ZOPSTORE WEB APP BUILT USING THREE LAYERED ARCHITECTURE IN JAVA SPRINGBOOT

Project report submitted in partial fulfilment of the requirement for the degree of Bachelor of Technology

in

Computer Science and Engineering/Information Technology

By

Sushant Rohan

Under the supervision of

Dr. Amit Kumar

to

Department of Computer Science & Engineering and Information Technology

Jaypee University of Information Technology
Waknaghat, Solan-173234, Himachal Pradesh

Certificate

I hereby declare that the work presented in this report entitled **“Zopstore Web App built using three tiered architecture in spring boot”** in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering/Information Technology** submitted in the department of Computer Science & Engineering and Information Technology**,** Jaypee University of Information Technology Waknaghat is an authentic record of my own work carried out over a period from January 2023 to May 2023 under the supervision of **Dr. Amit Kumar** (Assistant Professor (SG), Department of CSE, Jaypee University of Information Technology, Waknaghat).

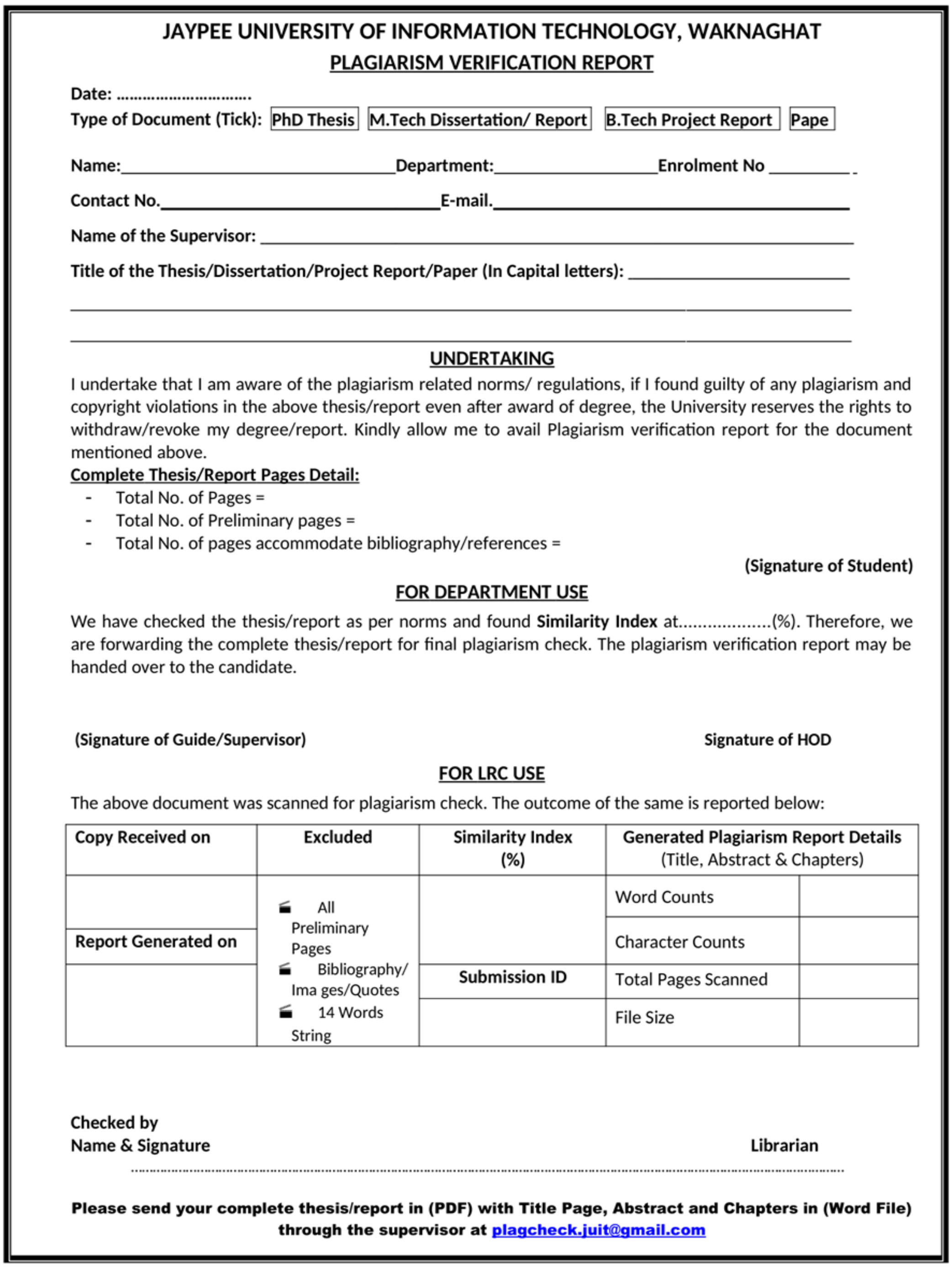
S.Rohan

Sushant Rohan
191267

This is to certify that the above statement made by the candidate is true to the best of my knowledge.

Dr. Amit Kumar
Assistant Professor (SG)
Computer Science & Engineering

Plagiarism Certificate



JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT

PLAGIARISM VERIFICATION REPORT

Date: ……………………….

Type of Document (Tick): PhD Thesis | M.Tech Dissertation/ Report | B.Tech Project Report | Pape

Name:____________________________**Department:**_________________**Enrolment No** _________

Contact No._____________________________**E-mail.**_____________________________________

Name of the Supervisor: ____

Title of the Thesis/Dissertation/Project Report/Paper (In Capital letters): _______________________

____

____

UNDERTAKING

I undertake that I am aware of the plagiarism related norms/ regulations, if I found guilty of any plagiarism and copyright violations in the above thesis/report even after award of degree, the University reserves the rights to withdraw/revoke my degree/report. Kindly allow me to avail Plagiarism verification report for the document mentioned above.

Complete Thesis/Report Pages Detail:

- Total No. of Pages =
- Total No. of Preliminary pages =
- Total No. of pages accommodate bibliography/references =

(Signature of Student)

FOR DEPARTMENT USE

We have checked the thesis/report as per norms and found **Similarity Index** at..................(%). Therefore, we are forwarding the complete thesis/report for final plagiarism check. The plagiarism verification report may be handed over to the candidate.

(Signature of Guide/Supervisor) **Signature of HOD**

FOR LRC USE

The above document was scanned for plagiarism check. The outcome of the same is reported below:

Copy Received on	**Excluded**	**Similarity Index (%)**	**Generated Plagiarism Report Details** (Title, Abstract & Chapters)	
	■ All Preliminary Pages ■ Bibliography/ Ima ges/Quotes ■ 14 Words String		Word Counts	
Report Generated on			Character Counts	
		Submission ID	Total Pages Scanned	
			File Size	

Checked by
Name & Signature **Librarian**

………

Please send your complete thesis/report in (PDF) with Title Page, Abstract and Chapters in (Word File) through the supervisor at plagcheck.juit@gmail.com

Acknowledgement

Firstly, I express my heartiest thanks and gratefulness to almighty God for His divine blessing in making us possible to complete the project work successfully.

I am really grateful and wish my profound indebtedness to Supervisor Dr. Amit Kumar, Assistant Professor (SG), Department of CSE Jaypee University of Information Technology, Wakhnaghat. It is their sincerity that prompted me throughout the project to do hard work using industry-adopted technologies. His endless patience, scholarly guidance, continual encouragement, constant and energetic supervision, constructive criticism, valuable advice, reading many inferior drafts, and correcting them at all stages have made it possible to complete this project.

I would also generously welcome each one of those individuals who have helped me straightforwardly or in a roundabout way in making this project a win. In this unique situation, I might want to thank the various staff individuals, both educating and non-instructing, which have developed their convenient help and facilitated my undertaking.

Finally, I must acknowledge with due respect the constant support and patients of my parents.

Sushant Rohan

191267

Table of Contents

List of Abbreviations

Abbr.	**Full Form**
IoT	Internet of Things
CRUD	Create Read Update Delete
API	Application Programming Interface
JPA	Java Persistence API
POJO	Plain Old Java Class
SQL	Structured Query Language
Java EE	Java Platform, Enterprise Edition
JPQL	Java Persistence Query Language
HQL	Hibernate Query Language

List of Figures

List of Tables

ABSTRACT

Sprint boot is one of the popular Java frameworks to create reliable and scalable web applications.. The three tiered architecture of spring boot consists of a presentation/controller layer, a layer for managing business logic(service layer) and a Data Transport Object(DTO) layer. The job of the presentation layer is to take appropriate response bodies, query parameters and so on from the client(web app, web page etc.) and to send responses accompanied by appropriate headers and status codes. Data layer or DTO or Repository layer's main task is to communicate with the database and perform one of the core functionalities like persisting an entity, reading from the database, updating and deleting entries from the database. Sitting between these two layers is the service layer. Service layer is responsible for handling business logic like validation, preparing responses and so on.

As spring boot comes with a lot of auto-configuration options, the boilerplate code is significantly reduced. This study would address the design, implementation and creation of API exploiting the three tiered architecture provided by spring boot. Three endpoints are available through the API: one for retrieving all categories, one for retrieving all goods, and one for retrieving all items inside a certain category.

One of the modules in Spring is Spring MVC which is used for creating web apps. The controller class annotated by @Controller handles the request and response part and transfers any incoming requests to one of the appropriate service classes handling business logic which would be calling one of the functions provided by Spring Data JPA or Hibernate. Both Spring Data JPA and Hibernate are ORM(Object Relational Mapping) tools and facilitate communication with the database. Hibernate is one of the popular implementation of the specification defined in Java Persistence API(JPA)

In this report, we'll go over the advantages of a three-layer design and how it keeps the code manageable and modular. Also, a quick glance shall be taken over the different modules provided by spring which aids the development speed and adds extra functionality on top of what was possible with spring. We would also be going to the benefits provided by TDD(Test Driven Development). We are using Junit5 and Mockito as our testing libraries.

CHAPTER 1

INTRODUCTION

1.1 Company

ZopSmart is a software solution technology company that provides you with all the tools to build your e-commerce business. ZopSmart has a suite of products that will help you build the perfect business you're aiming to open and run. It has different products such as Smart Store Eazy, Smart Payment Gateway, etc. Zopsmart is building next generation technology for the retail sector and their customers range from a small furniture shop to multinational retail chains and solutions include an e-commerce platform, digital Marketing, m-Commerce, automated logistics systems, management platform, order management platform, and iOT devices. It also provides software solutions to some of the top-most firms and has its own set of frameworks to work on.

Figure 1. Zopsmart Logo

1.2 Introduction

Spring-boot has emerged as one of the major tools to create scalable and reliable web applications especially with the rise of microservices and has been one of the catalysts in the shift from monolithic architecture to microservices that has been popular now-a-days. Based on the Spring framework, spring boot requires little boilerplate code to be written without too much configuration as the idea behind spring boot is “convention over configuration”. Also the compatibility with other spring modules like spring data and spring security makes sure that migrating from spring to spring boot is a seamless process.

The project consists of an ecommerce application with CRUD functionality written in Spring-boot. The project is written adhering to the three layered architecture. Namely the controller layer, the service layer and the Data Access Object(DAO) repository layer or the repository layer. Classes and methods at each layer have their own unit tests. The database of choice is postgreSQL, an open-source relational database. The project has been deployed adhering to the Continuous Integration and Continuous Delivery(CI/CD) principle of devops. This would allow the approved changes to be reflected in the deployed application.

1.3 Objectives

- To create structured, clean code following correct coding standards and design patterns

- To create an application that is structured, well-documented and therefore easy to maintain

- To have the project deployed on cloud and apply CI/CD principles.

1.4 Motivation

To apply industrial best practices and create a fast, scalable and secure web application.

