las aplicaciones se empaquetaban en archivos WAR y eran desplegadas en servidores web donde se ejecutaban (JBoss) junto con otras diez, veinte o treinta aplicaciones o incluso más. Ahora, cada aplicación es completamente operativa por sí sola y ese es uno de los requisitos para poder trabajar con imágenes y contenedores Docker. Esta forma de trabajar nos permitirá que cada aplicación se ejecute en un contenedor Docker separado. Esto será clave a la hora de trabajar con microservicios.

En cuanto a la configuración automática de librerías de terceros, Spring Boot incluye una serie de *starters* para cada una de ellas que nos permitirá facilitar y agilizar la configuración.

Un *starter* es un submódulo de Spring Boot enfocado a un tercero concreto. Dentro de los *starters* más famosos incluidos en este módulo encontramos los siguientes:

* Spring-boot-starter-activemq . Facilita la configuración e integración con mecanismos de mensajería asíncrona mediante JMS para Apache ActiveMQ. Es decir, incluye un soporte para integrarnos fácilmente con colas o *topics* de ActiveMQ.
* Spring-boot-starter-batch . Facilita la configuración y creación de aplicaciones *batch*.
* Spring-boot-starter-data-jdbc . Facilita la configuración y conexión a bases de datos mediante JDBC.
* Spring-boot-starter-data-jpa . Facilita la configuración y conexión a bases de datos haciendo uso de JPA con Hibernate como ORM.
* Spring-boot-starter-data-redis . Facilita la configuración e integración con cachés Redis.
* Spring-boot-starter-actuator . Incluye en las aplicaciones herramientas para monitorizar el estado de las aplicaciones (métricas y *health checks*).

Estos son solo algunos de los *starters* que incluye Spring boot. Para poder usarlos, solo necesitamos incluir su dependencia Maven en el archivo pom.xml que encontraremos dentro de las aplicaciones.

En la sección A fondo, en*Spring boot starters,*encontrarás un enlace en el que podrás ver todos los *starters* incluidos en este módulo.

8.5. Spring Data

En la mayoría de las aplicaciones web modernas es necesario acceder a alguna fuente de información externa o tener la capacidad de persistir datos de algún tipo en una base de datos.

A su vez, estos orígenes de datos pueden ser muy diversos. Podemos encontrarnos bases de datos relacionales de distintas marcas (MySQL, Oracle, etc.) o bases de datos no relacionales, como MongoDB, DynamoDB (la base de datos no relacional de Amazon Web Services), CosmosDB (la base de datos no relacional de Azure) o Neo4J, clústeres de ElasticSearch o incluso directorios LDAP.

Ante tanta variedad surge un problema claro. Una aplicación que haga uso de varios de ellos deberá tener implementaciones muy diferentes para acceder a los datos de cada origen. Spring Data se concibe como la solución a este problema. Su objetivo es facilitar el acceso y la explotación de los datos de una única forma, salvando así las diferencias técnicas que puedan existir entre los diferentes orígenes de datos.

Con Spring Data obtendremos una serie de interfaces que nos facilitarán el acceso a la información, pero no será necesario implementarlas, ya que esta será la función principal de este módulo.

Uno de los módulos principal dentro de Spring Data es Spring Data JPA. Con JPA podremos definir objetos Java como modelos, indicando cuál es el nombre de la «entidad» a la que pertenece (que podría traducirse en el nombre de una tabla, de un índice u otra cosa en función del tipo de origen de datos) y sus atributos, así como las relaciones entre entidades (en el caso de haberlas y de que el origen de datos final las soporte).

Para varias soluciones de persistencia, la combinación de Spring Data JPA y Spring Boot es suficiente. No obstante, cuando la implementación del acceso a los datos es muy específica, es necesario utilizar los submódulos de Spring Data correspondientes. Recordemos que son estos módulos los que incluirán la implementación del acceso a los datos. Un desarrollador trabajará únicamente con interfaces.

Algunos ejemplos de submódulos específicos de Spring Data para otras tecnologías de almacenamiento son:

* Spring-data-mongoDB.
* Spring-data-redis.
* Spring-data-solr.
* Spring-data-cassandra.
* Spring-data-elasticsearch.
* Spring-data-neo4j.
* Spring-data-hadoop.

8.6. Creación de una aplicación con Spring

Una vez listadas las principales características del *framework* que vamos a utilizar, lo siguiente es crear una aplicación de ejemplo con este *framework* para poder visualizar los conceptos estudiados.

En el siguiente vídeo, *Creación de una aplicación con Spring,* puedes ver cómo crear una aplicación en Spring desde cero, donde además se incluyen gran parte de los conceptos expuestos en esta sección.

Para crear una aplicación con Spring de manera sencilla, recomendamos utilizar la página web [Spring initializr](https://start.spring.io/), que nos permitirá configurar con una interfaz amigable los aspectos básicos de esta. A continuación, puedes ver la configuración que se seguirá para implementar todos los ejemplos de este tema:

A screenshot of a computer

Description automatically generated

Figura 2. Configuración de una aplicación Spring con Spring Initializr. Fuente: Spring Team, 2022.

Algunas consideraciones:

* Utilizaremos Maven como herramienta de gestión de dependencias. Esto creará un archivo pom.xml en nuestra aplicación con los detalles básicos de esta y con todas las dependencias que hayamos elegido.
* Utilizaremos la versión 3.0.6 de Spring Boot y 17 de Java. Si la versión 3.0.6 no estuviese disponible, escogeremos siempre la última versión estable que aparezca (que no sea SNAPSHOT, RC*x* o M*x*). Empaquetaremos siempre como JAR.
* Utilizaremos la dependencia Lombok para ayudarnos a generar código trivial (*getters*, *setters*) en las clases que realicemos. Utilizaremos la dependencia *Spring Boot DevTools* para facilitar el desarrollo, ya que, entre otras cosas, reiniciará automáticamente las aplicaciones cuando realicemos cambios en ellas. Por último, la dependencia de H2 nos servirá para utilizar una base de datos en memoria.

La aplicación que vamos a construir con Spring expondrá una API REST con diferentes operaciones, todas ellas relacionadas con los productos de una tienda cualquiera. Esta API tendrá las siguientes operaciones:

* Creación de un nuevo producto.
* Obtención de los datos de un producto.
* Eliminación de un producto.

Son operaciones muy sencillas que nos ayudarán a entender y visualizar algunos de los conceptos que hemos aprendido en la sección anterior. Una vez configurada nuestra aplicación en la web de Spring Initializr, haremos clic en *generate* para descargar un archivo ZIP que contendrá un esqueleto de la aplicación, a partir del cual iremos añadiendo funcionalidades.

En Eclipse, deberás importar la carpeta descomprimida de ZIP que has descargado mediante la opción *existing maven projects*. Además, recomendamos instalar la extensión Spring Tool Suite en Eclipse para poder desarrollar con más facilidades. Podemos instalar esta extensión a través del Eclipse Marketplace: Menú > Help > Eclipse Marketplace > buscamos «sts» > seleccionamos e instalamos la extensión.

