

Examen Spring Boot

Lorenzo Manuel Rosas Rodríguez

28 de septiembre de 2023

Índice

1	Resumen General.	3
2	Dependencias.	3
3	Estrucctura Proyecto	3
4	Base de datos	7
5	Conclusiones	7

Índice de figuras

3.1	Estructura carpetas general proyecto	3
3.2	Ejemplo ejecución método examen Controller Uno.	4
3.3	Ejemplo ejecución método examen Controller Dos.	4
3.4	Ejemplo manejador excepción.	5
3.5	Imagen creación relación One to Many, Uno.	5
3.6	Imagen creación relación One to Many, Dos.	5
3.7	Query nativa para el examen.	6
3.8	Ejecución tests controller.	6
3.9	Ejecución tests descritos en el examen.	6
3.10	Ejecución tests repositorio.	7
4.1	Archivo application.properties.	7
4.2	Archivo data.sql.	7
5.1	Comando ejecución correcta método PUT API REST en terminal.	8
5.2	Comando ejecución correcta método POST API REST en terminal.	8

1. Resumen General.

En este documento redactaré lo que considero más importante para entender el proyecto que he realizado para este examen para la oferta de empleo que se me ha ofrecido. Agradezco la oportunidad ofrecida, la demora y espero que cumpla con las exigencias del examen.

2. Dependencias.

El proyecto está realizado en la versión de **Spring Boot 3.0.11** y la versión de **Java 17**, ya que por lo que he leído son las versiones más estables y extendidas, aunque no tendría problema en trabajar con versiones más recientes, las cuales he desarrollado también.

3. Estructura Proyecto

El **proyecto** lo he dividido en la siguiente estructura de directorios, considerando que es la más extendida para trabajar:

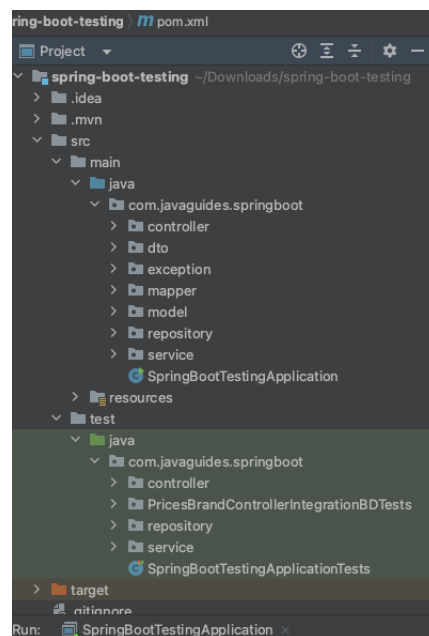


Figura 3.1: Estructura carpetas general proyecto

Cada carpeta contiene:

- **main/java/com.javaguides.springboot/src/controller**: contiene los métodos de la **Api Rest**, donde he implementado los típicos métodos **Get**, **Post**, **Update**, **Delete** además del método del propio examen, que he llamado **findByDateRequestIdProductIdBrand**. Para testearlos, proporciono la siguiente web al ejecutar el proyecto desde **Spring Boot**, <http://localhost:8080/swagger-ui/index.html>, donde se puede testear todos los métodos, además del requerido en el examen. Les muestro una captura del test del método del examen.

GET /api/prices/fechaAppli/{fechaAplicacion}/{prodId}/{brandId} Get method Price by fechaAplicacion prodId brandId Rest API with Priority Higher

Get method Price by fechaAplicacion prodId brandId Rest API with Priority Higher

Parameters

Name	Description
fechaAplicacion * required string(\$date-time) (path)	2020-07-14T14:00:11
prodId * required integer(\$int64) (path)	35455
brandId * required integer(\$int64) (path)	1

Execute Clear

Figura 3.2: Ejemplo ejecución método examen Controller Uno.

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:8080/api/prices/fechaAppli/2020-07-14T14:00:11/35455/1' \
  -H 'accept: */*'
```

Request URL

```
http://localhost:8080/api/prices/fechaAppli/2020-07-14T14:00:11/35455/1
```

Server response

Code	Details
200 Undocumented	<p>Response body</p> <pre>{ "price": 38.95, "brand_ID": 1, "price_LIST": 0, "product_ID": 35455, "start_DATE": "2020-06-15T16:00:00", "end_DATE": "2020-12-31T23:59:59" }</pre> <p>Response headers</p> <pre>connection: keep-alive content-type: application/json date: Wed, 27 Sep 2023 22:19:45 GMT keep-alive: timeout=60 transfer-encoding: chunked</pre>

Responses

Code	Description	Links
PricesExamDto	HttpStatus.OK	No links

Media type

/

Example Value Schema

```
{
  "price": 0,
  "brand_ID": 0,
  "price_LIST": 0,
  "product_ID": 0,
  "start_DATE": "2023-09-27T22:19:45.341Z",
  "end_DATE": "2023-09-27T22:19:45.341Z"
}
```

Figura 3.3: Ejemplo ejecución método examen Controller Dos.

Como se puede apreciar en la imagen, se devuelven los datos que se requieren en el examen: **identificador de producto, identificador de cadena, tarifa a aplicar, fechas de aplicación y precio final a aplicar.**

- **main/java/com.javaguides.springboot/src/dto**: Contienen **PricesDto** y **PricesExamDto**, que son las estructuras de datos que uso en **PriceBrandcontroller** para serializar correctamente en **Json** para la ejecución correcta de los métodos **Api Rest**.
- **main/java/com.javaguides.springboot/src/exception**: aquí he creado la clase **ResourceNotFoundException** para manejar correctamente las **excepciones** que se produzcan por ejemplo al no encontrar un elemento en la **BD**.

