

THREE LAYERED ARCHITECTURE

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

in

Computer Science and Engineering/Information Technology

By

Oshin Dhawan (191435)

to



Department of Computer Science & Engineering and Information Technology

Jaypee University of Information Technology Wagnaghat, Solan-173234,

Himachal Pradesh

DECLARATION

I hereby declare that this submission is my own work carried out at Zopsmart Technologies Pvt Ltd, Bangalore from February 2023 to May 2023 and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma from a university or other institute of higher learning, except where due acknowledgment has been made in the text.



SUBMITTED BY:

Oshin Dhawan

191435

Computer Science & Engineering and Information Technology Department.

Jaypee University of Information Technology, Waknaghat, Solan

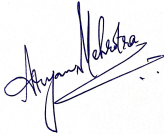
CERTIFICATE

I hereby declare that the work presented in this report entitled **THREE LAYERED ARCHITECTURE** 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 Wagnaghat is an authentic record of work carried out over a period from February 2023 to May 2023 under the supervision of **Mithali R Shetty (Senior Lead Engineer) and Aryan Mehrotra (SDE2)**. The matter embodied in the report has not been submitted for the award of any other degree or diploma.



Oshin Dhawan
191435

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



Mr. Aryan Mehrotra
SDE2
Zopsmart Technology
Dated: 13-05-2023

Dr. Diksha Hooda
Assistant Professor (SG)
Department of Computer Science & Engineering
Dated: 13-05-2023

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT
PLAGIARISM VERIFICATION REPORT

Date: 13/05/2023

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

Name: Oshin Dhawan Department: CSE Enrolment No 191435

Contact No. 9816140071 E-mail. 191435@juitsolan.in

Name of the Supervisor: Dr. Diksha Hooda

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

THREE LAYERED ARCHITECTURE

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 = 46
- Total No. of Preliminary pages = 9
- Total No. of pages accommodate bibliography/references = 1

(Signature of student)

FOR DEPARTMENT USE

We have checked the thesis/report as per norms and found **Similarity Index** at²⁰.....(%). 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)	
	<ul style="list-style-type: none"> • All Preliminary Pages • Bibliography/Images/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

This report is not just a result of hard work by me but there has been a joint contribution by a lot of other people who I would like to thank.

I would like to thank Rashmi Singh, Manager of HR in Talent Acquisition, of Zopsmart Technologies, Bangalore for allowing me to do an internship within the organization.

I also would like to thank Ms. Mithali R. Shetty and all the people that worked along with me at Zopsmart Technologies, Bangalore for their patience and openness. They created an enjoyable working environment.

I also would like to thank Mr. V. Vaishnav and Ms. Mahak Singhania for mentoring me throughout my internship and helping me learn new concepts and technologies. It is indeed with a great sense of pleasure and immense sense of gratitude that I acknowledge the help of these individuals.

I am highly indebted to Mr. Pankaj Kumar, Training & Placement Coordinator of our college for the facilities provided to accomplish this internship. I would also like to thank the Head of our Department Dr. Vivek Kumar Sehgal and the faculty for teaching us the skills required for this internship.

Special thanks to my mentor in the college Dr. Diksha Hooda who supported me throughout the whole and guided me to achieve the best.

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



Oshin Dhawan

191435

TABLE OF CONTENT

<u>CONTENT</u>	<u>PAGE NO.</u>
LIST OF ABBREVIATIONS	VII
LIST OF FIGURES	VIII
ABSTRACT	IX
CHAPTER - 1 INTRODUCTION	1
1.1 About the Company	1
1.2 Project Introduction	3
1.3 Project Description	6
1.4 Organization	7
CHAPTER - 2 LITERATURE SURVEY	8
CHAPTER -3 SYSTEM DEVELOPMENT	15
3.1 Technologies Required	15
3.2 Project Development Approach	31
3.3 Code Development	34
CHAPTER - 4 PERFORMANCE ANALYSIS	43
4.1 Unit Test Coverage	43
4.2 Linter Check	43
CHAPTER - 5 CONCLUSIONS	44
5.1 Results Achieved	44
5.2 Application Contributions	44
5.3 Limitations	45
5.4 Future Work/Scope	45
REFERENCES	46

LIST OF ABBREVIATIONS

SNSS	Social Networking Sites
CMS	Content Management System
UI	User Interface
ESR	Extended Support Release
HTML	Hypertext Markup Language
IP	Internet Protocol
LAN	Local Area Network
URL	Uniform Resource Locator
HTTP / HTTPS	Hypertext Transfer Protocol / Secure Hypertext Transfer Protocol
SVG	Scalable Vector Graphics
DOM	Document Object Model
JSON	Javascript Object Notation
SQL	Structured Query Language
W3C	World Wide Web Consortium
MVC	Model View Controller
VCS	Version Control System
XML	Extensible Markup Language

LIST OF FIGURES

Figure 1.	ZopSmart Logo
Figure 2.	Three-Layered Architecture
Figure 3.	Postman Endpoints
Figure 4.	Swagger Endpoints
Figure 5.	Software Development Cycle
Figure 6.	Scrum Life Cycle
Figure 7.	Setting up the database
Figure 8.	Product datastore layer code
Figure 9.	Brand datastore layer code
Figure 10.	Product service layer code
Figure 11.	Brand service layer code
Figure 12.	Product handler layer code
Figure 13.	Brand handler layer code
Figure 14.	Source file code
Figure 15.	Middleware code
Figure 16.	Swagger yaml file
Figure 17.	Swagger online editor
Figure 18.	Sample output for product_GetByID
Figure 19.	Sample output for brand_Get
Figure 20.	Postman Collection

Figure 21.	Unit Test Coverage Check
Figure 22.	Linters Check

ABSTRACT

Creating a web application is quite simple but the challenge comes when the code has to be tested, structured, cleaned, and maintained, and thus here we follow the Three Layered Architecture using Go language.

The three layers are handler, service, and datastore which are all independent of each other. The handler layer receives the request body and then parses anything that is required from that request. It then calls the service layer where all the logic of the program is defined, ensures that the response is in the required format, and writes it to the response writer. This layer further communicates with the datastore layer. It takes whatever it needs from the handler layer and then calls the datastore layer.

The datastore layer is where all the data is stored. It can be any data storage. The use case layer is the only layer that communicates with the data store. That is how we test each layer independently making sure that no layer affects the other.

CHAPTER - 1

INTRODUCTION

1.1 About The Company

➤ ZopSmart Technologies

They are a cutting-edge retail technology company that provides you with all of the tools you need to launch your own e-commerce venture [1]. They provide a portfolio of products that can help you achieve your goals quickly and easily, whether you are a traditional..store trying to extend your omnichannel business or an online-only shop looking to increase your e-commerce business. Zopsmart develops next-generation retail technology for customers ranging from small furniture stores to multinational retail chains. Their solutions include an e-commerce platform, Digital Marketing, m-Commerce, automated logistics systems, a management platform, an order management platform, and IoT devices. It also gives software solutions to some of the most prominent companies and has its own framework on which to operate.



Fig 1: ZopSmart Logo

➤ ZopSmart Solutions

- **e-Commerce:**

Give your clients a first-rate purchasing experience- Using straightforward search, you can quickly discover things. Product listing

that is personalized for ordering ease. Self-service rescheduling and returns ensure a smooth order experience.

- **m-Commerce:**

Offer your clients an excellent mobile purchase experience. A responsive website that adapts to all screen sizes. Use native Android as well as iOS mobile applications for a world-class experience. For convenience, the cart is persistent between devices.

- **e-Wallet:**

Increase consumer loyalty by using an integrated wallet - An integrated wallet provides for simple and quick checkout. Refunds and pocket rewards are used to encourage repeat purchases. For low-cost customer acquisition, use Wallet-credit as gift cards.

- **Order Management:**

Handle your purchases quickly and effortlessly - All of your purchases, together with associated data such as customer name, amount, and order status, will be shown in a single interface. As each process step is finished, the status of the order changes. You can add or delete items from your order.

- **Operations:**

By utilizing the Store-manager mobile app to manage daily tasks, you can provide your customers with outstanding service at the lowest possible cost. The inventory checker mobile app ensures catalog accuracy. Picker is a smartphone app that enables error-free and efficient selection and packing. The dispatch module will determine the most effective delivery route. Delivery smartphone app for reliable order delivery, risk-free payment collection, and exact return selection.