1.5 Libraries/Frameworks Used

a. Java: A high-level, platform independent object oriented programming language. The project uses Java 17 compiler along with gradle as its build tool

b. Spring: One of the more popular open-source frameworks, Spring is used to create Java corporate applications. Modern Java-based enterprise applications may be programmed and configured using its complete programming and configuration paradigm. Web applications, RESTful services, batch processing, messaging systems, and other types of applications can all be built with the help of the capabilities and modules that Spring offers.

c. Spring-boot: A popular sub project of the Spring framework called Spring Boot offers a quick and simple approach to develop Spring-based apps. It offers a pre-configured environment with a variety of features and components in order to streamline the creation and deployment of Spring applications.

d. JPA/Hibernate: One of the Java specifications for converting relational databases into Java objects is JPA. JPA offers a standardised method for leveraging Object-Relational Mapping (ORM) techniques to store Java objects in a relational database. Java classes can be mapped to database tables and vice versa using the interfaces and annotations defined by JPA. Hibernate is a well-known open-source Object-Relational Mapping (ORM) framework for Java. It offers a strong and adaptable technique to map Java objects to relational database tables and the other way around. JPA (Java Persistence API) is the foundation upon which Hibernate is based, and it offers a wealth of features that make it simple to create database-driven applications.

e. Docker: Developers can build, distribute, and operate applications in containers using Docker, a popular open-source framework for containerization. Containers make it simpler to create, test, and deploy programmes in many contexts since they are small, independent, and portable entities that may run on any operating system.

f. postgreSQL: For reliable applications, PostgreSQL is a popular open-source object-relational database management system (ORDBMS). Since its initial 1989 release as an academic project, PostgreSQL has developed into a reliable database platform used by several businesses.

g. git/github: Git is a distributed version control system that enables programmers to communicate with one another and track changes to source code. Linus Torvalds developed Git in 2005 to oversee the creation of the Linux kernel, but it has since grown to become one of the most widely used version control programmes worldwide.

   GIthub, a web-based platform offers hosting for Git-based version control repositories. Developers may monitor changes, store and manage their code, interact with others, and deploy their applications using this tool.

CHAPTER 2

LITERATURE REVIEW

RESTful API development has gained popularity in recent years because it provides a flexible and scalable means of interacting with data and creating applications.. Being a popular choice for creating scalable, high performing and reliable web applications, Spring boot has become the go-to choice for developing RESTful APIs.

One study By K. S. S. Kumar, A. S. Kumar, and K. Raja, "Performance Benchmarking of Spring Boot Microservices". This article uses benchmarking techniques to assess the performance of Spring Boot microservices and offers suggestions for enhancing microservice architecture performance.

Another study by V. Shrivastava, V. Tripathi, and R. Chugh, "A Comparative Study of Microservice Architectures Based on Spring Boot". This study examines various Spring Boot-based microservice designs and assesses the performance, scalability, and maintainability of each.

In a study by B. Benoit and A. Abousselham's "Securing Spring Boot Applications with JSON Web Tokens" discusses best practices and implementation information for using JSON Web Tokens (JWTs) to secure Spring Boot applications.

"Automatic Discovery of Spring Boot Applications in the Cloud" by G. Pierre, R. Zane, and S. Ben Mokhtar. In this study, a system for automatically locating and controlling the configuration and scalability of Spring Boot applications running in cloud environments is proposed.

A. Al-Riyami, M. Al-Zadjali, and K. Al-Nabhani's "Spring Boot with Docker: Containerizing a Microservice-Based Web Application". This study examines the performance and scalability of containerizing a Spring Boot microservices-based web application using Docker.

The official Spring and Spring boot documentations are also available and are quite robust in explaining the features provided to the user.

CHAPTER 3

SYSTEM DESIGN AND DEVELOPMENT

3.1 Overview

The aim of this section is to give a theoretical background and general overview of the architecture of the project. It outlines the work that has been done and also outlines the architecture of the project using pseudocode and visual aid wherever necessary. The chapter would also discuss the architecture of tools and frameworks used in the project to give a more detailed view of the underlying architecture of the project, especially of a RESTful API written using spring boot.

3.2 RESTful API

A set of rules for using in the development of web services are defined by the architectural style known as Representational State Transfer (REST). REST API is a straightforward and flexible method of accessing web services without any processing.

Because REST requires less bandwidth and is more suitable for internet use, it is often preferred to the more reliable Simple Object Access Protocol (SOAP) technology. It is used to obtain or provide data from an online service. Through the REST API, only HTTP requests are used for communication.

The key characteristics of a RESTful API are:

Resources: Anything that can be accessed or changed using a RESTful web service is a resource. URIs (Uniform Resource Identifiers), which are used to uniquely identify resources, are used to identify them.

HTTP methods: To perform actions on resources, RESTful web services use HTTP methods (GET, POST, PUT, DELETE, etc.). The terms "retrieving a resource" (GET), "creating a new resource" (POST), "updating a resource" (PUT), and "deleting a resource" (DELETE) all have distinct meanings. Hitting an appropriate API endpoint would also present itself with status codes to the client. Some of the common status codes are:

> 200 OK: This code indicates that the request has succeeded and the server has returned the requested data
> 201 Created: This code indicates that the server has successfully created a new resource as a result of the request.
> 204 No Content: This code indicates that the server has successfully processed the request, but there is no data to return in the response body.
> 400 Bad Request: This code indicates that the server cannot understand the request due to bad request from the client side
> 401 Unauthorised: This code indicates that the requested resource requires authentication, and the client has not provided valid credentials.
> 403 Forbidden: This code indicates that the client is not authorised to access the requested resource, even with valid credentials.
> 404 Not Found: This code indicates that the server could not find the requested resource.
> 500 Internal Server Error: This code indicates that the server has encountered an unexpected error while processing the request.

Representation: Resources can be represented in a variety of ways, including HTML, XML, and JSON. By specifying the right media type in the request, clients are able to indicate the format they prefer.

Stateless: Because RESTful web services are stateless, each request includes all the data required to fulfil that request. Between requests, the server doesn't keep track of any client state.

Cacheable: Clients or middlemen, including proxy servers, can store responses from RESTful web services in their caches. HTTP cache control headers are used to specify the caching rules.

Layered: RESTful web services can be layered, which allows clients to communicate through middlemen (like proxy servers) rather than the server directly.

3.3 Spring and Spring boot

Spring is an open source framework for building enterprise applications in Java. Combined with spring MVC, spring framework can help in building high-performing, scalable and reliable web applications and microservices. Spring MVC provides a model-view-controller architecture. The three layered architecture provided by Spring MVC makes it possible to create maintainable code with different functionality segmented at each layer.

Aforementioned, Spring follows a layered architecture that separates the concerns of an application into different modules. Here is an overview of the different layers of the Spring architecture:

i. **Controller/Presentation Layer**: The presentation layer is responsible for handling user interface components, such as web pages or user interfaces. Spring provides a web framework called Spring MVC that supports the development of web applications.

ii. **Service Layer**: The business logic is provided by the service layer and service layer sits between the Controller layer and the Data Access layer acting as a mediator between them. Spring provides a service layer framework called Spring Services that helps developers to create business services in a modular and reusable way.

iii. **Data Access Layer**: The data access layer provides access to the database or other data sources. Spring provides a data access layer framework called Spring Data that makes it easy for developers to access data from a variety of data sources, including relational databases, NoSQL databases, and web services.

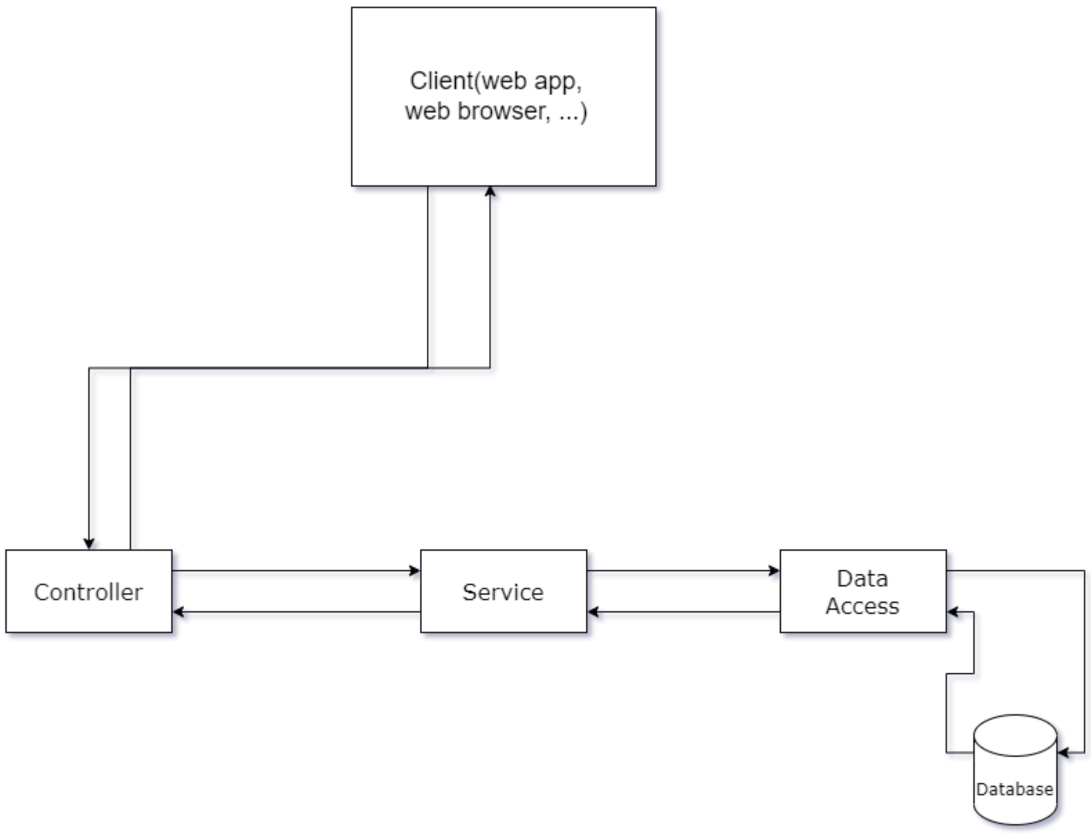


Figure 2. Architecture of Spring three tiered application

Spring framework acts as Inversion of Control(IoC) container for Java. In spring Inversion of Control is achieved by Dependency Injection(DI). Dependency Injection is a design pattern in Object Oriented Programming in which a class, object or a function receives another object or function that it depends on thereby making classes be independent of the creation of the object it depends on.

This means that a class using the services provided by objects of some other classes should not know anything about the creation of those objects and the services provided by those objects other than the services to be used by the class. Therefore, the class would be unaware of the implementation detail of the functionality provided by the service class. Hiding the implementation detail of a service providing class can be done using interfaces.

Making a client class unaware of the creation detail can be achieved using Injectors/Containers. Injectors are special classes whose main purpose is to create objects of the service classes. In spring and spring boot conventions, those objects are called beans. Apart from bean creation of service classes, injectors are responsible for providing the client classes with the beans of service classes that it requires which can be done at constructor level(Constructor Injection) or at the property level(Property Injection). This takes the responsibility of object creation of service classes from client classes to injectors.

3.4 Docker

Docker is an open source framework for creating, managing, running and deploying containers on servers and on cloud platforms. It provides a standardised way to containerize and deploy an application to server or cloud. A container can be defined as a sandbox process on a host machine that has been isolated from all other processes on the host. Docker is used as a tool to create containers running a particular application alongside its various dependencies as defined on Dockerfile. The Dockerfile is then applied to a container which is then run and managed. Both docker and virtual machines(VMs) try to solve the same problem. However, a container differs from VM in the sense that processes running on the container share the kernel of the underlying host machines. However in the case of VMs, multiple OS kernels can run on a host using hypervisor. Running and deploying applications on a container against VMs provides the following benefits:

i . **Resource Utilisation**: Since docker containers use the same underlying kernel as that of host and does not have a different guest OS with hypervisor running on top of the host.

This makes them more efficient in terms of resource utilisation and allows for more efficient use of hardware resources.

ii. **Isolation**: Unlike Virtual Machines which provide full machine level isolation, docker containers provide process level isolation..This means that Docker containers can be more easily scaled and managed, as they do not require as much overhead as VMs. However, for some of the use-cases a VM would be preferred if high level of security and isolation is a must have requirement.

iii. **Portability**: Docker containers are highly portable and can run on any machine that has Docker installed, regardless of the underlying hardware or operating system. VMs, on the other hand, require a hypervisor to be installed on the host machine, which can limit their portability.

iv. **Security**: VMs provide stronger isolation between applications, as they have their own guest operating system and kernel. This makes them a better choice for applications that require high levels of security and isolation.

