

Spring Boot Application using Three Layered Architecture

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

in

**Computer Science and Engineering/Information
Technology**

By

Sarthak Kumar(191387)

Under the supervision of

Mr. Prateek Thakral

to



Department of Computer Science & Engineering and
Information Technology

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

Certificate

Candidate's Declaration

I hereby declare that the work presented in this report entitled “Spring Boot Application using Three Layered Architecture” in fulfillment 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 July 2022 to May 2023 under the supervision of **Mr. Prateek** (Assistant Professor, Department of CSE, Jaypee University of Information Technology, Waknaghat).



(Student Signature)

Sarthak Kumar

191387

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

(Supervisor Signature)

Mr. Prateek

Assistant Professor (Grade II)

Computer Science & Engineering

Dated:



(Signature)

Ujjawal Mishra

Director of Engineering

Zopsmart

Dated: 8-May-2023

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 Paper

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)	
	<input type="checkbox"/> All Preliminary Pages <input type="checkbox"/> Bibliography/ Images/Quotes <input type="checkbox"/> 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

All compliments and praise are due to God who empowered me with strength and sense of devotion to successfully accomplish this project work successfully.

I am really grateful and wish my profound indebtedness to Supervisor **Mr. Prateek, Assistant Professor (Grade II)**, Department of CSE Jaypee University of Information Technology, Waknaghat. Deep Knowledge & keen interest of my supervisor in the field of “**Machine Learning**” to carry out this project. 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 like to express my heartiest gratitude to **Mr. Prateek**, Department of CSE, for his kind help to finish my 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.

Sarthak Kumar

191387

TABLE OF CONTENT

ABBREVIATIONS	vi
LIST OF FIGURES	vii
LIST OF TABLES	ix
1 INTRODUCTION	1
1.1 About Organization	1
1.2 Introduction	2
1.3 Problem Statement	4
1.4 Objectives	5
1.5 Methodology	6
1.6 Organization of Report	7
2 LITERATURE SURVEY	9
3 SYSTEM DEVELOPMENT	13
3.1 Overview	13
3.2 Overall Design	13
3.3 Software & Environment	14
3.3.1 Software Requirements	14
3.3.2 Hardware Requirements	26
3.4 Implementation & Deployment	26
3.5 Database Schema	27
4 EXPERIMENTS& RESULT ANALYSIS	29
4.1 Overview	29

4.2 Experiment Results	30
4.3 Outputs at Various Stages	30
4.4 Performance Analysis	48
5 CONCLUSIONS	49
5.1 Conclusion	49
5.2 Future Scope	50
REFERENCES	52

LIST OF ABBREVIATIONS

S. No.	Abbreviation	Full Form
1	DAO	Data Access Object
2	DTO	Data Transfer Object
3	REST	Representational State Transfer
4	API	Application Programming Interface
5	UI	User Interface

LIST OF FIGURES

Figure No.	Title of Figures	Page No.
1.1	Company Logo	1
1.2	Three Layered Architecture	2
3.1	System Design	13
3.2	Database Schema (Many to One)	27
4.1	SwaggerUI for API	31
4.2	Schemas in API	32
4.3	POST request for Category	33
4.4	GET request for Category	34
4.5	PUT request for Category	35
4.6	DELETE request for Category	36
4.7	GET request for Category when empty	37
4.8	POST with null category name	38
4.9	POST with category name already present	38
4.10	PUT request for Category when name already exists	39
4.11	POST request for Product	40
4.12	GET request for Product	41
4.13	GET request for products of a particular category	41
4.14	PUT request for Product	42
4.15	DELETE request for Product	43
4.16	GET request for Product when category	43

	with Id 2 is deleted	
4.17	GET request for Product when no products exists	44
4.18	POST request for Product with wrong Category Id	45
4.19	POST request for Product with name that already exists	46
4.20	POST request for Product with negative price	47
4.21	Unit Testing	48

ABSTRACT

A well-liked framework for creating dependable and scalable web apps is Spring Boot. A Controller layer, a Service layer, and a DAO layer make up its three-tiered architecture. Receiving incoming HTTP requests and sending acceptable replies are the responsibilities of the Controller layer, sometimes referred to as the Presentation layer. The Service layer, sometimes referred to as the Business layer, serves as a bridge between the Controller layer and the DAO layer and houses the business logic. Last but not least, interacting with the database and carrying out CRUD (Create, Read, Update, and Delete) activities fall within the purview of the DAO layer, also referred to as the Data layer.

Developers may quickly create RESTful APIs using Spring Boot that let users interact with the application's data. This study addresses the creation of an API utilising the aforementioned three-layer design. 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.

Using Spring MVC, the application's Controller layer manages incoming HTTP requests and transfers their processing to the appropriate Service layer method. The DAO layer is then utilised by the Service layer to retrieve or modify data as needed. Spring Data JPA, a potent ORM (Object-Relational Mapping) technology that makes database access and maintenance simple, is used by the DAO layer to communicate with the database.

In this report, we'll go over the advantages of a three-layer design and how it keeps the code manageable and modular. We'll also look at the several Spring Boot annotations that are applied to the application and how they cooperate to

support quick development. We will also go over the recommended practises for testing the application with JUnit and Mockito to make sure the API performs as anticipated.

Overall, this paper gives a thorough explanation of how a three-layer design was used to construct a Spring Boot API. It highlights how Spring Boot can be used to create RESTful APIs rapidly and that are simple to maintain while adhering to best practises.

CHAPTER 1

INTRODUCTION

1.1 About Organization

Innovative retail technology provider ZopSmart offers all the resources needed to establish an online business. ZopSmart provides a set of tools that can help you get there quickly and effectively whether you're a conventional store trying to develop an omni-channel business or an online-only shop looking to grow an e-commerce business.

E-commerce has been increasingly popular over time. And it is clear that the market for online stores offers a variety of possibilities. Let's say you're a shop looking to launch your own online business. ZopSmart can help companies create a strong foundation for their online growth using the many technologies at their disposal. Many vendors can use the products that ZopSmart offers to start their own online retail stores.



Figure 1.1: Company Logo

ZopSmart places a strong emphasis on increasing productivity and efficiency for our clients. Regardless of the method you employ, our products help customers increase their online retail sales as much as possible with the least amount of work

while also guaranteeing that they pay a reasonable price, which is feasible because many consumers will share the advantages.

With state-of-the-art concepts and technology, we'll make sure that everyone is completely satisfied—client and customer alike.

1.2 Introduction

Popular Java-based framework Spring Boot provides an easy and effective way to create reliable and scalable web applications. It is a favourite or more famous among developers since it enables quick application creation with little configuration. Spring Boot's assistance for creating APIs with a clear and user-friendly interface is one of its main benefits.

In this project report, we'll examine how a three-layer design might be used to create a Spring Boot API. The controller layer, service layer, and dao layer are the three logical levels that are divided into by the three-layer architectural design pattern. Building applications may be done in a more modular and scalable way because of this separation of concerns.

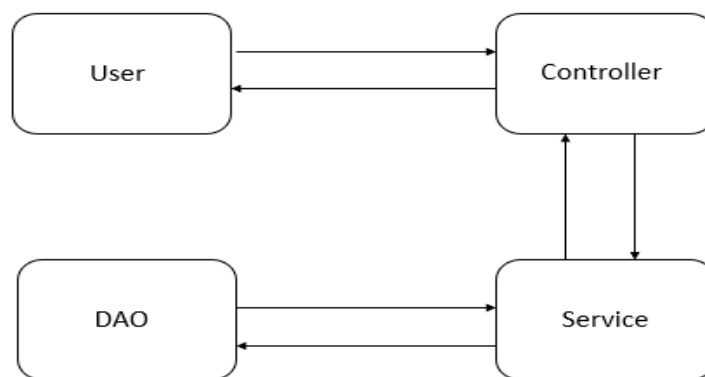


Figure 1.2: Three Layered Architecture

The input and output from users are handled by the controller layer. This layer would be in charge of managing requests and replies for a Spring Boot API. The application's business logic is contained in the service layer, sometimes referred to as the business layer. Data processing and business rule enforcement are within the purview of this layer. The dao layer, also known as the persistence layer, and it is in charge of communicating with the database and obtaining and storing information.

We can obtain a high level of abstraction and modularity in our Spring Boot API by adopting a three-layer architecture. This makes the application more scalable and maintainable as it expands and changes over time. It also makes it possible to test and debug the application more effectively.

We will be creating many APIs for an e-commerce application in our Spring Boot API, including ones for retrieving all categories, all items, and all products in a certain category. Three layers—the controller layer, the service layer, and the dao layer—will be used to implement these APIs.

The RESTful endpoints for each API will be located in the controller layer. These endpoints are in charge of managing incoming requests and providing the necessary responses. The service classes that implement each API's business logic will be found in the service layer. These classes will handle data processing, apply business rules, and invoke data layer functions. The repository classes that communicate with the database and carry out CRUD activities will be found in the dao layer.

To sum up, the three-layer architecture is an effective design pattern that can be used to create Spring Boot APIs that are scalable and maintainable. We can obtain

a high degree of abstraction and modularity by breaking the programme up into three logical levels, which enables easier testing and debugging of the application. Using this architecture, it is simple to implement APIs that fetch all categories, all products, and all products within a specific category. At last we will be deploying our RESTful API to a cloud server.

1.3 Problem Statement

The requirement for a dependable and effective method of keeping and retrieving data connected to items and categories is the issue that the API seeks to address. A system that can manage product-related data, such as product names, pricing, and classifications, in an organised and effective manner is essential for many firms that deal with a significant number of items. Additionally, they require a method for quickly and easily retrieving this data since it is frequently required for a variety of business operations like inventory management, sales analysis, and reporting.

Creating an API that can effectively and efficiently fulfil these needs is the difficult task. All product and category data ought to be accessible via the API, which should also be able to filter it according to particular standards like category. Additionally, it ought to offer endpoints for product creation and updating. The API must also be planned to support several requests at once, guarantee data integrity, and offer high performance and scalability.

The API will be created utilising a three-layer architecture with distinct controllers, services, and dao layers to address these difficulties. While the service layer will contain business logic and manage the communication between the presentation and data layers, the controller layer will manage requests and

responses to and from the API. All database interactions and data-related operations, such as saving and retrieving product and category data, will be handled by the data layer.

Additionally, Spring Boot, a well-known Java-based framework for creating web applications, will be used to build the API. A number of characteristics offered by Spring Boot may be used to create an API that is reliable, scalable, and fast. SwaggerUI, a user-friendly tool for creating interactive API documentation, will also be used to document the API.

The overall objective is to create an API that is dependable, effective, and capable of meeting the needs of companies that deal with a variety of products. The API should be simple to use, scalable, offer excellent speed and data integrity, and guarantee data security.

1.4 Objectives

This API's main goal is to make it easier for users to interact with product and category data. Through a clearly defined set of RESTful APIs, it aims to give users a seamless experience by making it simple to access and modify product and category data. This API provides data validation and error handling features in an effort to improve the management of product and category data.

Modularity, scalability, and extensibility are all considered in the design of the API. Users may easily incorporate it into their current systems and applications because of this. The API leverages contemporary technology and follows accepted industry standards for software development to guarantee that the code is tested, manageable, and reusable. The API also places a high priority on user

data security and includes the necessary procedures for authentication and authorisation.

The goal is to reduce time and effort for developers who are creating apps that need product and category data by offering a reliable and effective API. The API will simplify data administration and retrieval, freeing developers to concentrate on the main functionality of the application. The API may be linked with other platforms and apps according to RESTful principles, making it a flexible and adaptable solution for managing product and category data. Overall, the API offers both consumers and developers a trustworthy and effective solution.

1.5 Methodology

A systematic technique was used to design the Spring-Boot API in accordance with the stated criteria. Gathering requirements, planning, implementing, testing, and deploying were some of the stages that were included in the development process.

Knowing the API's needs was the first step in the technique. This was accomplished by looking at the problem description and comprehending our project's principal goal. The design of the API architecture came next when the requirements were crystal clear. For this project, a three-layer architecture including a controller layer, service layer, and dao layer was selected.

Our next step after designing the architecture was to put the API into practice. The API was created with the help of the Spring-Boot framework. Using Spring-Boot Data Access Objects, Spring-Boot Services, and Spring-Boot Controllers, the controller layer, service layer, and dao layer were each implemented.

