WEB APPLICATION IN GO USING THREE LAYER ARCHITECTURE

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

in

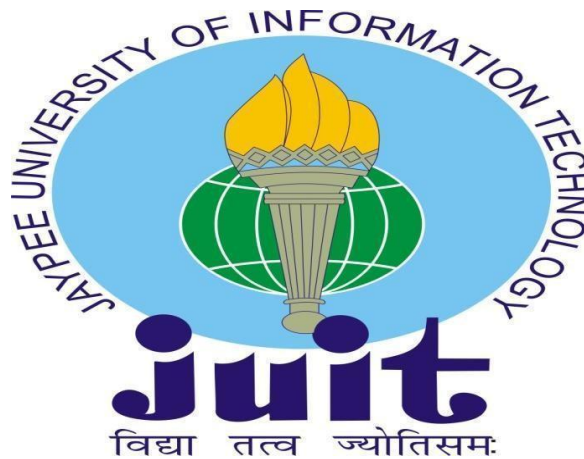**Computer Science and Engineering/Information Technology**

By

Anubhav Garg (191321)

Under the supervision of

Dr. Shubham Goel
Assistant Professor, CSE

to

Department of Computer Science & Engineering and Information Technology
Jaypee University of Information Technology Waknaghat, Solan-173234, Himachal Pradesh

CERTIFICATE

I hereby declare that the work presented in this report entitled "Web application in Go using Three Layer 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 Waknaghat is an authentic record of my own work carried out over a period from February 2023 to May 2023 under the supervision of Dr. Shubham Goel, Assistant Professor Computer Science and Engineering/ Information Department. The matter embodied in the report has not been submitted for the award of any other degree or diploma.

AnuGhav

Anubhav Garg
191321, CSE 4th Year

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

Dr. Shubham Goel
Assistant Professor
Computer Science and Engineering/ Information Department

Mr. Akshat Kumar Singhal
Director of Engineering
Zopsmart

I

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)	
	• 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

To begin, I would like to express our heartfelt gratitude to almighty God for his heavenly grace, which enabled me to successfully complete the project work.

I am extremely grateful and wish to express my deep gratitude to Supervisor **Dr. Shubham Goel, Assistant Professor,** Department of CSE & IT Jaypee University of Information Technology, Waknaghat. His never-ending patience, intellectual direction, persistent encouragement, constant and vigorous supervision, constructive criticism, helpful suggestions, and reading numerous versions and revising them at all stages allowed this project to be completed. I would like to express my heartiest gratitude to Dr. Shubham Goel**,** Department of CSE & IT, for his kind help to finish this project.

I would also like to express my gratitude to everyone who has assisted in making this project a success, whether directly or indirectly. In this unusual scenario, I'd like to express my gratitude to the different staff members, both teaching and non-teaching, who have provided valuable assistance and assisted the project. Finally, I must express my gratitude to my parents' unwavering support, patience and guidance.

Anubhav Garg
191321, CSE 4th Year

TABLE OF CONTENTS

LIST OF FIGURES/TABLES

ABSTRACT

According to a survey, a total of 5.18 billion people around the world were using the internet at the start of Q2 2023, which is equivalent to 64.6 percent of the world's total population. We use web-based applications on a daily basis, as it enables us to accomplish various business tasks.

But why now, in a mobile-everything era, companies launch web-based applications. The answer is simple, to acquire more users, as web applications do not require any installation. Hence, web apps make up the majority of modern web pages. We use web applications every day, maybe without even noticing these are actually web apps:

- Webmail services (Microsoft Outlook, Gmail)
- Online calendars (Google Calendar)
- eCommerce products (Shopify)
- Online marketplaces (Amazon)

Fig. 1 Client-Server Architecture
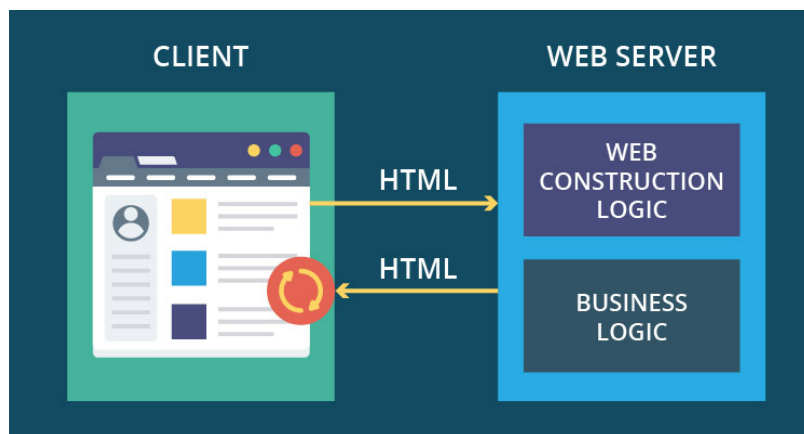
There are two components to a web app:

- **Front-end:** Users can see and interact with the user interfaces. Developed by front-end developers using HTML and CSS, with JavaScript to create a sophisticated, interactive UI.
- **Back-end:** Connect & exchange data with servers, databases, third-party APIs, which is achieved by back-end developers using Java, Node.js, Python, Go, etc.

VII

Chapter 01: INTRODUCTION

1.1 Introduction

1.1.1 Introduction of the company

ZopSmart Technology is a software solution firm that offers you all the resources you need to launch an online store. With the support of its product line, ZopSmart can help you create and manage the ideal company. It offers a variety of goods, including Smart Store Eazy and Smart Payment Gateway, among others. Zopsmart develops cutting-edge technology for the retail industry. Their clients range from independent furniture stores to large multinational chains, and their solutions include e-commerce platforms, digital marketing, mobile commerce, automated logistics systems, management platforms, order management platforms, and internet of things (IoT) devices. It has its own framework to work on and offers software solutions to some of the best companies.

Zopsmart Technology offers the tools and guidance needed to set up an online store, assisting offline businesses who want to expand online. Their objective is to provide mature, end-to-end products that add value to digitally savvy customers. They demonstrate to businesses how they may more effectively engage their customers, develop within set budgets, and launch their products online as soon as possible. They assist customers in reimagining their businesses, consumer interactions, and tested analytics through human-centered design methodologies with a shortened time-to-market.

1.1.2 Introduction of the project

The need for creating online apps that provide seamless user experiences has expanded significantly in recent years. Designing a well-structured API that interacts with the database and executes CRUD (Create, Read, Update, Delete) actions is necessary for creating a robust and scalable application. The performance and simplicity of the popular programming language Golang have attracted a lot of interest from the software development world.

The goal of this project is to provide a CRUD API in Golang utilizing a three-layered design that adheres to best practices. A design pattern known as three-layered architecture separates the handler layer from the service and store layer, improving the API's maintainability and scalability. When interacting with API endpoints, the handler

layer transmits requests to the service layer, which processes them and generates responses. The CRUD activities and database communication are handled by the store layer.

We will follow test-driven development (TDD) principles to ensure code quality, with unit tests and integration tests covering all critical components of the API. The API will be developed to be RESTful, adhering to the principles of HTTP methods, and will provide endpoints for creating, retrieving, updating, and deleting data from the database.

1.2.1 Introduction to API

"Application Programming Interface" is what API stands for. It is a collection of rules, procedures, and resources that enables information sharing and communication between various software programmes.

To put it simply, an API serves as a link between several software programmes, enabling communication and data sharing. REST APIs, SOAP APIs, and GraphQL APIs are just a few examples of the many ways that APIs can be implemented. Modern software development is impossible without APIs, which let programmers create sophisticated, complicated programmes that connect with other software services and systems.

RestAPI with Go

Representational State Transfer, sometimes known as REST, is an architectural design for distributed hypermedia systems.

We can streamline the overall system architecture and enhance interaction visibility by applying the generality principle to the component interface.

Several architectural restrictions aid in achieving a unified interface and regulating component behavior. A REST interface can be achieved by the following restrictions:

1. Each resource used in the interaction between the client and the server must be uniquely identified by the interface.

2. Resources should be represented consistently in the server response to prevent resource manipulation. These representations should be used by API consumers to alter the server's state of the resources.

3. Each resource representation should have enough details to explain how to interpret the message. It should also detail any extra operations the client may carry out on the resource.

4. The client should only have the application's starting URL when using hypermedia as the application state engine. All other resources and interactions should be dynamically driven by the client application using hyperlinks

Hyper-Text Transfer Protocol

The HTTP protocol is used to retrieve resources like HTML documents. It is a client-server protocol, meaning requests are made by the recipient, which is often the Web browser, and it serves as the basis for all data communication on the Internet. The various sub-documents that are retrieved, such as text, layout descriptions, photos, videos, scripts, and more, are combined to form a complete document.
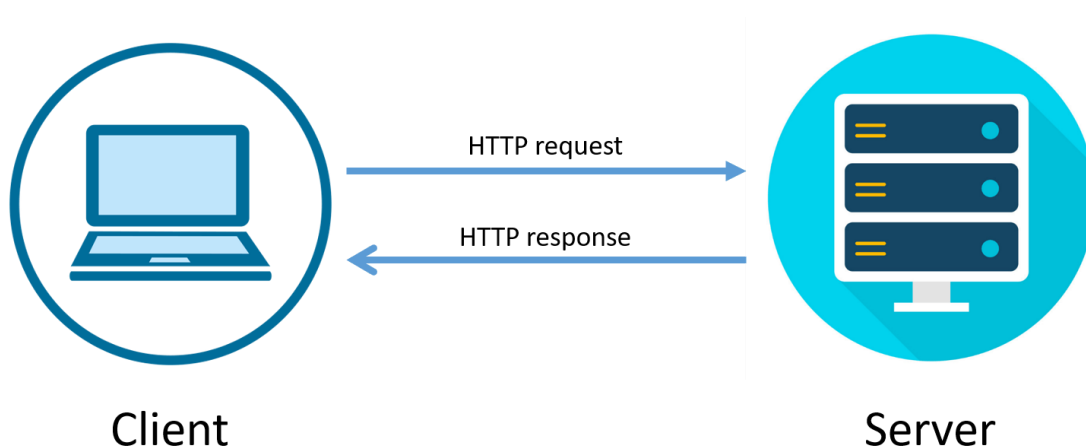
Fig. 1.1 Making request to a server

Clients and servers communicate by exchanging individual messages (as opposed to a stream of data). The messages sent by the client, usually a Web browser, are called *requests* and the messages sent by the server as an answer are called *responses*.

Status Codes

There are common status codes that are defined by HTTP and can be used to communicate a client's request's outcome. Five categories make up the status codes.

- **1xx: Informative -** Transfers information at the protocol level.
- **2xx: Success -** Represents a successful acceptance of the client's request.
- **3xx: Redirection -** Denotes that the client needs to take further action to fulfill their request.
- **4xx: Client Error -** Clients are blamed in this group of error status numbers.

3

● **5xx: Server Error -** Server's fault.