The difference between a VM and a container can be shown in the given given below:

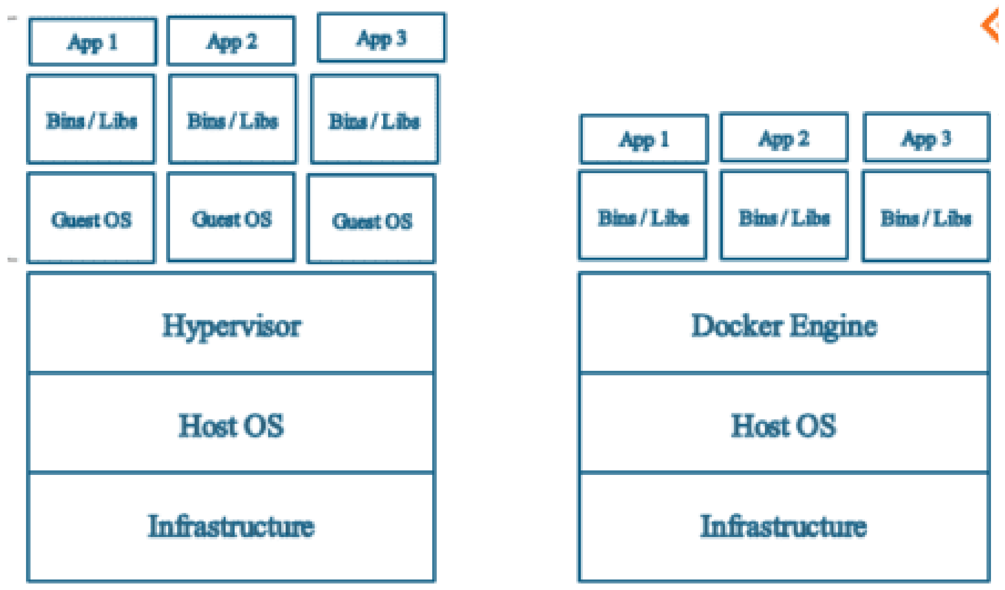


Figure 3. VM(left) vs Docker Container(right)

Docker is deemed lightweight as it only requires a docker engine to manage all docker containers running on the host as opposed to a hypervisor which would be managing all the Virtual Machines with their own OS.

The application along with its dependencies are defined in the Dockerfile. Dockerfile contains a set of instructions that docker uses to build a container. As mentioned, Dockerfile would contain all the dependencies and environments required by an application to run.

A docker image is created using the `docker build` command on a Dockerfile. This would pull all the dependencies from either a central repository or dockerhub and create a docker image which would be used to run on a container using the `docker run` command.

Figure 4. Docker Logo

3.5 git/github

Git is a distributed version control system(VCS) used to manage source code of a project. Git makes it easier to manage projects where there are a large number of contributors. Git tracks changes to a set of files over time and allows multiple developers to work on the same codebase simultaneously. A branch-based model is used by git to allow multiple developers to work on different branches simultaneously. On finishing their work, a developer may push to a central remote repository(like github). A developer may also keep their work in sync with other developers by pulling from the central remote repository.

Due to the distributed nature of the Git version control system, each developer has a complete copy of the repository on their local computer. This lowers the possibility of data loss in the event of a server failure and enables developers to operate offline.

Multiple developers can work together on the same codebase using Git, regardless of where they are in the world. Git additionally enables remote repositories(github being one of them), which can be used to synchronise updates across several devices and places.

Git uses a branching and merging model that enables developers to work independently on different versions of the source before merging their modifications back together. As a result, numerous developers can work together more easily and in parallel. Even in the case of conflicts(i.e merge or rebase conflicts), the ability to resolve those conflicts is very streamlined.

Git enables developers to quickly capture the current state of the files by committing changes to the codebase. Additionally, with the use of `git reset` and `git revert` it is simple for developers to roll back to a previous code version or undo modifications.

There are multiple workflows incorporated in git which has become the standard practice over the years. The project uses github flow which is a lightweight workflow for managing code and collaboration on projects. It is a branch-based workflow that simplifies the process of making changes and merging them into the main codebase. Github flow is based on collaboration and communication which makes it a good approach when multiple developers are working simultaneously. The steps involved in github flow are as follows:

i. **Create a Branch**: Create a new branch from the repository's main branch (often referred to as "master" or "main") when working on a new feature or bug repair. Give the branch a name that accurately describes the modifications being made i.e feature/<feature_name>, bugfix/<bug_name> or hotfix/<fix_name>

ii. **Make Changes**: Make modifications to the codebase on the branch and commit them as necessary. Pushing updates to the branch on GitHub lets others examine and collaborate on the changes as you make them.

iii. **Open a Pull Request**: Open a pull request on GitHub to merge the branch's changes with the main branch once they are finished. Assign reviewers to examine the changes after describing them in detail and any relevant background information.

iv. **Review and Discuss**: Once the modifications have been reviewed, reviewers can offer input and discuss any problems or ideas directly on the pull request. The assigner can then make changes as per the PR review and request for a review again.

v. **Merge and Deploy**: Once the changes are approved, merge the changes into the main branch, and deploy the changes to the appropriate environment (such as a staging or production environment).

Figure 5. Git logo

Figure 6. Github logo

3.6 github Actions and CI/CD Principles

Software development (Dev) and IT operations (Ops) teams are brought together by the practices and ideas of DevOps to work more closely and cooperatively throughout the software development lifecycle. Shortening the software development lifecycle and boosting delivery speed and efficiency are the main objectives of DevOps.

The goal of DevOps is to foster a culture of shared accountability for the creation, deployment, and maintenance of software through enhancing communication and collaboration between the development and operations teams. This is accomplished through streamlining the software development, testing, and deployment processes through the use of automation and tooling.

Common DevOps practices include infrastructure as code, automated testing, monitoring, and feedback loops. Continuous Integration/Continuous Deployment (CI/CD) is another common practice. DevOps enables teams to release new features and updates more frequently and with better confidence while lowering the risk of introducing bugs or resulting in downtime by automating the process of building, testing, and releasing code.

DevOps, as a whole, is a culture shift that places an emphasis on teamwork and communication as well as the use of automation and tooling to increase the efficacy and efficiency of software development and deployment. In today's fast-paced software development environment, where agility and speed are essential to be competitive, it has grown in significance.

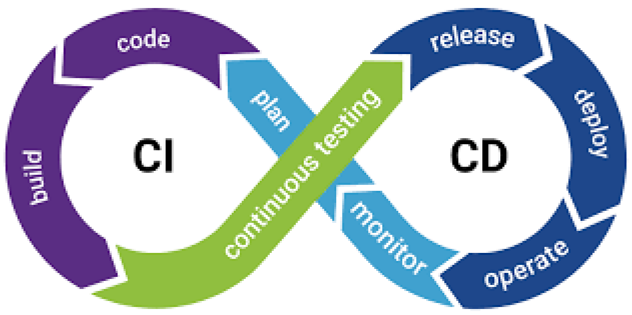


Figure 7. CI/CD

Developers can automate a variety of processes and workflows across their software development lifecycle with GitHub Actions, a robust and adaptable automation tool offered by GitHub. Without relying on other tools or services, it enables developers to automate the process of creating, testing, and deploying code from their GitHub repository.

A large selection of pre-built actions, or reusable chunks of code, are available through GitHub Actions. These tasks include developing and testing code, deploying apps, and sending notifications. These premade actions can be mixed as per user requirements to create unique actions which better match the requirements of a given project.

Additionally, developers can use JavaScript, shell scripts, or Docker containers to construct their own unique actions in GitHub Actions or even run shell scripts. Developers can now design unique workflows that connect with their current toolchains and procedures thanks to this.

A strong and adaptable infrastructure for automating software development workflows is offered by GitHub Actions. It integrates smoothly with GitHub repositories and supports a wide variety of programming languages, tools, and services. With GitHub Actions, developers can optimise their software development lifecycle, boost productivity, foster better teamwork, and minimise errors and downtime.

GitHub Actions

Figure 8. Github actions

3.7 Hardware and Software requirements

3.7.1 Hardware Requirements

- Processor: i3 or higher processor
- RAM: 2 GB or higher
- Hard Disk: 10 GB or higher

3.7.2 Software

- OS: Ubuntu 20.04 or higher

- Language: Java 17 with the following dependencies:
    - postgreSQL JDBC driver
    - Log4J logging library

- Build tool: gradle

- Framework: Spring boot(version 3.0.4) with the following modules:
    - Spring data JPA for persisting entity to the postreSQL database
    - Spring boot starter web
    - Spring boot webMVC
    - Spring boot validation
    - Hibernate validator

- Testing library: junit(version 5.8.1) and mockito for mocking dependencies

3.8 Database design

The choice of database was postgreSQL, a popular open-source relational database. PostgreSQL provides a wide range of functionality for data management and analysis. Known for its dependability and stability, PostgreSQL.

With capabilities like transaction management, crash recovery, and data integrity checks, it has a demonstrated history of being a very dependable database management system. PostgreSQL is a fantastic option for scalable applications since it can manage massive volumes of users and data.

High levels of flexibility are offered by PostgreSQL in terms of data formats, indexing choices, and query optimisation. Because of this, it is simple to use with a variety of applications and data formats. Strong security features in PostgreSQL include audits, data encryption, and user authentication and authorisation. Because PostgreSQL is open-source software, it is available for free and has a strong developer community that contributes to its growth and support.

PostgreSQL

Figure 9. PostgreSQL logo

The mapping of a Java entity to a postgreSQL table is handled by hibernate. In Hibernate, entity classes are mapped to database tables using XML mapping files(as in Spring framework) or annotations(as in Spring boot). The mapping defines the relationship between the properties of the entity class and the columns of the database table.
And example of entity class is given below:

```
@Entity
@Table(name = "users")
public class User {

    @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "username")
    private String username;

    @Column(name = "password")
    private String password;

    // getters and setters
}
```

In this case, the @Entity annotation is used to define that the following class is an Entity class which means that this class would be mapped to a table of the same name in the schema. The User class is mapped to the users table as specified in the @Table annotation.

Hibernate supports many-to-one, one-to-many, many-to-many, and other types of relationships between entities, which can also be mapped using annotations or XML configuration files.

Hibernate can also handle mapping inherited classes to database tables. Hibernate uses different strategies to map hierarchical classes to database tables.

One must take into account elements like performance, scalability, complexity, and the type of data being modelled in order to select the best inheritance approach for an entity hierarchy. It's crucial to make a sensible choice when selecting a strategy because it will also influence how the database structure is designed. In general, basic hierarchies with few subtypes are a suitable fit for the Single Table method, but complicated hierarchies with several subtypes are better suited for the Joined strategy.The four strategy provided by JPA are:

1. **MappedSuperclass:** In this strategy, one can define common properties and mappings that can be used by multiple entity classes, reducing duplication and improving maintainability. This can be applied at the class level by using @MappedSuperclass annotation.

2. **Single Table**: In this approach, each class hierarchy property is mapped to a separate database table. The discriminator column is used to distinguish between various entity kinds. Although it can result in a huge, sparse table with plenty of nullable columns, this approach is the most effective in terms of query performance. This can be applied by adding `@Inheritance(strategy = InheritanceType.SINGLE_TABLE)` annotation to the entity class level.

3. **Joined Table:** This approach maps each class in the hierarchy to a separate database table, and then uses a join operation to obtain the entire object. Due to the join operation, this technique may be less effective than the Single Table strategy but still produces a normalised database schema. This can be applied by

adding `@Inheritance(strategy = InheritanceType.JOINED)` annotation to the entity class level.

4. **Table per Class**: This method maps all of the class hierarchy's properties to a single database table without the need of any discriminator columns. Due to its avoidance of the join operation, this technique may provide a less normalised database schema than the Joined strategy, but it may also be more effective. This can be applied to an entity class by adding `@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)` annotation to the entity class.

For this project, the MappedSuperclass strategy is used owing to its efficient performance in joins and non-redundant nature.