A screenshot of a computer program

Description automatically generated

Figura 3. Extensión STS. Fuente: elaboración propia.

Los usuarios de IntelliJ también tienen disponible este plugin en la web de Jetbrains.

Una vez hayas importado el proyecto, verás que su estructura es la siguiente:

A screenshot of a black screen

Description automatically generated

Figura 4. Estructura de un proyecto con Spring. Fuente: elaboración propia.

Identificamos los siguientes elementos:

* Carpeta src/main/java: Es donde se encuentra todo el código fuente de la aplicación. Las clases de configuración, controladores, POJOs, repositorios de acceso a datos…
* Carpeta src/main/resources: Es donde se encuentran todos aquellos recursos externos al código principal, necesarios para el correcto funcionamiento de la aplicación. Un fichero muy importante dentro de esta carpeta es *application.yml* que, como veremos posteriormente, contendrá valores de configuración comunes a toda la aplicación.
* pom.xml: Es el archivo que define cuál es el nombre de la aplicación, la versión de Java que utiliza, y todas las dependencias externas (de Spring o de otros terceros) que son necesarias, así como sus versiones. Si abrimos este archivo veremos:
  + La declaración del *parent pom*: Esta aplicación *hereda* todas las dependencias de un *pom* padre, que es el de la versión 3.0.6 de Spring. Esta declaración hace que no sea necesario incluir explícitamente todas las dependencias de esa versión de Spring.
  + La información de nuestra aplicación.
  + La lista de dependencias de la aplicación: Veremos aquí todas las dependencias que en su momento incluimos en el paso anterior, más una más, spring-boot-starter-test, que nos da todo lo necesario para realizar pruebas unitarias en nuestra aplicación.

Si abrimos la carpeta con el código fuente, veremos que se ha creado la estructura de paquetes *com.unir.products.* Dentro de esta estructura incluiremos todo nuestro código. Abre la clase ProductsApplication.java. Verás algo similar a esto:

A computer screen with text

Description automatically generated

Figura 5. ProductsApplication, punto de entrada de la aplicación. Fuente: elaboración propia.

Esta clase es el punto de arranque de la aplicación de Spring. Y eso lo podemos saber por la anotación @SpringBootApplication que se encarga, por nosotros, de levantar toda la aplicación con la carga y configuración que ello conlleva. Cuando la aplicación se levanta, se carga el contexto de aplicación del que hablábamos y registra en él todos los *beans* que Spring encuentre. Habitualmente, no es necesario realizar ningún tipo de cambio en esta clase.

A continuación, crearemos varios paquetes más para guardar todas las clases que vamos a desarrollar.

* *Controller.* Aquí irá el controlador de las peticiones HTTP que recibamos. Gracias a este controlador podremos ver cuál es la API REST que expone nuestra aplicación para ser invocada por otras. Dentro de él tendremos la siguiente clase:
  + ProductsController.java

* *Service.* Dentro de este paquete tendremos la lógica de negocio de nuestra aplicación (controles, validaciones, etc.). Dentro de él tendremos las siguientes clases:
  + ProductsService.java. Será una interfaz. Es muy recomendable que cuando trabajemos con dependencias de Spring, lo hagamos a través de interfaces. De esta forma, el cliente estará liberado de cambios que pudiesen afectar a la implementación.
  + ProductsServiceImpl.java. Implementará la interfaz.

* *Data.* Dentro de este encontraremos todas las clases necesarias para implementar la persistencia y el acceso a datos de nuestra aplicación.
  + ProductRepository.java. Será el repositorio que utilizaremos para acceder a los datos almacenados en una base de datos en memoria H2. Aquí haremos uso de Spring Data y Spring Data JPA.

* *Model.* Aquí encontraremos los POJO (o modelos) que nos ayudarán a modelar las peticiones recibidas, las respuestas emitidas y cualquier tipo de información estructurada que sea necesaria. Además, dentro de este tendremos otros paquetes para modelar como objetos Java todos los mensajes que las diferentes capas intercambian entre sí. Principalmente:
  + *Request*. Modelos relacionados con las posibles peticiones que se pueden recibir.
  + *Pojo.* Modelos que representarán la información que almacenamos en la base de datos.

La estructura de la aplicación, teniendo en cuenta todo lo anterior, quedaría así una vez realizada la implementación:

A screenshot of a phone

Description automatically generated

Figura 6. Estructura del proyecto de Spring tras la implementación. Fuente: elaboración propia.

A continuación, veremos la implementación de cada capa.

*Resources: application.yml*

## Configuración de Spring

spring:

application:

name: ms-inventory-products

datasource:

url: jdbc:h2:mem:testdb

jpa:

show-sql: true

h2:

console:

enabled: true

path: /h2-console

settings:

web-allow-otrhers: true

## Configuración de Instancia

server:

port: 8088

En este archivo encontraremos toda la configuración relativa a la integración con la base de datos en memoria H2. Como podemos ver, solo es necesario definir la URL de la base de datos (en memoria), el *driver* y las credenciales (sa es el nombre de usuario por defecto y la contraseña puede ser cualquier valor). Spring boot y Spring data se encargarán de todo lo demás. Indicamos también el puerto en el que esta aplicación va a ejecutarse.

Es probable que *Spring Initializr* nos cree este archivo con el nombre *application.properties*. Simplemente lo renombramos. Son dos formas diferentes de especificar las propiedades de configuración.

Paquete *model.pojo*: Product.java

@Entity

@Table(name = “products”)

@Getter

@Setter

@AllArgsConstructor

@NoArgsConstructor

@Builder

@ToStering

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

@Column(name = “name”, unique = true)

private Sting name;

@Column(name = “country”)

private Sting country;

@Column(name = “description”)

private Sting description;

@Column(name = “visible”)

private Boolean visible;

}

Esta clase servirá para modelizar como objetos la información que estará almacenada en la base de datos. Las anotaciones *Entity* y *TableName* nos indican que estamos ante una entidad de una base de datos relacional y que el nombre de la tabla es *products*. El resto de las anotaciones de clase sirven para generar código de manera automática (*getters*, *setters*, constructores, método*toString, etc.*).

La definición de la tabla se obtiene a través de las anotaciones a nivel de atributo que encontramos dentro de esta clase. El atributo id será la clave principal de la tabla. Habrá otras cuatro columnas más, *name*, *country*, *description* y visible.

Paquete *model.request*: CreateProductRequest.java

@Getter

@Setter

@AllArgsConstructor

@NoArgsConstructor

public class CreateProductRequest {

private String name;

private String country;

private String description;

private Boolean visible;

}

Esta clase servirá para trasladar a un objeto Java el contenido de la petición HTTP que reciba el controlador en las operaciones de creación de productos (el cuerpo de la petición será un JSON). Spring tratará de convertir automáticamente el JSON recibido en un objeto de esta clase, utilizando internamente la librería Jackson. Para que este proceso de deserialización sea exitoso, debemos tener en cuenta que:

* El nombre de las claves del JSON debe coincidir con el nombre de los atributos de la clase Java. Si no se pudiese conseguir eso, usaríamos la anotación @JsonProperty encima del atributo Java, indicando entre comillas el nombre de la clave en el JSON, como en el caso de *country* (aunque en este ejemplo no sea necesario).
* El POJO debe tener definidos constructores con y sin argumentos para asegurar que no falle en ningún caso. Esto lo conseguimos con las anotaciones de Lombok,  NoArgsConstructor  y AllArgsConstructor .