Some important status codes are :

S.NO	STATUS CODE	DESCRIPTION
1	200 OK	REQUEST SUCCEEDED
2	201 CREATED	NEW RESOURCES CREATED
3	400 BAD REQUEST	INCORRECT SYNTAX
4	401 UNAUTHORIZED	AUTHENTICATION INFORMATION REQUIRED
5	403 FORBIDDEN	CLIENT DOES NOT HAVE ACCESS
6	404 NOT FOUND	REQUESTED RESOURCE NOT FOUND
7	500 INTERNAL SERVER ERROR	SERVER ENCOUNTERED UNEXPECTED CONDITION

Table 1.1 HTTP Status Codes

CRUD OPERATIONS

Create, Read, Update, and Delete, or CRUD, are the fundamental data management activities used in databases and APIs. By performing these actions, database engineers can modify data in a database by adding new records, retrieving old ones, updating old ones, and deleting old ones.

1. **Create:** Adding a new record to the database is done using this operation. For instance, the information from the form is saved as a new record in the database when you submit one to create a new account.

2. **Read:** Data from the database is retrieved using this procedure. For instance, the product information is pulled from the database and shown when you view a product page on an e-commerce website.

3. **Update:** This operation modifies the database's current data. For instance, the database is updated when you make changes to your profile information on a social media website.

4. **Delete:** The database can be cleaned up by using this action. For instance, a message gets destroyed from the database when you delete it from your inbox.

In order to create APIs and database-driven apps, CRUD procedures are essential since they let programmers manage data effectively and efficiently.

Golang

Programming in Go is procedural. Robert Griesemer, Rob Pike, and Ken Thompson at Google started working on it in 2007, but it wasn't released as an open-source programming language until 2009. Packages are used in the assembly of programmes to manage dependencies effectively. Like dynamic languages, this language also supports environment adoption patterns.

Go is a robust system-level programming language used to create large-scale distributed systems and network servers. It has been successful as a Java and C++ substitute. GoLang's syntax is considerably more like C while using fewer brackets, commas, and other symbols, making it more like Python as well. GoLang offers concurrency, multithreading, easy versioning, concurrency, quick compilation and execution, consistent language, creating with many languages, and easier maintenance.

Go programming language is developed by Robert Griesemer, Rob Pike, and Ken Thompson at Google. It is compiled and statically typed. Go is syntactically identical to C despite having memory safety, garbage collection, structural typing, and concurrency in the CSP tradition.

It's relatively simple to create a Web application in Go utilising a three-layered architecture. The difficulty arises when the code must be thoroughly testable, organised, tidy, and maintainable. Delivery, use-case, and datastore are the three separate layers. The request will be received by the delivery layer, which will then parse it for any necessary information. The response is then written to the response writer after calling the use case layer to check that it is in the correct format. The application's business logic is handled by the use case layer. This layer and the datastore layer will communicate. It calls the datastore layer after taking what it requires from the delivery layer. It applies the necessary business logic both before and after calling the datastore layer. The data is kept in the datastore. Any data storage device can be used. The only layer that interacts with the

datastore is the use case layer. In this manner, each layer can be checked independently from the others.

Key Features of Golang:
- Go is simple to learn and use because of this. Its clear syntax makes it a viable option for both inexperienced and seasoned programmers.
- Go features built-in concurrency support, enabling programmers to create scalable and effective code for multicore and distributed applications.
- Go uses automatic memory management, developers are relieved of the responsibility of managing memory allocation and deallocation.
- Rapid iteration during development is made possible by Go's short compilation times.
- Go can be built to work on a wide range of operating systems, including Windows, Linux, and macOS.
- Go is a statically typed language, therefore errors are caught at compile time rather than during runtime.

GO Basics
1. **Packages:** There are packages in every Go programme. In the package main, programmes begin to run. A package is a container that has different functions to carry out particular tasks. For instance, the math package offers the Sqrt() function to calculate a number's square root.
2. **Imports:** The import keyword imports the specified package from the provided directory or, if no path is specified, from the directory of $GOPATH. Importing merely entails moving the designated package from its source directory to the destination code, which is the main programme. In Go, import is crucial since it brings the packages that are absolutely needed to run programmes.
3. **Functions:** Functions are typically the set of instructions or statements in a programme that permit the user to reuse previously written code in order to save memory usage, speed up processing, and—most importantly—improve readability. In essence, a function is a group of statements that work together to complete a particular task and return the outcome to the caller. A function may also carry out a particular activity without producing a result.

4. **Named Return Values:** The return values of Go may have names. If so, they are handled as variables set out at the function's top.The definition of the return values should be documented using these names.The named return values are returned by a return statement without arguments. It's referred to as a "naked" return.Naked return statements should only be used in brief functions, like the one in this example. They may make lengthier functions harder to read.

5. **Shorthand variable declarations:** The **:=** short assignment statement may be used inside of a function in place of a var declaration with implicit type. The **:=** construct is not available outside of a function because every sentence starts with a keyword (var, func, and so forth).

6. **Zero Values:** Variables that are declared without a clear starting value are assigned zero.For numeric types, the zero value is 0; for boolean types, false; and for string types, "" (the empty string).

7. **Type Inference:** A variable's type is inferred from the value on the right side when it is declared without an explicit type being specified (either by using the := syntax or the var = expression syntax).

8. **For is Go's "While":**The while loop in Go is used to run a block of code until a specific condition is satisfied. In contrast to most programming languages, Go lacks a specific keyword for a while loop. However, we can mimic the functionality of a while loop using the for loop.

9. **Defer:** With a defer statement, a function's execution is postponed until the surrounding function completes its run.The inputs of the postponed call are assessed right away, but the function call is not carried out until the surrounding function has finished. Function calls that are deferred are pushed into a stack. Deferred calls are processed in last-in, first-out order when a function returns.

10. **Slices:** An array's size is fixed. On the other hand, a slice is a flexible, dynamic look into an array's components. Slices are far more prevalent than arrays in actual use.A slice with elements of type T is a type []T.

    There are various ways to make a slice in Go:

      ● Create a slice from an array using the []data type values format.
      ● The make() method is utilized

11. **Slice as reference to an array:** A slice only depicts a portion of the underlying array; it does not actually contain any data. A slice's underlying array's associated

elements are changed when the elements of the slice are changed. These modifications will be visible to other slices that use the same underlying array.

12. **Length and Capacity:** A slice has a capacity as well as a length. A slice's length is determined by how many elements it has. Counting from the first element in the slice, the capacity of a slice is the total number of elements in the underlying array. The expressions len(s) and cap(s) can be used to determine the length and capacity of a slice s.

Three Layered Architecture

The three main layers are:
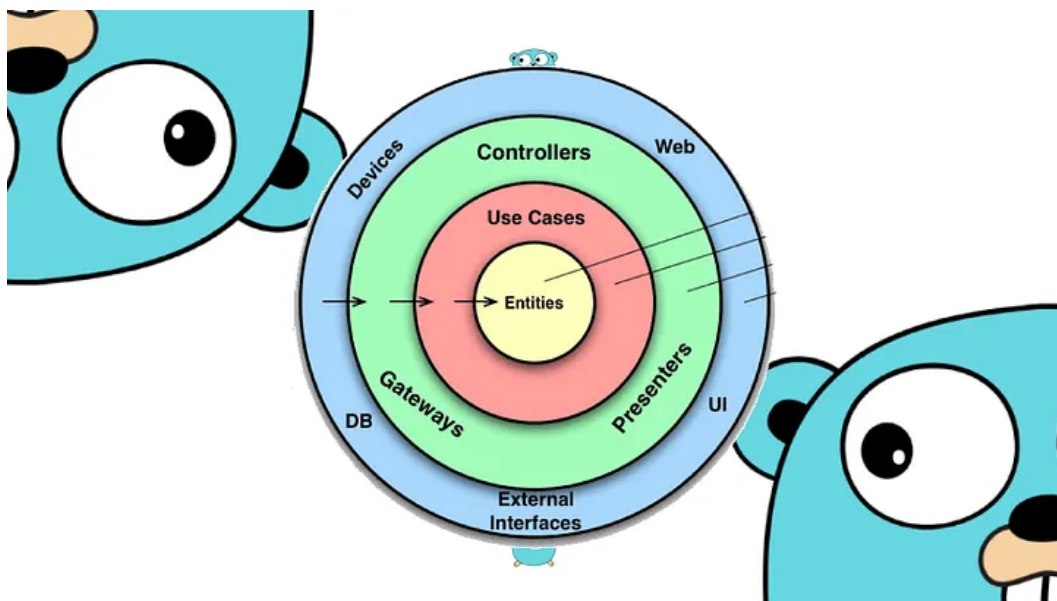
- Handler layer
- Service layer
- Store layer

Fig. 1.2 Three Layered Architecture

Handler Layer

The handler layer is also known as the delivery layer. The request will be received by the delivery layer, which will then parse it for any necessary information. The response is then written to the response writer after calling the use case layer to check that it is in the correct format.

Service Layer

It is also known as the Use-Case Layer. The application's business logic is handled by the use case layer. This layer and the datastore layer will communicate. It calls the datastore layer after taking what it requires from the delivery layer. It applies the necessary business logic both before and after calling the datastore layer.

Store Layer

It is also known as the Data-store layer. The data is kept in the datastore. Any data storage device can be used. The only layer that interacts with the datastore is the use case layer. In this manner, each layer can be checked independently from the others.

Since each layer operates independently from the others, only the delivery layer will change if the application develops to support gRPC. The datastore and use case layer won't change. The complete application need not change even if the datastore changes. It will only change at the datastore layer. This makes it simple to maintain the code, find and fix errors, and expand the programme.

In our project the store layer will be using MySQL to store and retrieve the data.

MySQL

MySQL is simply a database management system. A database is a planned collection of data. Anything might be it, from a simple grocery list to a photo gallery to the vast amount of information in a company network. Data included in a computer database must be added to, accessed, and processed using a database management system, such as MySQL Server. Computers are good at processing enormous volumes of data, thus database management systems—whether used as standalone programmes or as a component of other programs—are crucial to computing.

A relational database stores the data in separate tables rather than consolidating it into one enormous warehouse. The database structures are stored in physical files that are speed-optimized. A flexible programming environment is offered by the logical model, which includes objects like databases, tables, views, rows, and columns. You could create rules to govern the relationships between different data fields, such as one-to-one, one-to-many, unique, compulsory or optional, and "pointers" between different tables. Since a well-designed database upholds these constraints, your application won't ever run into inconsistent, duplicate, orphan, out-of-date, or missing data.

The SQL part of "MySQL" stands for "Structured Query Language". SQL is the most widely used standard language for database access. You might explicitly enter SQL (for instance, to generate reports), embed SQL statements into other languages' code, or use a language-specific API that hides the SQL syntax depending on your programming environment.

1.2 Problem Statement

To Create a CRUD API using three layered architecture in golang.

Customers API

- GET /customers/{id} - Get a customer by ID
- POST /customers - Create a new customer
- PUT /customers/{id} - Update an existing customer by ID
- DELETE /customers/{id} - Delete a customer by ID
- GET /customers - Get all customers
- GET /customers?vehicle=true - Get all customers with their vehicle details
- GET /customers?fuel_type={fuel_type} - Get all customers with vehicles that have a specific fuel type
- GET /customers?brand={brand} - Get all customers with vehicles that have a specific brand

Vehicles API

- POST /vehicles - Create a new vehicle
- PUT /vehicles/{id} - Update an existing vehicle by ID
- DELETE /vehicles/{id} - Delete a vehicle by ID

Use MySQL connection to access the tables. The name of the database should be ZopStore. There should be two tables Customers and Vehicles. For Customers tables the id should be uuid, age should be between zero to 100, gender should be enum, phone number should begin with code 91, and length should not be greater than 12.- For vehicle table the is should be uuid, type should be enum, and fuel_type should be enum too.