The ER-diagram of the database schema used in the project is given below:

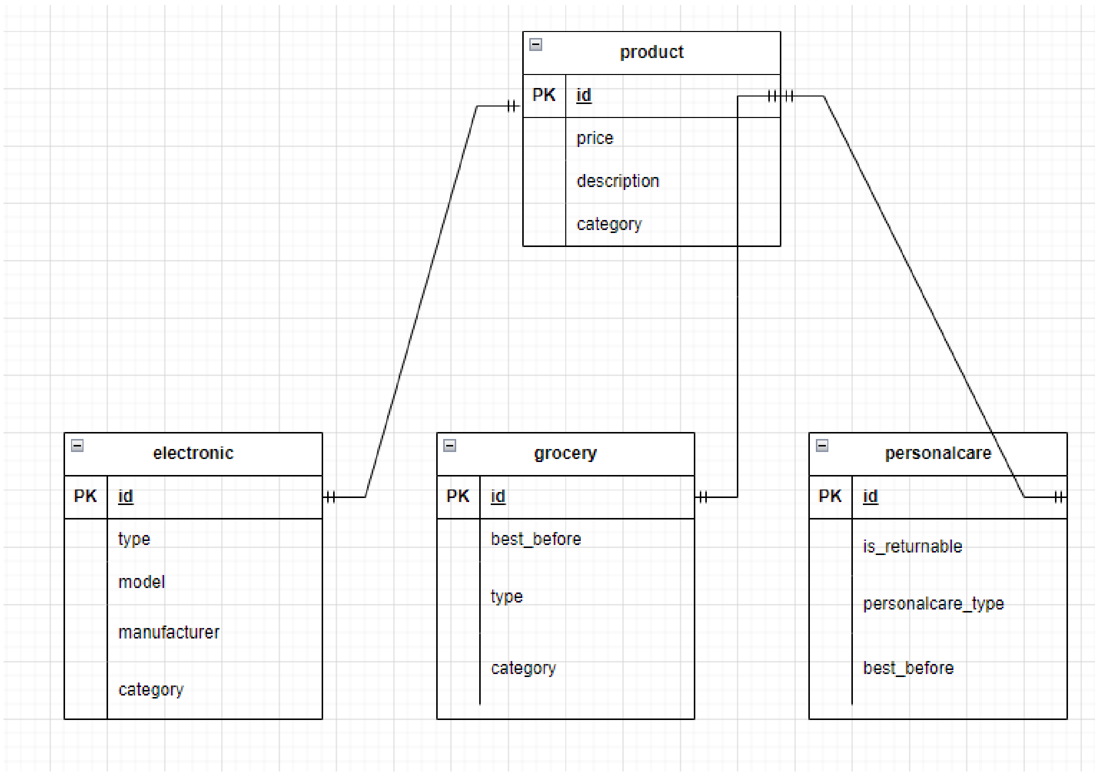


Figure 10. ER diagram for Zopstore

As seen from the ER diagram, using MappedSuperclass strategy gives us the ability to have one table per class with relation defined between them. The relationship is established as a foreign key between them. The foreign key constraint is defined in the child table referencing the id attribute in the parent product table.

The relation in this case is of type one-to-one and ON DELETE CASCADE was used. This means that if one were to delete a product with a specific id from the product table, the corresponding entry in the child tables(electronic, grocery and personalcare would also be deleted).

The following SQL DDL queries were run by hibernate as entities were mapped to postgreSQL schema tables:

```
CREATE TABLE elctronic (
    electronic_type smallint NOT NULL,
    manufacturer varchar(50),
    model varchar(50) NOT NULL,
    id bigint NOTNULL,
    primary key(id)
)

CREATE TABLE grocery (
    grocery_type smallint NOT NULL,
    best_before_date timestamp(6),
    id bigint NOTNULL,
    primary key(id)
)

CREATE TABLE personal_care (
    personalcare_type smallint NOT NULL,
    is_returnable boolean NOT NULL,
    id bigint NOTNULL,
    primary key(id)
)

CREATE TABLE product (
    id bigint NOT NULL,
    description text NOT NULL,
    price float(53) NOT NULL,
    product_type smallint NOT NULL,
    primary key(id)
)

ALTER TABLE IF EXISTS ELECTRONIC
ADD CONSTRAINT FOREIGN KEY(id) REFERENCES PRODUCT

ALTER TABLE IF EXISTS GROCERY
ADD CONSTRAINT FOREIGN KEY(id) REFERENCES PRODUCT

ALTER TABLE IF EXISTS PERSONALCARE
ADD CONSTRAINT FOREIGN KEY(id) REFERENCES PRODUCT
```

Aforementioned, the tables created would follow the entity class. The UML diagram of entity classes is given below:

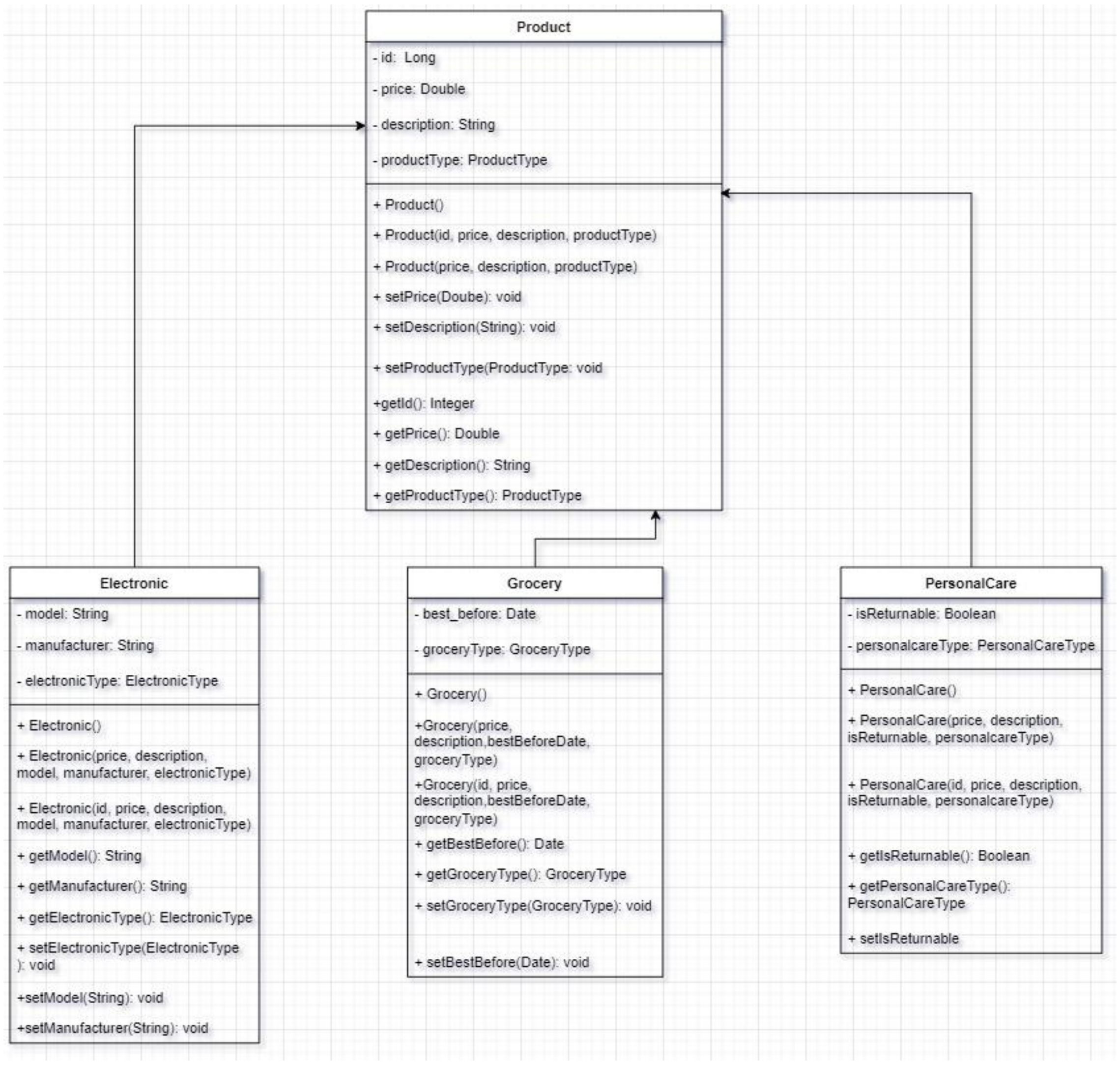

Product
- id: Long
- price: Double
- description: String
- productType: ProductType
+ Product()
+ Product(id, price, description, productType)
+ Product(price, description, productType)
+ setPrice(Doube): void
+ setDescription(String): void
+ setProductType(ProductType: void
+getId(): Integer
+ getPrice(): Double
+ getDescription(): String
+ getProductType(): ProductType
Electronic
- model: String
- manufacturer: String
- electronicType: ElectronicType
+ Electronic()
+ Electronic(price, description, model, manufacturer, electronicType)
+ Electronic(id, price, description, model, manufacturer, electronicType)
+ getModel(): String
+ getManufacturer(): String
+ getElectronicType(): ElectronicType
+ setElectronicType(ElectronicType): void
+setModel(String): void
+setManufacturer(String): void
Grocery
- best_before: Date
- groceryType: GroceryType
+ Grocery()
+Grocery(price, description,bestBeforeDate, groceryType)
+Grocery(id, price, description,bestBeforeDate, groceryType)
+ getBestBefore(): Date
+ getGroceryType(): GroceryType
+ setGroceryType(GroceryType): void
+ setBestBefore(Date): void
PersonalCare
- isReturnable: Boolean
- personalcareType: PersonalCareType
+ PersonalCare()
+ PersonalCare(price, description, isReturnable, personalcareType)
+ PersonalCare(id, price, description, isReturnable, personalcareType)
+ getIsReturnable(): Boolean
+ getPersonalCareType(): PersonalCareType
+ setIsReturnable

Figure 11. UML Class diagram for Zopstore

A standard specification for Java ORM (Object-Relational Mapping) technologies, JPA stands for Java Persistence API. Developers can use an ORM tool like Hibernate or EclipseLink to connect with a relational database utilising a set of APIs provided by JPA.

Without needing to write SQL queries, developers can use Java code to create, read, update, and delete objects by mapping Java classes to database tables using the annotations and interfaces defined by JPA.

The working of JPA can be explained using the following modules:

i. Entity: Entity classes are created by developers in JPA to represent objects that must be stored in the database. To indicate the mapping between the Java objects and database tables, these entity classes have the @Entity annotation and @Column or @JoinColumn annotations on their properties.

ii. Persistence Context: The persistence context is a collection of managed entity instances that reflects a specific database state. To establish and manage the persistence context, JPA offers the EntityManager interface. EntityManager is a tool that developers can use to conduct CRUD operations on entity objects.

iii. Mapping: JPA uses mapping annotations on the entity class properties to map entity classes to database tables. Developers can express relationships between entities by using the annotations @OneToOne, @OneToMany, @ManyToOne, and @ManyToMany.

iv. Transitions: Transaction management is provided by JPA to guarantee that database updates are atomic and consistent. The EntityManager interface allows developers to start, commit, and roll back transactions in case of failures.

v. Query: Querying: JPA offers a Query interface for running database queries. JPQL (Java Persistence Query Language) allows programmers to write queries in a high-level language that is comparable to SQL. After that, JPA converts JPQL queries into SQL ones that may be used to access the database.

An overview of the JPA structure is given below:

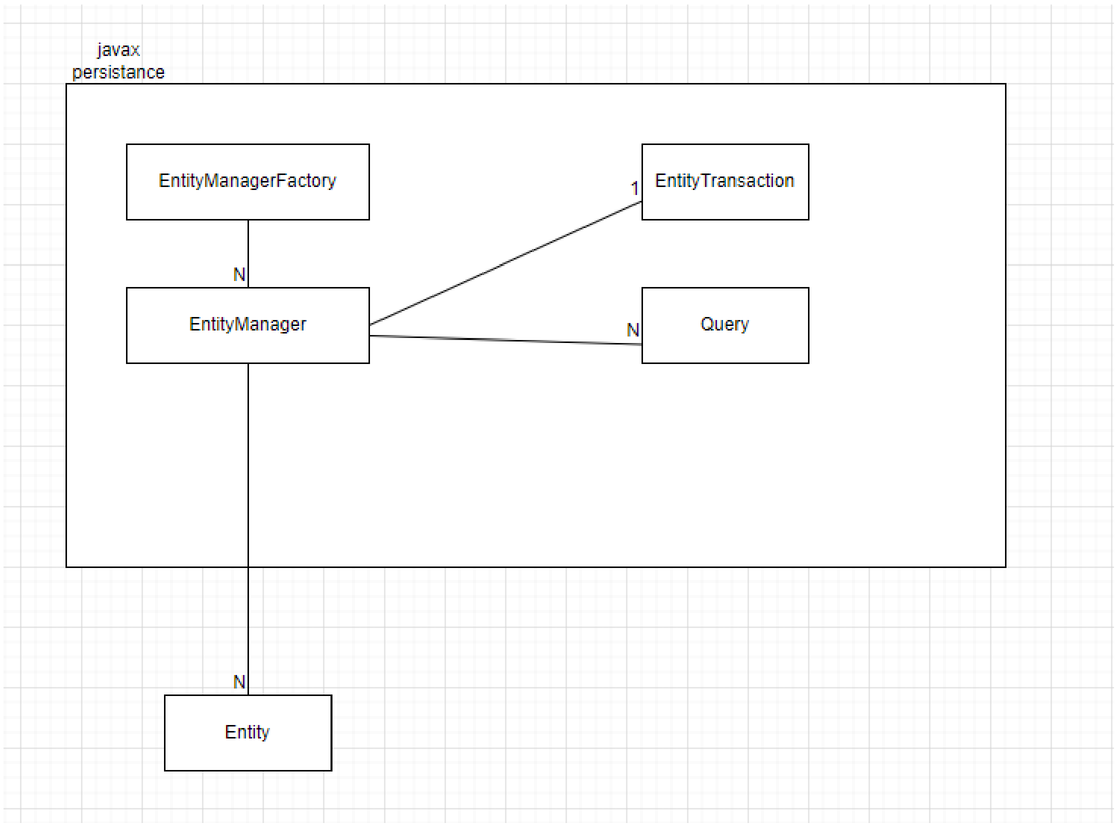


Figure 12. Architecture of javax persistence library

Hibernate is one of the popular implementations of the specifications provided by JPA. Hibernate is an Object-Relational Mapping (ORM) solution, bridging the gap between relational databases and object-oriented programming. Data is stored in tables in typical database applications, and SQL is used to access, modify, and delete data from these tables. In contrast, data is represented as objects in object-oriented programming, and these objects have qualities and behaviours.

