Obtener tarifa a aplicar

El objetivo el endpoint desarrollado es obtener la tarifa a aplicar a un producto en una rama (Brand).

# Historias de usuario

1. Como el vendedor Quiero obtener el precio aplicar Para cerrar una venta.

Para ello se habilitará un endpoint al que se le pasaran la fecha de aplicación la rama y el producto, este deberá devolver el precio a aplicar o un error cuando no encuentre precio para la información suministrada.

Cuando se encuentren varios precios a aplicar, se deberá elegir el precio con mayor prioridad.

# Diseño técnico

1. Requisitos

* Java sdk 11 (jdk-11.0.10)
* Maven (apache-maven-3.8.7)
* Postman
* Idle (IntelliJ IDEA 2022.3.1 (Community Edition)). Es occional.
* Git

1. Introducción

La aplicación esta desarrollada en Java y se usa principalmente el ecosistema de Spring Boot.

A continuación, describo lo utilizado en el desarrollo de la aplicación

Java 11. Permite programación funcional y otras ventajas.

Spring Boot: se ha usado la versión 2.7.7. Dentro del ecosistema de spring boot hemos usado los starters siguientes:

* spring-boot-starter-web: Empaqueta el spring framework web y es necesario para poder desarrollar el servicio Rest
* spring-boot-starter-data-jpa: empaqueta las funciones para acceso a base de datos. Contiene hibernate, un ORM que mapea las clases de java a tablas de base de datos.
* spring-boot-starter-test: empaqueta las funcionalidades para realizar test

Hay un repository del proyecto en GitHub [ralapont/api-tarifas: ejercicio api rest de tarifas (github.com)](https://github.com/ralapont/api-tarifas) donde se puede encontrar el proyecto

También usamos lo siguiente:

* Lombok: facilita la creación y mantenimiento de los javabeans.
* Modelmapper: facilita el mapeo entre javabeans. Mapea los dtos con las entities y viceversa.
* jackson-datatype-jsr310: permite el uso de los tipos de java 8 en los procesos de serialización de objetos
* Junit jupiter: creación de los test.

1. Base de datos

Como base de datos, en esta aplicación usamos H2 embebida. Los parámetros de conexión serán los siguiente:

* Datasource: jdbc:h2:mem:testdb
* Driver: org.h2.Driver
* Username: sa
* Password: password

Con la aplicación arrancada, se puede acceder al UI de H2 desde un browser con la url <http://localhost:8080/h2-ui>

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Las tablas se crean a partir de las clases Entities que definamos en el proyecto, en nuestro caso BrandEntity que almacena las ramas y PricesEntity que almacenara los precios por rama, producto y fechas de aplicación entre otros.

También estas tablas serán pobladas al arrancar la aplicación con los datos de ejemplo que nos suministran en la definición de la prueba. Para esto, creamos un fichero SQL que contiene los insert a las tablas data.sql que situamos en la carpeta de recursos.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Tabla

Descripción generada automáticamente con confianza media

Estas tablas están relacionadas como muestra el siguiente diagrama entidad relación:

Diagrama

Descripción generada automáticamente

En las siguientes tablas describimos los campos de las tablas

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| BRAND | | | | |
| Campo | Tipo | Descripción | Pk | Fk |
| Id | BigInt | Identificador de Brand | Marca de verificación con relleno sólido |  |
| Description | Character Varying | Nombre de Brand |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PRICES | | | | |
| Campo | Tipo | Descripción | Pk | Fk |
| Id | BigInt | Identificador de Price | Marca de verificación con relleno sólido |  |
| Brand\_id | BigInt | Identificador de Brand |  | Marca de verificación con relleno sólido |
| Start\_date | Timestamp | Fecha y hora de comienzo de aplicación del precio |  |  |
| End\_date | Timestamp | Fecha y hora de fin de aplicación del precio |  |  |
| Price\_list | Integer | Identificador de la tarifa aplicable |  |  |
| priority | Integer | Desambiguador de aplicación de precios. Si dos tarifas coinciden en un rango de fechas se aplica la de mayor prioridad (mayor valor numérico) |  |  |
| Price | Double Precisión | Precio final de venta |  |  |
| Curr | Charecter Varying | Codigo iso de la moneda |  |  |