Filters are given as follows:

a. if is vehicle true get all vehicle details for that particular customer

b. Get vehicles based on fuel_type

c. Get vehicles based on brand

Naming conventions should be snake_case only for DB naming convention and camelCase in the code.

1.3 Objectives

- To create testable, structured, clean and maintainable CRUD API by using industrial best practices.
- To understand Golang basic and advanced concepts.

1.4 Methodology

- Learn go basics
- Learn basic of mysql connection to golang
- Create tables in database
- Create the layers in Goland IDE by JetBrains.
- Write the test cases for all the layers using the Test Driven Development
- Write the function implementation in all the layers.
- Perform Dependency Injection

About Goland IDE

It is a Golang IDE provided by JetBrains. com. It provides support to create a go program on our local system, run it, debug it and many other functionalities. We can create go projects using Goland IDE.
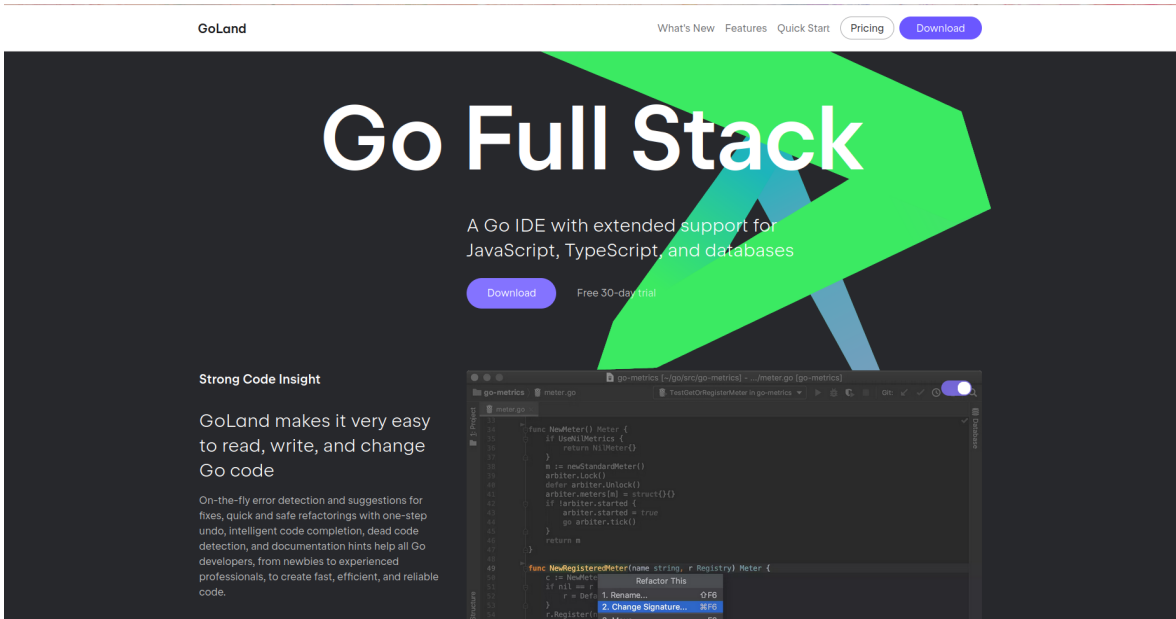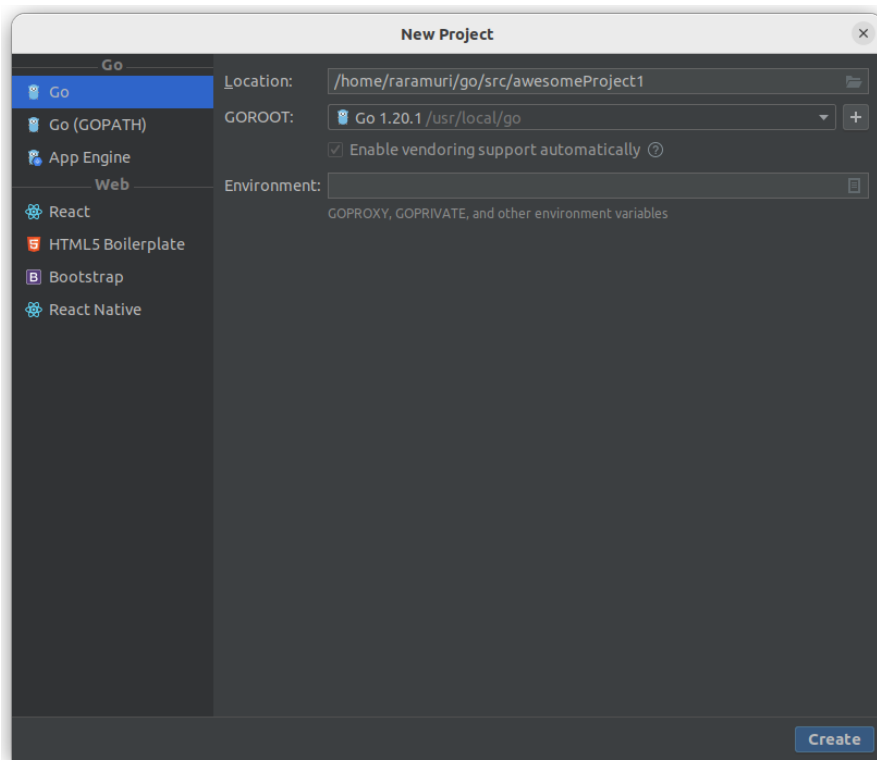
Fig. 1.3 Goland IDE by JetBrains

Fig. 1.4 Creating a New Project in Goland

Test Driven Development(TDD)

Test cases rather than the code that verifies them are written first under the "test driven development" methodology. Repetition of a brief development cycle is necessary. Automated unit tests are used in test driven development to guide design and unrestricted decoupling of dependencies.

The followings steps are involved in TDD approach:

- Add a test by creating a test case that fully explains the method. The developer must comprehend the features and requirements utilizing user stories and use cases in order to create the test cases.
- Make sure the new test case fails by running all the test cases.
- Make sure your code satisfies the test case.
- Execute the test scenarios.
- Code duplication is eliminated by refactoring the code.
- Repetition of the steps described above

Advantages of TDD

1. Continuous feedback about the functions is provided through unit tests.
2. Design quality improves, which also aids in good maintenance.
3. The use of test-driven development provides a safeguard against bugs.
4. TDD ensures that your application satisfies the requirements that have been set down for it.
5. The development lifecycle of TDD is quite brief.

Dependency Injection

Dependency injection, sometimes known as DI, is a technique that takes use of code decoupling by utilizing some of the greatest programming practices. Code segments that don't make sense together should either reside independently or be separated. To create this dependence, code injection is required. This is a very basic explanation of dependency injection.
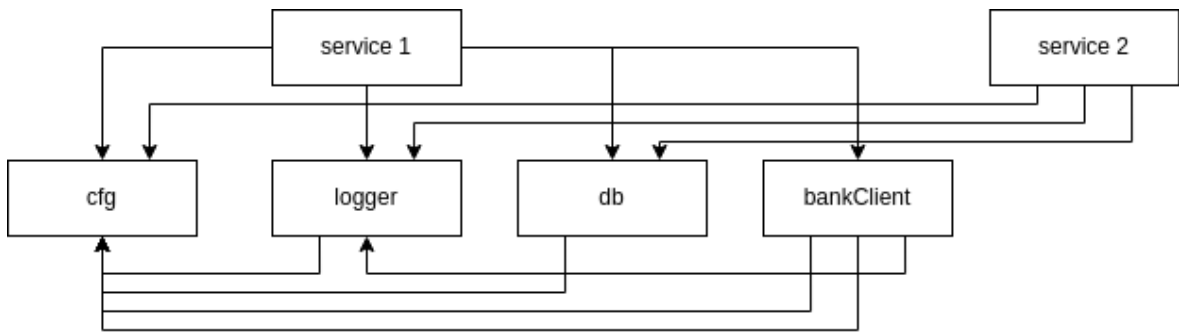
Fig. 1.5 Dependency injection

A design pattern called dependency injection can help you separate your implementation's external logic from it. An external API, database, etc., are frequently required for an implementation. The implementation should receive its dependencies and use them as necessary; it is not the implementation's job to be aware of these details. Consider the case when you have an implementation with the requirements listed below.

Avoiding injecting implementations (structs), and instead injecting abstractions (interfaces), is a crucial aspect of injecting dependencies. It enables you to change the implementation of specific dependencies quickly and to transition from the true implementation to a mock one. Unit testing depends on it fundamentally.

1.5 Organization

The report is organized in five chapters. All the chapters constitute important information regarding the project. First chapter consists of the basic introduction of the project, which is CRUD OPERATIONS, GOLANG & THREE LAYERED ARCHITECTURE. Second chapter has the literature survey of all the research papers we have used, we studied in the making of this project.Third chapter has the information about System development and the usage of the different softwares, database, feasibility study of the project, limitations, etc. The fourth chapter has the performance analysis. Chapter five has the conclusions of the API project and future work in the project .

Chapter 02: LITERATURE SURVEY

1. **GO Documentation** Designed at Google by R Griesemer, R Pike, and K Thompson, go is a statically typed, open source, compiled programming language. Go documentation has information about each and every topic listed or being used in golang

2. **MySQL Documentation** MySQL i.e My Structured Query Language is an open-source relational database management system that helps us to store data, fetch details, delete an entry etc.

3. **GoMock** is a mocking framework for the GO programming language and is used to integrate well with Go's built-in testing package.

4. **Git and Github Official documentation** that familiarizes you with the concepts of a version control system i.e Git and how it works with GitHub.

5. **Three Layered Architecture Industrial Documentation** on three layered architecture used in ZopSmart Technology by Milthali R. Shetty

6. **HTTP Documentation** about using status codes, client server architecture, requests , responses etc.

7. **SQL Mocks Documentation** inside the go documentation.

8. **Dedicated Training and upskilling platform** provided by the company.

Chapter 03: SYSTEM DESIGN & DEVELOPMENT

3.1 Analysis

Feasibility Study: With Create, Read, Update, and Delete operations made possible using an HTTP-based API, this project seeks to offer a straightforward yet effective method for managing data. The three-layered architecture concept will simplify future maintenance and growth while allowing for a distinct separation of issues.

In this feasibility study, the following project components will be examined:

- Technological feasibility
- Operational feasibility
- Financial feasibility

Technological feasibility: The project entails creating a CRUD API in the programming language Golang, which is well-suited for creating scalable and effective applications. Golang is a mature and widely-used programming language. Golang is a strong standard library that features built-in support for HTTP servers and clients, making it perfect for developing online applications. With the suggested three-layered design, it will be simpler to manage the codebase and maintain the application over time because there would be a clear separation of concerns.