Hibernate offers a mechanism to persist these objects to a relational database by mapping Java classes to database tables. It accomplishes this by defining the mapping between Java classes and database tables in metadata and configuration files. The mapping configuration details which columns in a database table correlate to which Java class properties.

An application uses Hibernate's session object to build a query when it wants to obtain data from the database. The query is constructed in Hibernate Query Language (HQL), a higher-level language than SQL and the query language used by Hibernate. Hibernate converts HQL requests into SQL requests that are then used to access the database.

Hibernate uses its mapping settings to generate Java objects that represent the data once it has been retrieved from the database. The application is then given the Java objects back. Using the session object, the application can then modify these objects and save them back to the database.

For database operations, the session object offers a transactional boundary. Hibernate establishes a database connection and initiates a transaction when a session is formed. This transaction contains the execution of all database actions carried out throughout the session. Hibernate commits the transaction or rolls it back if an error occurs when the session is closed.

Data caching is also supported by Hibernate, which can greatly enhance performance. First-level cache and second-level cache are two forms of caching offered by Hibernate. The second-level cache is shared by all of the application's sessions, while the first-level cache is connected to a single session object. In order to reuse the results of frequently conducted queries, Hibernate also provides query caching.

The architecture of Hibernate is given below:

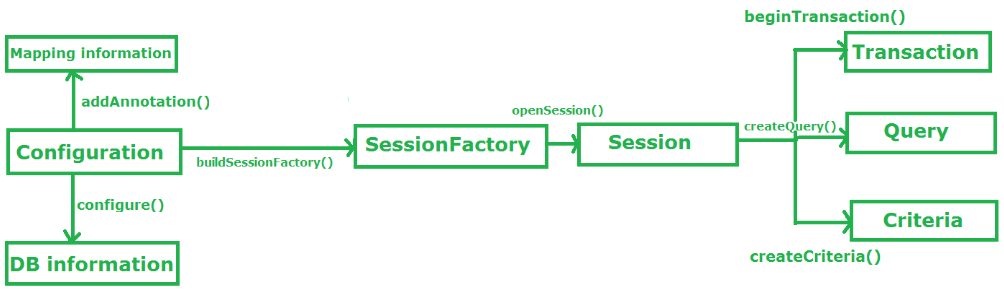


Figure 13. Architecture of Hibernate

To make a DAO(Data Access Object) or Repository interface, it should be annotated by @Repository annotation. To encapsulate an application's persistence layer, a design pattern is frequently employed called a repository class. It offers a central location where users can view data and carry out CRUD (Create, Read, Update, Delete) activities on database entities. A repository class serves the primary function of separating an application's business logic from the underlying data access layer, resulting in more modular, testable, and maintainable code.

In Spring Boot, the repository pattern is commonly used with the Spring Data JPA module to provide a simple and effective way to perform database operations. Spring Data JPA provides a powerful set of abstractions over the JPA (Java Persistence API) and Hibernate technologies, making it easy to interact with the database without having to write boilerplate code. The repository interface must implements on of the following JPA interfaces:

1. CRUD Repository
2. PagingAndSortingRepository
3. JPARepository

The class UML diagram of repository interfaces given by JPA is given below:

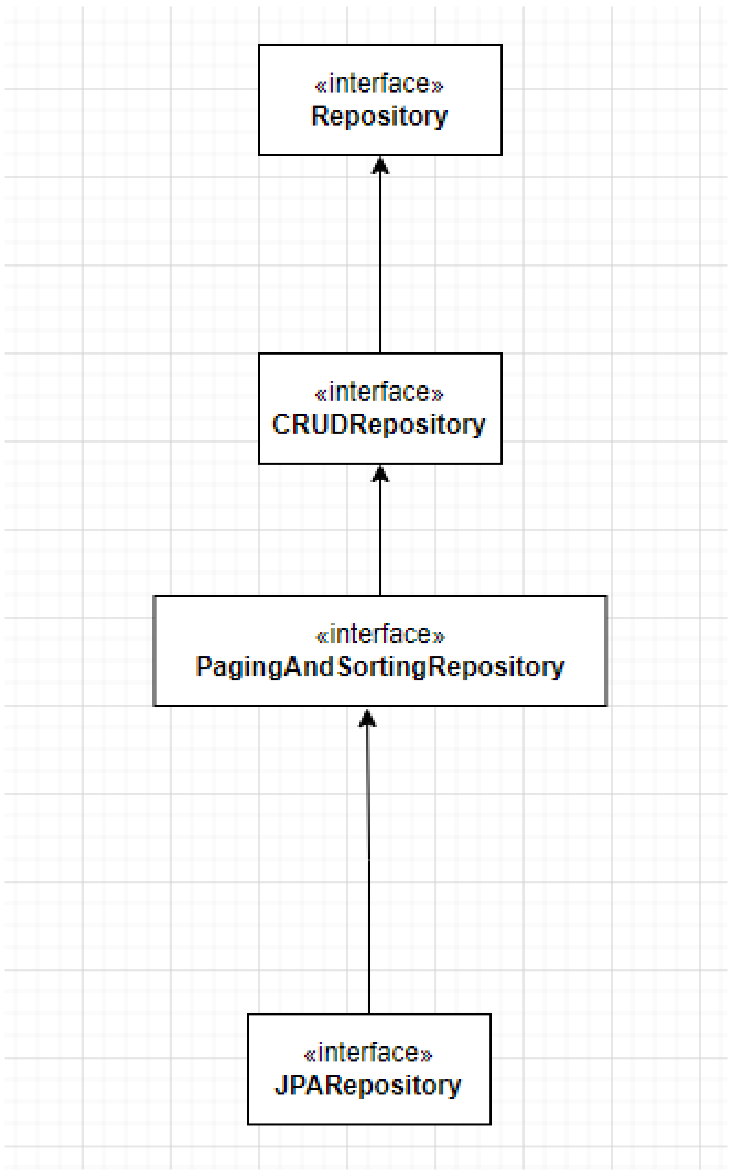


Figure 14. Class diagram of Repository interfaces

3.9 Implementation Detail

The project would support CRUD(Create, Read, Update, Delete) operations.

3.9.1 Create a Product

To create a product listing on the website, one would need to create a POST request to the /ecommerce/v1/product endpoint with the following JSON Request body

For creating electronic product,

```
{
    "_type": "ELECTRONIC",
    "price": number,
    "description": string,
    "model": string,
    "manufacturer": string,
    "electronicType": number,
    "productType": 0
}
```

For creating grocery product,

```
{
    "_type": "GROCERY",
    "price": number,
    "description": string,
    "groceryType": number,
    "bestBeforeDate": YYYY-mm-DD
    "productType": 1
}
```

For creating personalcare product,

```
{
    "_type": "PERSONALCARE",
    "price": number,
    "description": string,
    "personalcareType": number,
    "bestBeforeDate": YYYY-mm-DD
    "productType": 2
}
```

The endpoint will send back the following response code:

1. 201: On successfully creating a product
2. 400: Bad Request(Missing attribute in the JSON request body or wrong data)
3. 500: Internal Server error

The response body of this endpoint will be almost the same as that of request with _type attribute removed and unique id attribute added.

The response body would be in JSON format. For example, for an electronic product the response would be like this and all other entities would follow the same by adding and removing attributes from it.

```
{
    "Id": number
    "price": number,
    "description": string,
    "model": string,
    "manufacturer": string,
    "electronicType": number,
    "productType": 0
}
```

3.9.2 Read

For reading a product, a GET request needs to be made to the /ecommerce/v1/product endpoint. This would run the findById() function as defined in the JPA specification.

The response would be in JSON format with details of the product type. A sample JSON response for product of type electronic is given below:

```
{
    "Id": 1
    "price": "12999.90",
    "description":"Phone",
    "model": "iPhone",
    "manufacturer": "Apple",
    "electronicType": "MOBILE",
    "productType": 0
}
```

The response code for this case is 200. If a product for a given id does not exists, the following message is sent to the client:

```
{
    "message": "No product exists for given id"
}
```

In case invalid ID(negative or zero) is entered, the following message is sent as JSON response to the client with response code 400(Bad Request):

```
{
    "message":     "Invalid     input     parameters:
getProductById.id: ID cannot be zero or negative",
    "Method": "GET",
    "Uri": /ecommerce/v1/product/-1,
    "Timestamp": "2023-05-09T00:29:20.440663778"
}
```

In case invalid datatype is entered for example, string is entered instead of long/numeric in id field, the following error message encapsulated as JSON response is sent to the client with response code 400(Bad Request).

```
{
    "message": "Invalid input for id. Required type:
Long",
    "Method": "GET",
    "Uri": /ecommerce/v1/product/-1,
    "Timestamp": "2023-05-09T00:29:20.440663778"
}
```

3.9.3 Update

Adhering to RESTful convention, one might use PUT or PATCH methods to update an entity in the database. Both PUT and PATCH being methods to update, they have some differences between them.

The server's full resource is updated using the PUT method. With the PUT method, the server replaces the current resource with the new representation that the client sent. The client sends the whole representation of the resource. The server creates a new resource if one doesn't already exist. Because PUT requests are idempotent, several requests that are exactly the same will have the same result as a single request.

A portion of the resource on the server can be updated using the PATCH method. When a resource is updated using the PATCH technique, the server only replaces the fields that were present in the client's partial representation of the resource. Multiple identical PATCH requests may have different results since they are not idempotent.

Generally speaking, PUT should be used to update the entire resource, and PATCH should be used to change just a portion of the resource. The unique use case and the API design eventually determine whether to use PUT or PATCH.

The update operation is performed by using the PUT method on the endpoint /ecommerce/v1/product. The JSON object made during the request should be in the following format:

```
{
     "Id": number
    "_type": "ELECTRONIC",
    "price": number,
    "description": string,
    "model": string,
    "manufacturer": string,
    "electronicType": number,
    "productType": 0
}
```

The above JSON request body is for electronic products. For products of different types, the appropriate JSON request can be made by removing the attributes specific to electronic products and adding those attributes specific to those product in the JSON request object.

On successful completion of the update process, the entity is updated and persisted into the database. The updated values are returned to the client with response code 200. The format of the response body is given below:

```
{
    "Id": number
    "price": number,
    "description": string,
    "model": string,
    "manufacturer": string,
    "electronicType": number,
    "productType": 0
}
```

In case a user tries to update a product such that no product with given id exists, the following message is sent to the client:

```
{
     "message": "Update failed. No product exist for
given id"
}
```

In case a user tries to update a product with invalid data i.e. Zero or negative price, empty description, invalid date format and so on, the following JSON response is sent to the client side with status code 400(Bad Request).

```
{
    "message": "Description cannot be blank",
    "Method": "GET",
    "Uri": /ecommerce/v1/product/-1,
    "Timestamp": "2023-05-09T00:29:20.440663778"
}
```

3.9.4 Delete

One can delete a product from the database by hitting the API using DELETE method on endpoint /ecommerce/v1/product.

One can delete a product using its id. Id is sent as a path param in the API request.

On successfully deleting a product, the API would send an empty response with status code 204(No content).

If no product exists with the given id, the following message will be returned to the client

```
{
    "message": "Cannot delete. No product exist with
given id"
}
```

3.9.5 Validation

To verify the values of the fields in an entity class in JPA, utilise validation annotations from the javax.validation.constraints package. Constrictions like field length, necessary fields, range validation, and regular expression pattern validation can all be enforced with these annotations.

Use the @Valid annotation on the entity class or on the parameter of a controller function that accepts the entity as the request body to enable validation in JPA. When you add the @Valid annotation to an entity or method parameter, the validation framework will automatically validate the entity's fields before committing them to the database.

To add validation constraints to the fields of an entity, you can use annotations such as:

`@NotNull`: Specifies that the field must not be null.
`@Size`: Specifies the minimum and maximum size of a string or collection.
@Min and @Max: Specifies the minimum and maximum value of a numeric field.