```
if(priceBD.isPresent()){
    throw new ResourceNotFoundException("Price already exists in BD, ID -- "+price.getPrice_ID());
}
```

Figura 3.4: Ejemplo manejador excepción.

- **main/java/com.javaguides.springboot/src/mapper**: aquí he creado las clases para mapear correctamente los datos entre los objetos de las **entidades**, instanciadas en el repositorio, y los **dto**, especificados anteriormente.
- **main/java/com.javaguides.springboot/src/model**: aquí he creado las clases crear las entidades de la base de datos, en mi caso son **Prices** y **Brand**. Entre las dos entidades he creado la relación **One(Brand) to Many(Prices)**, de la siguiente forma:

```
@Schema(
    description = "Variable donde se almacena la coleccion de precios de la relación One-To-Many con prices. One(Price)-To-Many"
)
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER, mappedBy = "brand")
private Set<PRICES> prices = new HashSet<>();
```

Figura 3.5: Imagen creación relación One to Many, Uno.

```
@Schema(
    description = "Variable donde se almacena la Cadena, Brand, de la relación One-To-Many con prices. One(Price)-To-Many(Brand)"
)
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "BRAND_ID", referencedColumnName = "id")
private BRAND brand;
```

Figura 3.6: Imagen creación relación One to Many, Dos.

En esa relación destacamos dos tipos, **Lazy**, la carga de los objetos de la relación se producen a demanda, es decir, cuando un cliente los solicita, y **Eager**, la carga de los objetos de la relación se produce en el mismo momento.

- **main/java/com.javaguides.springboot/src/repository**: aquí he creado las **interfaces** para realizar la consulta en la base de datos. Para realizar la consulta pedida en el examen y su posterior testeo, he implementado una **query nativa**, interpretando en el enunciado que tendría que encontrar los **productos** que se encontraran entres las **START DATE** y **END DATE**, que coincidieran con **PRODUCT ID** y **BRAND ID**, y si encontraba más de uno, devolviera el de más prioridad, lo que se hace en **controller**.

```

("SELECT p from PRICES p where (:fechaAppli BETWEEN p.START_DATE AND p.END_DATE) AND p.PRODUCT_ID = :idProd AND p.brand.id = :idBrand"
RICES> findByDateRequestIdProductIdBrand(@Param("fechaAppli") LocalDateTime fechaAppli,
                                           @Param("idProd") long idProd, @Param("idBrand") long idBrand);

```

Figura 3.7: Query nativa para el examen.

- `main/java/com.javaguides.springboot/src/service`: para comunicar **repository** con **controller**.
- `test/java/com.javaguides.springboot/src/controller`: para testear que los métodos **API REST** de controller funcionan correctamente, simulado con **Mocks**. No usa los datos de la base de datos, los creo en el tests.
Nota: funcionan todos bien excepto el del metodo **POST** para crear **BRAND** y **PRICE**, la relación **ONE TO MANY** no la hace bien, si me dais más tiempo lo resuelvo sin problema.

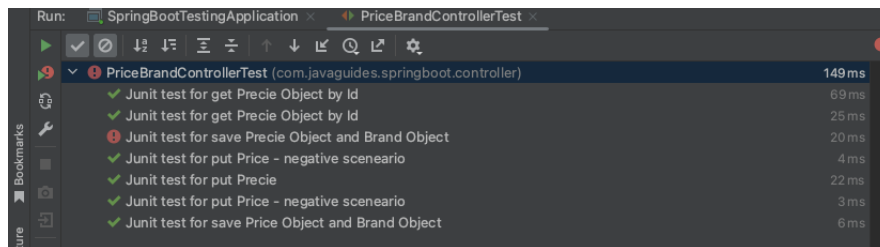


Figura 3.8: Ejecución tests controller.

- `test/java/com.javaguides.springboot/src/PricesBrandControllerIntegrationBDTests`: tests para testear el método que se pide en el **examen**, con los casos de prueba del mismo. Uso la base de datos creada con los datos iniciales que se da en el examen también. En mi caso, la ejecución de todos los tests es la **correcta**.

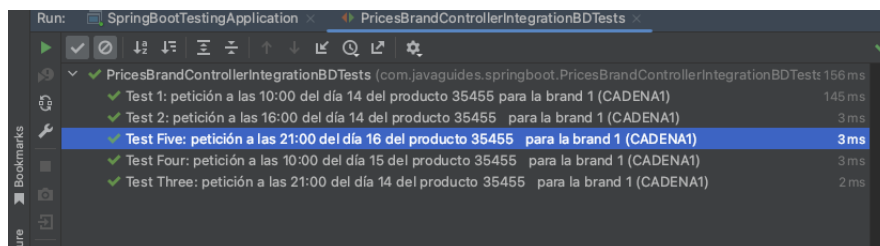


Figura 3.9: Ejecución tests descritos en el examen.

- `test/java/com.javaguides.springboot/src/PriceBrandRepositoryTests`: tests para testear el correcto funcionamiento del repository.

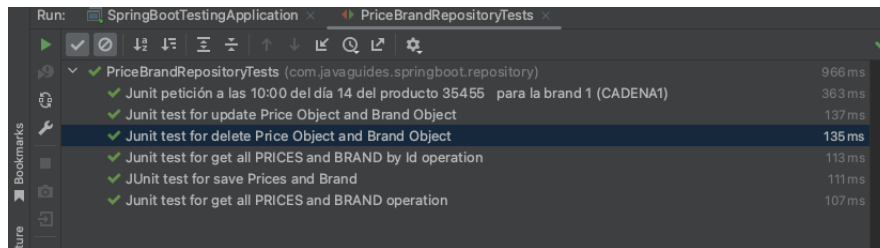


Figura 3.10: Ejecución tests repositorio.

4. Base de datos

Como se requería en el examen, la base de datos usada ha sido **h2**, lo cual especifico en el archivo `test/java/resources/application.properties` e inicializo en `test/java/resources/data.sql`