The API was tested once the implementation was finished to make sure it adhered to the specifications. There were several testing methods utilised, including unit testing using frameworks like Mockito and JUnit5. The API was published to an EC2 server operating on AWS for public usage following the conclusion of testing and the resolution of all issues. Our API was set up on EC2 in Docker containers.

Overall, the approach taken was successful in producing a usable and expandable Spring-Boot API. Prior to being installed on the production server, it made sure that the requirements were satisfied and the API was tested.

The three-layer design used allowed for a clear separation of concerns and made it simple to update and maintain the API. The deployment process was streamlined and effective thanks to the usage of Docker containers and the Spring-Boot framework, which allowed for quick development and simple deployment. As a consequence, the Spring-Boot API was developed using a systematic and efficient technique, producing a high-quality API that complied with the specifications.

1.6 Organization of Report:

Chapter 1: This chapter gives a brief introduction to our spring boot application and the three layered architecture and their benefits, its applications, the objective of the system, and its motivation. In this chapter, we also discuss the problem statement of our project around which our project aim revolves. We also discuss the methodology or solution that has been used to achieve our task, i.e. to build a spring boot RESTful API using the three layered architecture for an e-commerce application.

Chapter 2: This chapter contains literature surveys that provide a summary of individual papers and documentations which helped us to build our application or RESTful API using the proposed three layered architecture method. There have been solutions to various problems faced while building the project using different types of technologies, tools, languages and techniques by various research workers which have all had varying levels of success, and about the way/ approach that we decided to take to build our application.

Chapter 3: In this chapter, our main aim was to explain the step-by-step method we used to build the setup for the research done in the report. Our first step for this to start with was to start exploring various research papers and documents and even explore various e-commerce portals to get a list of features that we must need to include in our project or API at the backend. It mainly outlines the tools and technologies that have been used.

Chapter 4: In this Chapter, the report contains the results or outputs obtained at different stages of the project, so that we get a clear idea of the accuracy of our RESTful API built. In this chapter then we discuss the different types of use cases where we can use our application.

Chapter 5: This chapter contains a summary and conclusion from the research presented in the report and outputs that we got from the final project model. It also outlines areas for further research and work that can be added to it. It also contains the innovative work that was generated from the analysis of work and we also have a conclusion of the results achieved.

CHAPTER-2

LITERATURE SURVEY

Omar S. Gómez et al. proposed that creating REST APIs for CRUD activities in Spring Boot might be a tedious and time-consuming effort. They created the Domain-Specific Language (DSL) CRUDyLeaf to automate the creation of RESTful APIs from the Spring Boot entity model in order to solve this problem. The DSL is intended to lessen coding effort and increase developer productivity.

The DSL is thoroughly explained in the article, along with a step-by-step tutorial on how to create REST APIs using it. To assess CRUDyLeaf's efficacy, the authors also ran tests. The findings demonstrated that CRUDyLeaf shortens development times by up to 60% while enhancing code quality by cutting down on defects.

The authors drew attention to several of CRUDyLeaf's drawbacks, including its inability to manage intricate connections between things and the requirement that developers have a solid grasp of Spring Boot and RESTful APIs. They did, however, make the suggestion that these restrictions may be overcome in other studies.

Overall, by proposing a DSL that makes it easier to design REST APIs in Spring Boot, the article makes a significant addition to the area of software engineering. Insights on the advantages and disadvantages of employing such a tool are also given.

M. S. Takalika et al. conducted a study to evaluate how REST APIs were implemented. Their study aimed to identify the main difficulties that developers encountered when implementing RESTful web services and to suggest potential solutions to these difficulties.

The choice of proper HTTP methods, URI architecture, data encoding, and error handling were among the difficulties found. The authors put up a list of recommendations to assist developers in overcoming these difficulties, including following the REST limitations, utilising meaningful and descriptive URIs, and adopting standard data formats.

Their study emphasises how crucial it is to deploy REST APIs with thorough preparation and respect to RESTful principles. To guarantee the effective deployment of RESTful web services, developers may benefit greatly from the recommendations made by Kulkarni and Takalika.

Namrata Soni et al. proposed in their paper the deployment of a Spring Boot application as a REST API in Amazon Web Services (AWS) was suggested in their article. The writers describe the advantages of utilising AWS and talk about how crucial it is to deploy apps on the cloud. A step-by-step tutorial on how to deploy a Spring Boot application in AWS Elastic Beanstalk is also available.

The article outlines the architecture of a typical Spring Boot application and demonstrates how a REST API can be created from it. The authors then go over the various AWS services, such as Elastic Beanstalk, Amazon Elastic Container Service (ECS), and Amazon Lambda, that may be utilised for deployment.

The paper's primary concern is with the application's deployment in Elastic Beanstalk, a fully managed service for web application deployment. On how to set up the application, setup the environment, and deploy it to AWS, the authors give thorough instructions.

The document offers a helpful manual for programmers who wish to utilise AWS to deploy their Spring Boot apps in the cloud. Developers can easily follow along with and comprehend the deployment process thanks to the step-by-step directions and thorough explanations.

In conclusion, the delivery of high-quality software products depends on the use of automated testing methods. They contribute to a decrease in testing time, an increase in testing process effectiveness, and an improvement in the overall quality of the software product. As a result, it is imperative that software developers take automated testing methods into account while developing new software.

Sai Subramanyam Upadhyayula et al. discussed how to construct REST APIs with Spring Boot in their business. The authors emphasised the benefits of utilising Spring Boot for creating REST APIs, including how quickly the application can be set up, how simple it is to configure, and how many frameworks and tools are available. They also described how to define the API endpoints, implement the business logic, and handle errors while building a REST API using Spring Boot.

The authors concentrated on the practical elements of developing REST APIs, including examples of setups and code snippets for a variety of scenarios,

including managing authentication and authorisation, leveraging multiple data sources, and enhancing API performance.

Additionally, they talked about REST API testing and deployment, highlighting the value of automated testing and continuous integration and delivery (CI/CD) pipelines.

Overall, Sai Subramanyam Upadhyayula et al. offered a thorough tutorial for creating REST APIs with Spring Boot that covered both the theoretical ideas and the actual steps in the procedure. Developers who are new to creating REST APIs or who wish to advance their knowledge in this field can benefit greatly from their work.

CHAPTER-3

SYSTEM DEVELOPMENT

3.1 Overview

The basic aim of this chapter is to provide the theoretical background needed to comprehend the report's content. It mainly outlines the tools and technologies that have been used for building this project. This chapter aims to provide the reader with a clear understanding of the underlying principles of spring boot RESTful APIs and the various tools and techniques used to build an accurate application.

3.2 Overall Design

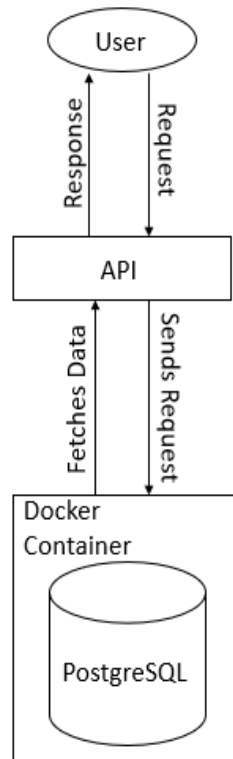


Figure 3.1: System Design

3.3 Software & Environment

In this section of the chapter we will be discussing about the different types of tools and technologies that we require to run our project. We will also be discussing the software and hardware requirements for this project.

3.3.1 Software Requirements

1. **Java:**

Sun Microsystems developed Java, an object-oriented programming language, in 1995. It was created as a substitute for the exceedingly challenging C ++ and is characterized by the lack of logical constraints and orderly development. In addition, the new strategy needed to address pressing problems with security, cross-platform compatibility, adaptability, etc.

Because Java was developed to streamline the programming process and get rid of extraneous steps and components, it also has advantages. Consequently, the following characteristics of this language should be emphasized:

- Convenience and simplicity: The language follows a fairly simple logic. A harsh code structure enables easy solution maintenance. The developer may quickly and simply explore any intricacy of code and enhance the necessary component. Additionally, specialized auto-complete and auto-correction features make the specialist's job easier.

- Versatility: Both frontend and backend component development may be done in Java. Additionally, IT experts use it to create solutions for banks and payment systems, as well as high-profile software for automobiles and smart homes.
- Cross-platform: The tagline for this functionality in Java is "Write once, run anywhere." This way of running programmes has the benefit of total byte-code independence from the hardware and operating system. Java code may therefore execute on any machine for which the JVM was designed. This will ensure that a programme runs smoothly on Linux, Windows, and other operating systems.
- Safety: The Java platform specifies a number of APIs for access control, secure transmission, public infrastructure, and authentication.
- Strong Stack: Most businesses utilize the Spring and Hibernate frameworks when working with Java. As a result, if necessary, it is simple to replace the development team. Many market professionals have experience with these technologies.
- Reliability: The inheritance hierarchy makes code easier to comprehend and causes fewer forced mistakes. Strong typing ensures accurate data interpretation.

2. **Gradle:**

Gradle is an effective build automation tool made to handle challenging build procedures with ease. Due to its efficiency and versatility, it is a commonly used open-source tool for Java applications. Gradle Java is an add-on to Gradle that offers more features for creating Java projects.

Developers can effectively create Java projects and manage dependencies with Gradle Java. It is a flexible tool for developers since it supports many different languages and operating systems. Additionally, it offers a thorough API for creating original plugins and extensions.

Gradle Java's build script DSL, which is incredibly versatile and simple to use, is one of its key features. This enables developers to produce build scripts that are specifically tailored to their needs. Additionally, Gradle supports incremental builds, which limits rebuilding to project components that have changed since the last build.

The support for dependency management in Gradle Java is another important feature. Repositories, local files, and remote URLs are just a few of the sources from which it may handle dependencies. This makes it simple for developers to manage dependencies and make sure that the libraries and frameworks they use in their projects are the right versions.

Additionally, Gradle Java offers superb integration with other programmes and systems, including Jenkins, IntelliJ IDEA, and Android Studio. This makes it simple for developers to integrate Gradle into their current processes and resources.

In conclusion, Gradle Java is a strong build automation tool that offers top-notch assistance for creating Java applications. It is a well-liked option among developers because to its versatility, performance, and simplicity of use. It is a flexible tool that can be used for a variety of projects thanks to its support for many languages and platforms.

3. **Spring Boot:**

Spring Boot is an open-source framework that is used to create standalone Spring-based applications. It is designed to simplify the process of developing and deploying microservices and web applications. Spring Boot is based on the popular Spring Framework and provides a fast and efficient way to build enterprise-grade applications.

Features of Spring Boot:

- **Spring Boot Starter:** Spring Boot Starter is a set of pre-configured dependencies that are commonly used in Spring applications. It helps developers to start building an application with minimal configuration.
- **Embedded Server:** Spring Boot comes with an embedded server that eliminates the need for deploying applications on a standalone server. The embedded server can be Tomcat, Jetty, or Undertow.
- **Auto-configuration:** Spring Boot provides a feature called Auto-configuration that automatically configures the application based on the dependencies that are included. It eliminates the need for writing boilerplate code and makes development faster.

- **Dependency Injection:** Dependency injection is a popular design pattern that is used to manage dependencies between objects in an application. It is an important feature of the Spring Framework and is widely used in Spring Boot applications. Dependency injection helps to decouple the code and makes it easier to test and maintain. In this report, we will discuss dependency injection in Spring Boot.

Benefits of using Spring Boot:

- **Faster development:** Spring Boot eliminates the need for writing boilerplate code and configuring the application. It provides a set of pre-configured dependencies and an embedded server that makes development faster.
- **Easier deployment:** Spring Boot applications can be deployed on a standalone server or as a containerized application. It provides an embedded server that eliminates the need for configuring the server.
- **Reduced maintenance:** Spring Boot provides a set of production-ready features that help developers to monitor and manage the application. It reduces the maintenance effort required to manage the application.

4. **Github:**

GitHub is a popular platform for software development that allows developers to collaborate on projects and manage their code repositories.

It was founded in 2008 and has since become one of the most widely used platforms in the software industry. In this report, we will discuss the key features and benefits of GitHub.