Paquete *data*: ProductRepository.java

package com.unir.products.data;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;

import com.unir.products.model.pojo.Product;

public interface ProductRepository extends JpaRepository<Product, Long> {

List<Product> findByName(String name);

Cuando trabajemos con Spring Data, solo seremos responsables de declarar las interfaces para acceder a los datos, pero no su implementación. Y este es el resultado. Nuestro  ProductRepository  hereda de la interfaz  JpaRepository  y provee a nuestro *bean* (ya que Spring registrará como *beans* todas aquellas clases que extiendan de esta interfaz de forma automática) de métodos como  findById ,  findAll , save  o  delete , entre otros. Podemos añadir más, si lo deseamos, incluyendo en la interfaz la firma del método, como se ve en el ejemplo para hacer búsquedas por la columna *name*. Esto funciona porque Spring analiza el nombre de los métodos y, en función de él, genera internamente una consulta SQL. Es por ello muy importante escribir correctamente el nombre de los métodos.

Todo el código necesario para conectar con la base de datos, componer la consulta SQL y procesar el resultado es gestionado por Spring Data.

En la sección de A fondo, dentro de *Creación de Queries JPA,* dispones de un enlace en el que podrás obtener más información sobre el nombre que puedes darle a los métodos en esta interfaz para construir otros tipos de consultas. En algunas ocasiones, algunas consultas son demasiado largas o complejas como para escribirlas en el nombre de un método y es mejor utilizar la anotación Query para indicar el código SQL de esta.

Paquete *service*: Interfaz ProductService.java

public interface ProductService{

List<Produc> getProducts();

Product getProduct(String productId);

Boolean removeProduct(String productId);

Product createProduct(CreateProductRequest request);

}

La interfaz de nuestro servicio expone las operaciones que permite realizar. Es una buena práctica que la comunicación entre las distintas capas de una aplicación se realice a través de interfaces y no usando las implementaciones directamente.

En el siguiente vídeo, *Conexión de una aplicación Spring y una base de datos SQL,* puedes ver cómo conectar tu aplicación hecha en Spring a una base de datos MySQL en local.

Paquete *service*: Implementación ProductServiceImpl.java

@Service

public class ProductsServiceImpl implements ProductsService {

@Autowired

private ProductRepository repository;

@Override

public List<Product> getProducts() {

List<Product> products = repository.findAll();

return products.isEmpty() ? null : products:

}

@Override

public Product getProduct(String productId) {

return repository.findById(Long.valueOf(productId)).orElse(null);

}

@Override

public Boolean removeProduct(String productId) {

Product product = repository.findById(Long.valueOf(productId)).orElse(null);

if (product != null) {

repository.delete(product);

return Boolean.TRUE;

} else {

return Boolean.FALSE;

}

}

@Override

public Product createProduct(CreateProductRequest request) {

if (request != null &&amp; StringUtils.hasLength(request.getName().trim())

&&amp; StringUtils.hasLength(request.getDescription().trim())

&&amp; StringUtils.hasLength(request.getCountry().trim()) &&amp; request.getVisible () != null) {

Product product = Product.builder().name(request.getName()).description(request.getDescription()).

country(request.getCountry()).visible(request.getVisible()).build();

return repository.save(product);

} else {

return null;

}

}

Esta clase contiene toda la lógica de negocio de la aplicación. Incluye, por tanto, cualquier tipo de validación o control que haya que hacer sobre los datos recibidos en el controlador. Su única dependencia (atributo Java) es un objeto de la clase  ProductRepository  que hemos visto antes.

Esta dependencia se está inyectando, en primer lugar, a través de una interfaz. En java no es posible instanciar una interfaz, la clase instanciada debe implementar la interfaz. Por tanto, Spring ya nos ayuda a elegir cuál debe ser el tipo de clase concreta (esto ocurre de forma totalmente transparente). Mientras que, por otro lado, la dependencia la hacemos de forma explícita con la anotación *autowired* definida a nivel de atributo. Con esa dependencia le estamos indicando al contenedor de Spring que debe tratar de buscar un *bean* que implemente la interfaz *ProductRepository* y que, si lo encuentra, le asigne su valor a la variable *repository*.

Te habrás dado cuenta de que la anotación *service* preside la declaración de esta clase. Esta anotación le indica al contenedor de Spring de que esta clase es un *bean* del nivel de servicio. Es decir, debe ser gestionada por su contenedor. Además, como no estamos especificando ningún tipo de *scope,* se usará el que viene por omisión, *singleton* (al igual que con *ProductRepository*).

Existen otras anotaciones similares a *service* que debes conocer:

* *Component.* Se utiliza para indicar que un *bean* es genérico (no pertenece a ninguna capa en particular).
* Repository. Se utiliza para indicar que un *bean* pertenece a la capa de persistencia. No es necesario incluirla si el *bean* implementa la interfaz JpaRepository.
* *Controller* y *RestController*. Se utilizan para indicar que el *bean* es un controlador de peticiones HTTP.

Por lo demás, esta clase implementa las operaciones de la interfaz asociada, al hacer uso de la capa de acceso a datos para añadir, eliminar y obtener nuevos productos.

Paquete *controller*: ProductsController.java

@RestController

@RequiredArgsConstructor

@Slf4j

public class ProductsController {

private final ProductService service:

@GetMapping(“/products”)

public ResponseEntity<List<Product>> getProducts(@RequestHeader Map<String, String> headers) {

log.info(“headers: {}”, headers);

List<Product> products = service.getProducts();

if (products != null) {

return ResponseEntity.ok(products);

} else {

return ResponseEntity.ok(Collections,emptyList();

}

}

@GetMapping(“/products/{productId}”)

public ResponseEntity<Product> getProducts(@PathVariable String productId) {

log.info(“Request received for product{}”, productId);

Product product = service.getProduct(productId);

if (products != null) {

return ResponseEntity.ok(products);

} else {

return ResponseEntity.notFound().build();

}

}

@DeleteMapping(“/products/{productId}”)

public ResponseEntity<Void> deleteProducts(@PathVariable String productId) {

Boolean removed = service.removeProduct(productId);

if (Boolean.TRUE.equals(removed)) {

return ResponseEntity.ok().build();

} else {

return ResponseEntity.notFound().build();

}

}

@PostMapping(“/products”)

public ResponseEntity<Product> getProduct(@RequestBody CreateProductRequest request) {

Product createrProduct = service.createProduct(request);

if (createdProduct !=null) {

return ResponseEntity.status(HttpStatus.CREATED).body(createdProduct);

} else {

return ResponseEntity.badRequest ().build();

}

}

}

Esta clase es la encargada de interceptar todas las peticiones HTTP que recibe nuestra aplicación a través de su API REST.