Operational feasibility: The RESTful architecture of the CRUD API, which adheres to well-established norms and standards, makes it simple to use and integrate with other systems. Users will be able to quickly generate HTTP requests for the API to create, read, update, and delete data using standard libraries or API clients. The suggested architecture will also make future maintenance and expansion easier because it makes it simple to replace specific parts or add new levels.

Financial feasibility: The financial viability of the project will be influenced by a number of variables, including development costs, hosting costs, and income potential. The complexity of the application and the team's level of expertise will determine the development costs.

3.2 REQUIREMENTS

The capability of the system to fulfill the conditions sought by the users is known as a system requirement.

By dividing the needs into functional and nonfunctional requirements, system requirement analysis is accomplished.

Functional requirements

1. **Authentication**
   - User authentication and authorization should be supported via the API.
   - Users ought to be able to make accounts and sign in with them.
   - User roles and permissions ought to be established and upheld.

2. **Data Management**
   - The Create, Read, Update, and Delete operations of the API should be able to manage data.
   - In a database or file system, data ought to be kept.
   - Data inputs should be verified and consistency monitored by the API.

3. **Data Filtering & Sorting**
   - The API should support sorting and filtering of data according to various criteria.
   - Users should have the option to query data using particular criteria.

4. **Error Handling**
   - The errors should be handled hasslefree and the status codes returned should be correct for all the edge cases.
   - Error messages should be accurate

5. **Testing**
   - API functions should pass all the test cases with 100 percent coverage.
   - Mocks should be used properly and dependency injection should be done for unit testing for all the layers.

6. **Documentation**
   - Documentation should be correct and sorted for all the functions including the correct naming conventions.

Non-Functional Requirements

1. Scalability
2. Performance
3. Readability
4. Reliability
5. Usable API
6. Maintainability

Technical Requirements

1. An IDE for writing clean code is called GOLand.
2. Postman, an API development and usage platform.
3. Mysql server offers connectivity and querying capabilities for database administration systems.
4. Swagger for API documentation
5. Git version control

Hardware Configurations

HP HP EliteBook 840 G7 Notebook PC
Memory 16.0 GiB
Disk Capacity 512 GB
Monitor 13''
Mouse
Keyboard

Table 3.1 Hardware Configurations

Software Configurations

Operating System Ubuntu
Language GO
Runtime environment GO runtime
Package Manager GO

Table 3.2 Software Configurations

3.3 IMPLEMENTATION

Project structure

1. **Project Folder Name :** VehicleStore
2. **SubFolders**
    -   **api:** Contains the swagger API documentation
    -   **driver:** MySql connection file.
    -   **entities:** go files having structures
    -   **handler:** Handler layer files.
    -   **service:** Service layer files
    -   **store:** Store layer files
    -   **postmanCollection:** Postman Collection for the API
3. **main.go**
4. **go.mod:** contains the dependencies
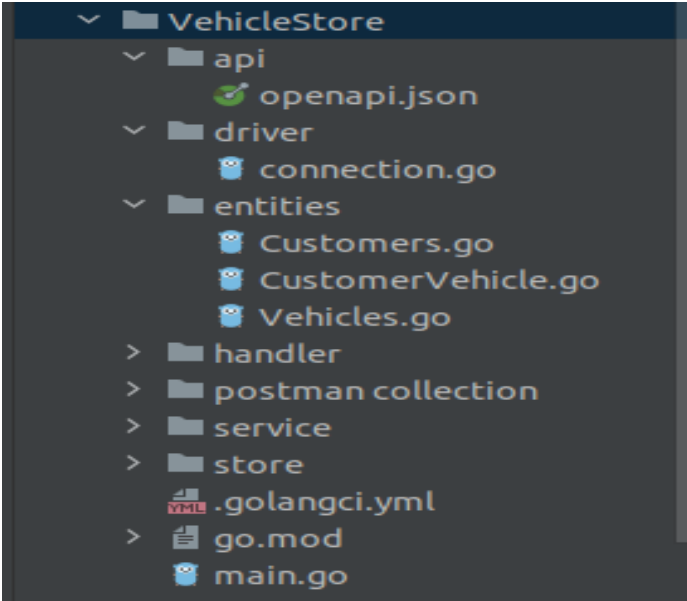
Fig. 3.1 Project Structure
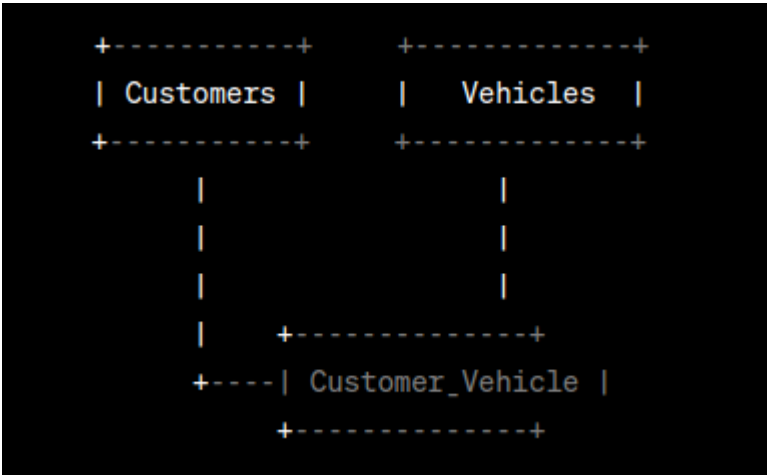
Fig. 3.2 DB SCHEMA

```
sh-4.4# mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 9
Server version: 8.0.31 MySQL Community Server - GPL

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

Fig. 3.3 MySQL docker image

```
mysql> use zopstore;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+--------------------+
| Tables_in_zopstore |
+--------------------+
| Customers          |
| Vehicles           |
+--------------------+
2 rows in set (0.01 sec)
```

Fig. 3.4 Tables in the Database

```
mysql> desc Customers;
+--------------+-------------------------------+------+-----+---------+-------+
| Field        | Type                          | Null | Key | Default | Extra |
+--------------+-------------------------------+------+-----+---------+-------+
| id           | varchar(255)                  | NO   | PRI | NULL    |       |
| name         | varchar(15)                   | NO   |     | NULL    |       |
| age          | int                           | YES  |     | NULL    |       |
| gender       | enum('male','female','others')| YES  |     | NULL    |       |
| phone_number | bigint                        | YES  |     | NULL    |       |
| city         | varchar(15)                   | YES  |     | NULL    |       |
| vehicle_id   | varchar(255)                  | YES  |     | NULL    |       |
+--------------+-------------------------------+------+-----+---------+-------+
7 rows in set (0.00 sec)
```

Fig. 3.5 Customer Description

```
mysql> desc Vehicles;
+-----------+-----------------------------------------+------+-----+---------+-------+
| Field     | Type                                    | Null | Key | Default | Extra |
+-----------+-----------------------------------------+------+-----+---------+-------+
| id        | varchar(255)                            | NO   | PRI | NULL    |       |
| type      | enum('2','4','6')                       | YES  |     | NULL    |       |
| fuel_type | enum('petrol','diesel','cng','electric')| YES  |     | NULL    |       |
| brand     | varchar(50)                             | YES  |     | NULL    |       |
| model     | varchar(50)                             | YES  |     | NULL    |       |
| colour    | varchar(25)                             | YES  |     | NULL    |       |
+-----------+-----------------------------------------+------+-----+---------+-------+
6 rows in set (0.00 sec)
```

Fig. 3.6 Vehicle Description

```
mysql> select * from Customers;
+--------------------------------------+------+-----+--------+--------------+-------+--------------------------------------+
| id                                   | name | age | gender | phone_number | city  | vehicle_id                           |
+--------------------------------------+------+-----+--------+--------------+-------+--------------------------------------+
| d2594bed-c7db-11ed-815b-5414f3f5a929 | ab   |  32 | male   | 910987654321 | spiti | 6effb3ce-c694-49d8-ad33-5a4f7b3441e8 |
+--------------------------------------+------+-----+--------+--------------+-------+--------------------------------------+
```

Fig. 3.7 Customer table