Features of GitHub:

- **Version Control:** GitHub provides powerful version control tools that enable developers to manage their code repositories efficiently. It allows developers to track changes to their code, collaborate with other developers, and revert to previous versions of their code when needed.
- **Collaboration:** Developers can collaborate on a project. It provides tools for creating issues, reviewing code changes, and merging changes into the code repository. This enables developers to work together on projects, share their knowledge and expertise, and contribute to the development of high-quality software.
- **Integrations:** GitHub provides a wide range of integrations with other software development tools and platforms. It integrates with popular project management tools like Trello and Asana, as well as with continuous integration and deployment tools like Jenkins and Travis CI.
- **Community:** GitHub has a large and active community of developers who contribute to the development of open-source projects. It provides a platform for developers to showcase their

work, collaborate with others, and contribute to the development of software that is accessible to all.

Benefits of GitHub:

- **Accessibility:** GitHub provides a platform for developers to make their code accessible to others. It allows developers to share their work with the world, collaborate on projects, and contribute to the development of open-source software.
- **Efficiency:** GitHub provides powerful tools for managing code repositories and collaborating on projects. This enables developers to work more efficiently, save time, and produce high-quality software.
- **Learning Opportunities:** GitHub provides a platform for developers to learn from others and contribute to the development of open-source projects. This enables developers to expand their knowledge and skills, learn new programming languages, and work on projects that have a real-world impact.
- **Reputation:** GitHub provides a platform for developers to showcase their work and build a reputation in the software industry. This can lead to job opportunities, networking opportunities, and other benefits.

5. Docker:

Developers may deploy, create, and execute apps in a lightweight container using the open-source technology known as Docker. Applications may now be packaged and operate in separate containers thanks to technology, enabling consistent performance across various settings.

Advantages of Docker:

- One of Docker's key advantages is that it offers a mechanism to guarantee that programmes will operate uniformly across various settings, making it simpler to create and deploy apps. With Docker containers, resources can be used more effectively because different containers can coexist peacefully on the same machine. This can result in substantial cost savings, especially for businesses that need to quickly grow their applications.
- Additional advantages of Docker include improved security, streamlined deployment, and quicker application delivery. Instead of taking days or weeks to create, test, and deploy an application, Docker makes it possible to do so quickly. In order to keep ahead of the competition, this can help organizations adapt to changing business requirements more swiftly.

Many different businesses use Docker for a range of functions. Running microservices, which are compact, independent components that can be deployed and scaled separately, is one such use case. The agility, flexibility, and scalability of an organisation may be increased by running microservices in Docker containers.

Cloud-native apps, which are created expressly for cloud settings, are likewise built and deployed using Docker. These applications may be deployed and maintained using Docker and are often created using a mix of microservices, containers, and other cloud-native technologies.

To sum up, Docker is a potent tool that businesses can use to develop, test, and deploy apps more rapidly and effectively. Docker makes it simpler to create and deploy programmes reliably across many settings by allowing developers to bundle and execute apps in lightweight containers. Docker is a crucial tool for any organisation wanting to stay competitive in the fast-paced business world of today because of its wide variety of advantages and use cases.

6. **PostgreSQL:**

Relational database management system (RDBMS) PostgreSQL is a potent, free software option that offers cutting-edge capabilities, security, and scalability. For businesses, developers, and academics that need a reliable database management system for storing, managing, and analysing complicated data, it is the perfect option.

Features and Capabilities:

PostgreSQL is a fantastic option for a variety of applications since it has a wide range of features and capabilities, including:

- **Data Integrity:** By imposing constraints, rules, and triggers, PostgreSQL offers strong data integrity. It supports check

constraints, exclusion constraints, unique keys, foreign keys, main keys, and foreign keys.

- PostgreSQL has sophisticated indexing capabilities, such as B-tree, Hash, GiST, SP-GiST, GIN, and BRIN indexes. These indexes facilitate quick data retrieval and enhance query performance.
- Possibilities for extensions include PostGIS, hstore, pl/python, and pl/pgsql. PostgreSQL is quite expandable. With the help of these extensions, PostgreSQL may be enhanced and tailored to developers' needs.
- High Availability: Through features like streaming replication, hot standby, and logical replication, PostgreSQL provides high Availability. These characteristics guarantee that data is accessible regardless of a hardware or software malfunction.
- Role-based access control, SSL encryption, and Kerberos authentication are just a few of the strong security features that PostgreSQL offers. Data is shielded from unauthorised access and security breaches thanks to these features.

Advantages:

- Open-Source: PostgreSQL is an open-source RDBMS, making it available for usage, modification, and distribution without charge. Because of this, it is a great option for developers and businesses who wish to avoid paying high licencing fees.

- Operating platforms such as Windows, Linux, and macOS are all compatible with PostgreSQL. Additionally, it supports a number of programming languages, such as PHP, Python, and Java.
- Scalability: PostgreSQL has a great degree of scalability and can manage massive amounts of data. Sharding and clustering are supported, enabling developers to spread data over a number of servers.
- Support from the community: PostgreSQL has a sizable and vibrant developer community that offers assistance, shares information, and contributes to the growth of the database.

Thus an effective and open-source RDBMS like PostgreSQL provides cutting-edge functionality, security, and scalability. For developers, businesses, and academics that need a reliable database management system for storing, managing, and analysing complicated data, it is the perfect option. PostgreSQL can handle a multitude of applications and use cases because of its extensive set of features and capabilities. For businesses looking for a dependable and affordable database management system, its open-source nature, interoperability, scalability, and community support make it a great option.

7. **Mockito:**

For testing Java-based apps, a popular open-source framework called Mockito is utilised. It offers a straightforward and understandable API for building mock objects and simulating their behaviour. The framework is

made to make it easier to test code that relies on other elements or services. Developers can separate the code being tested from its dependencies and concentrate on evaluating the specific functionality of the code by generating fake objects.

Several testing frameworks, such as JUnit and TestNG, can be used in conjunction with Mockito. It offers a collection of verification techniques to claim that particular interactions have taken place between the code under test and its dependencies and enables developers to easily mimic interfaces, abstract classes, and concrete classes.

Mockito encourages test-driven development (TDD) and enables developers to create test cases for their code as they write it, which is one of the main advantages of adopting it. As problems can be discovered early in the development process, this can result in more dependable and robust code.

The generation of spies, which are objects that wrap around actual objects and may be used to check the behaviour of the object being spied on, is another feature supported by Mockito. Testing dated or challenging-to-modify code might benefit from this.

Overall, Mockito is a strong and adaptable testing framework that enables developers to test certain functionality independently from the rest of the code, which may help them design more dependable and effective code.

8. IntelliJ IDEA:

To increase developer productivity, IntelliJ IDEA is an Integrated Development Environment (IDE) for JVM languages. By offering sophisticated code completion, static code analysis, and refactorings, it takes care of the tedious and repetitive duties for you and frees you up to concentrate on the positive aspects of software development, making it both efficient and pleasant. Cross-platform IDE IntelliJ IDEA offers a uniform user experience on Windows, macOS, and Linux.

Modern application development requires the use of a variety of languages, tools, frameworks, and technologies. Although IntelliJ IDEA is intended to be an IDE for JVM languages, it can be expanded with a variety of plugins to offer a polyglot experience.

3.3.2 Hardware Requirements

- Ram: 4GB or higher,
- Storage: 500GB,
- CPU: 2GHz or faster, and
- Architecture: 32Bit or 64Bit

It is essential to keep in mind that these parameters may alter depending on the application's specific requirements and projected demand. It is suggested to regularly check system performance and make relevant hardware configuration modifications. The hardware requirements may also significantly increase for bigger apps with heavy workloads and several concurrent users accessing the API.

3.4 Implementation & Deployment

As per system design provided in figure 3.1, we firstly need to download docker on our system. Then we pull an instance of PostgreSQL and run it on a docker container. We will be using this because at the time of deployment it will save us from setting up postgresql on a cloud system and saving our time.

We build our RESTful API using spring boot and build all the required endpoints. For the documentation of our API we use swaggerUI. After we are done building and testing our RESTful API we now only need to test it and at last we also deployed our API on AWS EC2 instance. For deployment, we first dockerized our application and pushed the image to “dockerHub”. Now we connected to our EC2 instance and pulled images of our application and postgresql images from docker hub. Then we run the containers and now anyone can access the deployed API from the public IP address available.

3.5 Database Schema

We used PostgreSQL as our database for our project.

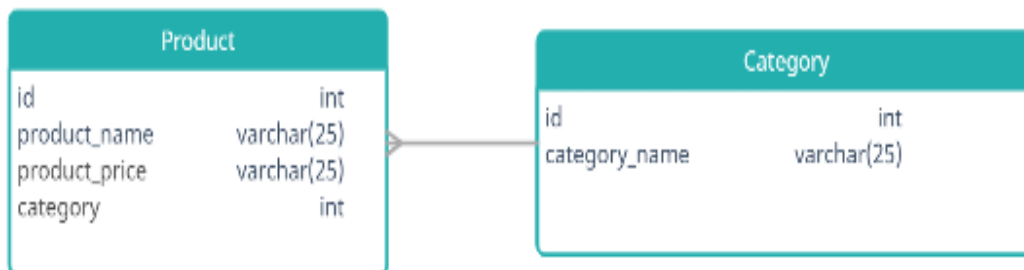


Figure 3.2: Database Schema (Many to One)

The Category table and the Product table are the two primary tables that make up the database structure for the API. The Category table includes details about each

category in the system, including the name and ID. The Product database holds data about each and every item in the system, including the product name, price, and category ID.

The Category and Product tables have a many-to-one connection, which means that while each category may include many items, each product may only belong to one category. Our tables have a many to one relationship. This is due to the fact that a category might either contain numerous goods or many products in a category. By including a foreign key column in the Product table that refers to the ID column in the Category database, this link is formed.

To guarantee data consistency and integrity, the schema also contains a number of constraints. For instance, the foreign key restriction prevents the addition of a product to a category that does not exist. Unique constraints are also included in the schema to guarantee that no two categories or items share the same name or ID.

Overall, the database architecture for the API is created to store and retrieve data about items and categories in an effective and efficient manner. The schema serves as a solid and dependable basis for the API since it is scalable, flexible, and easily adaptable to modifications and additions to the data model.

CHAPTER-4

EXPERIMENTS & RESULT ANALYSIS

4.1 Overview

The API's many endpoints for data retrieval and manipulation from the database were successfully developed. At every level of development, thorough testing was done to assure the API's dependability and usefulness. Even when dealing with a lot of data, the API continuously responded quickly. To minimise performance overhead, appropriate logging was provided to track the behaviour of the API, and the logging level was configured to capture only pertinent data.

The API was created using RESTful design principles, making it easy for developers to use and comprehend. In the event that there are any failures or exceptions when the API is being used, proper error handling has been put in place to deliver instructive error messages. To retain the code's flexibility and scalability, the API was created using a three-layer design that separates the presentation layer from the service layer from the data access layer.

The API had low hardware requirements, which made it simple to install on a variety of systems. By caching commonly used data and reducing the amount of database queries made, the API's speed was improved.

Overall, the API's deliverables met the project's objectives and exceeded expectations in terms of functionality and speed. The API may be tailored to match the particular requirements of diverse projects, and it has a lot of potential for usage in a variety of applications.

4.2 Experiment Results

The Spring-Boot application's API has been successfully tested and implemented. It has APIs for retrieving all goods, all categories, and all products in a particular category. After gradually adding and testing the stubs for these APIs, services, DAOs, and a PostgreSQL database were added.

To make sure the API performs as expected, test cases for controllers, services, and DAOs were created and run. To guarantee that any problems or errors are quickly found and fixed, appropriate logging was employed with appropriate log levels.

The response content may be modified as necessary, and the API was created to interact with request and response headers. To make sure the client receives the correct status code, a response code can also be specified.

In order to build and update goods, more APIs were introduced. To make sure they work as intended, these APIs underwent extensive testing as well. So, the Spring-Boot application's API has been successfully implemented and thoroughly tested. It offers the required tools for locating and modifying data about items and categories.

4.3 Outputs at Various Stages

In this section, we will be going through the outputs of our spring boot RESTful API for different types of input or requests from the user. We will also see the results for user requests where our RESTful API gives responses with perfect exceptions. Keeping in mind all the possible cases from a user, we perfectly built our API for handling all exceptions.