Pero ¿cuál es su API REST? Es muy sencillo intuir cuál es la API observando las anotaciones de cada uno de los métodos de esta clase, anotaciones pertenecientes al módulo *spring-web.* La API expuesta es la siguiente:

A blue and white table with text

Description automatically generated

Tabla 2. API REST expuesta por la aplicación de Spring. Fuente: elaboración propia.

La API expuesta es muy sencilla y algunos aspectos de esta podrían mejorarse, como quizá un mayor manejo de códigos de error (y no solo 400/ 404), sin embargo, nos sirve para ilustrar cómo nuestra aplicación expone el modo en el que los consumidores deben comunicarse con ella.

Este *bean* de Spring (es un *bean* porque está anotado con *RestController*, por lo que será gestionado a través del contenedor de Spring) tiene una única dependencia, expresada a través de su único atributo de instancia, y es el bean que hemos visto antes, *ProductsService*. En este caso, y para ilustrar otro modo de inyección de dependencias, la inyección ocurre por constructor.

A través de la documentación de Spring, intenta averiguar el propósito de las anotaciones @RequestBody ,  @RequestHeader ,  @PathVariable  y  @RequestParam .

La notación *autowired* no se encuentra al nivel del atributo, sino del constructor (de forma implícita). Esto hará que Spring busque *beans* del tipo del argumento del constructor (esto ocurre con todos los argumentos).

De esta forma, gracias a la anotación *RequiredArgsConstructor* de Lombok se crea un constructor para la clase con todos los atributos que estén marcados como final y Spring tratará de buscar candidatos para esos objetos. De esta forma, el código queda más limpio:

@RestController

@RequiredArgsConstructor

@Slf4j

public class ProductsController {

private final ProductsService service;

Como has podido observar a través de este ejemplo, cada capa y cada clase tienen una función muy específica. Por ejemplo, no incluimos ningún tipo de lógica de negocio en el controlador o en la capa de acceso a datos. Cada capa se ocupa exactamente de lo que debe hacer y nada más. Lo único que queda es ejecutar nuestra aplicación hecha con Spring para ver que todo funciona correctamente. Puedes usar Postman para realizar llamadas a la aplicación desplegada. Recuerda que el *host* es *localhost* y el puerto en el que se encuentra desplegado la aplicación por defecto es el 8088 (el valor por defecto es 8080). Puedes modificar el puerto mediante la variable de configuración *server.port* en el fichero *.properties*.

A screenshot of a computer

Description automatically generated

Figura 7. Ejecución de aplicación Spring desde Eclipse. Fuente: elaboración propia.

8.7. Eureka

Eureka es uno de los componentes principales del módulo *spring-cloud*, que se enfoca en los microservicios. Concretamente, Eureka se encarga del *service discovery*.

Eureka se comporta como un servidor cuyo trabajo principal es registrar y localizar todos los microservicios existentes de un sistema distribuido. Facilita también el balanceo de la carga entre las diferentes instancias de un microservicio concreto.

Los dos conceptos fundamentales que debemos entender con respecto a Eureka son:

* Debe haber una instancia que se comporte como el servidor, que registre y muestre información de los diferentes microservicios.
* Todos los microservicios del sistema deben registrarse como clientes del servidor Eureka. En el momento de su arranque, el microservicio se debe comunicar con el servidor para notificarle que está listo para ser consumido.

En el siguiente vídeo, *Configurando Eureka para registrar microservicios,* puedes ver cómo configurar Eureka para registrar el microservicio que creamos en los vídeos anteriores.

Eureka debe mantener constantemente el estado de todos los microservicios que están registrados. Para ello, los microservicios deben notificar su estado haciendo peticiones continuas a Eureka (por defecto, cada 30 segundos). Este tipo de peticiones se denominan *heartbeats,* ya que se asemejan a la idea de los latidos para indicar que el microservicio está vivo y funcionando.

Si tras varios períodos no se recibe un *heartbeat*, el servidor Eureka entenderá que el microservicio ha dejado de estar disponible y, por tanto, lo eliminará del registro. Si después de sacarlo, vuelve a recibir *heartbeats,* entenderá que vuelve a estar activo y será registrado de nuevo. Todo este proceso es gestionado automáticamente a través de las dependencias de Eureka para aplicaciones hechas con Spring (ya sean los clientes o el servidor).

El uso de Eureka en nuestra arquitectura de microservicios tiene numerosas ventajas:

* Abstracción de la localización física de los microservicios.
* Conocer en todo momento el estado de todos los microservicios.
* Soporte multiregión. Pensado especialmente para despliegues en la nube.

A continuación, veremos como implementar por nuestra cuenta un servidor de Eureka donde el microservicio que desarrollamos en la sección anterior se pueda registrar.

Para ello, volveremos a Spring Initializr y crearemos un nuevo proyecto, esta vez con diferentes dependencias:

A screenshot of a computer

Description automatically generated

Figura 8. Creación de un servidor de Eureka a través de Spring Initializr. Fuente: elaboración propia.

Generaremos e importaremos el código del proyecto en nuestro IDE (entorno de desarrollo integrado). Lo primero que haremos será renombrar el archivo de propiedades application.properties de la carpeta *src/main/resources* que pasará a llamarse application.yml. Cambiar la extensión del archivo nos permite variar la forma en la que escribimos las propiedades. Puedes elegir la opción que más te guste en tus proyectos. Incluiremos la siguiente configuración en el fichero:

server:

port: 8761

eureka:

client:

registerWithEureka: false

fetchRegistry: false

instance:

hostname: ${EUREKA\_HOST:localhost}

Como ves, la única diferencia entre .properties y .yml es la forma en la que se declaran las propiedades (*flat*vs *hierarchically*).

En esta configuración estamos indicando lo siguiente:

* Estamos levantando el microservicio (que posteriormente será un servidor de Eureka) en el puerto 8761.
* Este microservicio no se registrará en Eureka (lógico, dado que este es el propio servidor).

Una vez entendida esta configuración básica, debes añadir en la clase principal (EurekaApplication.java) la anotación *@EnableEurekaServer*, ya que Spring Intializr no lo hace por nosotros. Una vez hecho esto, podemos arrancar nuestro servidor de Eureka a través de la consola de Eclipse.

Una vez desplegado, accederemos a la consola de Eureka a través en la dirección http://localhost:8761. Dentro de esta consola veremos una sección llamada *Instances currently registered with* Eureka, que estará vacía, al igual que se muestra a continuación:



Figura 9. Aplicaciones registradas en Eureka al comienzo. Fuente: elaboración propia.

