Author: Mehmet Yiğit Aktaş

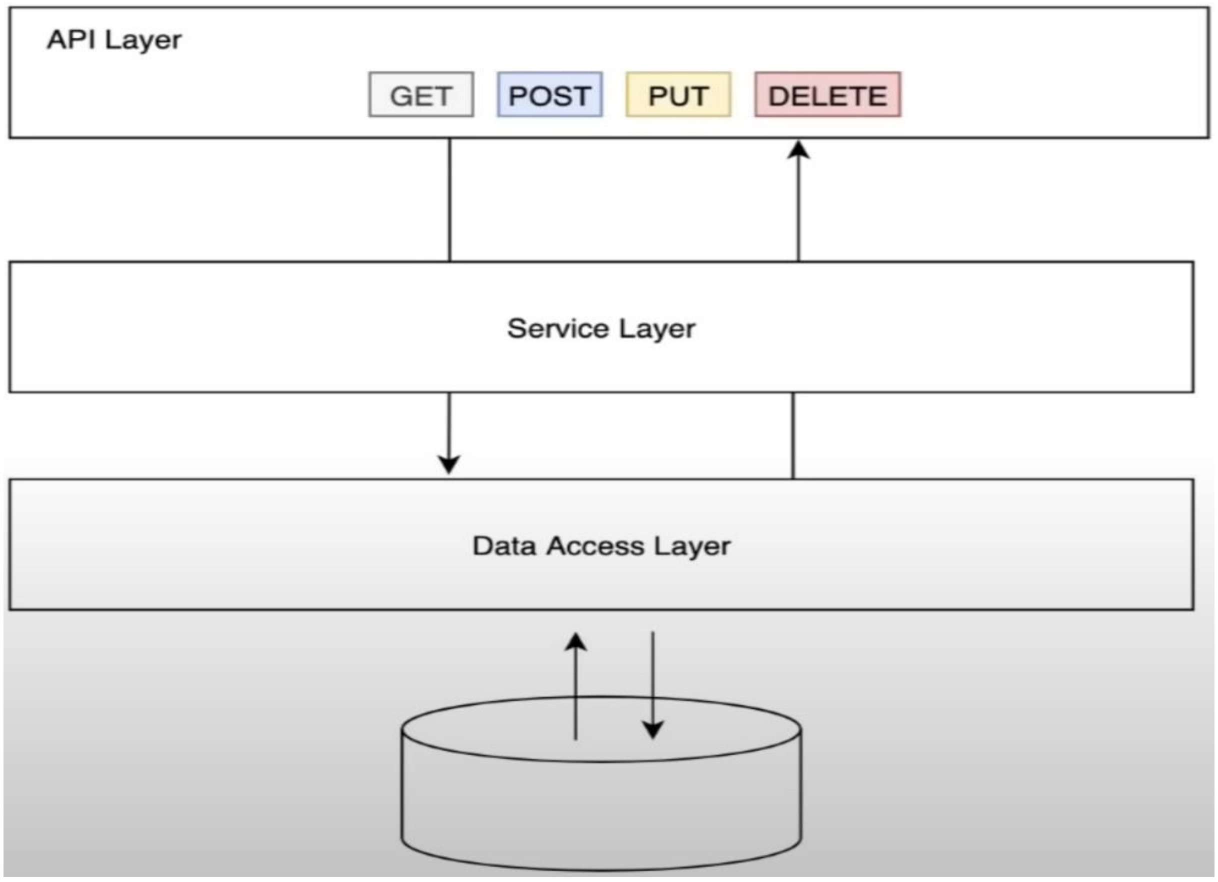
Vending Machine API Documentation

Introduction

This application simulates the backend and the database of a Vending Machine. The server is built with Java SpringBoot framework and the database is built with PostgreSQL.

Overview

The hierarchy of the server can be seen below:



Database/ Data Access Layer

Starting from the bottom, this project interacts with a local database to edit product fields like price and stock. Spring Data JPA and PostgreSQL Driver dependencies are used to integrate this database into the server. The ProductRepository class assumes the roles of Data Access Layer and extends the JPA interface to call necessary functions instead of writing each required SQL query by hand.

Service Layer

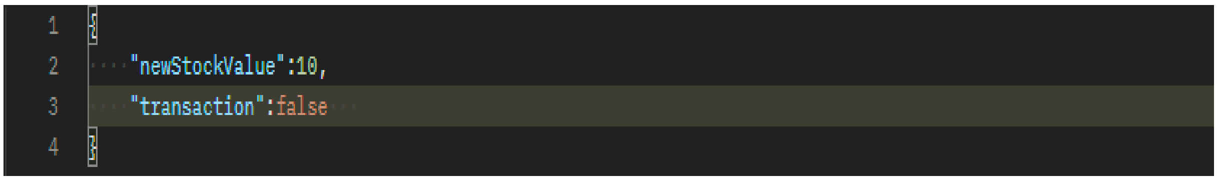
The ProductService class assumes the role of the Service Layer and holds a reference to the previous Data Access Layer in order to make necessary queries to the database according to the API Layer's requests. The business logic of this application is also written here. Most of the error handling is also written here such as solving queries to a product that does not exist in the database.