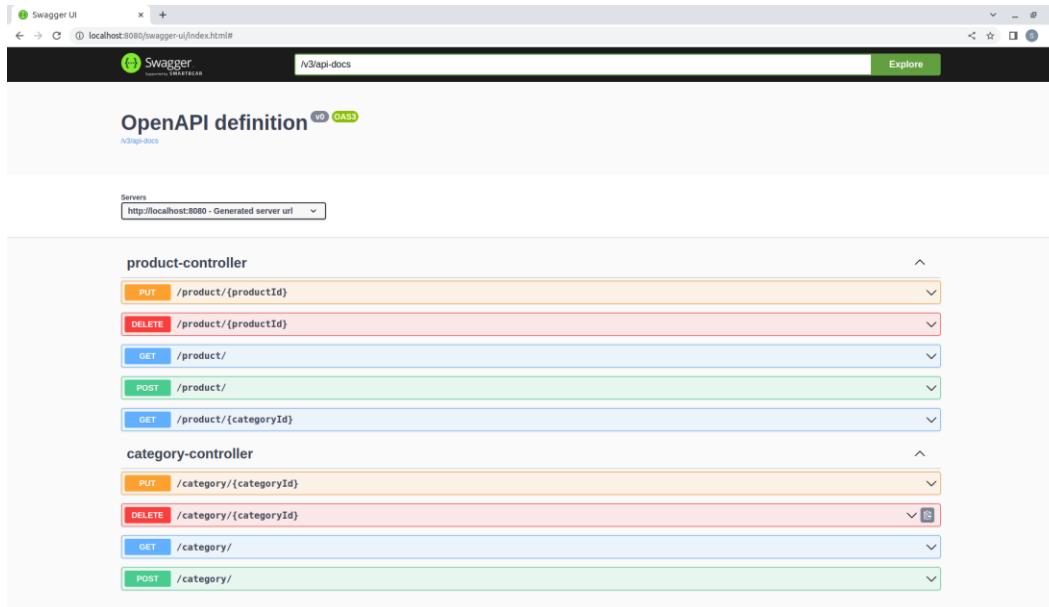


Figure 4.1: SwaggerUI for API

Developers may interact with APIs and see their structure using the open-source programme SwaggerUI. It is a web-based graphical user interface that gives a framework for testing and documenting RESTful APIs. By providing interactive API documentation that is simple to share with team members or incorporate into other apps, SwaggerUI makes API testing and development easier.

SwaggerUI is growing in popularity among API developers thanks to its simple interface and robust functionality. It enables programmers to quickly prototype and test their APIs to make sure they adhere to the required standards. Additionally, SwaggerUI offers a number of tools that assist programmers in locating and resolving problems, making it a crucial tool for creating reliable, high-quality APIs. It is more user friendly as compared to sending requests from the POSTMAN app.

```
Schemas

ProductDto {
  productName* string
  productPrice integer($int32)
  categoryId* integer($int32)
}

Category {
  categoryId integer($int32)
  categoryName string
}

Product {
  productId integer($int32)
  productName string
  productPrice integer($int32)
  category Category {
    categoryId integer($int32)
    categoryName string
  }
}

CategoryDto {
  categoryName* string
}
```

Figure 4.2: Schemas in API

Here, we used DTO for sending requests. Data transfer objects, or DTOs, are a crucial component of any modern API architecture. They act as a means of exchanging information between the controller, service, and data access levels, among other application layers. We will examine the use of DTOs in our implementation of the API in this report.

An API's maintainability, scalability, and performance may all be enhanced by using DTOs. We can make sure that the API is adaptive and flexible by isolating the data transmission from the business logic. This is crucial in large-scale applications because adjustments made to one component of the system may have an impact on the whole programme.

DTOs have been utilized to encapsulate the data being transmitted between the application's various tiers in our API implementation. This enables us to

guarantee that only essential data is delivered, enhancing efficiency and lowering the risk of data leakage or security flaws.

DTOs also assist us in making sure that the data being transported is consistent and complies with a certain contract or standard. When integrating with external systems, working with multiple teams, or using microservices, this is especially crucial. Conflicts may be avoided and we can make sure the API is still usable and interoperable by specifying a clear contract for the data transmission.

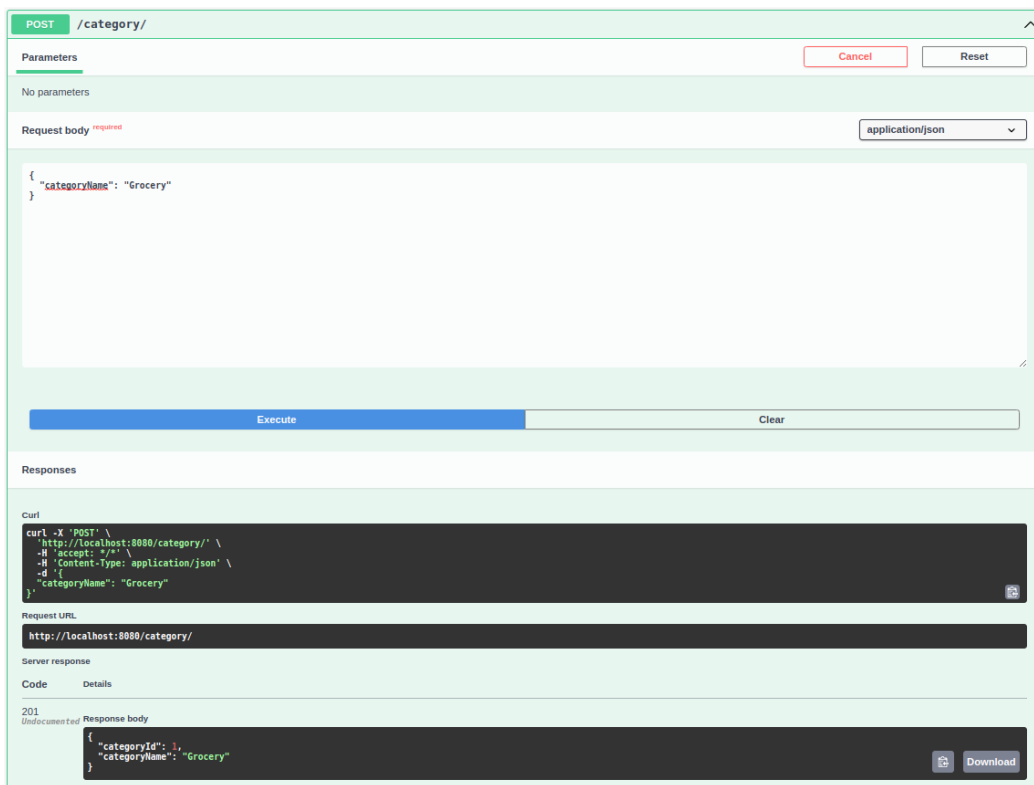


Figure 4.3: POST request for Category

In figure 4.3, we can see that we ran a POST request from our swagger API. As we know, POST requests create or enters new data into the database. Here, this request is used for the purpose of creating a new category in our database. We can

see that, when the request is executed then we get a response body with the data created and a response code, i.e, 201 (for Creation Success). From this response we confirm that we successfully created a new category from the POST request just by using the swaggerUI.

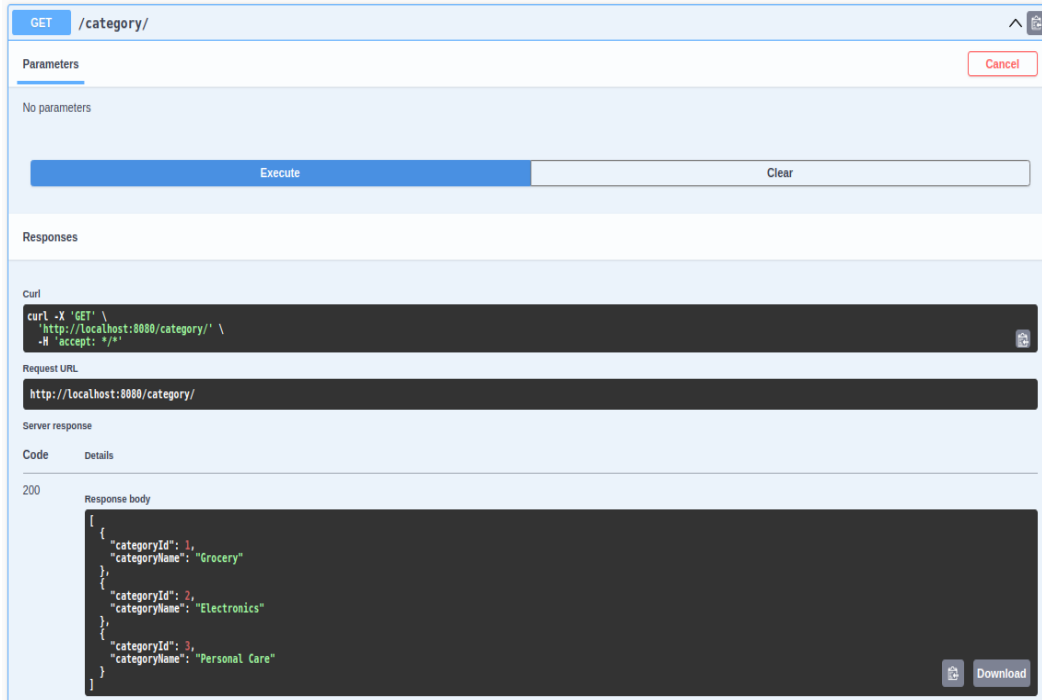


Figure 4.4: GET request for Category

In figure 4.4, we can see that we ran a GET request from our swagger API. As we know that GET requests fetches all the data from the database and returns it to the user as a response. Here, this request is used for the purpose of fetching all the categories in our database. We can see that, when the request is executed then we get a response body with the data available and a response code, i.e, 200 (for OK). From this response we confirm that we successfully fetched data from the database by using the GET request just by using the swaggerUI.

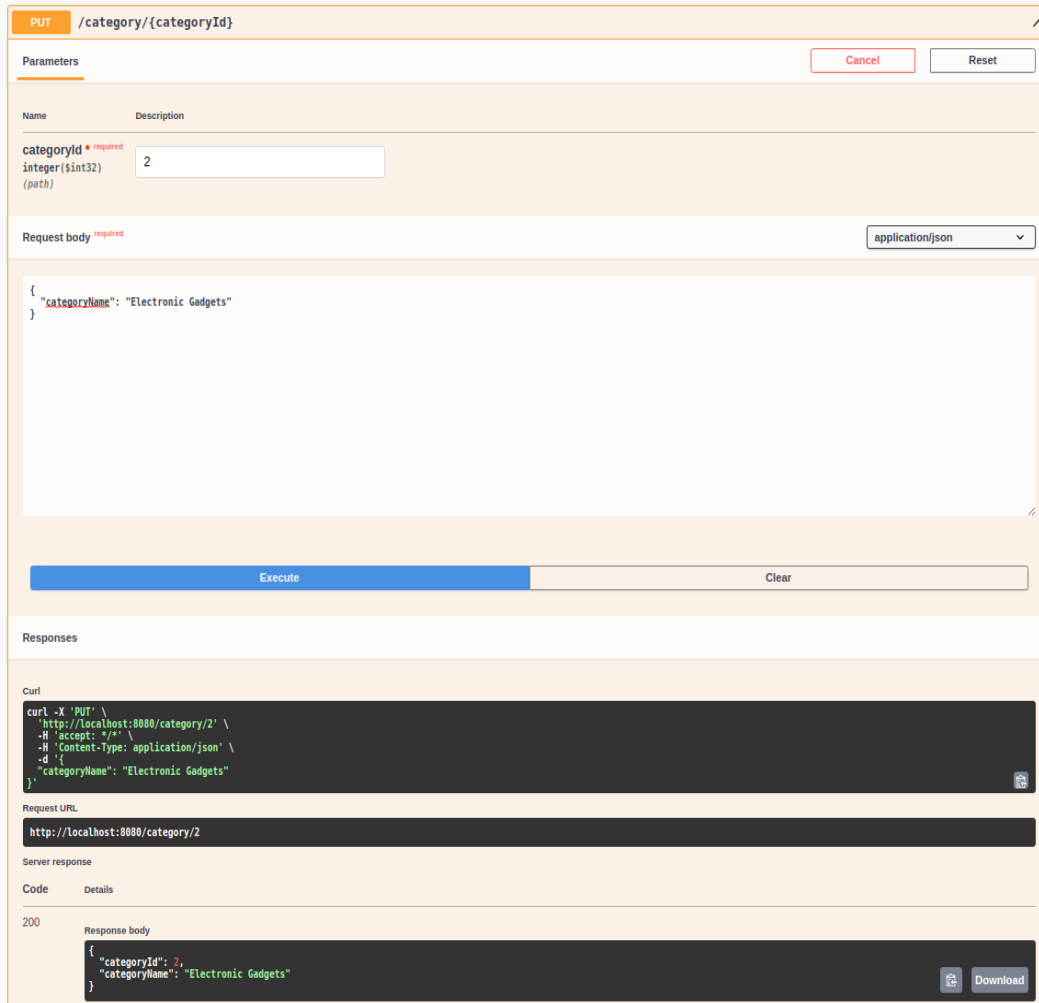


Figure 4.5: PUT request for Category

In figure 4.5, we can see that we ran a PUT request from our swagger API. As we know that PUT requests are used to update a record in the database. Here, this request is used for the purpose of updating category name in our database. It takes the id of the category as a path parameter. We can see that, when the request is executed then we get a response body with the data updated and a response code, i.e, 200 (for OK). From this response we confirm that we successfully updated a record database by using the PUT request just by using the swaggerUI.