Hibernate validator is one of the alternatives of using JPA validator.
A system called Hibernate Validation offers a selection of annotations for verifying the accuracy of objects and their fields. It is an independent framework for validating data that may be used with or without Hibernate ORM.

Based on the Java Bean Validation API (JSR 380), the Hibernate Validation framework offers extra limitations that can be incorporated into your validation rules. Hibernate Validation offers a number of frequently used constraints, including the following:

@NotNull: Specifies that the field must not be null.
@Size: Specifies the minimum and maximum size of a string or collection.
@Min and @Max: Specifies the minimum and maximum value of a numeric field.
@Email: Specifies that the field must be a valid email address.
@Past:Specifies a date that is in the past
@Future: Specifies a date.that is in the future.
@Pattern: Specifies a regular expression pattern that the field must match.

To validate an object using Hibernate Validation, one can use the Validator interface provided by Hibernate Validation

Based on the same Java Bean Validation API (JSR 380), JPA Validator and Hibernate Validator offer comparable capability for validating objects and their fields. There are some distinctions between the two, though.

The validation toolkit offered by the Java Persistence API (JPA) is called JPA Validator. It can be used to validate entities prior to their persistence in the database and is coupled with JPA. JPA Validator offers a small number of validation constraints, such as @IdClass and @Version, that are unique to JPA entities. Additionally, it facilitates the validation of entity-to-entity relationship mappings.

On the other hand, Hibernate Validator is a standalone framework for validating data that may be used with or without Hibernate ORM. It offers a wider range of validation restrictions, such as @Email and @Pattern, that are not unique to JPA entities. Additionally, group validation and specific validation constraints are supported by Hibernate Validator.

The validators applied to different entities are given below:

For Product entity:

id	@Positive
description	@NotNull, @NotEmpty, @NotBlank
price	@NotNull, @DecimalMin(value = “0”)
productType	@NotNull

Table 1. Constraint for Product entity

For Electronic entity,

model	@NotNull, @NotEmpty, @NotBlank
description	@NotNull, @NotEmpty, @NotBlank
electronicType	@NotNull

Table 2. Constraints for electronic entity

For Grocery entity,

bestBeforeDate	@Future, @NotNull
groceryType	@NotNull

Table 3. Constraints for grocery entity

For Personalcare entity,

isReturnable	@NotNull
personalcareType	@NotNull

Table 4. Constraints for personalcare entity

In conclusion, Hibernate Validator is a more versatile validation framework that can be used in a variety of applications, including those that do not utilise JPA, whereas JPA Validator is a more specialised validation framework that is optimised for usage with JPA entities. Your particular wants and requirements will determine which option you should select.

3.9.6 Global Exception Handling

Using the @ControllerAdvice annotation, one may build global exception handling in Spring Boot. A specialisation of the @Component annotation called @ControllerAdvice enables you to write exception-handling methods that can be used by several controllers at once.

The steps to implement global exception handling in spring boot are:

- Create an exception handler class and annotate it with @ControllerAdvice.

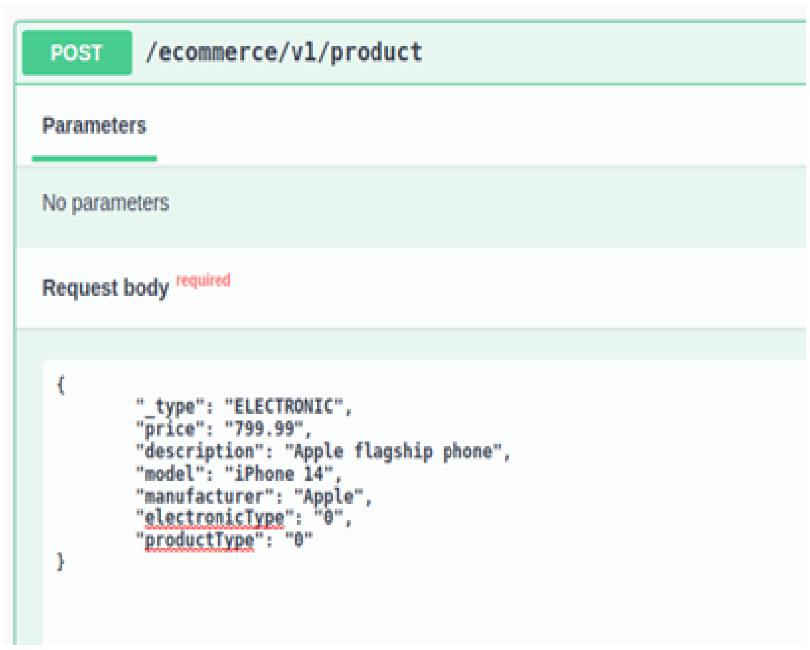
- Create methods in the class that handle exceptions and annotate them with @ExceptionHandler. These methods ought to be able to handle the particular exception type that is passed to them.

- Create a unique error response with information about the exception, such as the error message and status code, using the ResponseEntity class.
- One can also specify methods to deal with particular kinds of exceptions or choose a default exception handler to deal with any unhandled exceptions, if needed.

CHAPTER 4

EXPERIMENTS AND RESULT ANALYSIS

4.1 **Results and Demonstration**



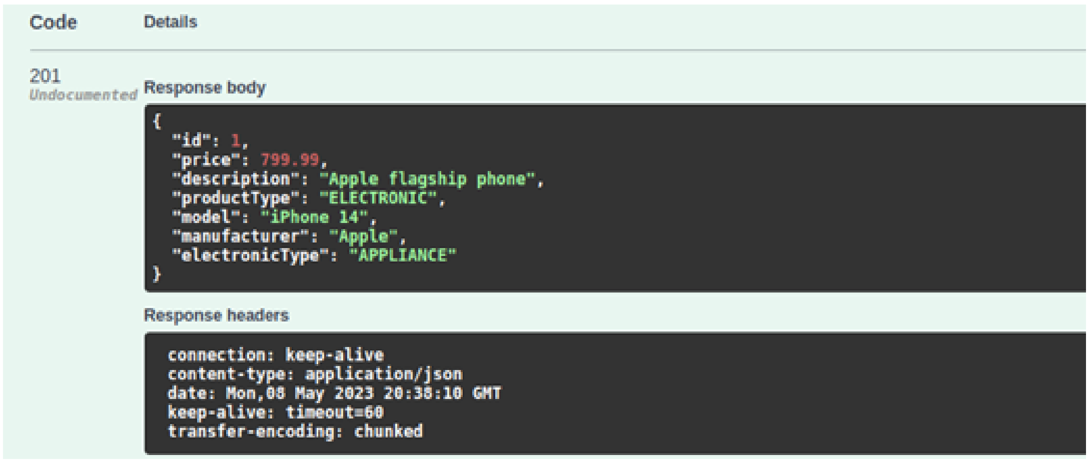
/ecommerce/v1/product

No parameters

```
{
        "_type": "ELECTRONIC",
        "price": "799.99",
        "description": "Apple flagship phone",
        "model": "iPhone 14",
        "manufacturer": "Apple",
        "electronicType": "0",
        "productType": "0"
}
```

Figure 15. Creating an Electronic product

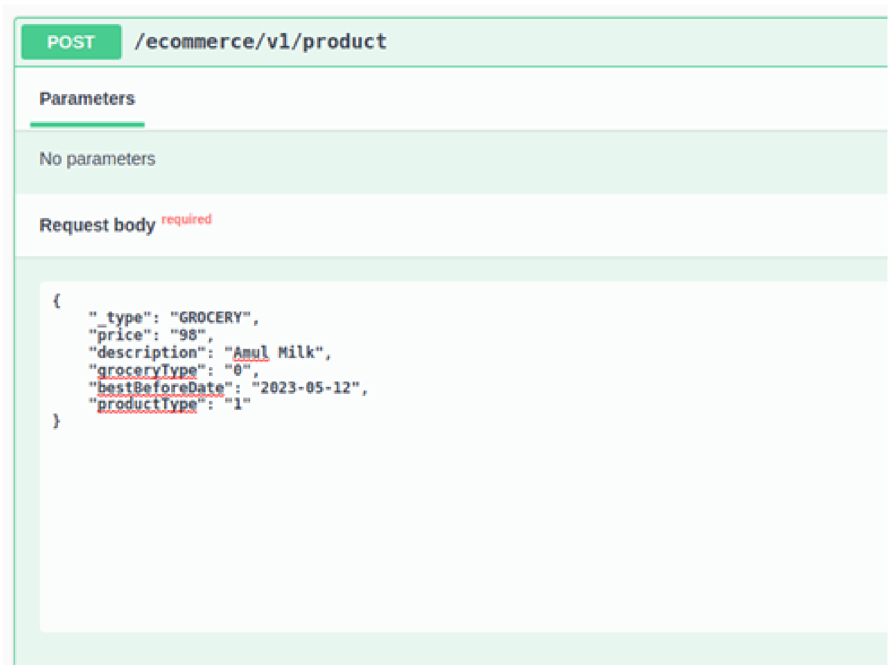
As evident in Fig. 15, a POST method is used to create a new entry into the database.
In this example, an entity of type electronic is persisted into the database. ElectronicType attribute is a Java enum mapped to ordinal values(0 for Appliance, 1 for Laptop and 2 for Mobile).
This POST method sends back the newly created entry with id added to the response with status code 201.



```
{
  "id": 1,
  "price": 799.99,
  "description": "Apple flagship phone",
  "productType": "ELECTRONIC",
  "model": "iPhone 14",
  "manufacturer": "Apple",
  "electronicType": "APPLIANCE"
}
```

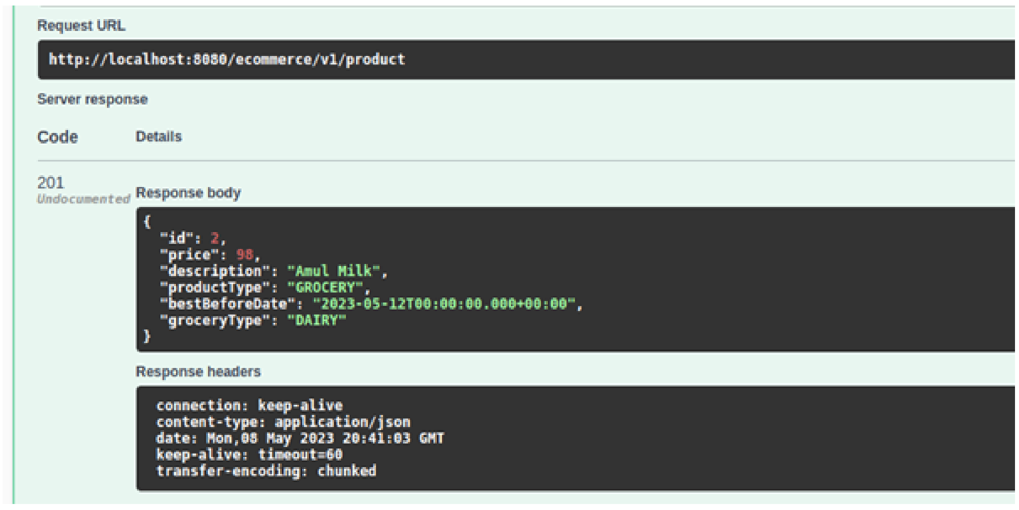
```
connection: keep-alive
content-type: application/json
date: Mon,08 May 2023 20:38:10 GMT
keep-alive: timeout=60
transfer-encoding: chunked
```

Figure 16. Response when successfully creating an electronic product entry



```
{
    "_type": "GROCERY",
    "price": "98",
    "description": "Amul Milk",
    "groceryType": "0",
    "bestBeforeDate": "2023-05-12",
    "productType": "1"
}
```

Figure 17. Creating a Grocery product entry


Request URL

```
http://localhost:8080/ecommerce/v1/product
```

Server response

Code Details

201 *Undocumented* Response body

```
{
  "id": 2,
  "price": 98,
  "description": "Amul Milk",
  "productType": "GROCERY",
  "bestBeforeDate": "2023-05-12T00:00:00.000+00:00",
  "groceryType": "DAIRY"
}
```