Ahora lo que resta es modificar nuestro microservicio inicial para que se registre como cliente de este servidor y pueda compartir su estado. Para ello, será necesario añadir una nueva dependencia en el archivo pom.xml del microservicio cliente, dentro de la sección de dependencias:

<dependency>

<groupId>org.springframeworks.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

Una vez añadida la dependencia de Eureka, lo siguiente es realizar algunas configuraciones en las *properties* del microservicio. El resultado final del archivo *application.yml* es el siguiente:

## Configuracion de Spring

spring:

application:

name: ms-inventory-products

datasource:

url: jdbc:h2:mem:testdb

jpa:

show-sql: true

h2:

console:

enabled: true

path: /h2-console

settings:

web-allow-others\_ true

## Configuracion de Instancia

server:

port: 8088

## Configuracion del cliente de Eureka

eureka:

client:

service-url:

defaultZone: ${EUREKA\_URL:http://localhost:8761/eureka}

instance:

preferIpAdress: true

hostname: ms-inventory-products

En esta configuración estamos indicando que el nombre de nuestra aplicación Spring (o microservicio) es *ms-inventory-products*, que el servidor de Eureka se encuentra disponible en la URL http://localhost:8761/eureka (o lo que indique la variable de entorno EUREKA\_URL) y que preferimos que se muestra la dirección IP al nombre del *host.* No es necesaria ninguna configuración adicional.

Si levantamos nuestra aplicación de nuevo, veremos que ahora sí está registrada en Eureka:

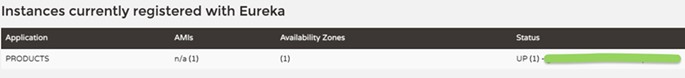


Figura 10. Servidor de Eureka con instancia registrada. Fuente: elaboración propia.

Gracias a esto, es posible realizar llamadas a los microservicios sin conocer necesariamente su dirección IP, ya que, gracias a clientes con Spring Feign o RestTemplate, podemos hacer uso de los nombres de las aplicaciones.

En la sección A fondo, dentro del recurso *RestTemplate,* tienes más información sobre este.

En el siguiente vídeo, *Peticiones HTTP entre microservicios usando su nombre gracias a Eureka,* puedes ver cómo beneficiarte de este servidor de registro para no incluir direcciones IP y puertos en las llamadas HTTP que realices entre los microservicios de tu sistema.

### **8.8. Spring Cloud Gateway**

Cloud Gateway (spring-cloud-gateway), en adelante CW, es otro de los **componentes principales del módulo** ***spring-cloud*.** Concretamente, el CW se encarga de implementar un servidor perimetral que actuará como ***proxy*** **inverso** y que será el **punto de entrada** de cualquier aplicación que quiera interactuar con nuestros microservicios.

Si bien es posible configurar de forma explícita las diferentes rutas de este proxy inverso, podremos utilizar la **combinación** de CW y de Eureka para no tener que escribir manualmente todos los endpoints que nuestros **microservicios** exponen a través de sus API (interfaces de programación de aplicaciones). Incluir este componente en nuestra arquitectura es extremadamente sencillo, ya que se hace a partir de los componentes que tenemos creados:

* El microservicio products es el encargado de gestionar los productos de una tienda mediante una base de datos H2 en memoria. Además, se registra en Eureka.
* El servidor de Eureka es el encargado del registro de todos los microservicios.

Para comenzar, crearemos un **nuevo proyecto de Java** con Maven haciendo una copia del microservicio products. Luego, hay que modificar, dentro del POM, el atributo  artifactId  para que sea Cloud-Gateway y las dependencias para quedarnos con las que nos interesan y añadir las correspondientes a CW.

Puedes ver el resultado final a través del siguiente enlace: <https://github.com/UnirCs/back-end-cloud-gateway/blob/master/pom.xml>

Dentro de la carpeta src/main/java borraremos todo el contenido (que estaba copiado de products) y crearemos un paquete que se llamará com.unir.gateway. Dentro de ese paquete, tendremos la única clase presente en este proyecto**, GatewayApplication.java,** que tendrá este contenido:

package com.unir.gateway;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication

@EnableDiscoveryClient

public class GatewayApplication {

public static void main(String[] args) {

SpringApplication.run(GatewayApplication.class, args);

}

}

Este componente se comportará como un ***proxy*** **inverso** y será nuestro **servidor perimetral.** Por tanto, tendremos que incluir una **pequeña configuración** en el archivo de configuración dentro de src/main/resources que, en este caso, será mediante YAML (aunque también podríamos haber elegido el formato de properties):

serve:

port: 8762

## Configuracion clasica de Eureka

eureka:

instance:

preferIpAddress: false

hostname: ${HOSTNAME:localhost}

client:

registerWithEureka: true

fetchRegistry: true

serviceUrl:

defaultZone: ${EUREKA\_URL:https://localhost:8761/eureka}

## Habilitamos el service Discovery (haciendo uso de Eureka)

## lower-case-serve-id necesario ya que por defecto se pone en mayúsculas, al cogerlo de Eureka

spring:

application:

name: gateway

cloud:

gateway:

discovery:

locator:

enabled: true

lower-case-service-id: true

default-filters:

- DedupeResponseHeader=Access-Control-Allow-Credentials Access-Control-Allow-Origin

globalcors: ## Configuracion de CORS a nivel Gateway

cors-configurations:

‘[/\*\*]’: ## Para todas las rutas

allowedOrigins: ${ALLOWED\_ORIGINS} ## Si ponemos ‘\*’ permitimos todos los orígenes. P

allowedHeaders: ‘\*’

allowedMethods:

- GET

- POST

## Habilitamos la API Actuator para poder acceder a /actuator/gateway/routes

## Exponemos los routings que se han detectado automáticamente (todos)

management:

endpoint:

gateway:

enabled: true

endpoints:

web:

exposure:

include:

- ‘\*’

Por tanto, indicamos que el servidor perimetral se está ejecutando en el puerto 8762 y que el nombre de este será gateway. Este servidor se **registrará en Eureka** de la misma forma que lo hace el microservicio products.

Es importante que la propiedad eureka.instance.hostname tenga el valor del host donde se está desplegado Eureka (en este caso, mientras probamos, localhost), ya que sino el CW no podrá utilizar todo el conocimiento de Eureka para redirigir las peticiones entrantes correctamente.

La parte de registro en Eureka es exactamente igual a la que vimos para products.

La propiedad final del fichero **habilita todos los** ***endpoints*** **del servidor,** lo que es necesario para que podamos acceder a la ruta http://localhost:8762/actuator/gateway/routes, donde encontraremos todas las rutas que CW está resolviendo.

Otro punto importante para tener en cuenta es la **configuración de CORS** que estamos aplicando, y que aplicaremos a todos los microservicios de nuestro back-end que accedan a través de este servidor perimetral (es decir, desde un navegador), dado que es el **punto de entrada** que los llamantes usarán. Indicamos:

* Los filtros por defecto a incluir en las respuestas.
* Los orígenes permitidos mediante la variable de entorno. Aquí deberíamos especificar el dominio donde nuestro front-end está desplegado.
* Los Headers permitidos, es decir, los que permitiremos en las peticiones.
* Los métodos HTTP que permitiremos en nuestras llamadas.

Para las tres últimas propiedades, el valor  “\*”  significa que se permite todo.