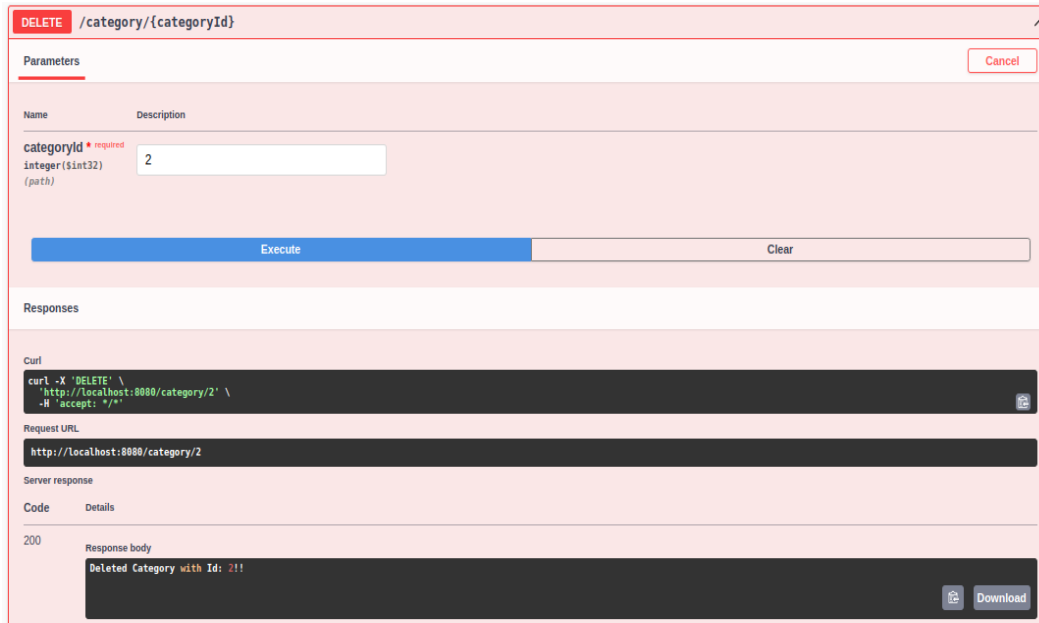


Figure 4.6: DELETE request for Category

In figure 4.6, we can see that we ran a DELETE request from our swagger API. As we know that DELETE requests are used to delete a record in the database. Here, this request is used for the purpose of deleting a category from our database. It takes the id of the category as a path parameter. We can see that, when the request is executed then we get a response body with the deleted message and a response code, i.e, 200 (for OK).

Now, as we have seen some normal inputs or normal response from our API for all the cases, so now lets see how our API behaves in abnormal conditions. That is, now we will be checking our RESTful API for abnormal inputs from users and test the exception handling for categories.

We will test our API for conditions like creating a category with name value as null or empty. or try updating or creating a category with name that already exists in database.

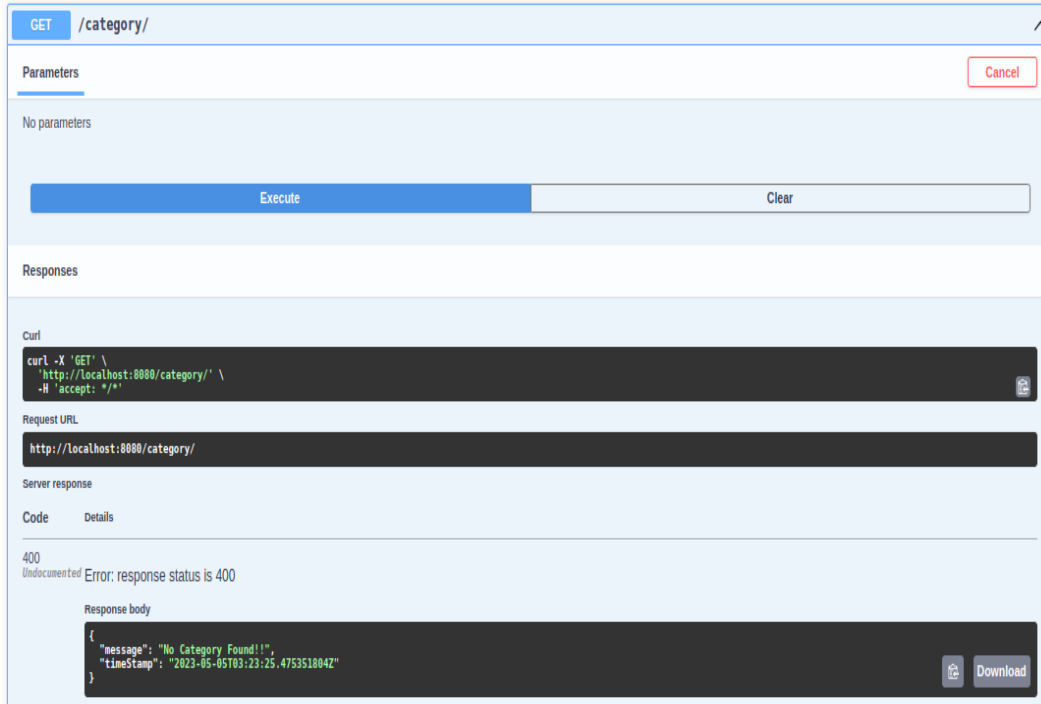


Figure 4.7: GET request for Category when empty

In figure 4.7, we can see that we ran a GET request from our swagger API. As we know that GET requests fetches all the data from the database and returns it to the user as a response. Here, this request is used for the purpose of fetching all the categories in our database. We can see that, when the request is executed then we get an error message in the response body with a response code, i.e, 400 (for Error). This happened because in the database, there were no data present in the category table.

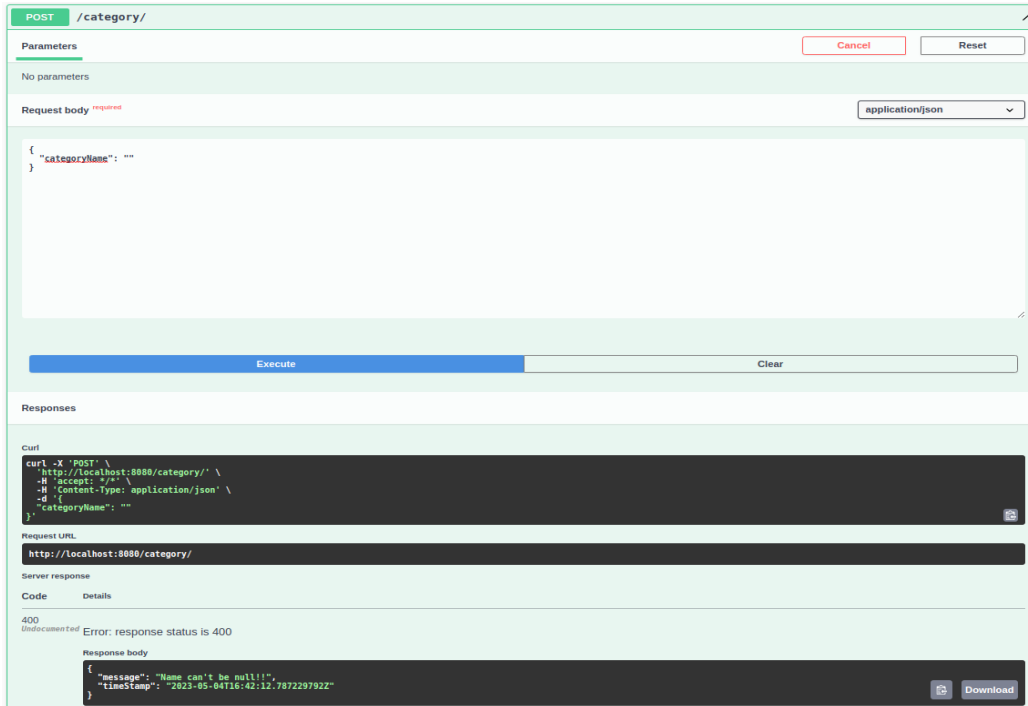


Figure 4.8: POST with null category name

In figure 4.8, we can see that we ran a POST request from our swagger API with an empty category name but we got a response body with the error message and a response code, i.e, 400 (for ERROR).

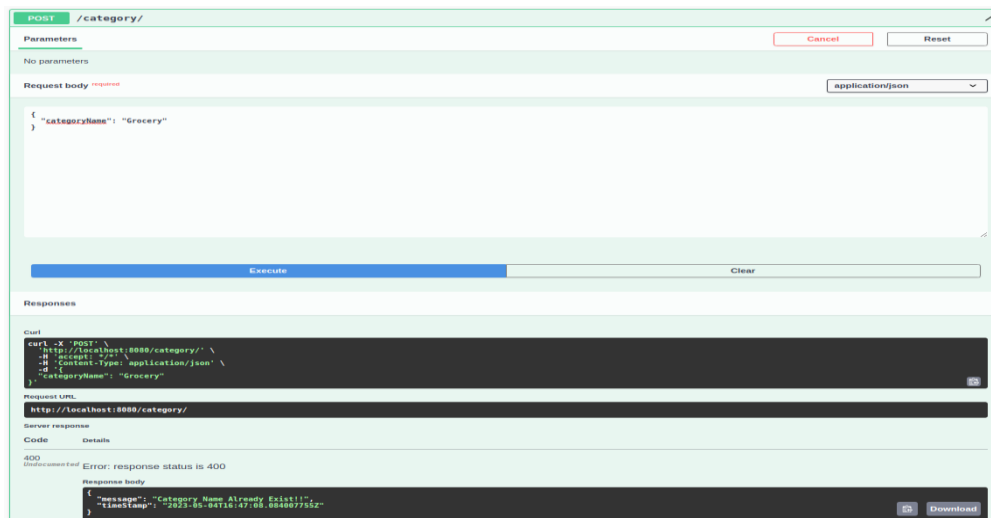


Figure 4.9: POST with category name already present

In figure 4.9, we can see that we ran a POST request from our swagger API with a category name that already exists in the database, but we get a response body with the error message and a response code i.e, 400 (for ERROR).

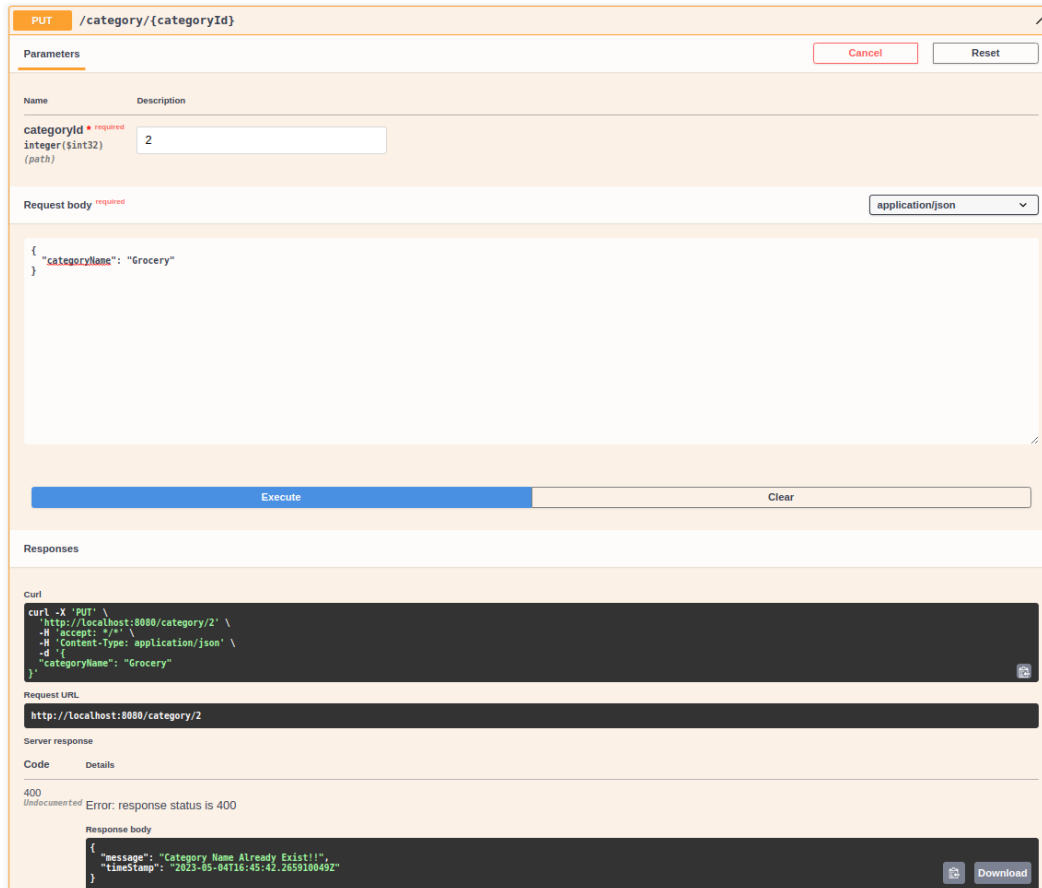


Figure 4.10: PUT request for Category when name already exists

In figure 4.10, we can see that we ran a PUT request from our swagger API. As we know that PUT requests are used to update a record in the database. Here, this request is used for the purpose of updating category name in our database. It takes the id of the category as a path parameter. We can see that, when the request is executed then we get a response body with the proper error message and a

response code, i.e, 400 (for Error). This is because there is already a category present with the given name.

After we have created some categories in our postgresql database, now we are ready to create some products for different types of categories that are available.

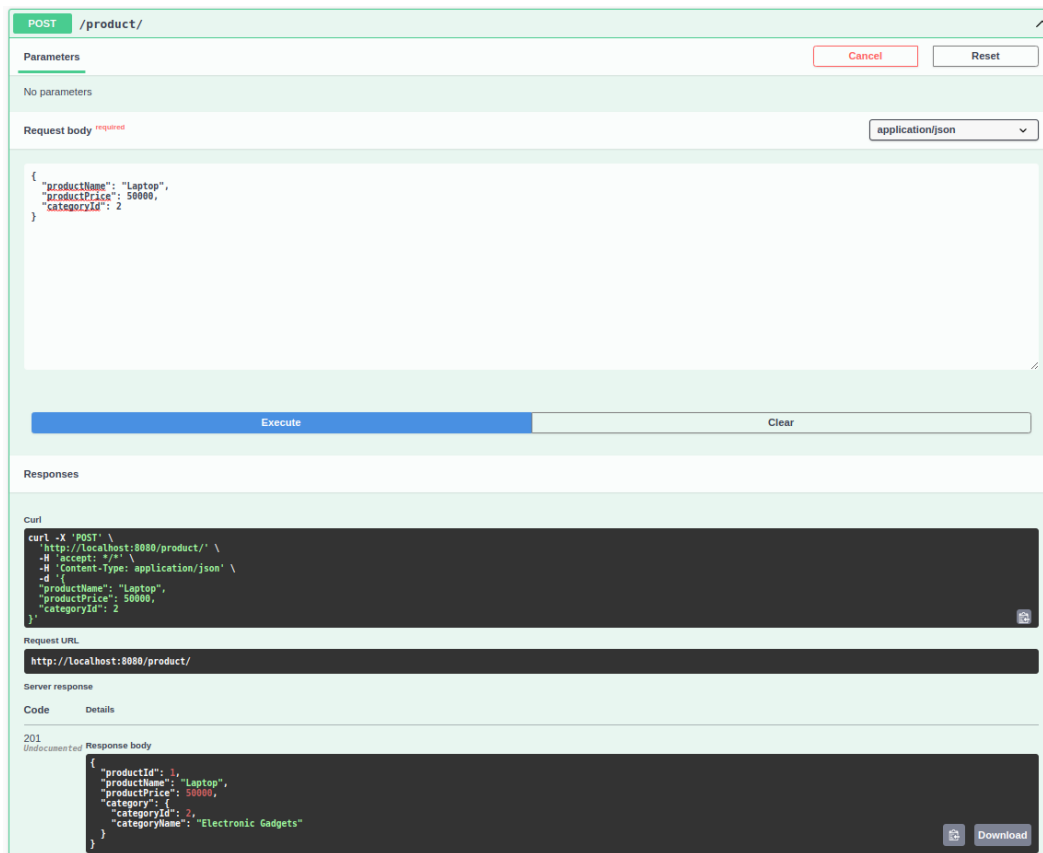


Figure 4.11: POST request for Product

In figure 4.11, we can see that we ran a POST request from our swagger API. Here, this request is used for the purpose of creating a new product in our database with product name, price and category id or type as request body from the user. We can see that, when the request is executed then we get a response body with the data created and a response code, i.e, 201 (for Creation Success).

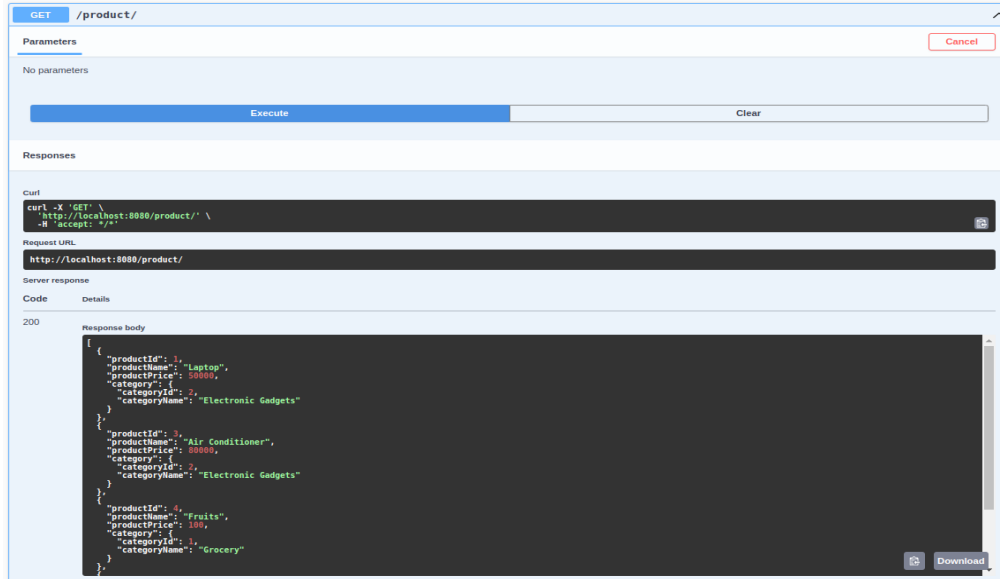


Figure 4.12: GET request for Product

In figure 4.12, we ran a GET request from our swagger API. This request is used for the purpose of fetching all the products in our database. We get a response body with the data fetched and a response code i.e, 200 (for OK).

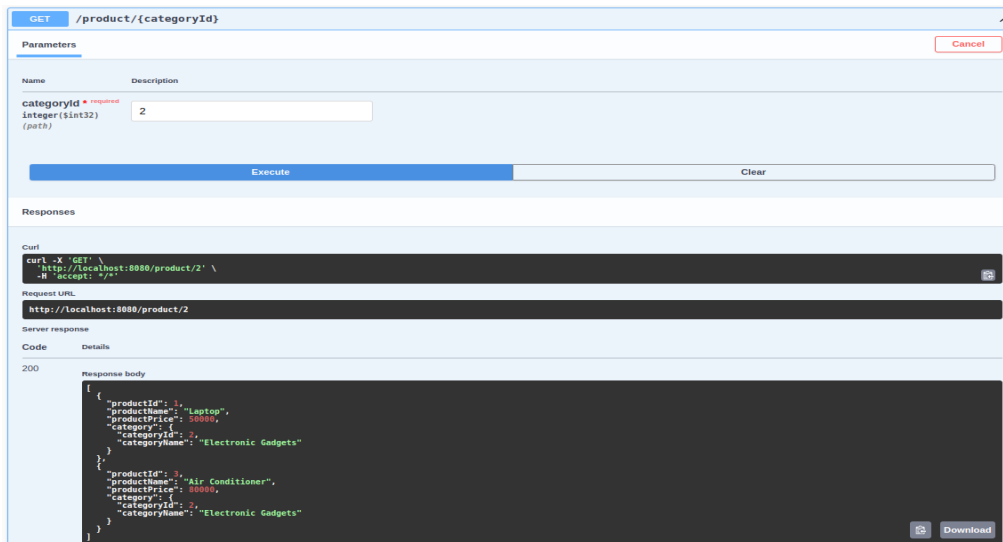


Figure 4.13: GET request for products of a particular category

In figure 4.13, we can see that we ran a GET request from our swagger API. Here, this request is used for the purpose of fetching all the products of a particular category (category id is taken as path parameter) in our database. We can see that, when the request is executed then we get a response body with the data fetched and a response code, i.e, 200 (for OK).

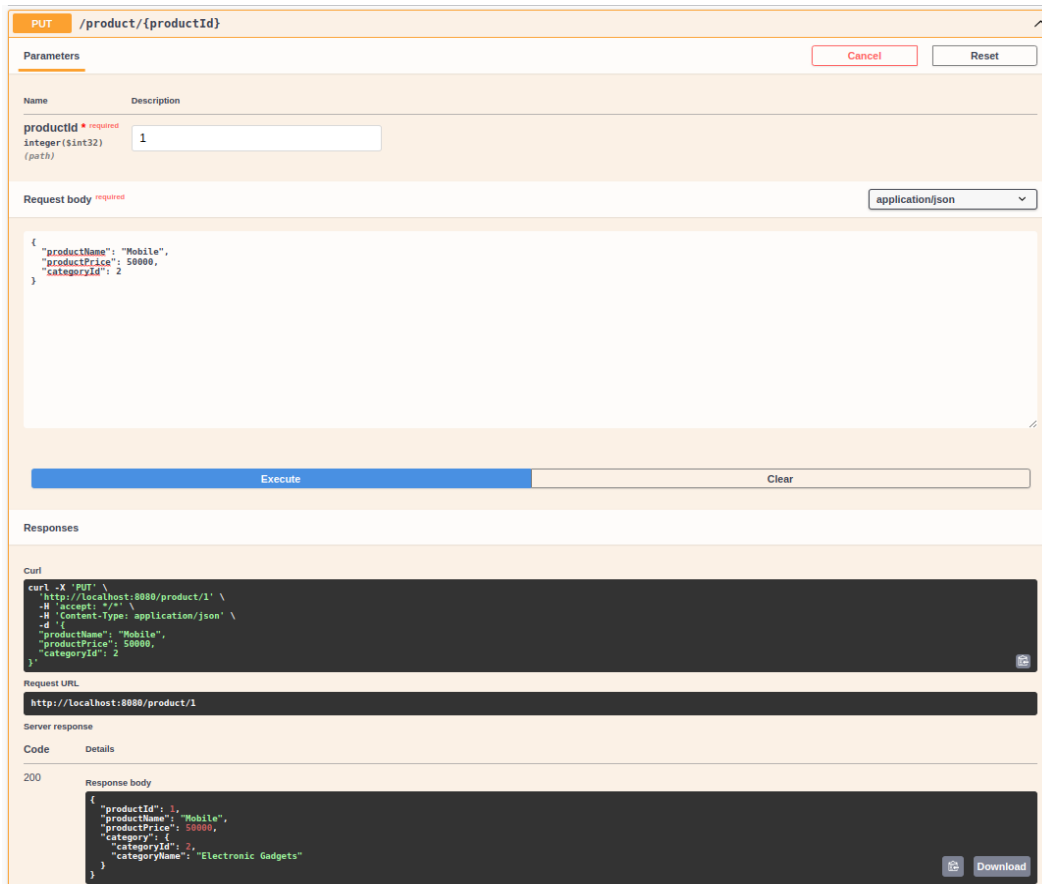


Figure 4.14: PUT request for Product

In figure 4.14, we ran a PUT request from our swagger API. This request is used for the purpose of updating a product in our database. It takes the id of the product to be updated as a path parameter. We can see that, when the request is executed then we get a response body with the data updated and a response code, i.e, 200

(for OK). From this response we confirm that we successfully updated a record database by using the PUT request just by using the swaggerUI.

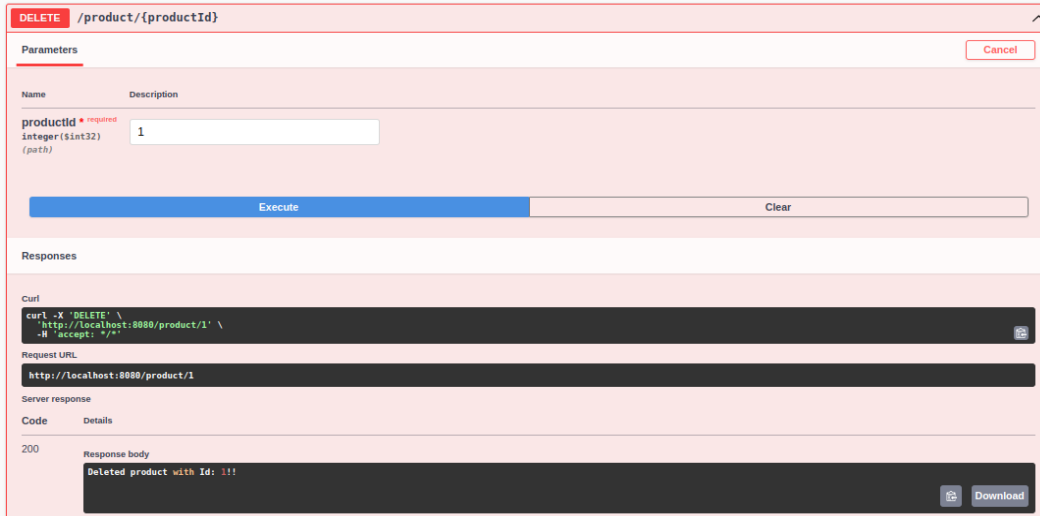


Figure 4.15: DELETE request for Product

In figure 4.15, we ran a DELETE request from our swagger API. This request is used for the purpose of deleting a product from our database. It takes the id of the product to be deleted as a path parameter. When the request is executed, we get a response body with the deleted message and a response code, i.e, 200 (for OK).

If we also delete any category, then all the products in that category also gets deleted due to cascading property that has been applied.

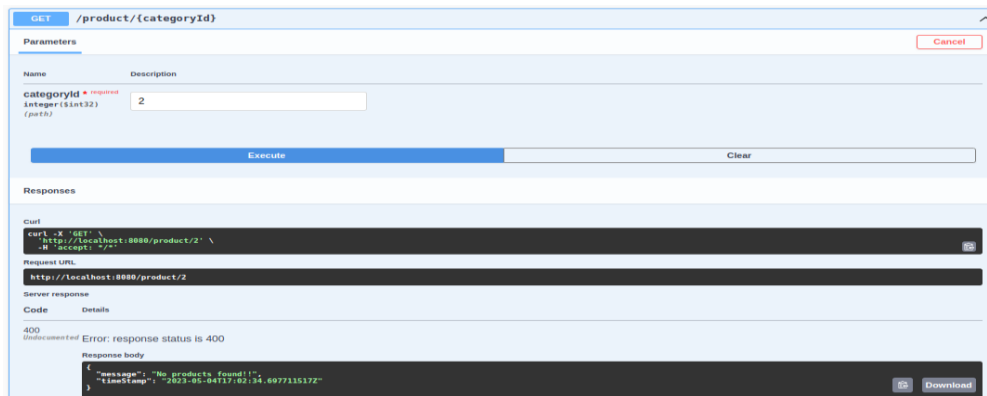


Figure 4.16: GET request for Product when category with Id 2 is deleted

In figure 4.16, we ran a GET request from our swagger API. Here, this request is used for the purpose of fetching all the products of a particular category (category id is taken as path parameter), but the category is already deleted from database. But we can see that, when the request is executed then we get a response body with a message that no product is available and a response code, i.e, 400 (for ERROR).

Now, as we have seen some normal inputs or normal response from our API for products, for all the cases, so now lets see how our API behaves in abnormal conditions. That is, now we will be checking our RESTful API for abnormal inputs from users and test the exception handling for products.

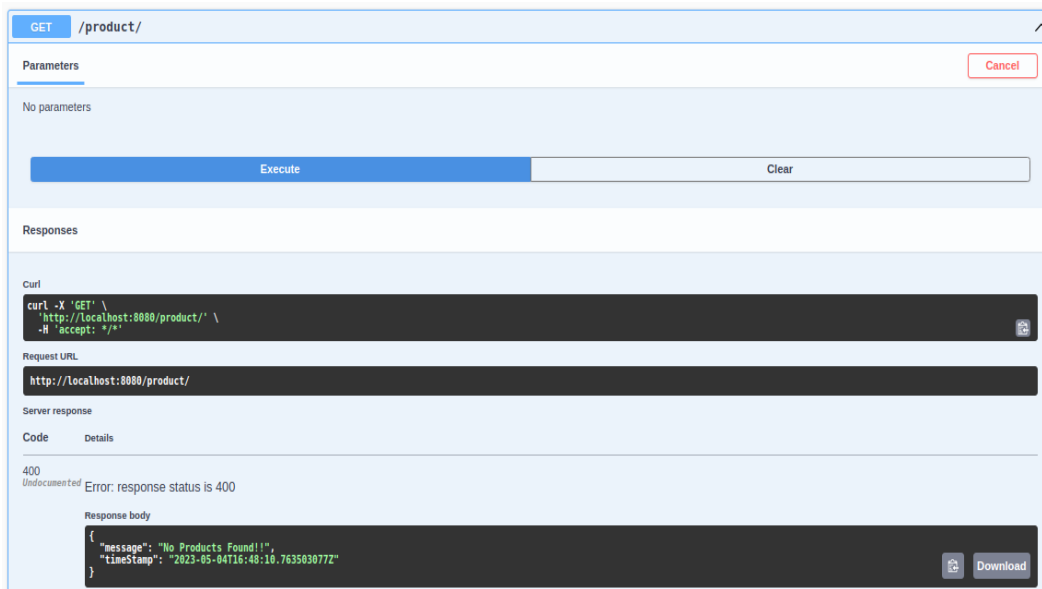


Figure 4.17: GET request for Product when no products exists

In figure 4.17, we ran a GET request from our swagger API. Here, this request is used for the purpose of fetching all the products but there are no products available in our database. So we can see that, when the request is executed then

we get a response body with a message that no product is available and a response code, i.e, 400 (for ERROR).

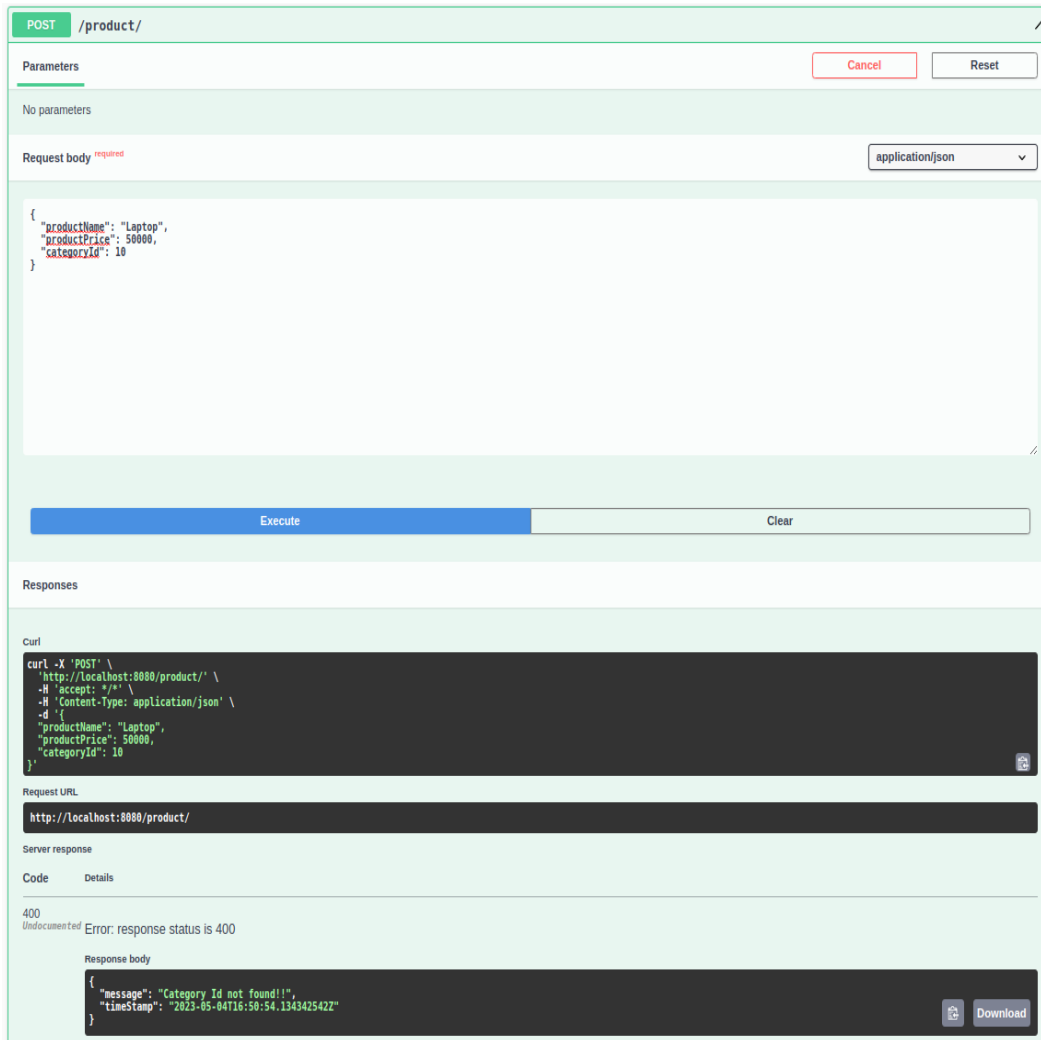


Figure 4.18: POST request for Product with wrong Category Id

In figure 4.18, we can see that we ran a POST request from our swagger API. This request is used for the purpose of creating a new product in our database with product name, price and category id or type as request body from the user. But we can see that, when the request is executed then we get an error message with the data created and a response code, i.e, 400 (for Error) because there is no category available with the given Id.