- **Monitoring and Analytics:**
Operate your business with little supervision - Continuous monitoring allows you to monitor every aspect of your operation. Notifications of deviations in the operations process to trigger immediate action
Intelligent analytics assists in detecting trends and improving procedures. Detailed logs for each action to assist you in troubleshooting.

1.2 Project Introduction

Go is a robust system-level programming language used to create large-scale server networks as well as distributed systems. It is frequently used as a replacement for C++ and Java. Go's syntax is comparable to C's, although it uses fewer brackets and commas, making it similar to Python's as well. Fast compilation and execution, faster code readability and debugging, simpler versioning, language consistency, easy interchange with other languages, simplified maintenance, and support for concurrency and multithreading are all advantages of the language. Go is a programming language created at Google by Robert Griesemer, Rob Pike, and Ken Thompson that provides memory security, garbage collection, structural type, and CSP-style parallelism.

➤ Three-Layered Architecture in Go:

It is straightforward to create a web application in Go using the Three-Layered Architecture, but making sure the code is tested, structured, clean, and maintained might be difficult.

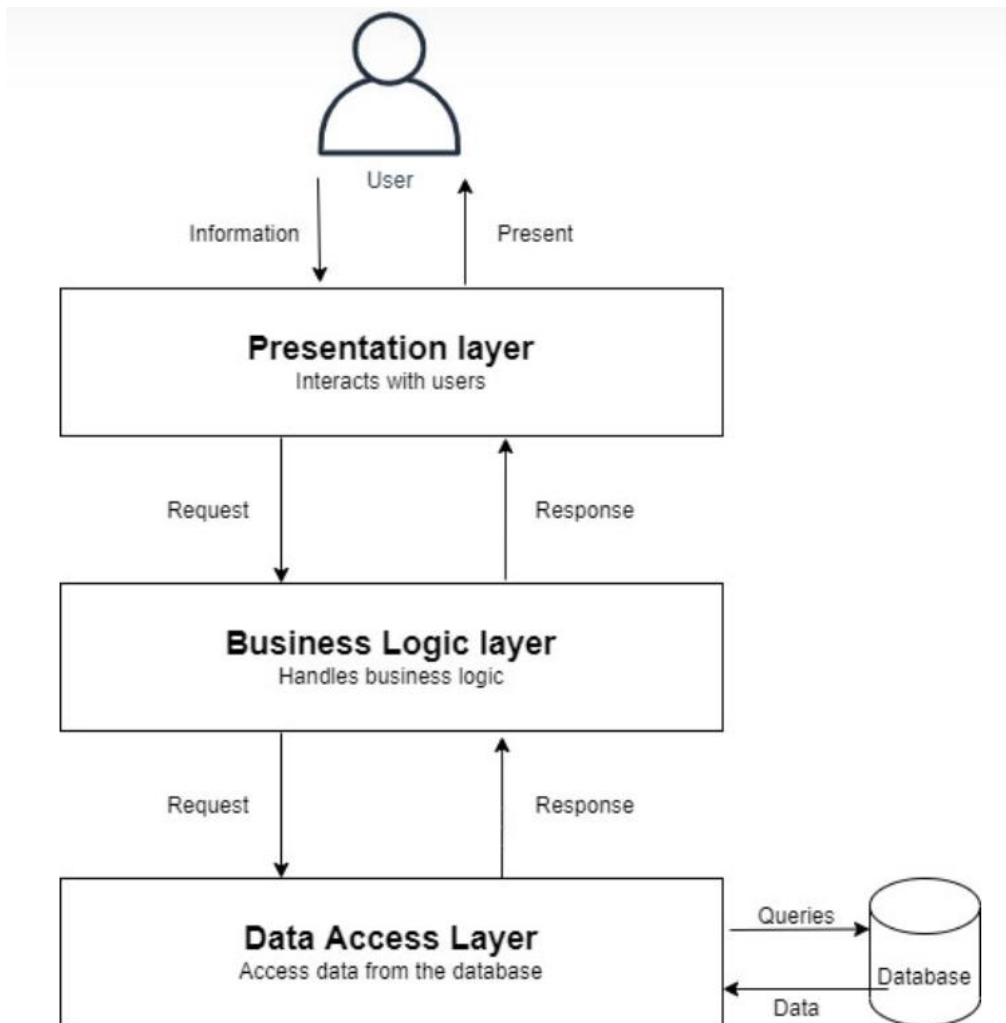


Fig 2. Three-Layered Architecture

The Three-Layered Architecture is made up of three distinct layers: the Presentation Layer, the Business Logic Layer, and the Data Access Layer.

The presentation layer, commonly referred to as the delivery layer, is in charge of receiving the request and processing any information it contains. It then calls the business layer, frequently referred to as the use-case layer, to ensure that the answer is in the correct format before sending it to the response writer.

The use-case layer handles the application's business logic and connects with the datastore layer. It gets what it needs from the delivery layer before calling the datastore layer. It implements the necessary business logic before and after calling the datastore layer, also known as the data access layer. The data store

is where the data is stored, which can be in any form. The only layer that interfaces with the datastore is the use-case layer. We guarantee that the system is testable and maintained by testing each layer independently of the others.

➤ *Advantages of three-layer architecture*

- **Explicit code:** The code is divided into layers. Instead of bundling all the code in one location, each one is dedicated to a specific duty such as interface, business processing, and querying.
- **Easy to maintain:** It would be easy to update anything since its duties separate each layer. The change can be limited to a single layer or affect only the layer closest to it without affecting the entire program.
- **Easy to develop, and reuse:** Because we already have a standard architecture, we can quickly alter a function. If we wish to switch from a Winform to a Web form, we only need to update the Presentation layer; the remaining levels may be reused entirely.
- **Easy to transfer:** We might save time migrating the application to others since they follow and use a common architecture.
- **Easy to distribute the workloads:** By separating the code into layers depending on responsibilities, each team/member may create code on each layer individually, allowing developers to better manage their workload.

1.3 Project Description

The principal objective was to create a product and branding management system utilizing a three-layered architecture. In this system, we developed and implemented a SQL database design to properly handle information about goods and brands for a food and beverage firm. The schema has two tables: "Products" and "Brands." Product ID, product name, description, price, quantity, category, and brand ID were all fields in the "Products" table. The product ID was the main key for the "Products" table, and the brand ID was a secondary key that referred to the "BrandID" field in the "Brands" table. The "Brands" table had fields for brand ID and brand name, with the brand ID serving as the primary key and the brand name serving as a unique identifier.

We also created five methods to communicate with the SQL database using Test Driven Development (TDD). The first function, `GetByID`, took a product ID as input and returned the product data from the database, comprising the product name, description, price, quantity, category, and brand name. The second function, `GetByName`, took a product name as input and returned the product data from the database. `GetAllProducts`, the third function, got all of the items and their associated information from the database. The fourth function, `Update`, received product information such as the product name, description, price, quantity, category, and brand name and updated the database record pertaining to that product. The function checked the supplied data before updating it and provided an error if it was incorrect. The fifth function, `Create`, took the product data as input and produced a new product record in the database, containing the product name, description, price, quantity, category, and brand name. The function evaluated the supplied data before putting it into the database and returned an error if it was incorrect.

In addition, we built a Swagger interface that enabled users to access the system's features. The `"products/id"` endpoint lets users get product data from

the database using the product ID. Users could also use the ID to change product information such as the product name, description, price, quantity, category, and brand ID. Similarly, the "product" endpoint enabled users to enter product details to create a new product record in the database. The input was checked before being inserted, and an error was returned if it was invalid.

We used a variety of technologies to build the system, including MySQL for the database, Postman for testing endpoints, Swagger for the API interface, Golang for system implementation, Lintercheck for code quality checking, and middleware to handle requests and responses between layers.

1.4 Organization

The rest of this report is organized as follows:

Chapter 2 provides the literature of the technologies required

Chapter 3 discusses system development and workflow

Chapter 4 presents performance analysis

Chapter 5 covers conclusions and future scope.