A continuación, **desplegaremos los tres componentes:** el servidor de Eureka, nuestro microservicio de products y el servidor perimetral CW. Por tanto, podemos confirmar que los dos componentes se encuentran registrados en Eureka accediendo a http://localhost:8761.

A screenshot of a computer

Description automatically generated

Figura 11. Microservicio y CW registrados en Eureka. Fuente: elaboración propia.

También, podemos ver las rutas que está resolviendo CW de forma automática al estar conectado al **servidor de descubrimiento Eureka,** al acceder a http://localhost:8762/actuator/gateway/routes:

[

{

“predicate”: “Paths: [/gatewat/\*\*], match trailing slash: true”,

“metadata”: {↔},

“route:id”: “ReactiveCompositeDiscoveryClient\_GATEWAY”,

“filters”: [↔],

“uri”: “lb://GATEWAY”,

“order”: 0

},

{

“predicate”: “Paths: [/ms-inventory-products/\*\*], match trailing slash: true”,

“metadata”: {↔},

“route:id”: “ReactiveCompositeDiscoveryClient\_MS-INVENTORY-PRODUCTS”,

“filters”: [↔]

“uri”: “lb://MS-INVENTORY-PRODUCTS”,

“order”: 0

}

]

Esto quiere decir que todas las peticiones que el CW reciba y que comiencen por la cadena /ms-inventory-products/  serán redirigidas automáticamente a la aplicación ms-inventory-products, que es el nombre de nuestro microservicio.

Probar esto es sencillo, ya que utilizaremos **Postman** para crear un nuevo producto en la base de datos H2 del microservicio y haremos la llamada contra el microservicio directamente:

A screenshot of a computer

Description automatically generated

Figura 12. Petición HTTP  POST  al microservicio de products para crear un nuevo producto. Fuente: elaboración propia.

Como se puede observar, el producto se ha creado correctamente en la base datos H2. A continuación, haremos una **llamada al microservicio** para listar todos los productos disponibles, pero lo haremos a través del **servidor perimetral CW.** Si todo funciona correctamente, no solo debería reenviarse bien la petición, sino que, también, deberíamos ver, por ejemplo, el cepillo de dientes que acabamos de crear.

A screenshot of a computer

Description automatically generated

Figura 13. Petición HTTP  GET  para obtener todos los productos a través del CW. Fuente: elaboración propia.

Como puedes ver, el CW ha sido capaz de **redirigir** la petición correctamente y recibimos como **respuesta el listado** de productos que contiene, únicamente, el producto que acabamos de crear.

En el siguiente vídeo llamado Uso de Spring Cloud Gateway se ilustra el funcionamiento de Cloud Gateway.

### **8.9. Creación de imágenes Docker**

Actualmente, una arquitectura orientada a microservicios depende, totalmente, de los **contenedores** y de los **orquestadores**. En cuanto a las tecnologías para los contenedores, **Docker** es, sin duda, la más famosa y conocida.

Un contenedor es una especie de paquete con una finalidad concreta que contiene en su interior todo lo necesario para llevar a cabo esa finalidad. Por tanto, ofrecen un grado elevado de autosuficiencia.

Los contenedores son una de las formas de **virtualización** más utilizadas hoy en día, ya que con ellos podemos generar el **entorno** necesario para ejecutar un microservicio en poco tiempo.

Dentro del contenedor, encontraremos todas las **herramientas** necesarias para que la aplicación funcione:

* El código de la aplicación.
* Las bibliotecas y las dependencias necesarias.
* Los archivos de configuración.

Para crear un contenedor es necesario, en primer lugar, crear una **imagen**. Una imagen es un archivo, a modo de **plantilla**, a partir de la cual Docker parte para crear otras imágenes más específicas o nuevos contenedores. Para construir una imagen será necesario incluir en la raíz de nuestro proyecto un nuevo archivo llamado **Dockerfile**. En este especificaremos las **acciones** por realizar para generar un paquete que incluya toda nuestra aplicación y que permita que esta sea desplegada y ejecutada de forma correcta.

A continuación, se muestra un ejemplo de Dockerfile sencillo:

#

# Build del proyecto (Multi-Stage)

# --------------------------------

#

# Usamos una imagen de Maven para hacer build de proyecto con Java 17

# Llamaremos a este sub-entorno “build”

# Copiamos todo el contenido del repositorio

# Ejecutamos el comando mvn clean package (Genera un archivo JAR para el despliegue)

FROM Maven:3.8.5-openjdk-17 AS build

COPY . .

RUN mvn clean package

# Usamos una imagen Openjdk v17

# Exponemos el puerto que nuestro componente va a usar para escuchar peticiones

# Copiamos desde “buil” el JAR generado (la ruta de generacion es la misma que veriamos en local)

y lo movemos y renombramos en destino como #

# Marcamos el punto de arranque de la imagen con el comando “java -jar app.jar” que ejecutará nuestro componente.

FROM openjdk:17

EXPOSE 8088

COPY --from=build /target/inventory-products-0.0.1-SNAPSHOT.jar app.jar

ENTRYPOINT [“java”, “-jar”, “/app.jar”]

En ese Dockerfile hay dos etapas o stages claramente diferenciadas: la primera se encarga de **compilar** y **empaquetar** el código fuente desarrollado, mientras que la segunda **ejecuta** el paquete generado el paso anterior. Algunas consideraciones:

* Estamos tomando como imagen base  openjdk:17  y  Maven:3.8.5-openjdk-17 .
* Exponemos al mundo el puerto 8088, que está indicado en application.yml.
* Copiaremos el archivo  inventory-products-0.0.1-SNAPSHOT  al directorio de trabajo de Docker, bajo el nombre de  app.jar .
* Al crearse el contenedor, su punto de arranque será ejecutar el comando  java -jar /app.jar .

Si intentásemos ejecutar el comando  docker build  desde el directorio en el que se encuentra el Dockerfile (la raíz del proyecto), obtendríamos algo similar a esto:

[[~~ JLUMOS ~~ proudcts]$ docker build .

[+] Building 4.6s (7/7) FINISHED

=> [internal] load build definition from Dockerfile

=> => transferring dockerfile: 164B

=> [internal] load.dockerignore

=> => transferring context:2B

=> [internal] load metadata for docker.io/library/openjdk.11

=> [internal] load build context

=> => transferring context: 59.32MB

=> CACHED [1/2] FROM docker.io/library/openjdk:11

@sha256:3eea2dbe168ebcb93426283c2ea06fbaab95673c2f50ed3ed7e86689e73ce790

=> [2/2] COPY target/products-0.0.1-SNAPSHOT.jar. app.ar

=> exporting to image

=> => exporting layers

=> => writing image sha256:a12a98d660cb831a6eef6f266d604d937a4104efd2e27c339c2890e38d4053559

Una vez creada nuestra imagen, podremos crear un contenedor que la ejecute. Para ello, ejecutaremos el comando  docker run -p 8088:8088 <<Firma de la imagen>> , como se muestra en la Figura 14:

[~~ JLUMOS ~~ proudcts]$ docker run -p 8080:8080

a12a98d660cb831a6ef6f266d604d937a4104efd2e27c339c2890e38d4053559

A black rectangular object with white text

Description automatically generated

Figura 14. Creación y arranque de un contenedor asociado a una imagen con Docker. Fuente: elaboración propia.