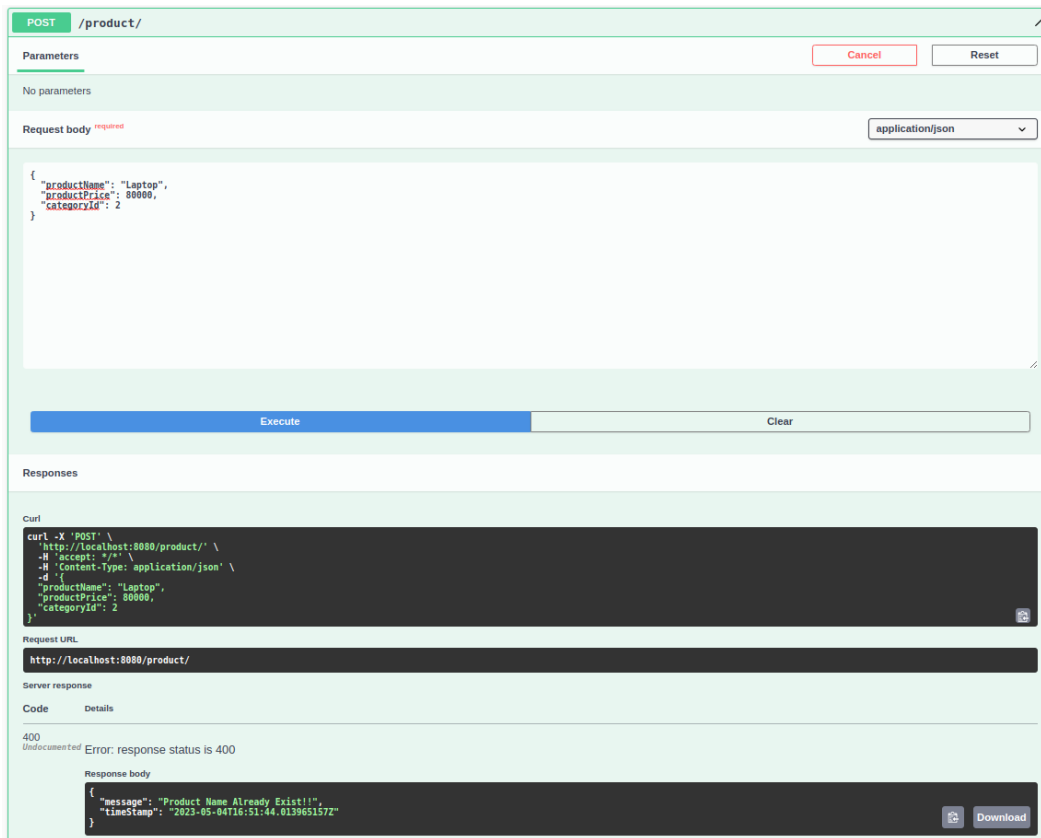


Figure 4.19: POST request for Product with name that already exists

In figure 4.19, we can see that we ran a POST request from our swagger API. This request is used for the purpose of creating a new product in our database with product name, price and category id or type as request body from the user. But we can see that, when the request is executed then we get an error message with the data created and a response code, i.e, 400 (for Error) because the product with the provided name already exists in our database.

Similarly, it handles the exception when we try to create a product without giving a name. It shows the required message. All these exceptions are also handled in PUT request also, to have a proper validation on data being entered by the user

and to remove or avoid insertion of garbage data from our database which may lead to various errors.

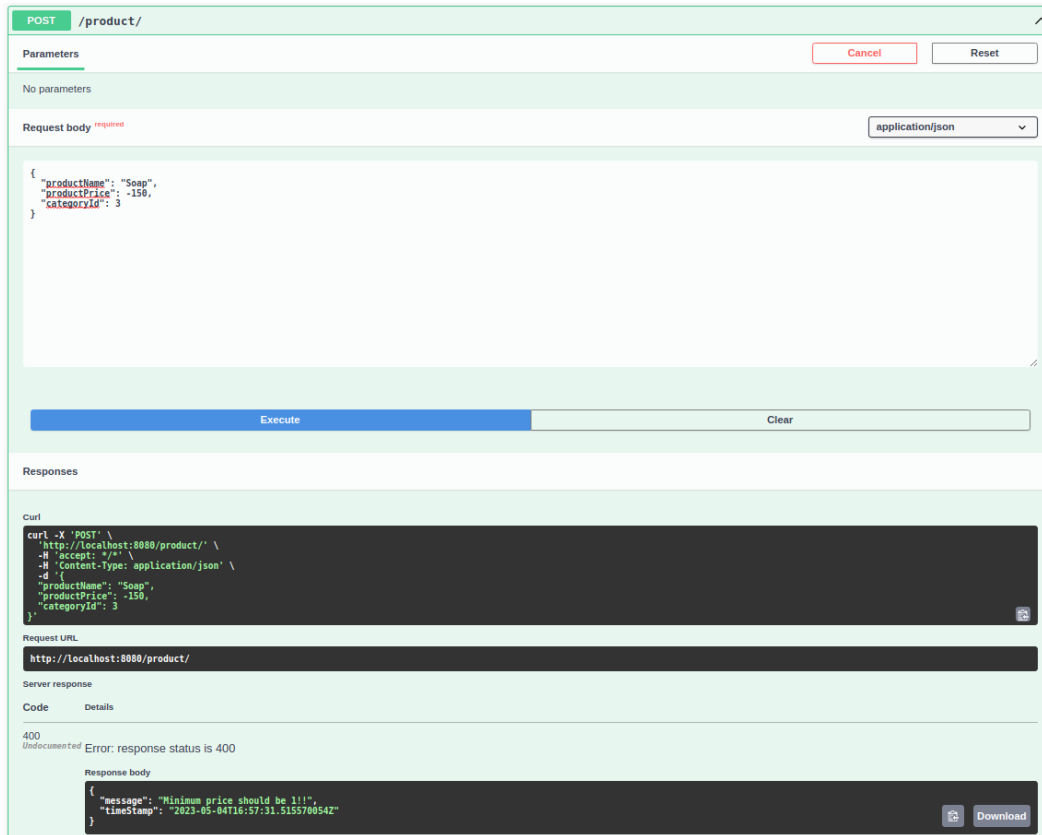
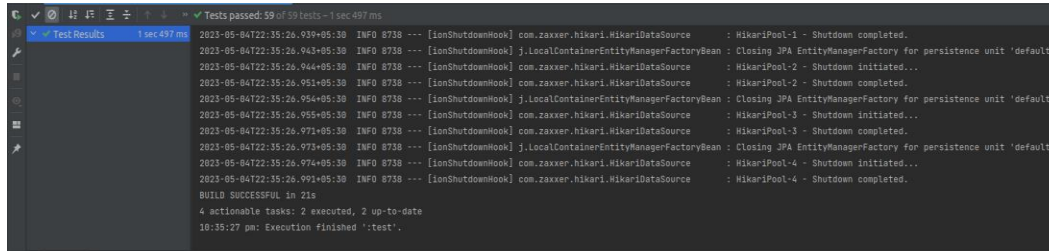


Figure 4.20: POST request for Product with negative price

In figure 4.20, we can see that we ran a POST request from our swagger API. This request is used for the purpose of creating a new product in our database with product name, price and category id or type as request body from the user. But we can see that, when the request is executed then we get an error message with the data created and a response code, i.e, 400 (for Error) because the user tried to create a product with a price having a negative value.

There are more validations added to our API to remove the insertion or updation in our database that may lead to several errors in future.

4.4 Performance Analysis



```
Tests passed: 59 of 59 tests - 1.97 sec
Test Results 1.97 sec
2023-05-04T22:35:26.939+05:30 INFO 8738 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Shutdown completed.
2023-05-04T22:35:26.943+05:30 INFO 8738 --- [ionShutdownHook] j.LocalContainerEntityManagerFactoryBean : Closing JPA EntityManagerFactory for persistence unit 'default'
2023-05-04T22:35:26.944+05:30 INFO 8738 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource : HikariPool-2 - Shutdown initiated...
2023-05-04T22:35:26.951+05:30 INFO 8738 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource : HikariPool-2 - Shutdown completed.
2023-05-04T22:35:26.954+05:30 INFO 8738 --- [ionShutdownHook] j.LocalContainerEntityManagerFactoryBean : Closing JPA EntityManagerFactory for persistence unit 'default'
2023-05-04T22:35:26.955+05:30 INFO 8738 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource : HikariPool-3 - Shutdown initiated...
2023-05-04T22:35:26.971+05:30 INFO 8738 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource : HikariPool-3 - Shutdown completed.
2023-05-04T22:35:26.974+05:30 INFO 8738 --- [ionShutdownHook] j.LocalContainerEntityManagerFactoryBean : Closing JPA EntityManagerFactory for persistence unit 'default'
2023-05-04T22:35:26.974+05:30 INFO 8738 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource : HikariPool-4 - Shutdown initiated...
2023-05-04T22:35:26.991+05:30 INFO 8738 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource : HikariPool-4 - Shutdown completed.
BUILD SUCCESSFUL in 21s
4 actionable tasks: 2 executed, 2 up-to-date
10:35:27 pm: Execution finished 'test'.
```

Figure 4.21: Unit Testing

Performed unit test coverage and found that all 59 tests written for different cases and scenarios, ran successfully i.e PASSED. I used Mockito and JUnit5 to write test cases.

CHAPTER-5

CONCLUSIONS

5.1 Conclusion

The Spring Boot-based REST API that we created for this project has many endpoints that serve for different purposes. All categories, all products, and all items that fall within a certain category may all be fetched using the API. Three layers—a controller layer, a service layer, and a DAO layer—make up the architecture of our API. The DAO layer interacts with the database to get and save data while the service layer performs the business logic. To make sure the API works properly, we have also created test cases for the controller, service, and DAO levels.

In conclusion, we have successfully used Spring Boot to construct a reliable and scalable REST API. The codebase is simple to maintain and change as necessary thanks to the 3-layer design. We have been able to store vast volumes of data effectively and retrieve it rapidly thanks to our usage of the PostgreSQL database. The API is universal and can be simply deployed on any platform thanks to the usage of Docker and PostgreSQL.

This API's flexibility is one of its most important benefits. It may be used to create a variety of applications, including inventory management systems and e-commerce platforms. It is simple to create specialised interfaces for various applications since it is possible to retrieve all categories, all goods, and all items that fall under a certain category.

Another useful element of the API is the logging capability. It enables tracking and debugging of mistakes, which is crucial when dealing with complex

programmes. The logging function aids in locating any potential performance bottlenecks in the codebase.

It is now simpler to interact with the API thanks to the introduction of Request and Response Headers. Developers can transmit crucial data, including authentication tokens, in the headers, aiding in the API's security. Another essential component of creating a strong API is the capability to establish a response code. Developers can use it to let the client application know whether or not the request was successful.

A product creation and update API has also been implemented, enhancing the API's capability. Developers may now create and update goods programmatically, which will save them time and effort.

In conclusion, our Spring Boot-based REST API provides a flexible, scalable, and reliable tool for creating a variety of apps. It is a great option for developers due to its three-layer design, use of a PostgreSQL database, test cases, logging functionality, Request and Response Headers, and capacity to build and update products. We are sure that this API will assist programmers in creating top-notch apps that satisfy their business needs.

5.2 Future Scope

The RESTful API developed in this project has a lot of future enhancements that can improve its performance, security, and user-friendliness. By putting these improvements into place, the API can help organizations and developers accomplish their objectives quickly and effectively.

The security of the API will be improved by the implementation of authentication and permission procedures. While authorization will specify the degree of access that each user has to the API, authentication will make sure that only authorised users can access it.

Another crucial aspect is caching, which may enhance the efficiency of the API by lowering the amount of queries made to the database. Response times will be quicker and scalability will be improved with the use of a caching mechanism.

When working with large datasets, it's crucial to have the ability to filter and paginate the data. Clients will be able to access exactly the data they require by using the API's filtering and pagination capabilities, which will decrease network traffic and boost performance.

For developers that wish to utilize the API, clear and thorough documentation is crucial. Developers will benefit from clear documentation that explains how to use the API, what data is accessible, and what it can and cannot accomplish.

A method of restricting the volume of traffic to the API is rate limitation. It can assist in preventing misuse, enhancing efficiency, and guaranteeing that all customers are using the API fairly.

Software updates may be distributed quickly and effectively thanks to continuous integration and deployment (CI/CD), an automated method. The API may be updated quickly and seamlessly without compromising its availability by using a CI/CD workflow.

REFERENCES

- [1] O. Gómez, R. Rosero, and K. Cortés-Verdín, "CRUDyLeaf: A DSL for Generating Spring Boot REST APIs from Entity CRUD Operations," *Cybernetics and Information Technologies*, vol. 20, no. 3, pp. 3-14, 2020.

- [2] Kulkarni, Chaitanya & Takalikar, Mukta. "CSEIT183535 | Analysis of REST API Implementation." (2020), *International Journal of Computer Science and Information Technology Research*, vol. 8, no. 3, pp. 156-163.

- [3] R. K. Soni, N. Soni, R. K. Soni and N. Soni, "Deploy a Spring Boot Application as a REST API in AWS," *Spring Boot with React and AWS: Learn to Deploy a Full Stack Spring Boot React Application to AWS*, pp. 41-75, 2021.

- [4] S.P.R. Katamreddy and S.S. Upadhyayula, "Building REST APIs Using Spring Boot," in *Beginning Spring Boot 3: Build Dynamic Cloud-Native Java Applications and Microservices*, Berkeley, CA: Apress, 2022, pp. 161-184.