CHAPTER - 2

LITERATURE SURVEY

1. GO

Go, often known as Golang, is an open-source compiled and statically typed computer language created by Google. Rob Pike, Ken Thompson, and Robert Griesemer invented the language, which was originally made accessible to the public in November 2009.

Go is a general-purpose programming language with an easy-to-use syntax and a robust standard library. It is particularly good at developing scalable and highly accessible online applications, in addition to command-line apps, desktop apps, and even mobile apps.

➤ Advantages of Golang:

- **Simple syntax:**

The grammar is straightforward and succinct, and the language is devoid of superfluous features. This makes it simple to develop understandable and maintainable code.

- **Easy to write concurrent programs:**

Concurrency is built into the language. As a result, creating a multithreaded program is a breeze. This is accomplished through the use of Goroutines and channels, which will be covered in the next chapter.

- **Compiled language:**

Go is a compiled programming language. The source code gets

transformed into a native binary. This is lacking in interpreted languages like JavaScript, which is utilized in nodejs.

- **Fast compilation:**

The Go compiler is incredible because it was built from the ground up to be quick.

- **Static linking:**

Static linking is supported by the Go compiler. The complete go project may be statistically linked into a single large binary that can be quickly deployed in cloud servers without regard for dependencies.

2. WEB DEVELOPMENT

The process of generating and managing websites and online applications is known as web development. It includes a wide range of jobs, from the creation of a simple static website to the creation of complicated web-based apps, e-commerce solutions, and social network services. Web engineering, web design, web content creation, client-side scripting, web server and network security settings, and e-commerce development are some of the responsibilities involved in web development.

Web development is often done by a team of specialists, with each team member specialized in a certain area of web development. Front-end developers, for example, are in charge of the visual and interactive parts of a website, such as its layout, user interface, and functionality. Back-end developers, on the other hand, concentrate on the website's server-side components, such as the database, server-side scripting, and APIs. Full-stack engineers are skilled in both front-end and back-end development and are capable of handling all parts of web development.

Content Management Systems (CMS) have grown to be a popular solution for non-technical users to maintain the content of their websites without requiring technical skills. CMS systems can be built from the ground up, be proprietary, or open source. In a larger sense, a CMS serves as a bridge between the database and the user via the browser.

Smaller organizations may only need a single permanent or contract developer, or secondary assignment to related job positions like a graphic designer or information systems technician. While larger organizations and businesses frequently have dedicated web development teams that adhere to standard methods like agile methodologies. Instead of being the purview of a single department, web development can also be a collaborative effort between departments.

3. CRUD OPERATIONS

CRUD stands for Create, Read, Update, and Delete. It refers to the four fundamental functionalities of persistent storage in computer programming. These operations are mapped to conventional HTTP methods, SQL statements, or DDS actions and may be implemented in relational database systems. CRUD can also refer to user-interface principles that enable viewing, finding, and editing information via computer-based forms and reports.

CRUD operations are used to read, create, update, and remove entities. Before sending it back to the service for an update, data from a service can be updated by altering the setting's properties. CRUD is data-focused and uses common HTTP action verbs.

Every programmer has worked with CRUD functionality at some time because it is present in the majority of apps. Forms are used in CRUD applications to access databases and return data. James Martin's book *Managing the Database*

Environment from 1983 is credited with popularizing the phrase "CRUD operations." 1990's "From Semantic to Object-Oriented Data Modeling" essay by Haim Kilov also made mention of CRUD activities.

Here's a breakdown of **CRUD** operations:

- **CREATE:** executes the INSERT command to add a new entry to the database.
- **READ:** depending on the input parameter's primary key, reads the table's records.
- **UPDATE:** Executes an UPDATE statement on the table based on the supplied primary key for a record specified in the statement's WHERE clause.
- **DELETE:** Deletes a specified row in the WHERE clause.

CRUD activities are carried out in accordance with the system's requirements, and various users may have various CRUD cycles. A consumer, for example, may utilize CRUD to establish an account and access it when returning to a certain site. Users may then alter personal information or billing details. A product record may be created by an operations manager, and then line items may be modified.

CRUD processes were the core of most dynamic websites throughout the Web 2.0 era. Therefore, it is critical to distinguish between CRUD and HTTP action verbs. For example, "POST" is used to add a new record, "PUT" or "PATCH" is used to change an existing record, and "DELETE" is used to delete a record. Users and administrators can utilize CRUD to edit, remove, create, or explore online records.

For performing CRUD activities, application designers have numerous alternatives. To perform processes, one of the most efficient methods is to establish a collection of stored procedures in SQL.

Here are some common naming conventions for CRUD stored procedures:

- The procedure name should finish with the CRUD operation's implemented name, and the prefix should not match that of the prefix for other user-defined stored procedures.
- If you put the table name after the prefix, CRUD methods for the same table will be grouped together.
- You can edit the database schema after adding CRUD procedures by identifying the database object where CRUD operations will be conducted.

4. UNIT TESTING

It is a critical sort of software testing in which individual units or components of a software program are tested to ensure that they work as intended. Developers use it during the development process to isolate a part of code and test its accuracy. Individual functions, methods, processes, modules, or objects might be considered units.

In SDLC, STLC, and V Models, unit testing is the initial level of testing performed before integration testing. It is a WhiteBox testing approach that is normally conducted by developers, although in practice, due to time restrictions or developer unwillingness to test, QA engineers also undertake unit testing.

➤ *Why is unit testing important?*

Inadequate unit testing can result in large expenses for flaw correction during System Testing, Integration Testing, and even Beta Testing after application

release. Effective unit testing performed early in development saves time and money in the long run. Excellent unit tests serve as project documentation, aid in code reuse, and allow developers to easily comprehend and modify the testing code base.

There are two types of unit testing:

- Manual
- Automated

While there is no preference in software engineering, automation is preferable since it saves time and resources. Unit testing requires developers to write a chunk of code to test a particular function in a software application and can involve the usage of a UnitTest framework to create automated test cases.

Black box testing, white box testing, and grey box testing are the three types of unit testing approaches.

Unit testing code coverage strategies include:

- Statement Coverage
- Decision Coverage
- Branch Coverage
- Condition Coverage
- Finite State Machine Coverage.

These strategies assist in ensuring that unit tests cover all potential circumstances and that the code is resilient and dependable.

In summary, unit testing is an important part of the software development process since it assures code quality and dependability. It aids in the early detection of defects, saves time and money, and aids developers in understanding the codebase. Manual unit testing is favored over automated

unit testing, and there are several ways available to obtain thorough code coverage.

5. GOMOCK

GoMock [3] is a mock framework for the Go programming language that is used for testing. It is approved by the github.com/golang organization, interacts easily with the built-in testing package, and offers a configurable expectation API.

The use of GoMock entails four basic steps:

1. The first step is to construct a mock for the interface you want to simulate using `mockgen`.
2. Create a `gomock` object in your test.
3. Send it to your mock object's `Object()` [native code] method to get a fake object.
4. Configure your mock's expectations and return values by invoking `EXPECT()` on it.
5. Call `Complete()` on the fake controller to ensure that the mock's requirements are satisfied.

CHAPTER - 3

SYSTEM DESIGN AND DEVELOPMENT

3.1 Technologies Required

➤ Go Programming Language:

Go, often known as Golang, is an open-source compiled and statically typed computer language created by Google. Rob Pike, Ken Thompson, and Robert Griesemer invented the language, which was originally made accessible to the public in November 2009.

Go is a general-purpose programming language with an easy-to-use syntax and a robust standard library. It is particularly good at developing scalable and highly accessible online applications, as well as command-line apps, desktop applications, and even mobile applications.

GOPATH and GOROOT are two significant environment variables in Go. The former gives your workspace's location, while the latter specifies the location of your Go SDK.

Go Programming Tools:

Go is well-equipped with various powerful tools that help developers write better code. These tools include:

- gofmt: format automatically Use tabs for indentation and blanks for alignment in your Go source code.
- golint: identifies styling issues in the code.

Garbage Collection:

Because Go employs garbage collection, memory management is handled automatically, allowing developers to easily construct concurrent programs.

Simple Language Specification:

The whole Go language specification is contained on a single page, making it simple and straightforward.

Open Source:

Go is an open-source project, and anybody may help create and enhance it.

Popular Products Built with Go:

Go has been used to develop many popular products, including

- Kubernetes, developed by Google.
- Docker, a containerization platform.
- Dropbox has migrated its performance-critical components from Python to Go.
- Infoblox's next-generation networking products are developed using Go.

Variables

A variable is a designated memory area that stores a certain type of data. Variables in Go may be declared by using the `var` keyword followed by the variable's name and type.

For example, `var age int` defines an int variable named "age." Variables can also be initialized with a value when declared using the syntax `var name type = value`. For example, `var score float64 = 9.5` initializes a variable named "score" of type float64 with a value of 9.5.

Type Inference

Type inference is a Go feature that enables the compiler to automatically deduce the type of data of a variable based on its value. When a variable is

initialized with a value, its kind can be inferred, allowing the type declaration to be omitted.

```
var name = initial_value
```

Go will automatically infer the type of that variable from the initial value.

For example, *name := "John"*

declares a variable named "name" and initializes it with a string value "John".

The type of variable is automatically inferred as a string.

Multiple variables can be declared using a single statement.

```
var name1, name2 type = initialvalue1, initialvalue2
```

This is the syntax for multiple variable declarations.

For example, *var x,y int = 10,20*

declares two variables named "x" and "y" of type int and initializes them with the values 10 and 20 respectively.

Go also provides another concise way to declare variables. This is known as short-hand declaration and it uses " := " operator.

```
name := initial_value
```

It is the short-hand syntax to declare a variable.

For example, *age := 30*

declares a variable named "age" of type int and initializes it with a value of 30.

Functions

In Go, functions are chunks of code that execute certain tasks. They receive parameters as input, process them, and return output values.

In Go, the syntax for defining a function is:

```
func functionName(param1 type1, param2 type2) returnType {  
    //function body  
}
```

The function name is accompanied by the input arguments, which are separated by commas in brackets. Each parameter has a name as well as a type. The data type of the value returned by the function is specified by the returnType.

In Go, functions can return multiple values, which are provided in brackets following the function parameters:

For example, *func swap(a, b int)(int, int){}* is a function named “swap” that takes two integer parameters and returns two integers.

Named return values can also be used to make the code more readable.

For example, *func calculate(x,y int)(result int) {}* declares a function named “calculate” that takes two integer parameters and returns an integer value named “result”

Go Packages

Packages are used in Go to organize and reuse code. A package is a grouping of similar Go source files in the same directory. Go has a standard package library that can be exported and used in any Go program.

The main function of Go is the point of entry for program execution. Every executable Go program must have a main function that is located in the main package. The main package is Go's default package for creating executable

applications. The package name should be specified in the first line of every Go source file using the syntax `package packagename`.

Package

`packagename` specifies that a particular source file belongs to package `packagename`. This should be the first line of every Go source file.

Init

Function

An `init` function may be found in every Go package. There must be no return type and no arguments in the `init` function. In our source code, the `init` method cannot be invoked explicitly. After the package is initialized, it will be called automatically. The syntax for the `init` function is as follows:

```
func init() {  
    // initialization tasks go here  
}
```

The `init` function can be used to execute initialization duties as well as to validate the program's correctness before execution begins.

The order of initialization of a package is as follows:

1. Package-level variables are initialized first.
2. The function `init` is called next. A package can have numerous `init` functions (either in a single file or spread over multiple files), and the compiler calls them in the order they are provided to it.
3. When a package imports other packages, the imported packages are the first to be initialized.
4. Even if a package is imported from numerous packages, it will only be started once.

Arrays

A collection of items of the same kind is called an array. An array, for example, is formed by the numbers 5, 8, 9, 79, and 76. Combining values of multiple kinds, such as an array containing both strings and integers, is not permitted in Go.

Declaration

An array is of the type '[n]T'. The letter 'n' signifies the number of items in an array, while the letter 'T' represents the type of each element. The type additionally includes the number of components 'n'.

```
var a [3]int
```

The preceding code declares a 3-dimensional integer array. Every item in an array is immediately allocated the array type's zero value. Because 'a' is an integer array in this example, all of its elements are assigned to 0, the zero value of 'int'.

In Go, arrays are value types rather than reference types. This implies that when they are allocated to a new variable, they are assigned a duplicate of the original array. If you alter the new variable, the changes will not be reflected in the original array.

The for loop may be used to traverse across array

```
import "fmt"
func main(){
    a := [...]float64{67.7,89.8,21,78}
    // looping from 0 to the length of the array
    for i := 0; i < len(a); i++ {
        fmt.Printf("%d the element of a is %2.f\n", i, a[i])
    }
}
elements:
```

The range version of the for loop in Go is a more efficient and succinct approach to iterate over an array. 'range' delivers the index as well as the value at that index.

Slices

A slice is an array's handy, versatile, and powerful wrapper. Slices do not have their own data. These are simply pointers to existing arrays.

Creating a Slice

'[]T' represents a slice having elements of type 'T'. A slice does not have its own data. It is nothing more than a representation of the underlying array. Any changes to the slice will be reflected in the underlying array. The number of items in the slice is represented by the slice's length. The slice's capacity is the number of items in the underlying array beginning at the index from which the slice is formed.

Creating a slice using make:

```
func make([]T, len, cap) []T
```

`make` can be used to create a slice

Channels

In Go, channels are used to interact between goroutines. They function in the same way as pipes do, with data provided from one end and received from the other.

Declaring channels

Every channel may only transmit one type of data; additional types are not authorized. "chan T" specifies a channel of type T. The zero value of a channel is nil, and it, like maps and slices, should be defined using make.

Example:

```
package main

import "fmt"

func main() {
    var a chan int

    if a == nil {
        fmt.Println("Channel a is nil, defining it now")
        a = make(chan int)
        fmt.Printf("Type of a is %T", a)
    }
}
```

In this example, "a" is a nil int channel, as defined by the if expression. When data is sent to a channel, the transmitter is disabled until the channel is read by a receiver, and vice versa. This feature allows goroutines to successfully interact without the use of explicit locks or conditional variables.

Deadlock

A deadlock can arise when a goroutine is waiting to send data on a channel but there is no recipient. A deadlock can also arise if a goroutine is waiting to receive data on a channel but no other goroutine is transmitting data on that channel.

Layered Architecture

A layered architecture is divided into three distinct layers: delivery, use-case, and datastore. Each instruction and learning with the others via an interface allows for easy application maintenance and expansion.

Delivery Layer:

- Accept the request and parse everything necessary from it.
- Calls use case layer
- Check that the response is in the proper format and send it to the response writer.

Use-Case Layer:

- Business logic
- Communicates with datastore layer

Data Store Layer:

- If the datastore changes, the complete application does not need to update. Just the datastore layer will be modified.
- Easy to isolate any bugs, maintain the code and grow the application.

Dependency Injection

Dependency injection (DI) is a coding approach in which an object's dependencies are given when the object is initialized. This gives you more flexibility over when to add new dependencies and when to reuse old ones. DI eliminates the connectivity between objects and their dependents, making code maintenance and modification easier.

Factory Method

The factory method is a design pattern that allows objects to be created without providing their particular classes. Instead of utilizing a function `Object() { [native code] }` call directly, a factory function may be used to generate objects. Factory approaches are classified into two types: basic factories and interface factories. Straightforward factories yield struct instances, whereas interface factories return interfaces, allowing the behavior to be defined without revealing implementation specifics.

➤ **MySQL:**

MySQL [] is a prominent relational database management system (RDBMS) used by both small and big enterprises. MySQL, which is developed, sold, and maintained by the Swedish corporation MySQL AB, is highly regarded for a variety of reasons, including

- It is open-source and free to use.
- It has a powerful collection of capabilities that can handle a significant portion of the functionality provided by more costly database solutions.
- It uses the SQL data language standard.
- It works with a wide range of operating systems and computer languages, including PHP, PERL, C, C++, and JAVA..
- Even with enormous datasets, it executes rapidly and efficiently.
- It is very compatible with PHP, the most extensively used web development language.
- Large databases can be supported, with a theoretical limit of 8 million gigabytes (TB).
- The open-source GPL license allows for extensive customization.

Some of the key features of MySQL include

- A multi-layered, modular design.
- Full multithreading and support for multiple CPUs.
- Transactional and non-transactional storage engines.
- A high-speed, thread-based memory allocation system.
- In-memory heap table support.
- The ability to handle large databases.
- Support for both client/server and embedded systems.
- Compatibility with a wide range of platforms.

Plenty of good websites, including Facebook, Wikipedia, YouTube, Flickr, and Google, which use MySQL (not for search). It's also popular for content management systems (CMS) like WordPress, Drupal, Joomla, and phpBB. Furthermore, a big number of web developers all around the world utilize MySQL to create online applications.

➤ **Docker:**

Docker [6] is a software platform that makes it easier to design, test, and deploy applications. This is accomplished by packaging software into standardized units known as containers, which include everything the software requires to function, such as libraries, system tools, code, and runtime. Docker allows applications to be instantly deployed and scaled in any environment, providing developers with confidence that their code will work smoothly.

Docker is a component of the Moby project, which is a framework for creating, operating, and managing containers on servers and in the cloud. The name "docker" can refer to both the tools (commands and daemon) and the Dockerfile file format. Formerly, deploying a web application involved purchasing a server, installing Linux, configuring a LAMP stack, and

executing the program. Yet, as cloud-based systems have grown in popularity, the idea of a server has been abstracted into a software-based container that can run on any hardware.

Containers are a mixture of the Linux operating system with a hyper-localized runtime environment, allowing them to be highly portable and scalable.

➤ Understanding Containers

Container technology can be divided into three categories: builder, engine, and orchestration.

- **Builders** are tools used to create containers, such as Dockerfile for Docker or distro builder for LXC.
- **Engines** run the container, such as the docker command and daemon for Docker, or contained and postman for other engines.
- **Orchestration** technologies, such as Kubernetes and OKD, manage a lot of containers at once.

Containers contain both software and its settings, saving system administrators time and effort as compared to installing an application from a traditional source. Dockerhub and Quay.io are repositories of pre-built pictures that container engines can utilize.

One of the most appealing features of containers is their ability to "die" gently and revive as needed for load balancing. Containers are built to emerge and vanish invisibly, making them incredibly scalable and cost-effective. Container orchestration systems, such as Kubernetes and OKD, can handle huge numbers of containers efficiently.

➤ Why use Docker?

Open-source technologies enable developers to select the tools that best suit their needs. Docker is a compact and clean environment ideal for developers

who wish to test without the need for complicated orchestration. If Docker is not accessible, Podman can be used instead. To maintain long-term flexibility and avoid vendor lock-in, it is critical to favor open-source and open-standards-based solutions.

Finally, Docker facilitates application deployment by abstracting hardware and offering standardized containerized environments. Containers are great for cloud-based applications because they are very portable and scalable. Managing many containers is simple and efficient with the right container orchestration tools. Open-source solutions such as Docker and Podman provide developers with flexibility and choice while avoiding vendor lock-in.

➤ **Postman:**

Postman[7] is a robust API testing, development, and modification tool. It has over 5 million monthly users and provides a user-friendly graphical interface that enables various sorts of HTTP queries (e.g., GET, POST, PUT, PATCH), environmental savings, and API-to-code translation for languages such as JavaScript and Python.

To interact with endpoints, Postman offers several methods that include:

- GET: Retrieve information
- POST: Add information
- PUT: Replace information
- PATCH: Update certain information
- DELETE: Delete information

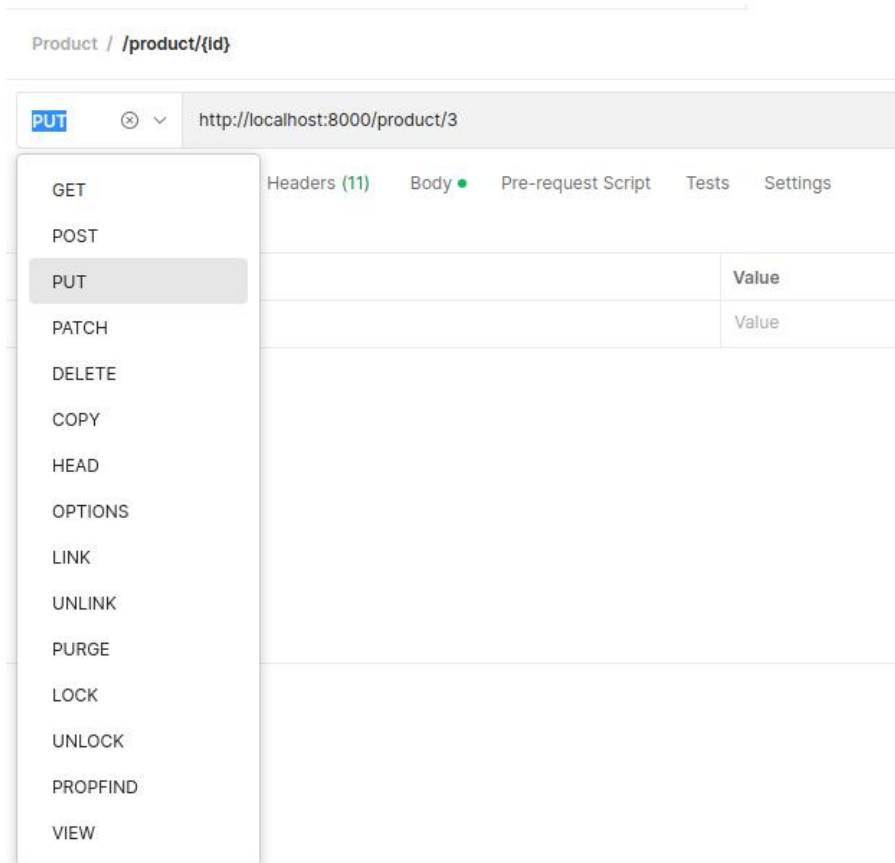


Fig 3. Postman Endpoints

When testing APIs with Postman, different response codes can be obtained. Some of the most common ones include

- 100 Series: Temporal responses, for example, '102 Processing.'
- 200 Series: Responses, where the client accepts the request and the server, processes it successfully, for instance, '200 Ok.'
- 300 Series: Responses related to URL redirection, for example, '301 Moved Permanently.'
- 400 Series: Client error responses, for instance, '400 Bad Requests.'
- 500 Series: Server error responses, for example, '500 Internal Server Error.'

Postman provides collections to help arrange tests. These are folders where requests are saved and can be organized in any way the team sees fit. They can also be exported and imported.

Also, Postman users may utilize variables to build distinct setups. A URL variable, for example, might be addressed towards distinct test environments (for example, dev-QA), allowing tests to be conducted in different contexts utilizing existing queries.

➤ Swagger:

Swagger is a tool that allows you to define the structure of your APIs in a machine-readable language, enabling the development of attractive and interactive API documentation automatically. It can also produce client libraries and automate testing in a variety of computer languages.

Swagger operates by requesting a YAML or JSON file with a full description of the API in accordance with the OpenAPI Standard. This file contains information on the API's available operations, arguments, return values, and authentication requirements. It may also include conditions of use, contact information, and license information.

```
openapi: 3.0.3
info: <3 keys>
servers: <1 item>
tags: <2 items>
paths: <4 keys>
components: <2 keys>
```

Fig 4. Swagger endpoints

A Swagger specification can be written by hand or generated automatically from comments in your source code. In swagger.io/open-source-integrations,

you may find a list of tools that can help you build Swagger from code. You may ease API development, testing, and documenting by using Swagger to explain the structure of your API, making it more accessible to others.

➤ **Version Control:**

Version control is a critical component of software development because it provides a centralized mechanism for programmers to access, monitor, update, and report on project progress. Github is a popular and free version control application that includes functionality such as pull requests, merge, and fork.

Version control isn't only for software engineers. A Version Control System (VCS) may also help graphic and web designers maintain track of all versions of an image or layout. Designers may use a VCS to easily restore chosen files to a prior state, analyze changes over time, and discover who last edited anything that may be creating an issue. It also enables users to retrieve files that have been lost or destroyed.

Git is a popular version control system (VCS) that offers various benefits. Users may work offline, commit changes locally, and subsequently push those changes to a centralized repository. Git also includes a distributed design, which implies that developers may work on an identical project at the same time without clashes. Git also supports branching, which allows programmers to collaborate on multiple features at the same time without interfering with each other's work.

To use Git efficiently, you must first grasp its foundations. Git differs from other VCSs in how it saves and thinks about information. Understanding these distinctions will thus assist users in avoiding misunderstanding when using the tool.

3.2 Project Development Approach

- Software Development Life Cycle

The SDLC (Software Development Life Cycle) is a framework that describes the activities that must be completed at each stage of the software development process. A development team inside a software organization follows this structure. The SDLC is a thorough strategy that explains how to create, maintain, and replace specific software. This life cycle presents a mechanism for enhancing software quality and the development process as a whole. The SDLC is often known colloquially as the software development process.



Fig 5: Software development cycle

- SDLC Models

The Software Development Life Cycle (SDLC) is a project management model that explains the processes required in producing an information system, from early feasibility analysis through post-development maintenance. Several SDLC models, often known as Software Development Process Models, have been developed to assist software development. Each model follows a unique set of phases to guarantee effective software development.

The following are some of the most widely used and important SDLC models:

- Waterfall model
- Iterative model
- Spiral model
- V-shaped model
- Agile model.

- Scrum Model

Scrum is a software development technique that is agile and based on incremental and iterative procedures. It is intended to be adaptive, quick, flexible, and effective, with the purpose of providing value to the client throughout the project's life cycle. Scrum aims to assure customer happiness via open communication, joint ownership, and continual development. The development process begins with a high-level concept of what needs to be produced, followed by the formulation of a prioritized list of characteristics (product backlog) desired by the product owner.



Fig 6. Scrum Life Cycle

The Scrum technique is distinguished by small, periodic pieces of work known as Sprints, which generally span 2-4 weeks and are used for feedback and reflection. Each Sprint is a self-contained entity that creates a releasable product increment that may be supplied to the client with minimum effort upon request.

The process starts with a list of project objectives/requirements that will be used to create the project plan. The customer prioritizes these goals by weighing their worth and cost, which defines the number of iterations and future delivery.

3.3 Code Development

Below I have provided the screenshots of the code used to create the product-brand management system in an orderly manner.

a. DATABASE:

First the database for the product and brand is set. The table for both is created in MySQL.

- setup.sql

```
1  USE zopstore;
2
3  CREATE TABLE brands(
4      id INT NOT NULL AUTO_INCREMENT,
5      name VARCHAR(50) NOT NULL,
6      PRIMARY KEY(id),
7      UNIQUE(name)
8  );
9
10 CREATE TABLE products(
11     id INT NOT NULL AUTO_INCREMENT,
12     name VARCHAR(50) NOT NULL,
13     description VARCHAR(500),
14     price INT NOT NULL ,
15     quantity INT NOT NULL,
16     category VARCHAR(50) NOT NULL,
17     brand_id INT NOT NULL,
18     status enum('Available','Out of Stock', 'Discontinued') NOT NULL,
19     PRIMARY KEY (id),
20     FOREIGN KEY (brand_id) REFERENCES brands (id)
21 );
22
23 INSERT INTO brands VALUES (1,'Amul');
24 INSERT INTO brands VALUES (2,'Cadbury');
25 INSERT INTO brands VALUES (3,'Nestle');
26
27 INSERT INTO products VALUES(1, 'Ghee', 'Amul Ghee 250gm', 150, 250, 'Food', 1, 'Available');
28 INSERT INTO products VALUES(2, 'Chocolate', 'DairyMilk silk 60gm', 100, 0, 'Food', 2, 'Out of Stock');
29 INSERT INTO products VALUES(3, 'Instant Coffee', 'Instant coffee 50gm jar', 170, 50, 'Food', 3, 'Available');
```

Fig 7. Setting up the database

b. THREE-LAYER ARCHITECTURE CODE:

1. Data store layer
2. Service layer
3. Handler layer

1. DATA STORE LAYER

- *productstore.go*

```
httpproduct.go x productservice.go x productstore.go x
1 package product
2
3 import ...
4
10
11 type Store struct { 12 usages ± Oshin
12 }
13
14 func New() *Store {...}
15
16 // GetAllProducts will retrieve the complete database of the output table
17
18 // GetAllProducts(ctx *gofr.Context, brand bool) ([]models.Product, error) {...}
19
46
47 // GetByID will retrieve the data of the given output id
48
49 // GetByID(ctx *gofr.Context, id int, brand bool) (*models.Product, error) {...}
50
69
70 // GetByName will retrieve the data of the given output name
71
72 // GetByName(ctx *gofr.Context, name string, brand bool) ([]models.Product, error) {...}
73
180
181 // Create will add a new product to the database with the given brand ID.
182 // It returns the number of rows affected (should be 1) and any error that occurs
183
184 // Create(ctx *gofr.Context, product *models.Product, brandID int) (int64, error) {...}
185
123
124 // Update will edit the output details for a particular id in the database.
125 // It returns the number of rows affected (should be 1) and any error that occurs.
126
127 // Update(ctx *gofr.Context, id int, product *models.Product, brandID int) (int64, error) {...}
128
145
```

Fig 8. Product datastore layer code

- *brandstore.go*

```
brandservice.go x brandstore.go x
1 package brand
2
3 import ...
4
8
9 type Store struct { 5 usages ± Oshin
10 }
11
12 func New() *Store { 5 usages ± Oshin
13     return &Store{}
14 }
15
16 // GetByID will retrieve the data of the given brand ID.
17 // It returns the number of rows affected (should be 1) and any error that occurs.
18
19 // GetByID(ctx *gofr.Context, id int) (models.Brand, error) { 3 usages ± Oshin
20     brand := models.Brand{}
21
22     row := ctx.DB().QueryRowContext(ctx, query: "SELECT id ,name FROM brandsbrands...", id)
23     err := row.Scan(&brand.ID, &brand.Name)
24
25     if err != nil { models.Brand{}, err }
26
27     return brand, nil
28 }
29
30 // Create will add a new brand to the database with the given name.
31 // It returns the last inserted ID and any error that occurs.
32
33 // Create(ctx *gofr.Context, brand models.Brand) (int64, error) { ± Oshin
34     result, err := ctx.DB().ExecContext(ctx, query: "INSERT INTO brands ...", brand.Name)
35
36     if err != nil { 0, err }
37
38     lastID, _ := result.LastInsertId()
39
40     return lastID, nil
41 }
42
43 // Update will edit the brand details for a particular id in the database.
44 // It returns the number of rows affected (should be 1) and any error that occurs.
45
46 // Update(ctx *gofr.Context, id int, brand models.Brand) (int64, error) {...}
47
```

Fig 9. Brand datastore layer code

2. SERVICE LAYER

- *productservice.go*

```
1 package product
2
3 import ...
4
5
6
7
8
9
10
11
12 // Service is a struct that holds the product store and brand store instances
13 type Service struct {...}
14
15
16
17 // New is a factory function that returns a new Service instance
18 func New(productStore stores.ProductStore, brandStore stores.BrandStore) *Service {...}
19
20
21
22
23 // GetAllProducts is a function that retrieves all products from the product store
24 func (s *Service) GetAllProducts(ctx *gofr.Context, brand bool) ([]models.Product, error) {...}
25
26
27 // GetProductByID is a function that retrieves a product from the product store by ID
28 func (s *Service) GetProductByID(ctx *gofr.Context, id int, brand bool) (*models.Product, error) {...}
29
30
31 // GetProductByName is a function that retrieves a list of products from the product store by name
32 func (s *Service) GetProductByName(ctx *gofr.Context, name string, brand bool) ([]models.Product, error) {...}
33
34
35 // CreateProduct is a function that creates a new product in the product store
36 func (s *Service) CreateProduct(ctx *gofr.Context, product *models.Product) (int64, error) {...}
37
38
39 // UpdateProduct is a function that updates a product in the product store
40 func (s *Service) UpdateProduct(ctx *gofr.Context, id int, product *models.Product) (int64, error) {...}
41
42
43 // checkMissingName is a helper function that checks if a product's name is missing and returns an error if it is
44 func checkMissingName(product *models.Product) error {...}
45
46
```

Fig 10. Product service layer code

- *brandservice.go*

```
1 package brand
2
3 import ...
4
5
6
7
8
9
10
11
12 type Service struct {
13     5 usages ± Oshin
14     store stores.BrandStore
15 }
16
17 func New(brandStore stores.BrandStore) *Service {
18     4 usages ± Oshin
19     return &Service{store: brandStore}
20 }
21
22 func (s *Service) GetBrandByID(ctx *gofr.Context, id int) (models.Brand, error) {
23     2 usages ± Oshin
24     return s.store.GetByID(ctx, id)
25 }
26
27 func (s *Service) CreateBrand(ctx *gofr.Context, brand models.Brand) (int64, error) {
28     2 usages ± Oshin
29     rows, err := s.store.Create(ctx, brand)
30     if rows < 1 || err != nil {
31         return rows, err
32     }
33     return rows, nil
34 }
35
36 func (s *Service) UpdateBrand(ctx *gofr.Context, id int, brand models.Brand) (int64, error) {
37     2 usages ± Oshin
38     rows, err := s.store.Update(ctx, id, brand)
39     if rows < 1 || err != nil {
40         return rows, err
41     }
42     return rows, nil
43 }
44
```

Fig 11. Brand service layer code

3. HANDLER LAYER

- *httpproduct.go*

```
1 package product
2
3 import ...
17
18 type Handler struct { 6 usages ± Oshin
19     svc services.ProductService
20 }
21
22 func New(svc services.ProductService) *Handler {...}
27
28 // Index function handles the request for the Index.
29 // It retrieves parameters from the request context and calls the corresponding service function
30 // to get products by name or all products.
31 // If a name and organization are provided, the name is concatenated with the organization name.
32 func (h *Handler) Index(ctx *gofr.Context) (interface{}, error) {...}
61
62 // Read function handles HTTP GET requests to retrieve a single product by ID.
63 // It extracts the product ID from the URL path parameters and the 'brand' query parameter.
64 // If the 'brand' parameter is set to 'true', the response will include brand information.
65 func (h *Handler) Read(c *gofr.Context) (interface{}, error) {...}
98
99
100 // Create handler creates a new product using the data from the request body
101 func (h *Handler) Create(ctx *gofr.Context) (interface{}, error) {...}
119
120 // Update handler updates a product by retrieving its ID from the path parameter
121 func (h *Handler) Update(ctx *gofr.Context) (interface{}, error) {...}
156
```

Fig 12. Product handler layer code

- *httpbrand.go*

```
1 package brand
2
3 import ...
13
14 type Handler struct { 5 usages ± Oshin
15     svc services.BrandService
16 }
17
18 func New(svc services.BrandService) *Handler { 4 usages ± Oshin
19     return &Handler{
20         svc: svc,
21     }
22 }
23
24 func (h *Handler) Read(ctx *gofr.Context) (interface{}, error) { ± Oshin
25     id, err := strconv.Atoi(ctx.PathParam(key: "id")) // getting the id
26     if err != nil : nil, errors.InvalidParam{} ↗
29
30     result, err := h.svc.GetBrandByID(ctx, id) // calling service
31     if err != nil : nil, errors.EntityNotFound{} ↗
34
35     return result, nil // returning the result
36 }
37
38 func (h *Handler) Create(ctx *gofr.Context) (interface{}, error) {...}
58
59 func (h *Handler) Update(ctx *gofr.Context) (interface{}, error) {...}
94
```

Fig 13. Brand handler layer code

c. MAIN

- main.go: source file

```
package main

import ...

func main() {
    oshin := Oshin{}
    app := gofr.New()

    app.Server.ValidateHeaders = false

    app.Server.UseMiddleware(middleware.Middleware)
    app.Server.UseMiddleware(middleware.WareOrg)

    storeP := productStore.New()
    storeB := brandStore.New()
    serviceP := productService.New(storeP, storeB)
    productHandler := product.New(serviceP)

    app.REST(entity: "product", productHandler)

    store := brandStore.New()
    service := brandService.New(store)
    brandHandler := brand.New(service)

    app.REST(entity: "brand", brandHandler)
    app.Start()
}
```

Fig 14. Source file code

Upon running the main file using:

go run main.go,

- Http server established on port 8000.
- Metric server which is part of the GOFr framework, starts on port 2121.
- Upon hitting the endpoint on port 8080 on Postman, the control transfers to the main and then our middleware for authentication.

d. MIDDLEWARE

- middleware.go:

```
func MiddleWare(handler http.Handler) http.Handler { 2 usages ↕ Oshin
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        head := r.Header.Get(key: "X-API-KEY")
        path := strings.Split(r.URL.String(), sep: "/")[1]
        path = strings.Split(path, sep: "?")[0]

        allowedMethods := map[string][]string{...}

        methods, ok := allowedMethods[head]
        if !ok {...}

        if !contains(methods, r.Method) || (head == "brand-r" && strings.Contains(path, substr: "product")) ||
            (head == "brand-w" && strings.Contains(path, substr: "product")) {...}

        handler.ServeHTTP(w, r)
    })
}

func contains(s []string, e string) bool {...}

func WareOrg(handler http.Handler) http.Handler { 2 usages ↕ Oshin
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        org := r.Header.Get(key: "X-ORG")

        switch r.Method {
        case http.MethodGet:
            url := r.URL
            q := url.Query()
            q.Add(key: "organization", org)
            url.RawQuery = q.Encode()
            r.URL = url

        case http.MethodPost:
            ctx := context.WithValue(r.Context(), constants.CtxValue, org)
            r = r.WithContext(ctx)
        }
    })
}
```

Fig 15. Middleware code

e. SWAGGER DOCUMENTATION

- swagger.yaml and swagger online editor

```

1  openapi: 3.0.3
2  info:
3    title: ZopStore - OpenAPI 3.0
4    description: >-
5      Zopstore will contain the details of all the products and brands in its
6      database store.
7    version: 1.0.11
8  servers:
9    - url: http://localhost:8000
10 tags:
11 - name: Product
12   description: Get Information about products at Zopstore.
13 - name: Brand
14   description: Get Information about brands at Zopstore
15 paths:
16   /product/{id}:
17     get:
18       tags:
19         - Product
20       summary: Get the product details
21       description: Retrieve an existing product information by ID
22       operationId: GetProductByID
23       parameters:
24         - name: id
25           in: path
26           description: enter ID of product to retrieve
27           required: true
28           schema:
29             type: integer
30         - name: brand
31           in: query
32           description: enter "true" to get brand name otherwise "false"
33           required: false
34           schema:
35             type: string
36         - name: X-ORG
37           in: header

```

Fig 16. Swagger yaml file

Product Get Information about products at Zopstore. ^

- GET** /product/{id} Get the product details
- PUT** /product/{id} Update an existing product
- GET** /product Retrieve all products or products by name
- POST** /product Create a new product

Brand Get Information about brands at Zopstore ^

- GET** /brand/{id} Get the brand details
- PUT** /brand/{id} Update an existing Brand
- POST** /brand Create a new brand

Fig 17. Swagger online editor

f. POSTMAN

Product / **GetByID** Save ... ✎ 🗨

GET ▼ `http://localhost:8000/product/2` Send ▼

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

Query Params

Key	Value	Description	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (7) Test Results 🌐 Status: 200 OK Time: 18 ms Size: 560 B 📄 Save as Example ...

Pretty Raw Preview Visualize JSON ▼ 🔍

```

1  |
2  |   "data": {
3  |     "id": 2,
4  |     "name": "Chocolate",
5  |     "description": "Dairy Milk silk 60gm",
6  |     "price": 130,
7  |     "quantity": 60,
8  |     "category": "Food",
9  |     "brand": {
10 |       "id": 2
11 |     },
12 |     "status": "Discontinued"
13 |   }
14 |

```

Fig 18. Sample output for product_GetByID

Brand / **/brand/{id}** Save ... ✎ 🗨

GET ▼ `http://localhost:8000/brand/1` Send ▼

Params Auth Headers (8) Body Pre-req. Tests Settings Cookies

Query Params

Key	Value	Description	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (7) Test Results 🌐 200 OK 4 ms 432 B 📄 Save as Example ...