```
mysql> select * from Vehicles;
+--------------------------------------+------+-----------+--------+----------+--------+
| id                                   | type | fuel_type | brand  | model    | colour |
+--------------------------------------+------+-----------+--------+----------+--------+
| 6effb3ce-c694-49d8-ad33-5a4f7b3441e8 | 4    | diesel    | toyota | fortuner | white  |
+--------------------------------------+------+-----------+--------+----------+--------+
1 row in set (0.00 sec)
```

Fig. 3.8 Vehicle table

The "Customers" table in this graphic includes details about the customers, including their ID (as a UUID), name, age, gender, phone number, city, and the foreign key for the car they own. The "Vehicles" table includes details about the vehicles, such as their ID (as a UUID), type (2, 4, or 6 wheelers), fuel type (petrol, diesel, cng, or electric), brand, model, and color.

The "Customer_Vehicle" table, which is a many-to-many relationship, represents the relationship between the "Customers" and "Vehicles" databases. The customer ID and the vehicle ID, which connects them, are both contained in this table.

22

The tables might be utilized using the following API endpoints:

API for users

- Get a customer by ID with the help of the GET /customers/id> command.
- Add a brand-new client.
- Update an existing customer by ID using PUT.
- Delete a customer by ID using DELETE.
- Retrieve customers using GET. Get every client.
- GET /customer?Get all clients' car details if vehicle=true is set.
- GET /customer?Get all clients with vehicles that use a certain fuel type by using the fuel_type= fuel_type parameter.
- GET /customer?Get all clients with automobiles of the given brand using the query brand="brand"

API for Vehicles

- Make a new car using POST /vehicles.
- Update a current vehicle by ID with PUT /vehicles/
- Delete a car using its ID by typing DELETE /vehicles/

SYSTEM DESIGN

The implementation of each layer is done separately. First of all the entities i.e structs which are to be used in the project are created in a separate folder named entities. The entities folder has three go files, Customers.go, Vehicles.go, CustomerVehicle.go. Each of them contains different structs named as Customers, Vehicles, and CustomerVehicle respectively.

Next part is making the SQL connection, i.e database to go. Similar tables have to be made in MySQL DB , i.e Customer and Vehicles, keeping in mind the schema.

Now, to create the first layer , i.e the handler layer. Inside the handler layer there is a file named interface.go, it contains an interface which has all the methods of the next layer. These methods are called inside the handler.

The handler layer has the following structure:

    ☐ **handler**

    customer

        customer.go

        customer_test.go

    vehicles

        vehicles.go

        vehicles_test.go

    interface.go
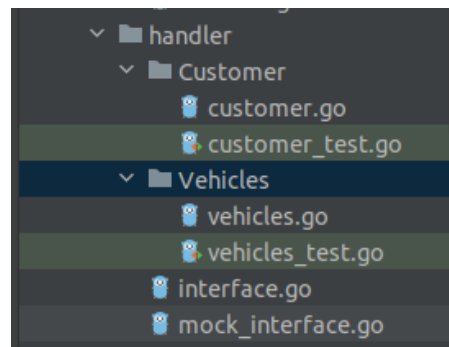
    mock_interface.go

Fig. 3.9 Structure of handler layer

The mock_interface.go file is created to create mocks for testing purposes using mock gen.

Mocks & their advantages

Mock testing involves isolating the code from others while testing it without the distraction of dependencies and other factors, such as network issues and traffic swings. Mock objects, which mimic the behavior of real things and display true attributes, are used to substitute the dependent items. Mock testing's guiding principle is to prioritize testing over prioritizing dependencies. Here, we'll talk about the following subjects:

Uses of mocks are as under:

- When conducting unit testing, it is more beneficial.
- when you desire to stay away from outside dependence.
- Even though you wish to use dummy objects to speed up the testing process.
- Even so, it's important to understand how the test will look in advance.

Internal working of mocks

It's a kind of unit testing that makes it possible to make claims about how the code driving the test interacts with other system modules.

1. When performing mock testing, dependencies are swapped out for objects that mimic the behavior of the crucial ones. It is founded on verification using behavior.
2. The mock object builds a fake interface to represent the real object's interface. Thus, it is known as mock.
3. Instead of concentrating on the entire code, it highlights the specific section that will be tested.
4. The mock object only reads test data from a local disc and responds with it.
5. No changes to the codebase are necessary for mocking.
6. During testing, fake objects are used in place of the inherited class when there are dependencies in the case of constructors and other methods.
7. In contrast to conventional unit testing, assertion is carried out by fake objects that have been initialized beforehand with respect to the type of method calls that should be made and the expected behavior.
8. Mocking is used for protocol testing, which examines how APIs should be used and how they will respond when implemented properly.

Note: Mock Gen auto generates the mock functions for the given methods in the interface. Then the next layer is the service layer, which checks the logics in the methods and functions.

The structure of the service layer is :

⬜ **service**

customer

     customer.go

     customer_test.go

vehicles

vehicle.go

vehicle_test.go

interface.go

mock_interface.go
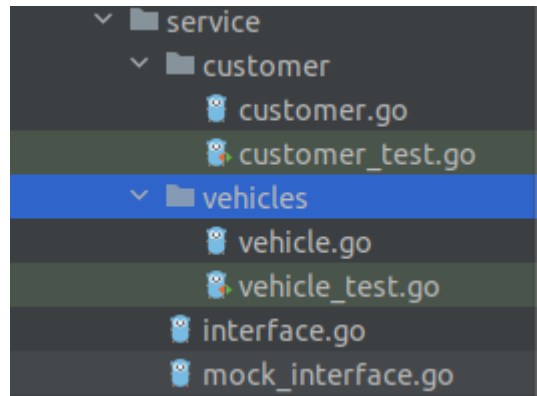
Fig. 3.10 Structure of service layer

The service layer has the interface of the methods of the store layer, and those methods are called in the service layer.

Another reason to create mocks is that since we are calling service in handler , store in service and when the testing is performed this testing will also be calling the similar functions in a similar way, so there will be an integrated testing for all these layers but we won't be able to do the unit testing so, in order to do that we create the mocks of these functions.

In the store layer where db queries are used and the database connection is established.

The structure of store layer is given as follows:

- **store**

    customer

    　　customer.go

    　　customer_test.go

    vehicle

    　　vehicle.go

    　　vehicle_test.go

26
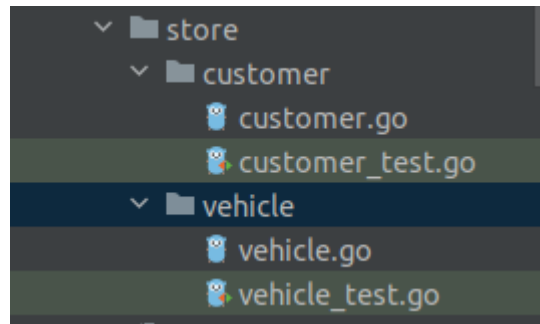
Fig. 3.11 Structure of store layer

The store layer does not implement go mock, it uses SQL mocks to mock the DB, so that the changes are not made to the really existing database.

SOL Mocks

A mock library that uses the sql/driver is called sqlmock. With which any sql driver behavior can be simulated in testing without the need for a genuine database connection. It assists in maintaining the proper TDD process.

This library enables numerous connections and concurrency. It also enables mocking and named sql parameters for go1.8 context-related features. It does not call for changing your source code in any way. Any sql driver method behavior can be mocked using the driver.It has rigorous expectation order matching by default. It has no reliance on outside parties. We mock the sql connection in this layer and then write the test cases using these mocks.

Docker

Docker is an open platform for developing, sharing, and executing applications. With the use of Docker, you can separate your programmes from your infrastructure, enabling quick software delivery. The same way you manage your applications, you can use Docker to manage your infrastructure. By using Docker's technique for swiftly shipping, testing, and deploying code, you may significantly reduce the amount of time it takes between writing code and running it in production.

We set up docker images of MySQL and operated on the image rather than the real MySQL table.

CODE

```go
package driver

import ...

type Database struct {  1 usage   ▲ Nidhi
    DB *sql.DB
}

// SQLConnection  method for creating database connection with golang.
func (d Database) SQLConnection() (*sql.DB, error) {  1 usage   ▲ Nidhi
    var err error
    d.DB, err = sql.Open( driverName: "mysql",  dataSourceName: "root:Nidhi@24@tcp(localhost:3306)/zopstore")

    if err != nil {
        return d.DB, errors.New( text: "unable to open Database")
        panic( v: "Connection Failed")
    }

    return d.DB, nil
}
```

Fig. 3.12 Sql Driver Connection

```go
package entities

type Customers struct {        ▲ Nidhi
    ID          string  `json:"id"`
    Name        string  `json:"name"`
    Age         int     `json:"age"`
    Gender      Genders `json:"gender"`
    PhoneNumber int     `json:"phoneNumber"`
    City        string  `json:"city"`
    VehicleID   string  `json:"vehicleId"`
}
type Genders string // ENUM datatype  23 usages  ▲ Nidhi
const (
    Male   Genders = "Male"    no usages  ▲ Nidhi
    Female Genders = "Female"  no usages  ▲ Nidhi
    Others Genders = "Others"  no usages  ▲ Nidhi
)
```

Fig. 3.13 Entities - customers.go (having Customers struct)

```go
package entities

type Vehicles struct {        ▲ Nidhi
    ID       string `json:"id"`
    Type     Wheels `json:"type"`
    FuelType Fuels  `json:"fuelType"`
    Brand    string `json:"brand"`
    Model    string `json:"model"`
    Color    string `json:"color"`
}
type Wheels string  15 usages  ▲ Nidhi

const (
    Two  Wheels = "2"  no usages  ▲ Nidhi
    Four Wheels = "4"  no usages  ▲ Nidhi
    Six  Wheels = "6"  no usages  ▲ Nidhi
)

type Fuels string  16 usages  ▲ Nidhi

const (
    Petrol   Fuels = "Petrol"    no usages  ▲ Nidhi
    Deisel   Fuels = "Deisel"    no usages  ▲ Nidhi
    Cng      Fuels = "CNG"       no usages  ▲ Nidhi
    Electric Fuels = "Electric"  no usages  ▲ Nidhi
)
```

Fig. 3.14 Entities - Vehicles.go(having struct vehicles)

```
package entities

type CustomerVehicle struct {   10 usages   ± Nidhi
    Customers
    Vehicles
}
```

Fig. 3.15 Entities - CustomerVehicle.go(having CustomerVehicle struct)

```
// TestCreateHandler is a test function to test the
// CreateHandler method  in the handler layer
func TestCreateHandler(t *testing.T) {   ± Nidhi *
    tests := []struct {
        desc     string
        method   string
        customer entities.Customers
        expCode  int
        expErr   error
    }{
        { desc: "Customer Created",  method: http.MethodPost,
            customer: entities.Customers{ ID: cId,  Name: "Emily",  Age: 24,  Gender: entities.Genders("Female"),
                PhoneNumber: 917777777777,  City: "Chicago",  VehicleID: "ab9d6925-287b-40c7-b2e1-175928235cd2"},
            expCode: http.StatusCreated,  expErr: nil},

        { desc: "Wrong method",  method: http.MethodGet,  customer: entities.Customers{ ID: cId,  Name: "Alfie",  Age: 24,
            Gender: entities.Genders("Male"),  PhoneNumber: 913333333333,  City: "London",
            VehicleID: "ab9d6925-287b-40c7-b2e1-175928235cd2"},  expCode: http.StatusMethodNotAllowed,
            expErr: errors.New( text: "Wrong method")},
    }

    ctrl := gomock.NewController(t)
    mock := handler.NewMockCustomerService(ctrl)
    for i, tc := range tests {
        byteCustomer, err := json.Marshal(tc.customer)
        if err != nil {
            log.Println(err)
        }
```

Fig. 3.16 Handler Layer customer_test.go file(test function for creating a customer)

```go
// TestGetById is a test function to test the
// GetById method in the handler layer
func TestGetById(t *testing.T) {
    tests := []struct {
        desc     string
        id       string
        method   string
        expCode  int
        expErr   error
    }{
        { desc: "Customer details fetched", id: cId, method: http.MethodGet, expCode: http.StatusOK, expErr: nil},
        { desc: "Wrong method", id: cId, method: http.MethodDelete, expCode: http.StatusMethodNotAllowed, expErr: errors
    }
    ctrl := gomock.NewController(t)
    mock := handler.NewMockCustomerService(ctrl)
    for i, tc := range tests {
        r := httptest.NewRequest(tc.method, "/customers/"+tc.id, body: nil)
        w := httptest.NewRecorder()
        mock.EXPECT().GetByIdService(tc.id).Return(tc.expErr).MaxTimes( n: 1)
        h := NewCustomerHandler(mock)
        h.GetById(w, r)

        assert.Equalf(t, tc.expCode, w.Code, "Test case #{i} failed.")

    }
}
```

Fig. 3.17 Handler test function for get by Id

```go
package handler

import ...

// CustomerHandler struct which takes Object of CustomerService type
type CustomerHandler struct {  8 usages  ±Nidhi
    customService handler.CustomerService // here CustomeService is the interface of service layer.
}

// NewCustomerHandler is a factory function.
func NewCustomerHandler(customService handler.CustomerService) CustomerHandler {  7 usages  ±Nidhi
    return CustomerHandler{ customService: customService}
}

// CreateHandler takes in the customer details and creates a customer
func (ch CustomerHandler) CreateHandler(w http.ResponseWriter, r *http.Request) {  3 usages  ±Nidhi
    if r.Method != http.MethodPost {
        w.WriteHeader(http.StatusMethodNotAllowed)
        w.Write([]byte("Method used is incorrect"))
        return
    }

    customerBody, err := io.ReadAll(r.Body)
    if err != nil {
        w.Write([]byte("error in reading the body"))
        return
    }
    var customer entities.Customers
```

Fig. 3.18 Handler Layer customer.go for create customer method

31

```go
func (ch CustomerHandler) GetById(w http.ResponseWriter, r *http.Request) { 3 usages  Nidhi *
    vars := mux.Vars(r)
    id := vars["id"]
    if r.Method != http.MethodGet {
        w.WriteHeader(http.StatusMethodNotAllowed)
        w.Write([]byte("Method used is incorrect"))
        return
    }
    var customer entities.Customers
    url := r.URL.Path
    p := strings.Split(url, sep: "/")
    id = p[len(p)-1]
    if id == "" {
        w.WriteHeader(http.StatusNotFound)
        return
    }
    customer, err := ch.customService.GetByIdService(id)
    if err != nil {
        w.Write([]byte(err.Error()))
        return
    }
    result, err := json.Marshal(customer)
    if err != nil {
        w.Write([]byte(err.Error()))
        return
    }
    w.Write(result)
    return
```

Fig. 3.19 Get BY ID method in handler layer

```go
// returns the status code http.StatusCreated when the vehicle is created.
func (vh VehicleHandler) PostHandler(w http.ResponseWriter, r *http.Request) { 3 usages  Nidhi *
    if r.Method != http.MethodPost {
        w.WriteHeader(http.StatusMethodNotAllowed)
        w.Write([]byte("Method used is incorrect"))
        return
    }
    var vehicle entities.Vehicles
    vehicleBody, err := io.ReadAll(r.Body)
    if err != nil {
        log.Println(err)
        return
    }
    err = json.Unmarshal(vehicleBody, &vehicle)
    if err != nil {
        log.Println(err)
        return
    }
    err = vh.vehiService.PostService(vehicle)
    if err != nil {
        w.Write([]byte(err.Error()))
        return
    }
    w.Header().Set( key: "Content-Type", value: "application/json")
    w.WriteHeader(http.StatusCreated)
    w.Write(vehicleBody)
}
```

Fig. 3.20 Create vehicle in vehicle.go

```go
func (vh VehicleHandler) PutHandler(w http.ResponseWriter, r *http.Request) {  3 usages   ±Nidhi*
    vars := mux.Vars(r)
    id := vars["id"]
    if r.Method != http.MethodPut {
        w.WriteHeader(http.StatusMethodNotAllowed)
        w.Write([]byte("Method used is incorrect"))
        return
    }
    url := r.URL.Path
    p := strings.Split(url, sep: "/")
    id = p[len(p)-1]
    if id == "" {
        w.WriteHeader(http.StatusNotFound)
        return
    }
    vehicleBody, err := io.ReadAll(r.Body)
    if err != nil {
        log.Println(err)
        return
    }
    var vehicle entities.Vehicles
    err = json.Unmarshal(vehicleBody, &vehicle)
    if err != nil {
        log.Println(err)
        return
    }
    err = vh.vehiService.PutService(id, vehicle)
    if err != nil {
```

Fig. 3.21 Update Vehicle in vehicle.go for handler layer

```go
//  TestVehiclePost is  a test function for PostHandler
func TestVehiclePost(t *testing.T) {  ±Nidhi*
    tests := []struct {
        desc    string
        method  string
        vehicle entities.Vehicles
        expCode int
        expErr  error
    }{
        { desc: "vehicle created",  method: http.MethodPost,  vehicle: entities.Vehicles{ ID: vId,  Type: entities.Wheels("F
        { desc: "Wrong method",  method: http.MethodGet,  vehicle: entities.Vehicles{ ID: vId,  Type: entities.Wheels("Two")
    }
    ctrl := gomock.NewController(t)
    mock := handler.NewMockVehicleService(ctrl)
    for i, tc := range tests {
        byteVehicles, err := json.Marshal(tc.vehicle)
        if err != nil {
            log.Println(err)
            return
        }
        r := httptest.NewRequest(tc.method, target: "/vehicles", bytes.NewReader(byteVehicles))
        w := httptest.NewRecorder()
        mock.EXPECT().PostService(tc.vehicle).Return(tc.expErr).MaxTimes( n: 1)
        h := NewVehicleHandler(mock)
        h.PostHandler(w, r)
        assert.Equalf(t, tc.expCode, w.Code, "Test case #{i} failed.")
    }
}
```

Fig. 3.22 Test function for create vehicle in vehicle_test.go

```go
package handler

import "go-daily-assignment-noida/VehicleStore/entities"

type CustomerService interface {  3 usages  ± Nidhi
    CreateService(entities.Customers) error
    GetByIdService(string) (entities.Customers, error)
    GetAllService(string, string, string) (entities.CustomerVehicle, error)
    UpdateService(string, entities.Customers) error
    DeleteService(string) error
}

type VehicleService interface {  3 usages  ± Nidhi
    PostService(entities.Vehicles) error
    PutService(string, entities.Vehicles) error
}
```

Fig. 3.23 Interface.go in handler layer


```go
package service

import ...

type CustomerServ struct {  7 usages  ± Nidhi
    customStore service.CustomerStore
}

func NewCustomerService(customStore service.CustomerStore) CustomerServ {  6 usages  ± Nidhi
    return CustomerServ{ customStore: customStore}
}

// CreateService is method for creating a customer in service layer
// return an error if details are invalid
func (cs CustomerServ) CreateService(customer entities.Customers) error {  ± Nidhi

    phoneNum := strconv.Itoa(customer.PhoneNumber)
    // condition for length of phone is not 12
    if len(phoneNum) != 12 : errors.New("phone number length should be 12")  ↗  else if phoneNum[:2] != "91"  : erro

    // condition for age of customer
    if customer.Age < 0 || customer.Age > 100 : errors.New("age should be between 0 and 100")  ↗
    if customer.Name == "" || customer.Age == 0 || customer.PhoneNumber == 0 || customer.City == "" || customer.Ge
        return errors.New( text: "Missing Values are not allowed")
    }
    return cs.customStore.CreateStore(customer)
}
```

Fig. 3.24 Create customer method in service layer

34

Fig. 3.25 GetById and get All method in service layer

Fig. 3.26 Test cases for Create customer in service layer

35

```go
func TestGetbyIdService(t *testing.T) {  ±Nidhi

    tests := []struct {
        desc    string
        id      string
        expRes  entities.Customers
        expErr  error
    }{
        { desc: "Customer details fetched",  id: cId,  expRes: entities.Customers{ ID: cId,  Name: "Emily",  Age: 24,  Gender: entities
        { desc: "Invalid Id",  id: "",  expRes: entities.Customers{},  expErr: errors.New( text: "Invalid Id")},
    }

    ctrl := gomock.NewController(t)
    mock := service.NewMockCustomerStore(ctrl)

    h := NewCustomerService(mock)
    for i, tc := range tests {

        mock.EXPECT().GetByIdStore(tc.id).Return(tc.expErr).MaxTimes( n: 5)
        _, err := h.GetByIdService(tc.id)
        if err != nil {
            log.Println(err)
        }
        assert.Equalf(t, tc.expErr, err, "Test case #{i} Failed")
    }

}
```

Fig. 3.27 Test function for GetByID in service layer

```go
// CreateService takes the vehicle details and creates the vehicle
func (vs VehicleServ) PostService(vehicle entities.Vehicles) error {  2 usages  ±Nidhi *
    if vehicle.Brand == "" || vehicle.Model == "" {

        return errors.New( text: "Brand or model missing")
    }
    if vehicle.Type != "Two" && vehicle.Type != "Four" && vehicle.Type != "Six" {
        return errors.New( text: "Type is invalid")
    }

    if vehicle.FuelType != "petrol" && vehicle.FuelType != "diesel" &&
        vehicle.FuelType != "cng" && vehicle.FuelType != "electric" {

        return errors.New( text: "Fuel type is invalid")
    }

    if vehicle.Color == "" {

        return errors.New( text: "Colour is empty")
    }
    err := vs.vehiStore.CreateStore(vehicle)
    if err != nil {
        log.Println(err)
        return err
    }
    return nil
}
```

Fig. 3.28 Method create vehicle in service layer

36

```go
// UpdateService updates the vehicle taking id as a parameter
func (vs VehicleServ) PutService(id string, vehicle entities.Vehicles) error {    2 usages    ≗ Nidhi *
    //var vehicle entities.Vehicles
    if vehicle.Brand == "" || vehicle.Model == "" {
        return errors.New( text: "Brand or model missing")
    }
    if id == "" {
        return errors.New( text: "Invalid Id")
    }

    if vehicle.Type != "Two" && vehicle.Type != "Four" && vehicle.Type != "Six" {
        return errors.New( text: "Type is invalid")
    }

    if vehicle.FuelType != "petrol" && vehicle.FuelType != "diesel" && vehicle.FuelType != "cng" &&
        vehicle.FuelType != "electric" {

        return errors.New( text: "Fuel type is invalid")
    }
    if vehicle.Color == "" {

        return errors.New( text: "Colour is empty")
    }
    return vs.vehiStore.UpdateStore(id, vehicle)
}
```

Error string should not be capitalized or end with punctuation mark

Fix error string format Alt+Shift+Enter   More actions... Alt+Enter

Fig. 3.29 Update Vehicle Method in service layer

```go
func TestPostVehicle(t *testing.T) {   ≗ Nidhi *
    Test := []struct {
        desc     string
        vehicle entities.Vehicles

        expErr error
    }{
        { desc: "Incorrect type",   vehicle: entities.Vehicles{ ID: "964f534e-c23a-11ed-afa1-0242ac120002",
            Type: entities.Wheels("One"),  FuelType: entities.Fuels("petrol"),  Brand: "TATA",  Model: "Prima",  Color: "Red"},
            expErr: errors.New( text: "Type is invalid")},
        { desc: "Brand name is empty",  vehicle: entities.Vehicles{ ID: "964f534e-c23a-11ed-afa1-0242ac120002",
            Type: entities.Wheels("Four"),  FuelType: entities.Fuels("petrol"),  Brand: "",  Model: "Prima",  Color: "Red"},
            expErr: errors.New( text: "Brand or model missing")},
    }

    ctrl := gomock.NewController(t)
    mock := service.NewMockVehicleStore(ctrl)

    h := NewVehicleService(mock)
    for i, tc := range Test {
        mock.EXPECT().CreateStore(tc.vehicle).Return(tc.expErr).MaxTimes( n: 2)
        err := h.PostService(tc.vehicle)
        assert.Equalf(t, tc.expErr, err,  msg: "TestCase:%v", i)

    }
}
```

Fig. 3.30 Test function for create vehicle in service layer

37

```
// CreateStore implements the sql query to create the customer.
func (d Database) CreateStore(customer entities.Customers) error {  3 usages   Nidhi
    _, err := d.db.Exec( query: "insert into Customers(id, name,age, gender, phone_number,city, vehicle_id) values(?,?,?,?,?
    if err != nil {
        return err
    }

    return nil
}

// GetByIdStore takes the id and returns the rows of customer
func (d Database) GetByIdStore(id string) (entities.Customers, error) {  4 usages   Nidhi
    customRows := d.db.QueryRow( query: "select * from Customers where id = ?", id)

    var c entities.Customers

    err := customRows.Scan(&c.ID, &c.Name, &c.Age, &c.Gender, &c.PhoneNumber, &c.City, &c.VehicleID)
    if err != nil {
        return entities.Customers{}, err
    }
    return c, nil
}
```

Fig. 3.31 Create Customer and GetByIdStore in store layer

```
// UpdateStore implements the update query for a table
func (d Database) UpdateStore(id string, customer entities.Customers) error {  3 usages   Nidhi
    //var customer entities.Customers
    result, err := d.db.Exec( query: "Update Customers set name = ?, age = ?, gender = ?, phone_number = ?, city = ? where i
    if err != nil : err
    rowAffected, err := result.RowsAffected()
    if err != nil : err
    if rowAffected == 0 {
        log.Println(err)
        return err
    }
    return nil
}

// DeleteStore implements the delete query
func (d Database) DeleteStore(id string) error {  2 usages   Nidhi
    result, err := d.db.Exec( query: "delete from Customers where id = ?", id)
    if err != nil : err
    rowAffected, err := result.RowsAffected()
    if err != nil : err
    if rowAffected == 0 {
        log.Println(err)

    }
    return nil
}
```

Fig. 3.32 Update customer and Delete customer in store layer

38

```
func TestCreateStore(t *testing.T) { ▲Nidhi *                                                                      ▲12
    db, mock, err := sqlmock.New(sqlmock.QueryMatcherOption(sqlmock.QueryMatcherEqual))
    if err != nil {
        log.Println(err)
    }
    defer db.Close()
    tests := []struct {
        desc    string
        expRes entities.Customers
        expErr error
    }{
        { desc: "Customer created", expRes: entities.Customers{ ID: "2a43cf1e-d6ae-4710-a109-ec7e627e0c26",
            Name: "Tony", Age: 50, Gender: entities.Genders("Male"), PhoneNumber: 918888888888, City: "New York",
            VehicleID: "964f534e-c23a-11ed-afa1-0242ac120002"}, expErr: nil},
    }
    store := NewCustomerStore(db)
    d := Database{ db: store.db}
    for i, tc := range tests {
        c := entities.Customers{ ID: "2a43cf1e-d6ae-4710-a109-ec7e627e0c26", Name: "Tony", Age: 50,
            Gender: entities.Genders("Male"), PhoneNumber: 918888888888, City: "New York",
            VehicleID: "964f534e-c23a-11ed-afa1-0242ac120002"}
        mock.ExpectExec( expectedSQL: `INSERT INTO Customers VALUES (?,?,?,?,?,?,?)`).WithArgs(c.ID, c.Name,
            c.Age, c.PhoneNumber, c.Gender, c.City,
            c.VehicleID).WillReturnResult(sqlmock.NewResult( lastInsertID: 1, rowsAffected: 1))
        err = d.CreateStore(c)
        assert.Equalf(t, err, tc.expErr, msg: "Test case %v failed", i)
    }
}
```

Fig. 3.33 Test Function for create customer in store layer

```
func TestGetCustomer(t *testing.T) { ▲Nidhi                                                                      ▲12
    // Create a mock database connection
    db, mock, err := sqlmock.New(sqlmock.QueryMatcherOption(sqlmock.QueryMatcherEqual))
    if err != nil {
        log.Println(err)
    }
    defer db.Close()

    test := []struct {
        desc    string
        id      string
        rows    *sqlmock.Rows
        expRes entities.Customers
        expErr error
    }{
        { desc: "Customer fecthed", id: "2a43cf1e-d6ae-4710-a109-ec7e627e0c26", rows: sqlmock.NewRows([]string{"Id", "Name", "
            AddRow( values...: "2a43cf1e-d6ae-4710-a109-ec7e627e0c26", "Tony", 50, "Male", 918888888888, "New York", "964f534e-
        { desc: "Invalid Id", id: "", rows: sqlmock.NewRows([]string{}).AddRow(), expRes: entities.Customers{}, expErr: errors.Ne
    }
    store := NewCustomerStore(db)
    d := Database{ db: store.db}

    for i, tc := range test {
        mock.ExpectQuery( expectedSQL: "select * from Customers where id = ?").WithArgs(tc.id).WillReturnRows(tc.rows).WillRe
        _, err = d.GetByIdStore(tc.id)
        assert.Equalf(t, err, tc.expErr, "Test case #{i} Failed")
    }
}
```

Fig. 3.34 Test Function for GET customer in store layer

39

Fig. 3.35 Create vehicle in store layer

Fig. 3.36 Test function for create vehicle in store layer

Chapter 04: EXPERIMENTS AND RESULT ANALYSIS

RESULTS

1. All the test functions in every layer passed with coverage 100%
2. All the end points are working as expected
3. All the status codes are correct

The end point testing is done on Postman.

POSTMAN

Postman is an API platform for building and using APIs. Postman enhances cooperation and streamlines each phase of the API lifecycle in order to assist you in designing better APIs more quickly.
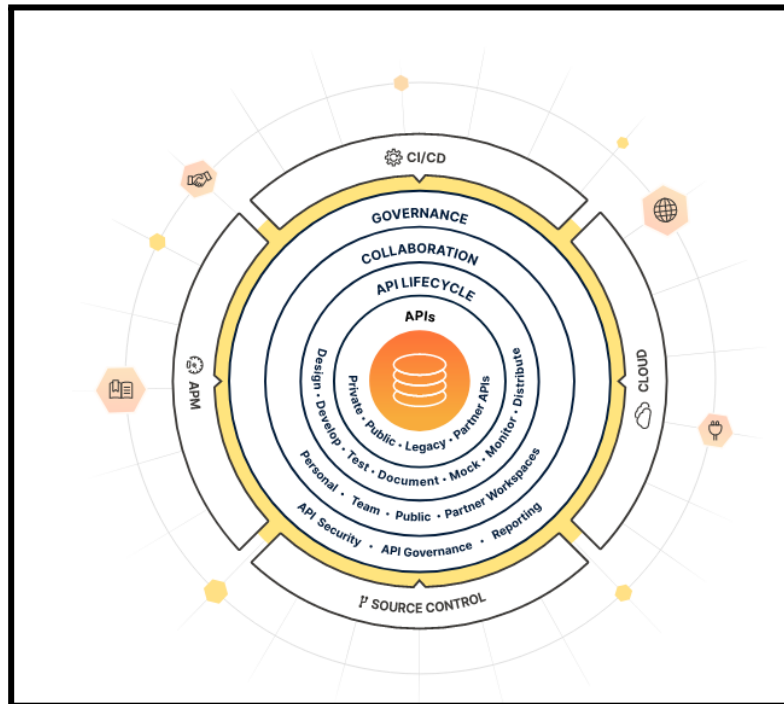
Fig. 4.1 Postman

Everything connected to APIs, including API specifications, documentation, workflow recipes, test cases and results, metrics, and more, may be stored and managed by Postman.
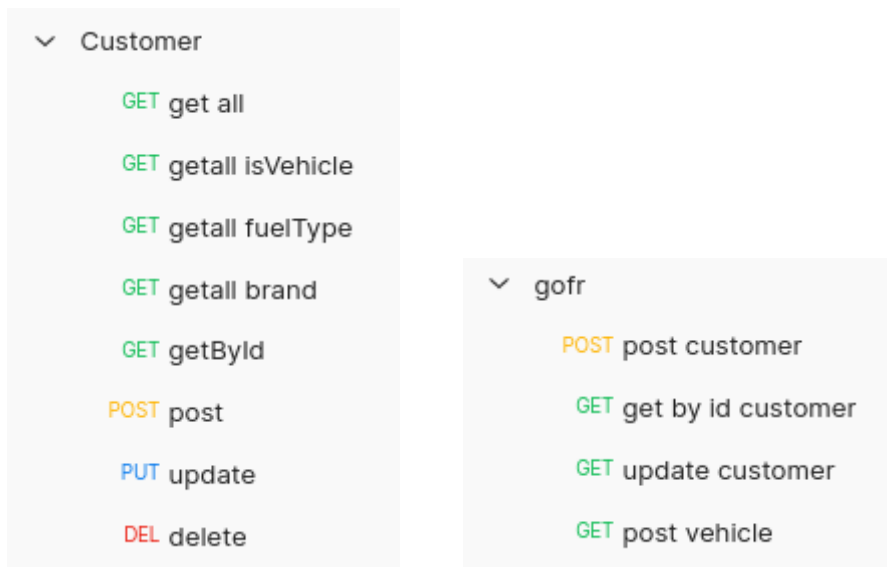
Fig. 4.2 Postman Collections
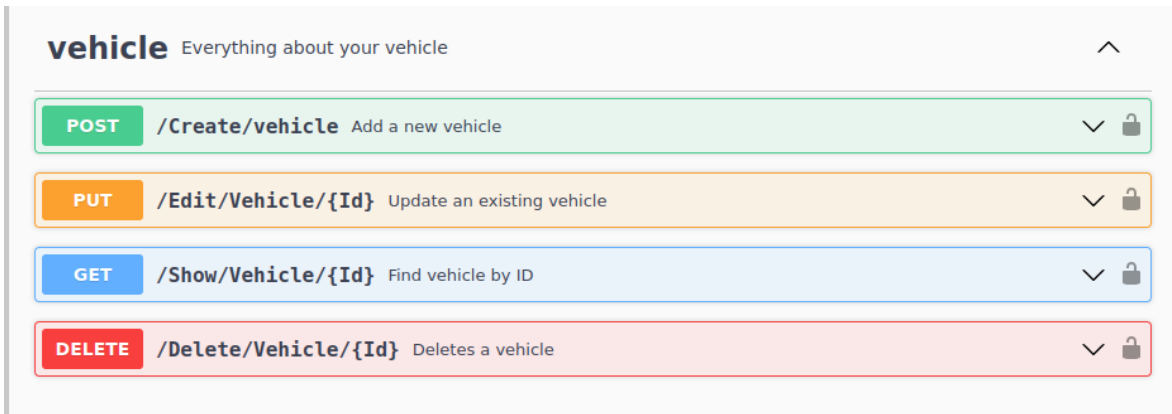
Fig. 4.3 Swagger - API documentation for vehicle

ENDPOINTS RESULT FROM POSTMAN