API Layer

The ProductController class assumes the role of the API layer and holds a reference to the previous Service Layer in order to direct the request from the client and call necessary functions of the service class to apply business logic. It performs this operation by creating endpoints for HTTP requests, store the body of the request in DTO's (Data Transfer Objects) and apply some simple preliminary logic so that required function of the service class is called. The base URL for the API is : localhost:8080/api/v1/product

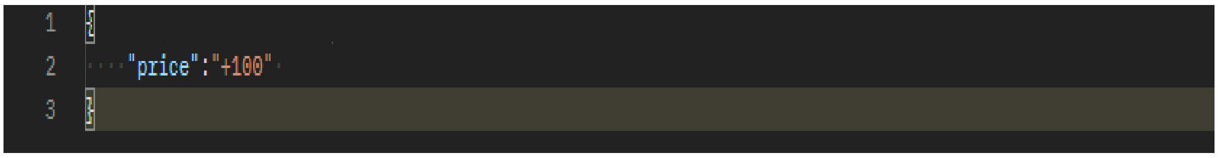
HTTP Method	URLS	Descriptions
GET	localhost:8080/api/v1/product	Returns all the products in the database as a List
PUT	localhost:8080/api/v1/product /{productId}/stocks	Either decreases the stock of the given product by 1 or resupplies the product. JSON body determines which function to call.
PUT	localhost:8080/api/v1/product /{productId}/prices	Either raises or discount the price of the given product. JSON body determines which function to call

The responses to these requests may return an 200(OK) and specifies which operation has occurred, or return a 400(Bad Request) with the reasoning or a 404(Not Found) when the given product id does not exist in the database. The stocks put method expects a JSON body in the form:



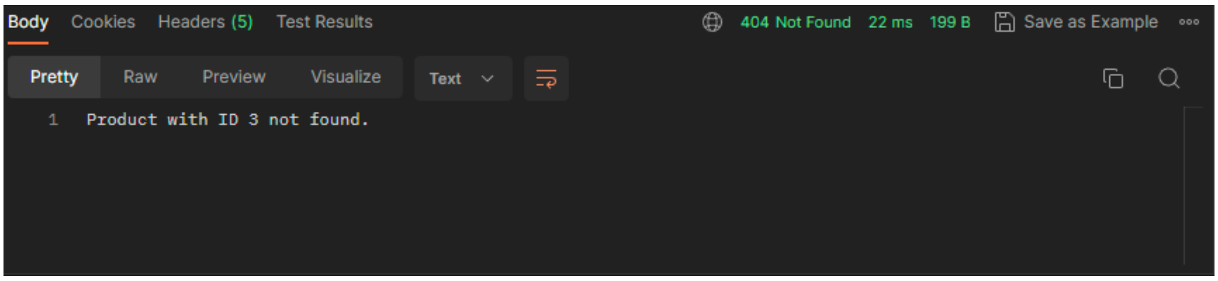
```
{
    "newStockValue":10,
    "transaction":false
}
```

Meanwhile the prices put method expects a JSON body in the form:



```
{
    "price":"+100"
}
```

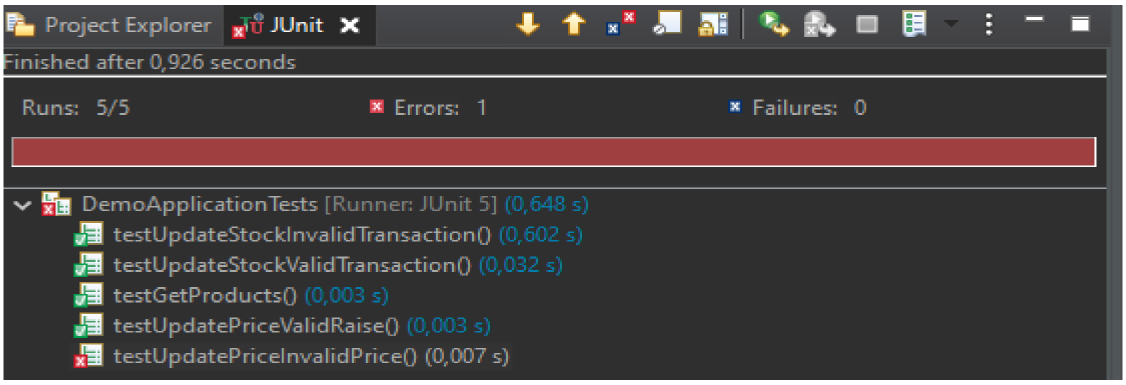
An example error message would be:



```
Product with ID 3 not found.
```

Unit Tests

The server passes 4/5 unit test written for checking stable construction of the server. The results can be seen below:



```
Finished after 0,926 seconds
Runs: 5/5          Errors: 1          Failures: 0
DemoApplicationTests [Runner: JUnit 5] (0,648 s)
    testUpdateStockInvalidTransaction() (0,602 s)
    testUpdateStockValidTransaction() (0,032 s)
    testGetProducts() (0,003 s)
    testUpdatePriceValidRaise() (0,003 s)
    testUpdatePriceInvalidPrice() (0,007 s)
```

Conclusions

By utilizing three different layers which have connections to each other, the server simulates the backend of a vending machine by processing requests and making changes in the database if the request is sent to proper endpoint with proper JSON body.