1. Implementación

La estructura de paquetes de la aplicación la podemos representar en el siguiente diagrama:

Diagrama

Descripción generada automáticamente

* Controller: Implemeta el Servicio Rest (PriceController) y el tratamiento de errores lanzado por el controller (PriceControllerAdvice). Usa el paquete Dtos para mandar información al exterior. Usa el paquete Exception para comunicar errores de negocio con el exterior
* Service: Implementa la capa de negocio (PriceService). Usa el paquete entities para interaccionar con la base de datos y el paquete Dtos para interaccionar con el Controller
* Repository: Implementa el acceso a la base de datos (PricesRepository). Usa el paquete Entities donde se definen los datos a mapear con la base de datos.

Para obtener los precios por Brand, producto y fecha de aplicación se usa una query nombrada en JPQL, esta query se puede encontrar en la clase PriceEntity. La podemos ver a continuación

*select p*

*from PRICES p*

*where p.brandId.id = ?1*

*and p.productId = ?2*

*and p.startDate <= ?3*

*and p.endDate > ?3*

También tenemos los siguientes paquetes:

* Config: paquetes que instancian y configuran Beans de spring para inyectarlos en los anteriores (ejemplo el Modelmaper)

El siguiente gráfico, muestra los paquetes y sus clases implementadas en la aplicación

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

1. Pruebas

Para la implementación de los test se ha usado la herramienta Junit 5 jupiter.

En la clase PricesControllerTest se implementan los test de integración del endpoint. No se mockea y en el primer test que aparece, getRateToApplyTest, se ejecutan los cinco test requeridos, es decir:

* Test 1: petición a las 10:00 del día 14 del producto 35455 para la brand 1 (ZARA)
* Test 2: petición a las 16:00 del día 14 del producto 35455 para la brand 1 (ZARA)
* Test 3: petición a las 21:00 del día 14 del producto 35455 para la brand 1 (ZARA)
* Test 4: petición a las 10:00 del día 15 del producto 35455 para la brand 1 (ZARA)
* Test 5: petición a las 21:00 del día 16 del producto 35455 para la brand 1 (ZARA)

En el segundo test se prueba que pasa si con los datos de entrada, no se obtiene ningún precio.

En la siguiente figura, se muestra los resultados del test ejecutados desde el idle.

Una captura de pantalla de una computadora

Descripción generada automáticamente

En la clase PriceServiceTest se implementan dos test

En el primer test se mockea las llamadas al repository y al modelmapper.

En la segunda prueba se testea el caso de que no se encuentren precios para los datos de entrada.

En la siguiente figura se puede ver la ejecución de los test del servicio

Una captura de pantalla de una computadora

Descripción generada automáticamente

En la clase PriceRespository se hace la prueba de la query creada para obtener los precios por rama, producto y fecha de aplicación.

En la siguiente figura, se muestra la ejecución de este test desde el idle.

Texto

Descripción generada automáticamente

Con los test implementados, se cubre el 100% del Controller, Service y Repository

Texto

Descripción generada automáticamente

1. Operativa

Para la construcción y ejecución de la aplicación se usa la herramienta Mavel.

En el fichero pom.xml se pueden encontrar las dependencias, etc.

A continuación, describimos como compilar, desplegar y ejecutar la aplicación usando comandos Maven desde la consola.

Situandose en el directorio root del proyecto:

mvn clean package: compila la aplicación y genera el fichero jar

Interfaz de usuario gráfica, Tabla

Descripción generada automáticamente

mvn test: ejecuta los test

Texto

Descripción generada automáticamente

mvn exec:java -Dexec.mainClass="com.example.apitarifas.ApiTarifasApplication": ejecuta la aplicación

Texto

Descripción generada automáticamente

A continuación, desde la herramienta Postman se puede invocar al servicio rest

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente