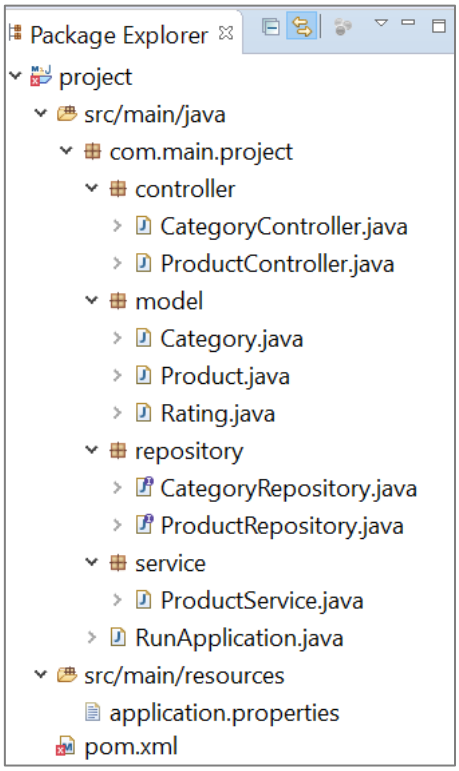
Technical Documentation

Introduction

The objective of this document is to explain the development of an application that structures a database with Products and Categories. It should be able to manage and get records using Restful endpoints with JSON data.

Project Structure



Tools and technologies used

- Spring Initializr
- Java 8
- Maven
- Framework Spring Boot 2.6.1
- Dependencies: Rest Repositories, H2 Database e Spring Data JPA
- Eclipse IDE Oxygen

Description of the Project Structure

Model

Category.java – file that represents the Category entity. Each category record has an id, name and may be associated to one product, defined by the product_id.

Product.java – file that represents the Product entity. Each product has an id, title, price, description, image (stored as an URL), rating and may have a lot of categories

Rating.java – file that represents the Rating entity. This object belongs to a Product and is divided into Rate and Count

Repositories

CategoryRepository.java – interface that implements a method to find categories by their name. Its implementation is done through Spring functionalities that implement CRUD actions based on the method's name

ProductRepository.java – interface that includes three methods: one returns a list of Products based on their number of categories, another one returns a list of products grouped by category and a last one that returns a list of products ordered by price. Each method is developed considering a query that returns the intended products

Service

ProductService.java – implements the API operations, resorting to the methods implemented in the Repositories classes

Controllers

CategoryController.java – file that develops the endpoints that represent the requested operations, related to Categories

ProductController.java – file that develops the endpoints that represent the requested operations, related to Products

Other files

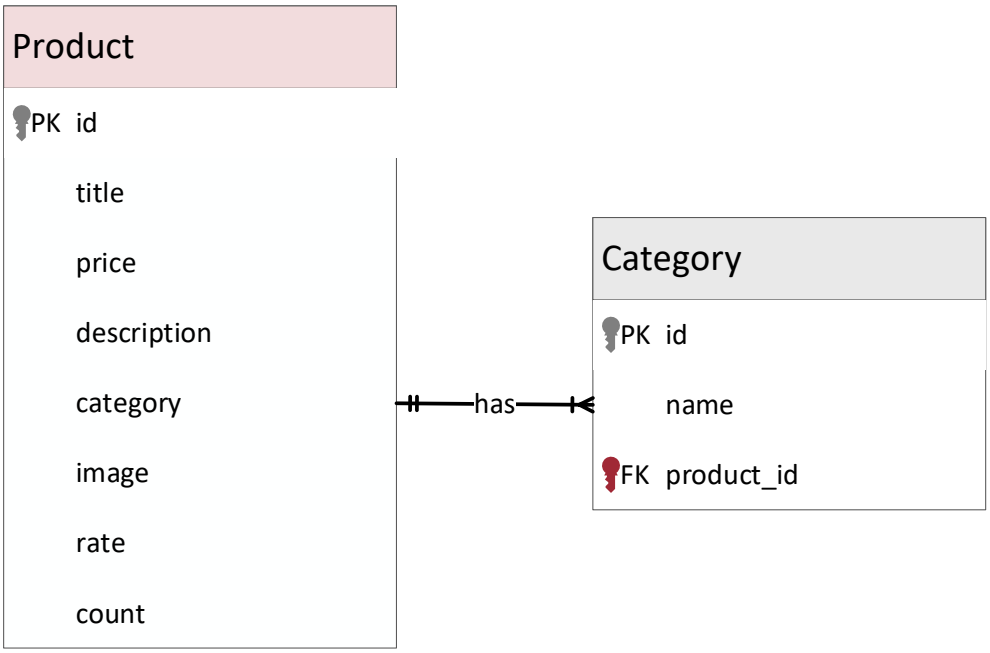
RunApplication.java – responsible for running the application. Also, it fills the database when the server starts up, using a JSON response from a provided endpoint (https://fakestoreapi.com/products)

application.properties – defines the database H2 parameters. These parameters allow to access the database through a browser and to run queries

pom.xml – includes Maven, JPA and Rest dependencies

Implemented Solution

To develop this application, it was implemented a database resorting to the H2 Database system. This database is populated when the server starts up, in `RunApplication.java`, using JSON data sent in the provided endpoint. The database is structured with two tables: Product and Category. A product can have many categories and a category record can have one product. Next, we have an Entity Relationship Diagram with the tables and how they relate:



In order to develop an API with the requested operations, it was created endpoints for each one. Next, it is shown details about each operation.

1. Add products

This implementation is done in the `ProductController` class, with the `save(List<Product> products)` method, defined in the `ProductService` class. First, the list of products submitted is stored in the table and then each category is added one by one in the Category table, defining the product_id according to its product.

Example of JSON data sent:

```
[{"title":"a","price":22.3,"description":"c","category":
"ss,aa,cc","image":"ff","rating":{"rate":4.1,"count":259}},{"tit
le":"b","price":11.2,"description":"d","category":
"dd","image":"kk","rating":{"rate":2,"count":100}}]
```

Endpoint: http://localhost:8080/products (POST)

2. Add categories

This implementation is done in the `CategoryController` class. One category is added if another category with the same name does not exist in the table. If that category already exists, a warning is shown with that information.

The content is sent in JSON format and each Category must be added one by one, filling the Name column.

Example of JSON data sent: `{"name":"test"}`

Endpoint: http://localhost:8080/categories (POST)

3. Return a list of products filtered by the number of categories. This parameter is optional and when it is not sent, it should return a list of all products

When implementing this operation, it was defined two endpoints to make the parameter optional. According to the number received, for example 5, it is obtained a list of products that have 5 associated categories. If no number is sent, then we get a list with every product added. Here, it was defined the `findProductsByCategoryCount (@Param("category_count") Integer categoryCount)` method in `ProductRepository.java`, which returns a list of products according to the defined query.

Endpoints: http://localhost:8080/listByCategoryCount and http://localhost:8080/listByCategoryCount/{count} (GET)

4. Return a list of products grouped by category

This operation was developed in the `listProductsGroupedByCategory()` method in `ProductRepository.java`, which includes a query to obtain the wanted list of products.

Endpoint: http://localhost:8080/groupByCategory (GET)

5. Return a list of products ordered by price

This operation was developed in the `listProductsOrderedByPrice()` method in `ProductRepository.java`, which includes a query to obtain the wanted list of products.

Endpoint: http://localhost:8080/orderByPrice (GET)

6. Update a product

This development uses the same method to add a product, `save(List<Product> products)`, from `ProductService.java`. In this case, it is expected an id parameter in the endpoint, that represents the id of the product that is going to be updated. The categories associated to the product to be updated are deleted from the database and new categories and connections are created in order to ensure old categories stop being associated to the updated products.

Example of JSON data sent:

```
{"title":"a","price":22.3,"description":"c","category":
"ss,aa,cc","image":"ff","rating":{"rate":4.1,"count":259}}
```

Endpoint: http://localhost:8080/products/{id} (PUT)

Tools used to test the application

To test the endpoints, I accessed the website https://reqbin.com/. The queries developed were tested in the H2 console, through the URL http://localhost:8080/h2-console.

Steps to run the application

1. Extract the project from the zip file
2. From the root folder “project”, open command-line interface and execute:
   `mvn spring-boot:run`