Pretty Raw Preview Visualize JSON ▼ 🔍

```

1  |
2  |   "data": {
3  |     "id": 1,
4  |     "name": "Amul"
5  |   }
6  |

```

Fig 19. Sample output for brand_Get

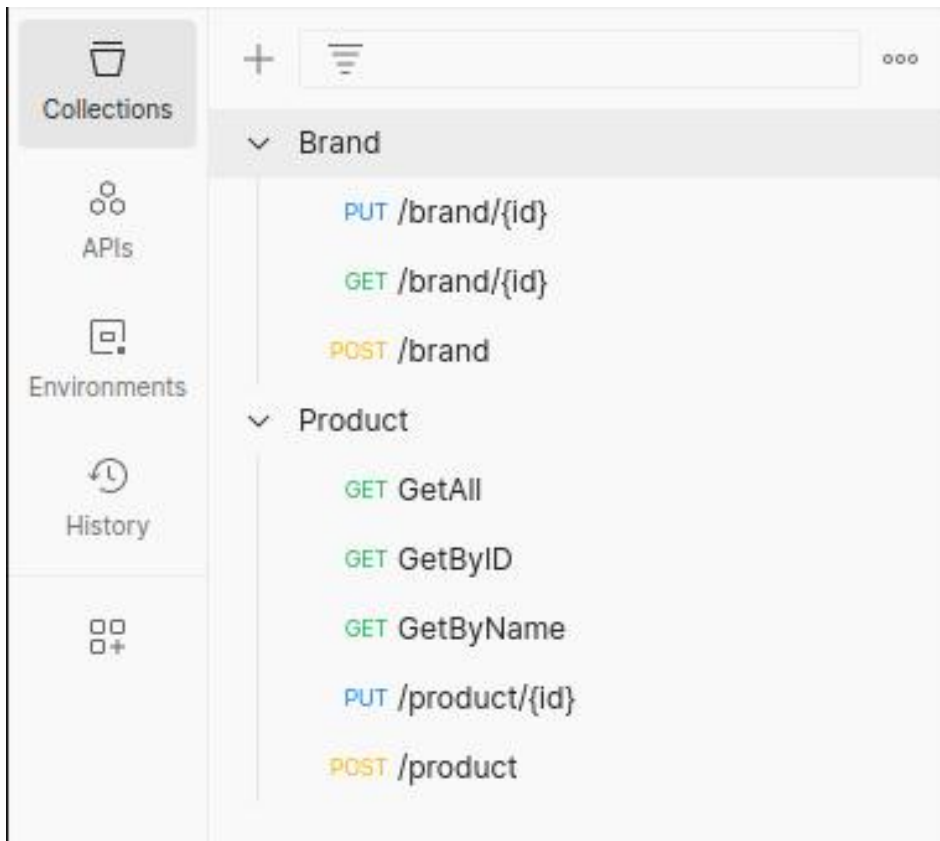


Fig 20. Postman collection

CHAPTER - 4

PERFORMANCE ANALYSIS

4.1 Unit Test Coverage

Ran unit test coverage and discovered that all 32 tests passed, i.e. PASS, with a total coverage of 100%.

```
raramuri@Raramuri:~/go-dailiy-assignment/ZopStore$ go tool cover -func coverage.out
github.com/Zopsmart-Training/go-dailiy-assignment/internal/http/brand/httpbrand.go:18:      New      100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/http/brand/httpbrand.go:24:      Read      100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/http/brand/httpbrand.go:38:      Create    100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/http/brand/httpbrand.go:59:      Update    100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/http/product/httpproduct.go:22:    New      100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/http/product/httpproduct.go:32:    Index    100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/http/product/httpproduct.go:65:    Read      100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/http/product/httpproduct.go:92:    Create    100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/http/product/httpproduct.go:121:   Update    100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/services/brand/brandservice.go:14:   New      100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/services/brand/brandservice.go:18:   GetBrandByID 100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/services/brand/brandservice.go:22:   CreateBrand 100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/services/brand/brandservice.go:31:   UpdateBrand 100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/services/product/productservice.go:19:   New      100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/services/product/productservice.go:27:   GetAllProducts 100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/services/product/productservice.go:32:   GetProductByID 100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/services/product/productservice.go:37:   GetProductByName 100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/services/product/productservice.go:42:   CreateProduct 100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/services/product/productservice.go:57:   UpdateProduct 100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/services/product/productservice.go:69:   checkMissingName 100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/stores/brand/brandstore.go:12:    New      100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/stores/brand/brandstore.go:16:    GetByID    100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/stores/brand/brandstore.go:29:    Create    100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/stores/brand/brandstore.go:41:    Update    100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/stores/product/productstore.go:14:    New      100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/stores/product/productstore.go:19:    GetAllProducts 100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/stores/product/productstore.go:48:    GetByID    100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/stores/product/productstore.go:71:    GetByName  100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/stores/product/productstore.go:103:   Create    100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/internal/stores/product/productstore.go:126:   Update    100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/middleware/middleware.go:12:           Middleware 100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/middleware/middleware.go:41:           contains  100.0%
github.com/Zopsmart-Training/go-dailiy-assignment/middleware/middleware.go:51:           WareOrg   100.0%
total:                                         (statements) 100.0%
raramuri@Raramuri:~/go-dailiy-assignment/ZopStore$
```

Fig 21. Unit Test Coverage Check

4.2 Linter Check

A linter check was performed using the command ‘golanci-lint run’, which ensures that the program is properly structured and follows standard code rules such as no gocognit complexity or funlen to be 0 and so on. In this project, no linter mistakes were discovered.

```
raramuri@Raramuri:~/go-dailiy-assignment/ZopStore$ golanci-lint run
raramuri@Raramuri:~/go-dailiy-assignment/ZopStore$
```

Fig 22. Linters check

CHAPTER - 5

CONCLUSIONS

5.1 Results Achieved

Throughout the course, I successfully learned and applied the principles of GoLang, MySQL, and unit testing. The primary goal was to obtain practical skills in these technologies, which I did by creating a web application that performs basic CRUD operations. To ensure a well-structured and maintained codebase, I created the application using a three-layered architecture. I also used Postman for efficient testing and verification of the application's functioning.

5.2 Application Contributions

GoLang has become a popular language in both open-source and commercial applications. It has various benefits, including concurrency support, robust typing, and fast performance. Among the major examples of GoLang usage in the business are:

- **Docker:** Docker, a containerization platform, is built with GoLang as one of its major components. Because of its simplicity, speed, and compatibility with Linux containers, the language is an excellent candidate for Docker's infrastructure.
- **Swagger:** Swagger, a popular framework for designing, implementing, and documenting RESTful APIs, has included GoLang in its ecosystem. The efficiency of GoLang and its ability to handle concurrent queries contribute to Swagger's great speed.
- **Postman:** GoLang support has been added to Postman, a popular API testing and development tool. GoLang's simplicity and speed make it ideal for effectively processing API requests and answers.

5.3 Limitations

While the built program executes backend operations correctly, it needs a frontend component. To improve the entire user experience, a user-friendly and visually appealing front-end is required. As a result, future work should focus on creating and implementing a front-end interface to supplement the existing backend capabilities.

5.4 Future Work / Scope

The following areas for future improvement can be addressed in order to further improve the application:

1. **Frontend Development:** The inclusion of a frontend component would substantially improve the application's usability and appearance. This includes creating user interfaces that are easy to use, adding interactive elements, and maintaining a consistent user experience.

2. **Functionality Expansion:** Additional features can be introduced to make the application more comprehensive and feature-rich. Advanced CRUD operations, user authentication and authorization, data validation, and interaction with external services or APIs are all possible.

The program may be developed into a more robust and user-friendly solution by focusing on these areas of future growth.

REFERENCES

- [1] About Zopsmart Technologies from ZopSmart.com <https://zopsmart.com>
- [2] *Documentation Go*. Available at: <https://go.dev/doc/>
- [3] *Gomock gomock package - github.com/golang/mock/gomock - Go Packages*.
Available at: <https://pkg.go.dev/github.com/golang/mock/gomock>
- [4] *Swagger* <https://swagger.io/>
- [5] *MySQL documentation MySQL*. Available at: <https://dev.mysql.com/doc/>
- [6] *Docker*. Available at: https://hub.docker.com/_/docker-docs
- [7] Postman (2023) *Overview, Postman Learning Center*. Available at: <https://learning.postman.com/docs/introduction/overview/>