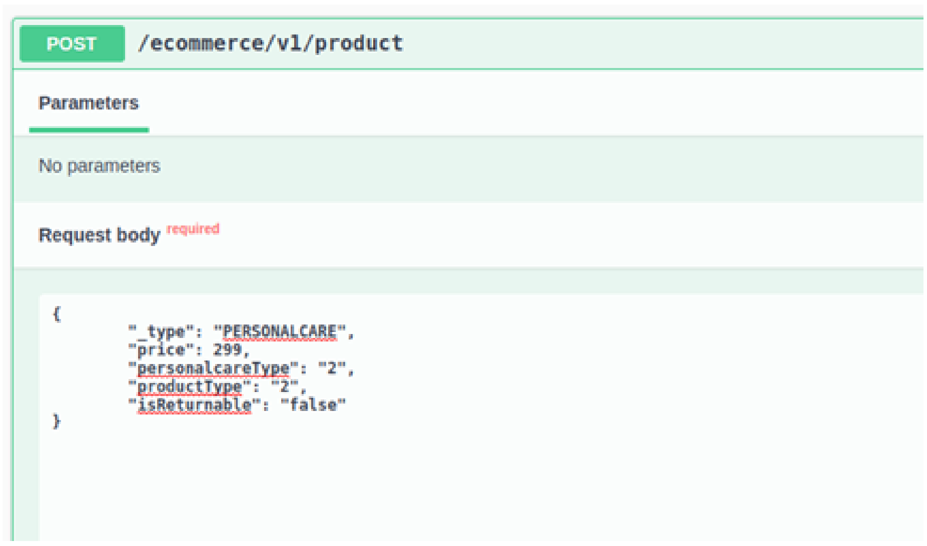
Response headers

```
connection: keep-alive
content-type: application/json
date: Mon,08 May 2023 20:41:03 GMT
keep-alive: timeout=60
transfer-encoding: chunked
```

Figure 18. Response for successfully persisting Grocery entity

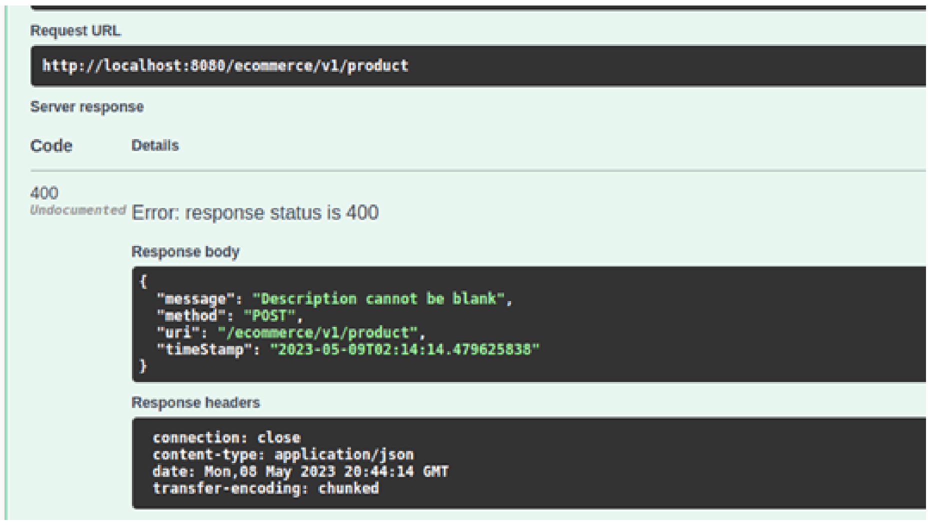
In this example, a grocery entity is persisted into the database. Validation is used on all the fields. For example, the request will send back a 400 bad request status code if the bestBeforeDate entered is in the past. Here groceryType is mapped to Java enum(0 stands for Dairy, 1 stands for Fruit, 2 stands for Meat and 3 stands for Vegetable).

In case of successful creation, the Grocery entity along with its id is sent back to the client as JSON. The status code in this scenario would be 201



```
{
        "_type": "PERSONALCARE",
        "price": 299,
        "personalcareType": "2",
        "productType": "2",
        "isReturnable": "false"
}
```

Figure 19. POST on create product with incomplete/wrong data



```
http://localhost:8080/ecommerce/v1/product
```

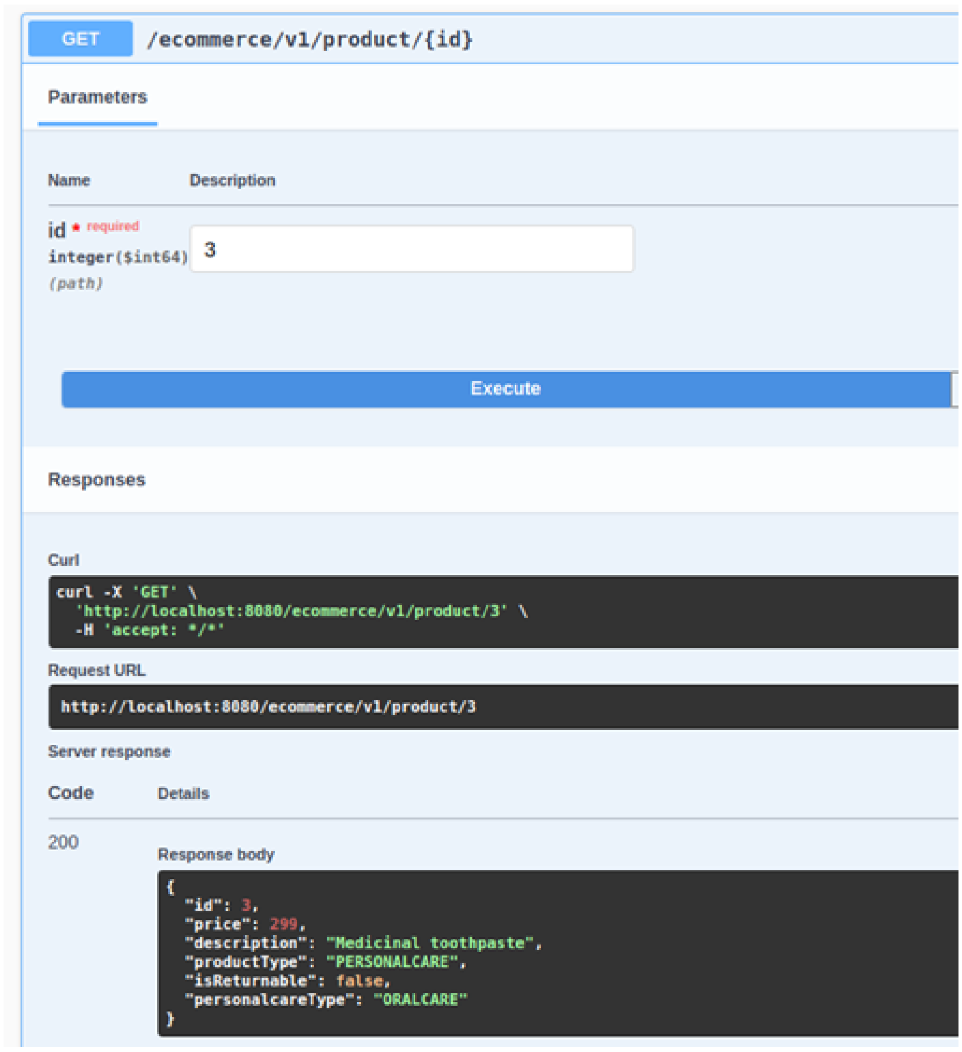
400 *Undocumented* Error: response status is 400

```
{
  "message": "Description cannot be blank",
  "method": "POST",
  "uri": "/ecommerce/v1/product",
  "timeStamp": "2023-05-09T02:14:14.479625838"
}
```

```
connection: close
content-type: application/json
date: Mon,08 May 2023 20:44:14 GMT
transfer-encoding: chunked
```

Figure 20. Response on trying to create product listing with wrong/incomplete data

In this scenario, a user tries to create a product listing by sending incomplete data as the request body. In this case, the description field is left empty. The server would respond with status code 400 with appropriate error messages. Examples of incomplete/invalid data includes negative or zero price, empty string or string containing only whitespace for keys requiring string.



```
/ecommerce/v1/product/{id}
```

id * required
integer($int64)
(path)
3

Curl

```
curl -X 'GET' \
  'http://localhost:8080/ecommerce/v1/product/3' \
  -H 'accept: */*'
```

Request URL

```
http://localhost:8080/ecommerce/v1/product/3
```

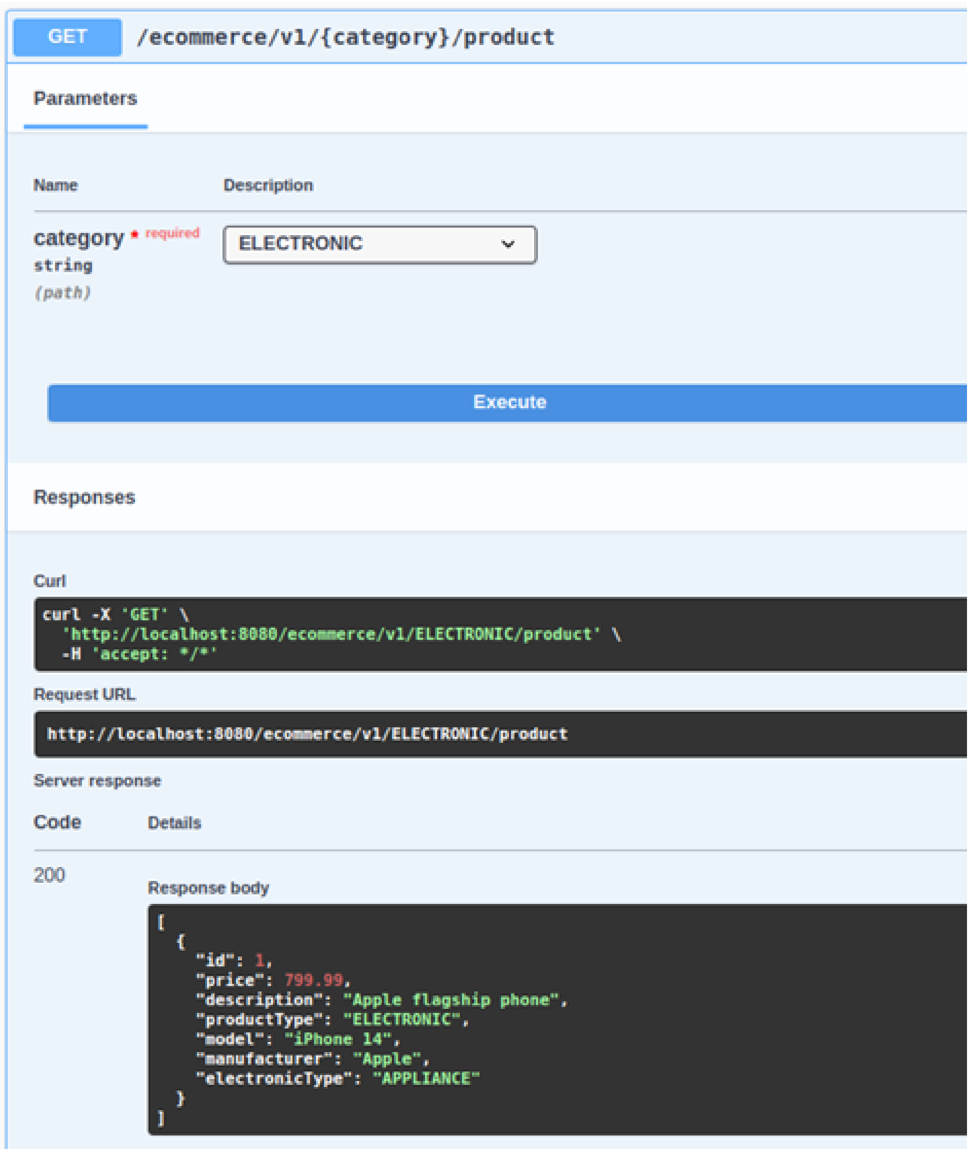
200

Response body

```
{
  "id": 3,
  "price": 299,
  "description": "Medicinal toothpaste",
  "productType": "PERSONALCARE",
  "isReturnable": false,
  "personalcareType": "ORALCARE"
}
```

Figure 21. Fetching a product by its id

In this case, a product is being fetched from the database given by their id. Depending upon the type of the product which the id belongs to, an appropriate sql query is run giving all the details of the product.