```json
{
    "id":"f03d0d86-feda-4202-8990-3122ebaa1d10",
    "type":"4",
    "fuel_type":"diesel",
    "brand":"tata",
    "model":"tiago",
    "color":"brown"
}
```
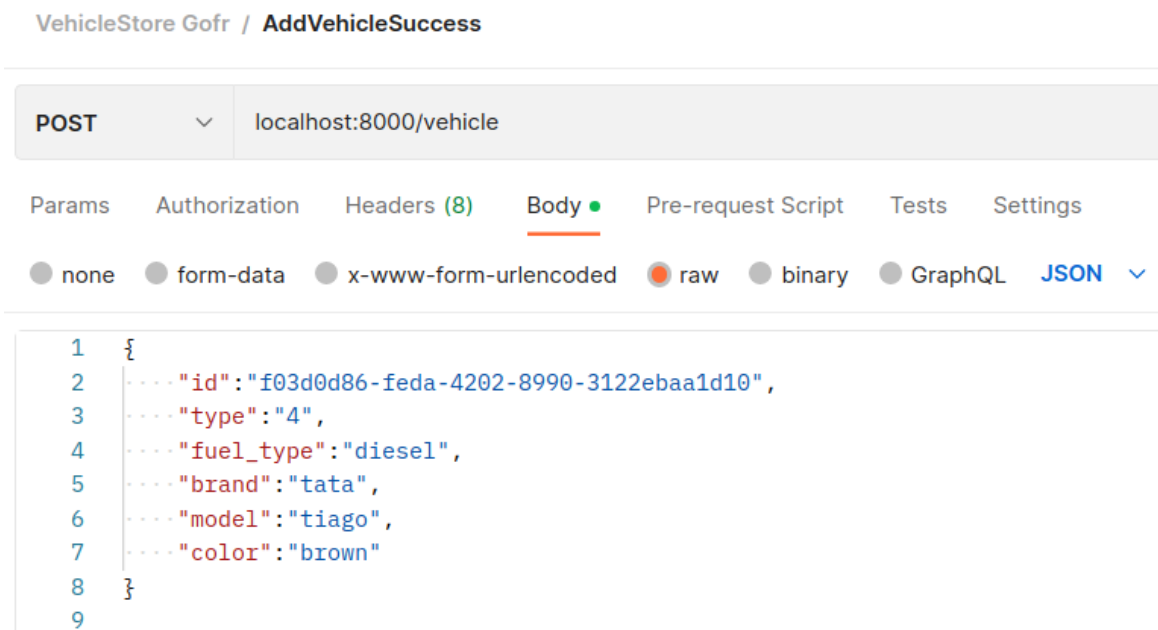
Fig. 4.4 Add vehicle Endpoint
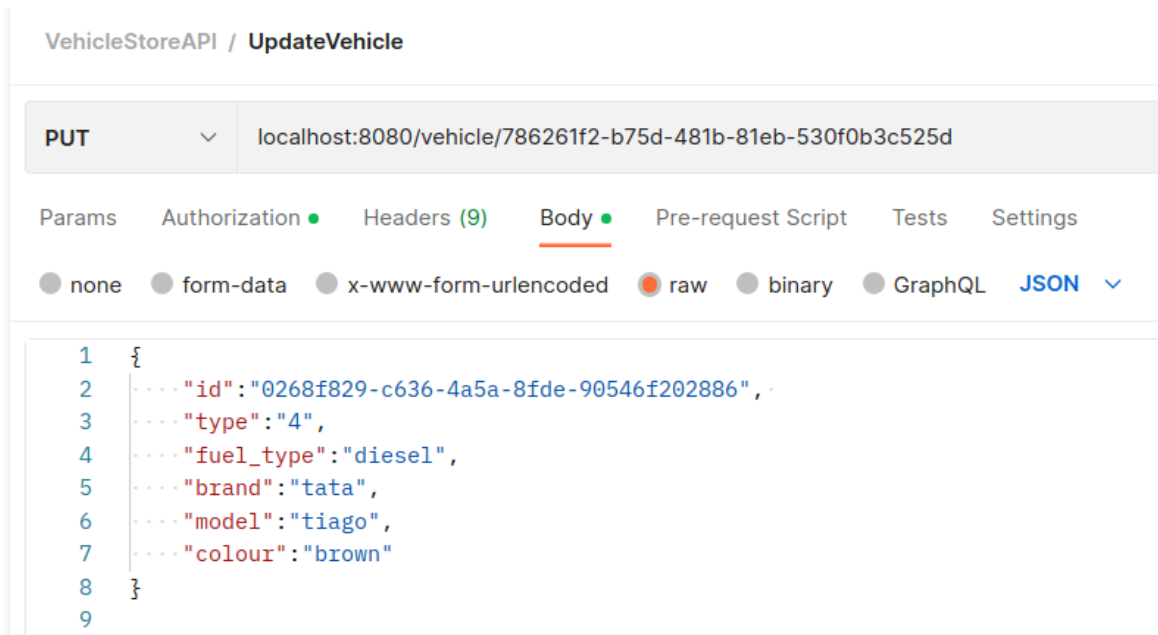
Fig. 4.5 Update vehicle endpoint
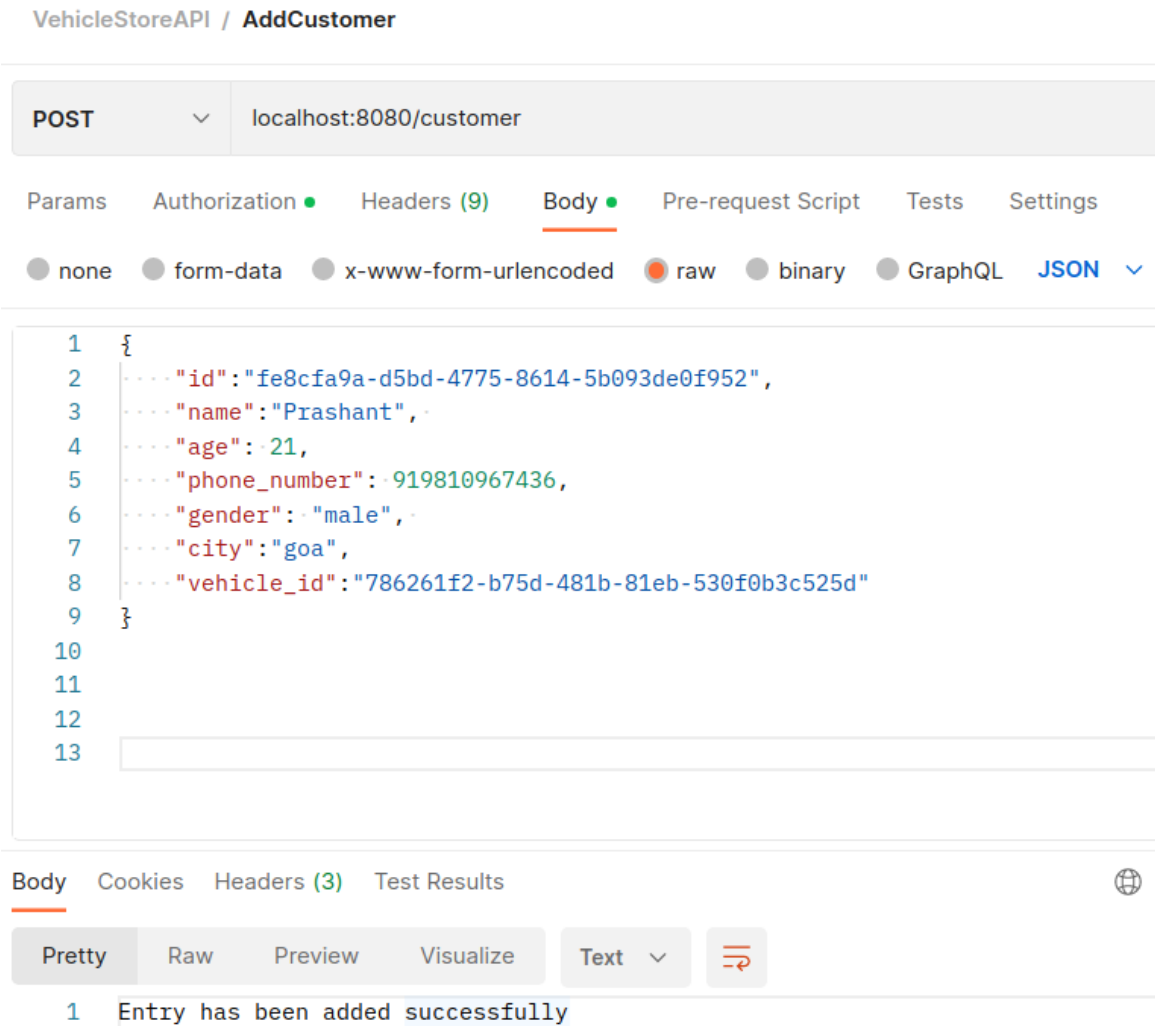
Fig. 4.6 Add Customer endpoint

Swagger

The Swagger API tool is used to create the API documentation. In Swagger, you can specify the internal organisation of your APIs so that computers can comprehend them. The capacity of APIs to explain their own structures is at the heart of everything excellent in Swagger. What makes it so amazing? However, by scanning your API's structure, we can automatically generate beautiful and interactive API documentation. We can also look at other possibilities, including automated testing and creating client libraries for your API in several languages automatically. Swagger achieves this by asking your API for a comprehensive YAML or JSON answer that provides a description of your whole API.

45

Fig. 4.7 Swagger -API documentation for customer

Fig. 4.8 GetAll endpoint

Chapter 05: CONCLUSIONS

5.1 CONCLUSIONS

Overall, best practices and techniques were taken into account when developing the ZopStore Programming interface. In a three-layer engineering approach, the handler, service, and storage levels were explicitly segregated from one another. The probability of defects was decreased by using test-driven development across the entire development cycle.

While middleware was used to provide overarching concerns like confirmation and logging to the Programming interface, data set movements were used to monitor changes to the data set blueprint. The store layer's unit testing was made possible via SQL deriding, while the handler layer's unit testing was made possible with mockgen and mux.

Metrics were used to carefully monitor and analyze execution, and steps were taken to simplify the programming interface and ensure excellent performance even under heavy load. Additionally, the programming interface was completely documented using Swagger and Postman, with clear and comprehensive documentation available for all endpoints.

Finally, the communication between development teams was enhanced via GitHub Activities for automated testing, code inclusion, and linter tests. In general, the ZopStore Programming interface was created employing a variety of best practises and approaches, yielding an excellent, dependable, and performant Programming interface that addressed the issues faced by its clients.

5.2 OBJECTIVES ACHIEVED
1. Versatile Program
2. Easily executable
3. Secure
4. Practically usable
5. Enhanced Quality

5.3 FUTURE WORK

This API can be created using frameworks. Frameworks increase the quality of the code and help build a better API. Here at ZopSmart technologies we have our in-house Go Framework named **gofr.** This API can be reconciled with the gofr framework.

  For quicker reaction times and increased variety, the programming interface can be improved. This can be achieved using techniques like load testing, execution profiling, and code improvement.

  The programming interface can be communicated on cloud platforms like AWS, Sky Blue, or Google Cloud, offering additional benefits like adaptability, unchanging quality, and cost-effectiveness.

  In general, the ZopStore Programming interface has a few exciting open doors for future turn of events and development, and it will be fascinating to see how it grows over time.

References

All the references are taken from go documentation present online and in-house learning platform Skillup offered by Zopsmart technologies.

[1] "Documentation - The Go Programming Language." https://go.dev/doc/

[2] "Tutorial gomock · GitHub." https://gist.github.com/thiagozs

[3] "sqlmock package - github.com/data-dog/go-sqlmock - Go Packages." https://pkg.go.dev/github.com/data-dog/go-sqlmock

[4] "mux package - github.com/gorilla/mux - Go Packages." https://pkg.go.dev/github.com/gorilla/mux

[5] "MySQL Documentation - MySQL." https://dev.mysql.com/doc

[6] "Linux Foundation - Decentralized innovation, built with trust." https://www.linuxfoundation.org

[7] "Docker Docs: How to build, share, and run applications | Docker .." https://docs.docker.com

[8] "Getting started | Prometheus." https://prometheus.io/docs/prometheus/latest/getting_started/

[9] "Kubernetes Documentation | Kubernetes." https://kubernetes.io/docs/home

[10] "Documentation | Harness Developer Hub." https://developer.harness.io/docs

major condensed 1