Si todo funciona correctamente, nuestra aplicación se habrá desplegado sin problemas y será accesible a través de http://localhost:8088/. No obstante, observaremos **errores en el despliegue**, ya que, como ha sido declarada un cliente de Eureka, esta intentará conectarse al servidor y, dado que no lo hemos desplegado, no estará disponible. Entonces, desplegaremos en otro contenedor una imagen correspondiente al servidor Eureka para que nuestro microservicio pueda registrarse. De nuevo, crearemos un Dockerfile en la raíz del proyecto del servidor de Eureka, que tendrá este contenido:

#

# Build del proyecto (Multi-Stage)

# --------------------------------

#

# Usamos una imagen de Maven para hacer build de proyecto con Java 17

# Llamaremos a este sub-entorno “build”

# Copiamos todo el contenido del repositorio

# Ejecutamos el comando mvn clean package (Genera un archivo JAR para el despliegue)

FROM Maven:3.8.5-openjdk-17 AS build

COPY . .

RUN mvn clean package

# Usamos una imagen Openjdk v17

# Exponemos el puerto que nuestro componente va a usar para escuchar peticiones

# Copiamos desde “buil” el JAR generado (la ruta de generacion es la misma que veriamos en local)

y lo movemos y renombramos en destino como #

# Marcamos el punto de arranque de la imagen con el comando “java -jar app.jar” que ejecutará nuestro componente.

FROM openjdk:17

EXPOSE 8761

COPY --from=build /target/eureka-0.0.1-SNAPSHOT.jar app.jar

ENTRYPOINT [“java”, “-jar”, “/app.jar”]

Como podemos observar, lo único que se modifica es el puerto que vamos a exponer (8761) y el nombre del archivo JAR que queremos incluir en el contenedor. Una vez creada la imagen, procederemos a ejecutarla de la misma forma y, cuando se haya ejecutado correctamente y tengamos nuestro servidor Eureka desplegado en local (recordemos que podemos acceder al dashboard de Eureka para comprobarlo), ejecutaremos los siguientes comandos en la consola, que nos devolverá la dirección IP donde se está ejecutando el servidor de Eureka:

* docker ps
* docker inspect -f ‘{{range.NetworkSettings.Networks}}{{-IPAddress}}{{end}}’ <<container-id>>

[[~~ JLUMOS ~~ proudcts]$ docker ps

CONTAINER ID NAME COMMAND CREATED STATUS PORTS NAMES 3321c8e43ea8 65f2e8c23c1c

“java -jar /app.jar” 12 minutes ago Up 12 minutes 0.0.0.0:8761->8761/tcp sharp\_wu

]$ docker inspect -f ‘{{range.NetworkSetting.Networks}}{{.IPAddress}}{{end}}’ 3321c8e43ea8

]$

Pero ¿para qué queremos esta dirección? Recordemos que en la configuración del microservicio inventory-products  hacíamos referencia a la **variable del entorno**  EUREKA\_URL  y, dado que estamos desplegando nuestra imagen en Docker, no podemos aludir a  localhost , ya que estaríamos indicando a la propia instancia y no a la de Eureka. Entonces, para solventar este problema, necesitamos la **dirección IP**que reemplazará a la que hay en el fichero de configuración. Para lograr esto, lo haremos tal y como se indica a continuación:

[~~ JLUMOS ~~ proudcts]$ docker run -p 8080:8080 --env eureka\_url=

http://17.0.2:8761/eureka dce91a1bf1c7c8089c2acd5d103360c2d97ffa8ed1b59f34d7dda50ef0586f12

A black rectangular object with white text

Description automatically generated

Figura 15. Arranque del contenedor con la variable de entorno. Fuente: elaboración propia.

Inmediatamente después del arranque del microservicio, este intentará **conectarse** con el servidor de Eureka, algo que podemos ver a través de los ***logs*** **emitidos** por este cliente de Eureka. Por su parte, el servidor también nos indica que ha registrado una **aplicación nueva:**

A screen shot of a computer

Description automatically generated

Figura 16. El registro de los logs del microservicio  inventory-products  en Eureka. Fuente: elaboración propia.



Figura 17. Logs del servidor de Eureka cuando se registra un nuevo cliente. Fuente: elaboración propia.

Por último, si accedemos con el navegador al dashboard de Eureka a través de la dirección http://localhost:8761 veremos, de una forma más visual e intuitiva, que, efectivamente, nuestro microservicio se ha **desplegado**:

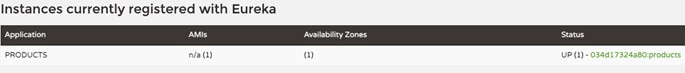


Figura 18. Aplicaciones registradas en Eureka. Fuente: elaboración propia.

También puedes repetir los mismos pasos que hemos realizado para el microservicio  inventory-products  con el servidor perimetral CW y levantar, en local, los tres componentes.

### **8.10. Despliegue en entornos reales**

Hasta ahora, nos hemos limitado a desplegar los componentes de nuestro back-end en nuestra **máquina local.** Si bien esto es una excelente opción para poder probar el software desarrollado y ser más eficientes a la hora de encontrar defectos, no se trata del **escenario final** deseado.

Por norma general, nuestro objetivo será desplegar nuestro back-end en una infraestructura que tenga una conectividad con el dominio donde se encuentra funcionando el front-end de la aplicación.

Para realizar este despliegue, se pueden considerar, a grandes rasgos, las siguientes opciones:

* Hacer uso de un servidor existente, ya sea propio o de una empresa que nos lo facilite.
* Hacer uso de la nube pública (Azure, Amazon Web Services o Google Cloud), privada o híbrida.
* Hacer uso de aplicaciones web similares a Vercel, pero orientadas a back-end.

En esta ocasión, escogeremos la última opción y utilizaremos una **aplicación web** que nos permitirá, de una forma sencilla, desplegar los componentes que hemos ido desarrollando.

Para realizar esto, utilizaremos **Railway**, una plataforma como servicio (PaaS) lanzada en 2020 orientada a desplegar los componentes de back-end, tanto para individuos como para empresas, que ofrece una **capa gratuita** que nos permitirá desplegar hasta tres servicios. Si queremos desplegar más, debemos hacer uso de varias cuentas o pasar a un modelo de pago por uso.

Puedes obtener más información en su documentación a través del siguiente enlace:

<https://docs.railway.app/>

Para poder dar continuidad al material ya visto, se agradece realizar el registro a través del siguiente enlace:

<https://railway.app/?referralCode=65Joj_>

A screenshot of a phone

Description automatically generated

Figura 19. Railway. Fuente: Railway.app.