GET /ecommerce/v1/{category}/product

category * required
string
(path)
ELECTRONIC

Curl

```
curl -X 'GET' \
  'http://localhost:8080/ecommerce/v1/ELECTRONIC/product' \
  -H 'accept: */*'
```

Request URL

```
http://localhost:8080/ecommerce/v1/ELECTRONIC/product
```

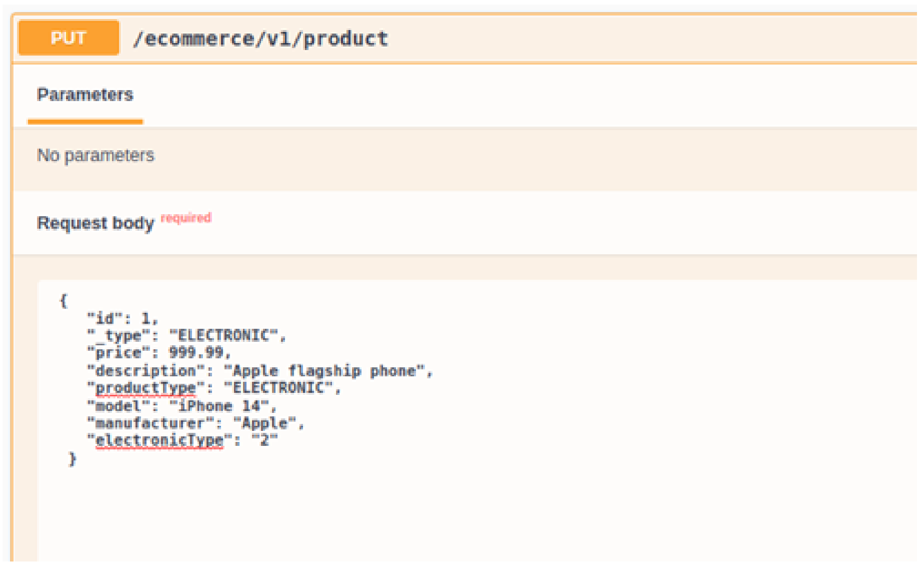
200

Response body

```
[
  {
    "id": 1,
    "price": 799.99,
    "description": "Apple flagship phone",
    "productType": "ELECTRONIC",
    "model": "iPhone 14",
    "manufacturer": "Apple",
    "electronicType": "APPLIANCE"
  }
]
```

Figure 22. Fetching all products belonging to a particular category

For this as seen on Figure 22, all products belonging to a particular category can be fetched. The endpoint in question would be /ecommerce/v1/{category}/product on which GET method shall be used. Category is a query parameter and can have values ELECTRONIC, GROCERY. PERSONALCARE.



PUT /ecommerce/v1/product

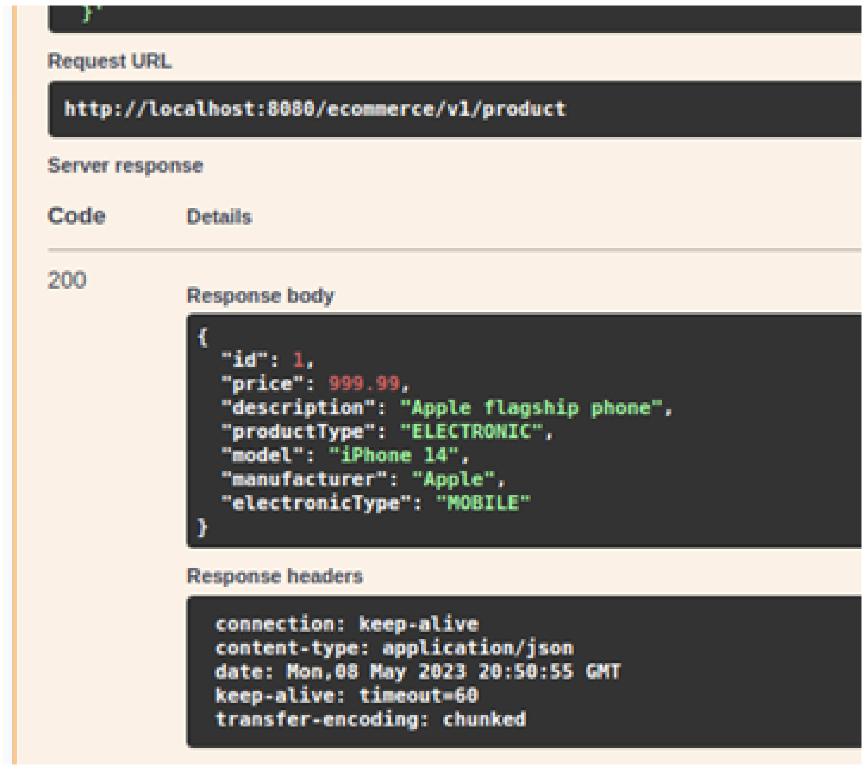
Parameters

No parameters

Request body required

```
{
    "id": 1,
    "_type": "ELECTRONIC",
    "price": 999.99,
    "description": "Apple flagship phone",
    "productType": "ELECTRONIC",
    "model": "iPhone 14",
    "manufacturer": "Apple",
    "electronicType": "2"
 }
```

Figure 23. Update a product listing



Request URL

```
http://localhost:8080/ecommerce/v1/product
```

Server response

Code Details

200

Response body

```
{
  "id": 1,
  "price": 999.99,
  "description": "Apple flagship phone",
  "productType": "ELECTRONIC",
  "model": "iPhone 14",
  "manufacturer": "Apple",
  "electronicType": "MOBILE"
}
```

Response headers

```
 connection: keep-alive
 content-type: application/json
 date: Mon,08 May 2023 20:50:55 GMT
 keep-alive: timeout=60
 transfer-encoding: chunked
```

Figure 24.

Figure 25. Response body after updating

PUT is used as the preferred method to update a product listing over PATCH for this. In PUT, the entire entity object which needs to be updated is sent. On validation, the updated entity is persisted into the database and the updated values of product are sent back as JSON response with response code 200.

4.2 Unit Testing

Being one of the major techniques used for testing, Unit testing involves evaluating individual software application units or components separately from the rest of the system to make sure they function as expected.

Unit testing's objective is to find bugs early in the development cycle, when fixing them will be simpler and less expensive. In a continuous integration and delivery (CI/CD) pipeline, unit tests are often automated and run.

Developers create test cases that simulate the behaviour of the code under test in order to create unit tests. Various scenarios, such as normal inputs, boundary cases, and error conditions are frequently covered by the tests.

A good unit test should have good code coverage i.e the line covered by unit tests should be about 95%.

The following were the code coverage of different layers of the system:

service	93%
controller	85%

Table 5. Unit test coverage

4.3 Integration Testing

Integration testing is a type of software testing aiming to verify whether different components of a system are working correctly with one another. Integration testing's goal is to find errors in the interfaces and interactions between various modules and to make sure the integrated system functions as intended.

Establishing the test plan for integration testing is the initial step. This should have a thorough plan outlining the modules to be evaluated, the sequence in which they will be integrated, the testing strategy to be employed, and the testing results that should be anticipated.

The execution of integration testing requires the setup of an appropriate test environment. The network, software, and hardware settings necessary to evaluate the integrated system should be included in this. And most importantly, good test data should be there to do better integration testing. To account for different eventualities, the test data should contain both valid and invalid input data.

Also, it is necessary to create test cases for all potential module integration scenarios. These test cases should be created to confirm both the overall functionality of the integrated system and the accuracy of the interactions between the modules.

4.4 End-To-End Testing

End-to-end testing tries to validate the complete system, including all of its interrelated parts and subsystems, from beginning to end. Ensuring that the software programme behaves as anticipated in a real-world situation is the goal of end-to-end testing.

The following scenarios were tested

1. Creating

Test conditions	**Expected Result**	**Status Code**	**Actual Result**
Missing attribute	Validation Error	400	As expected(Pass)
Invalid input	Validation	400	As expected(Pass)
Mismatch input type	Validation Error	400	As expected(Pass)

Table 6. Scenarios tested for creating a product

2. Reading

- findById

Test conditions	**Expected Result**	**Status Code**	**Actual Result**
Zero or negative	Validation Error	400	As

id			expected(Pass)

Table 7. Scenarios tested for fetching a product

- findAllByCategory()

Test conditions	**Expected Result**	**Status Code**	**Actual Result**
Invalid entry in category path param	Validation Error	400	As expected(Pass)

Table 8. Scenarios tested for fetching all products from a category

3. Update

Test conditions	**Expected Result**	**Status Code**	**Actual Result**
ID with no product existing	Validation error	400	As expected(Pass)
Missing attribute	Validation Error	400	As expected(Pass)
Invalid input	Validation	400	As expected(Pass)
Mismatch input type	Validation Error	400	As expected(Pass)

Table 9. Scenarios tested for updating a product

4. Delete

Test conditions	Expected Result	Status Code	Actual Result
ID with no product existing	Validation error	400	As expected(Pass)
Invalid input	Validation	400	As expected(Pass)
Mismatch input type	Validation Error	400	As expected(Pass)

Table 10. Scenarios tested for deleting a product

CHAPTER 5

CONCLUSIONS

5.1 Conclusions

We successfully developed a Spring Boot-based REST API for this project that includes numerous endpoints that are used for various functions. The API allows users to retrieve all products, categories, and objects that fit under a specific category. The architecture of our API is composed of three layers: a controller layer, a service layer, and a DAO layer. While the service layer handles the business logic, the DAO layer interacts with the database to retrieve and save data. We have also developed test cases for the controller, service, and DAO levels to ensure the API functions correctly.

In conclusion, we have created a dependable and scalable REST API using Spring Boot. The 3-layer nature of the codebase makes it easy to maintain and modify as needed. Using the PostgreSQL database has allowed us to efficiently store large amounts of data and retrieve it quickly. Because Docker and PostgreSQL are used, the API is ubiquitous and can be easily deployed on any platform.

One of the most significant advantages of this API is its versatility. Applications ranging from inventory management systems to e-commerce platforms can be made using it. Since it is possible to access all categories, all products, and all items that fit under a certain category, creating specialised interfaces for diverse applications is straightforward.

5.2 Future Scope

Future improvements to the RESTful API created for this project have the potential to boost its usability, performance, and security. By implementing these upgrades, the API can assist businesses and developers in efficiently achieving their goals.

The adoption of authentication and authorisation methods will increase the security of the API. Authentication will ensure that only users with the proper authorization can use the API, whereas authorisation will define the scope of each user's access to the API.

Caching, which can improve API efficiency by reducing the number of database requests done, is a further important consideration. By using a caching method, response times will be quicker and scalability would be enhanced.

The ability to filter and paginate the data is essential when working with enormous datasets. Filtering and pagination features of the API will allow clients to only get the data they need, reducing network traffic and improving speed.

5.3 Limitations

- Limited number of filters
- Limited data
- Authentication and protection API would need to be there

REFERENCES

[1] Roy T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," PhD dissertation, University of California, Irvine, 2000.

[2] Leonard Richardson, Sam Ruby, and David Heinemeier Hansson, "RESTful Web Services," O'Reilly Media, Inc., 2007.

[3] S. Y. Lee, J. H. Park, and H. S. Kim, "Spring Boot: A Framework for Building Web Applications with Minimal Configuration," in Proceedings of the International Conference on Software Engineering and Knowledge Engineering, San Francisco, USA, 2014, pp. 775-780.

[4] S. Y. Lee, J. H. Park, and H. S. Kim, "Spring Boot: A Framework for Building Web Applications with Minimal Configuration," in Proceedings of the International Conference on Software Engineering and Knowledge Engineering, San Francisco, USA, 2014, pp. 775-780.

[5] S. Basar and M. Demirbas, "Spring Boot vs. WildFly Swarm: A Performance Comparison," in Proceedings of the International Conference on Computer Science and Engineering, Istanbul, Turkey, 2017, pp. 320-325.

[6] R. Alotaibi, M. Alshammari, and A. Aljuhani, "Spring Boot Security: An Analysis of Best Practices," in Proceedings of the International Conference on Computer Science and Applications, Kuala Lumpur, Malaysia, 2019, pp. 107-111.

[7] R. Maier, S. Schmid, and A. Winter, "Design and Implementation of a Microservice Architecture with Spring Boot," in Proceedings of the International Conference on Advanced Information Systems Engineering, Ljubljana, Slovenia, 2016, pp. 285-300.

[8] P. Di Tommaso, A. Caciagli, and R. Giorgi, "Developing Cloud-Native Applications with Spring Boot and Kubernetes," in Proceedings of the International Conference on Cloud Computing and Services Science, Porto, Portugal, 2018, pp. 214-221.

[9] J. Ullman and J. Widom, "A First Course in Database Systems," 3rd ed. Prentice Hall, Upper Saddle River, NJ, USA, 2008.

[10] J. U. Adogbeji, K. A. Adegboyega, and T. M. Folorunso, "Evaluating the Performance of Spring Boot Framework for Web Application Development," in Proceedings of the International Conference on Computational Science and Computational Intelligence, Las Vegas, USA, 2019, pp. 515-520.

[11] M. A. Tahir, M. S. Abro, and A. Memon, "Microservices with Spring Boot and Spring Cloud: An Overview," in Proceedings of the International Conference on Internet of Things and Cloud Computing, Cambridge, UK, 2017, pp. 92-97.