```
spring.datasource.url=jdbc:h2:mem:testdb;DB_CLOSE_ON_EXIT=TRUE
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# Also disable the automatic schema generation on application start
spring.datasource.initialization-mode=never

spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
spring.h2.console.settings.trace=true

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.jpa.defer-datasource-initialization=true

spring.jpa.hibernate.ddl-auto=create-drop
```

Figura 4.1: Archivo application.properties.

```
INSERT INTO brand (id, name, description) VALUES
(1, 'Brand type One Example', 'Brand type One Example');

INSERT INTO prices (BRAND_ID, START_DATE, END_DATE, PRICE_LIST, PRODUCT_ID, PRIORITY, PRICE, CURR) VALUES
(1, '2020-06-14 00:00:00', '2020-12-31 23:59:59', 1, 35455, 0, 35.50, 'EUR');
INSERT INTO prices (BRAND_ID, START_DATE, END_DATE, PRICE_LIST, PRODUCT_ID, PRIORITY, PRICE, CURR) VALUES
(1, '2020-06-14 15:00:00', '2020-06-14 18:30:00', 2, 35455, 1, 25.45, 'EUR');
INSERT INTO prices (BRAND_ID, START_DATE, END_DATE, PRICE_LIST, PRODUCT_ID, PRIORITY, PRICE, CURR) VALUES
(1, '2020-06-15 00:00:00', '2020-06-15 11:00:00', 3, 35455, 1, 30.50, 'EUR');
INSERT INTO prices (BRAND_ID, START_DATE, END_DATE, PRICE_LIST, PRODUCT_ID, PRIORITY, PRICE, CURR) VALUES
(1, '2020-06-15 16:00:00', '2020-12-31 23:59:59', 4, 35455, 1, 38.95, 'EUR');
```

Figura 4.2: Archivo data.sql.

5. Conclusiones

Yo pienso que he solventado todos los requisitos del examen, salvo métodos **POST** y **PUT** de la **API REST**, los cuales si me dáis mas tiempo resuelvo. Los cuales se ejecutan bien

con los siguientes comandos de terminal introduciendo un **id** de **brand** válido en la petición, pero si no se introduce **id** de **Brand** válido da error.

```
* --> curl -i -X PUT http://localhost:8080/api/prices/price/1 -H 'Content-Type: application/json'
* -d '{"prices": 30.5, "curr": "EUR", "price_ID": 1, "start_DATE": "2020-06-14T00:00:00",
* "end_DATE": "2020-12-31T23:59:59", "price_LIST":1, "product_ID": 35455, "priority":0, "brand_ID": 1}'
*/
```

Figura 5.1: Comando ejecución correcta método PUT API REST en terminal.

```
* PricesDB - Localización: /src/main/java/com.javasdboss.springboot007/src/PricesDB
* --> curl -i -X POST http://localhost:8080/api/prices/price -H 'Content-Type: application/json' -d
* '{"price": 22.2, "curr": "EUR", "price_ID": 8, "start_DATE": "2020-06-14T00:00:00", "end_DATE":
* "2020-12-31T23:59:59", "price_LIST":1, "product_ID": 35455, "priority":0, "brand_ID": 1}'
*/
```

Figura 5.2: Comando ejecución correcta método POST API REST en terminal.

En cuanto a **foreign key**, en la relación **One To Many**, interpreto es la clave primaria con la que referencia la tabla con la otra en la relación, **BRAND ID**.

Espero que se queden reflejados todos mis conocimientos, además tengo más, y si tenéis alguna crítica que me haga mejorar no dudéis en decírmela, muchas gracias y un cordial saludo.