Lo primero que tendremos que hacer, obligatoriamente, es **crear una cuenta** en esta aplicación y **enlazarla**con nuestra cuenta de GitHub. Si la aplicación considera que la cuenta es válida, se habilitará la capa gratuita y podremos empezar a trabajar.

En el caso de que esto no sea así, porque la cuenta de GitHub sea demasiado nueva o porque Railway no la considere segura, no quedará otra opción que pasar a un **modelo de pago** por uso. En este caso, se recomienda utilizar la opción de prepago (Developer Plan) por las siguientes razones:

* En el modelo de pago por uso, Railway regala 5 dólares al mes. Es decir, si recargamos nuestra cuenta con 5 dólares (es el mínimo), tendremos 5 dólares extra para gastar antes de que se comiencen a gastar los 5 dólares con los que hemos cargado la cuenta.
* En el caso de que gastemos los 5 dólares y aquellos con los que hayamos cargado la cuenta, todos los servicios se detienen y no se aplica ningún cargo más.

A screenshot of a computer

Description automatically generated

Figura 20. Developer Plan en Railway. Fuente: Railway.app.

Una vez tengamos una cuenta creada, iremos al dashboard de la aplicación donde, en este momento, no habrá nada creado.



Figura 21. Railway dashboard. Fuente: Railway.app.

Haremos click en «+ New Project» y seleccionaremos la opción de despliegue mediante un **repositorio de GitHub.** Con esta opción, podremos **desplegar el código** tanto desde repositorios públicos como privados, por lo que no tendrás que marcar tu repositorio como público en ningún caso (salvo que quieras hacerlo).

A screenshot of a phone

Description automatically generated

Figura 22. Tipología de proyectos. Fuente: Railway.app.

A continuación, veremos la **lista de repositorios** de nuestra cuenta de GitHub. Simplemente, tenemos que seleccionar el componente a desplegar. En este ejemplo vamos a desplegar Eureka.

A screenshot of a phone

Description automatically generated

Figura 23. Despliegue con repositorio de GitHub. Fuente: Railway.app.

Railway nos preguntará si queremos desplegar el componente tal cual o si queremos añadir algún **tipo de variable** a este despliegue, por lo que le daremos valor a las diferentes variables del entorno o a las configuraciones específicas de despliegue que presente nuestro componente. Para todos los **componentes**(microservicios) que despleguemos, tendremos que especificar obligatoriamente el **puerto** para utilizar el que tenemos marcado en nuestro código (en el caso de Eureka, es el 8761).

A screenshot of a computer

Description automatically generated

Figura 24. Añadir variables de entorno. Fuente: adaptado de Railway.app.

En el apartado «Settings» podemos ver la información del servicio:

* La rama de la cual se está leyendo el código (master por defecto). Si no se realiza ningún cambio en la configuración, Railway desplegará una nueva versión del servicio con cada nuevo commit en la rama seleccionada, aunque se puede deshabilitar (sin embargo, si se hacen muchos commits es recomendable tenerlo habilitado).
* El dominio del servicio. Inicialmente, no habrá, pero crearemos uno haciendo click en «Generate Domain» para poder tener un nombre de dominio para usar en otros microservicios (por ejemplo, para que se registren en Eureka).
* La información de cómo se está construyendo el paquete de despliegue (se hace por defecto con el Dockerfile que tenemos en los repositorios).

A screenshot of a computer

Description automatically generated

Figura 25. Configuración de servicio. Fuente: Railway.app.

A screenshot of a computer

Description automatically generated

Figura 26. Configuración de servicio. Fuente: Railway.app.

Si todo funciona correctamente, deberíamos ver algo similar a lo expuesto en la Figura 26 de la pestaña «Deployments» de nuestro servicio de Eureka:

A screenshot of a phone

Description automatically generated

Figura 27. Despliegue activo. Fuente: Railway.app.

Haciendo click en la URL facilitada, deberíamos ver el dashboard de Eureka. Notarás que no se tuvo que indicar el puerto en la URL, al contrario de lo que hacíamos en local. Esto es posible gracias al binding que hemos hecho, lo que le da valor a la variable del entorno  PORT , que hace que el servicio de Railway escuche el puerto especificado.

* Si observamos algún otro deploy en curso, podremos abortarlo; aunque puede tratarse del despliegue inicial, antes de la modificación la variable del entorno del puerto, que aún no ha terminado. De esta manera, podremos ver los logs del build y de la ejecución haciendo click en «View Logs».
* Si deseamos detener el servicio para dejar de generar gastos, podemos hacer click en los tres puntitos que hay al lado de «View Logs» y elegir la opción «Remove». Esto detendrá la ejecución del servicio y se recomienda realizar esta operación antes de terminar la sesión de trabajo para evitar gastos no deseados.

A screenshot of a phone

Description automatically generated

Figura 28. Detención de despliegues. Fuente: Railway.app.

* Si queremos volver a desplegar lo mismo que ya teníamos desplegado, simplemente, iremos al último despliegue detenido y haremos click en «Rollback».

A screenshot of a phone

Description automatically generated

Figura 29. Reinicio de despliegues. Fuente: Railway.app.

Ahora bien, ¿qué ocurriría si queremos que otro microservicio se registre en Eureka? En nuestros despliegues locales indicábamos que Eureka se alojaba en  http://localhost:8761/eureka . ¿Qué **valor**tendríamos que darle a la **variable del entorno** EUREKA\_URL  ahora? Pues bien, si el host ahora es https://back-end-eureka-production.up.railway.app  (dato proporcionado por Railway), la variable del entorno debería tener el valor  https://back-end-eureka-production.up.railway.app/eureka .

Otro detalle que debemos tener en cuenta es que, dentro de Railway, no se permite el tráfico de red interno. Esto implica utilizar el nombre de host y no la dirección IP cuando registremos nuestros microservicios en el Eureka desplegado.

Esto podemos conseguirlo fácilmente con la siguiente configuración en cada uno de los componentes que se registran con Eureka:

## Configuration for Eureka

eureka:

client:

service-url:

defaultZone: ${EUREKA\_URL:http://localhost:8761/eureka}

instance:

preferIpAdderss: false

hostname: ${HOSTNAME:localhost}

non-secure-port-enabled: false

secure-port-enabled: true

non-secure-port: 80

secure-port: 443

* Se especifica que no se prefiere la dirección IP, por lo que se registrará el componente con el nombre de dominio.
* El hostname tendremos que incluirlo por la variable del entorno. Es el host que nos da Railway.
* Railway solo permite tráfico entrante por el puerto 80 o 443, por tanto, deshabilitamos el 80 y activamos el 443. De esta forma, todo el tráfico irá al puerto seguro por HTTPS.

No dudes en poner esto en práctica. Intenta desplegar Eureka, CW y el microservicio  inventory-products  en tu cuenta.

A continuación, veremos el vídeo llamado Despliegue de microservicios en entornos reales en donde se muestra, de forma visual, cómo desplegar imágenes Docker en Railway.

A continuación, veremos el vídeo llamado Despliegue de bases de datos en entornos reales para entender cómo desplegar una base de datos relacional en Railway.