

CRUD API IN GOLANG USING THREE LAYERED ARCHITECTURE

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

in

Computer Science and Engineering/Information Technology

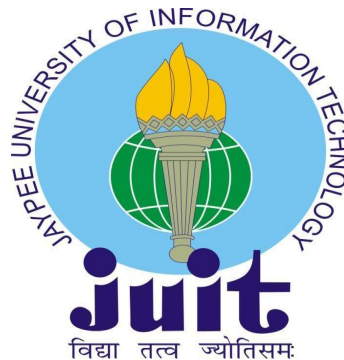
By

NIDHI RAJPUT 191207

Under the supervision of

DR. HARI SINGH

to



Department of Computer Science & Engineering and Information
Technology

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

Certificate Candidate's Declaration

I hereby declare that the work presented in this report entitled “**CRUD API IN GOLANG USING THREE LAYERED ARCHITECTURE**” in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology in Computer Science and Engineering** submitted in the department of Computer Science & Engineering and Information Technology, Jaypee University of Information Technology Wagnaghat is an authentic record of my own work carried out over a period from Feb 2023 to April 2023 under the supervision of **DR. HARI SINGH** Assistant Professor (SG), and training mentor **Miss CHAITHRA AV** SDE 2 (Zopsmart Technology).

The matter embodied in the report has not been submitted for the award of any other degree or diploma.

(Student Signature)

NIDHI RAJPUT 191207

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

(Supervisor Signature)

DR. HARI SINGH

ASSISTANT PROFESSOR (SG)

Computer Science and Engineering/Information Technology

Dated:

ACKNOWLEDGEMENT

First of all, I would like to express my heartiest gratitude and gratefulness to almighty god for his divine blessings helped me to complete the project work successfully in the given period of time.

I extend my heartiest thanks to my project Supervisor DR. HARI SINGH, Assistant Professor (SG), Department of Computer Science & Engineering, Jaypee University of Information Technology, Wanknaghat. Vast knowledge and keen interest of our supervisor in the field of Machine Learning helped me a lot to execute this project. His endless patience, scholarly guidance, continual encouragement, constant, energetic supervision, constructive criticism, valuable advice, reading many inferior drafts and correcting them at all the stages have made it possible to complete this project.

I would also generously thank each one of those individuals who have helped me directly or indirectly in successfully carrying out the execution of this project. In this situation, I also want to thank the various staff individuals, both educating and non-instructing, which have developed their convenient help and facilitated my undertaking.

Last but not the least, I must acknowledge with due respect the constant support and faith of my parents.

NIDHI RAJPUT, 191207

TABLE OF CONTENTS

TITLE	PAGE NO.
LIST OF ABBREVIATIONS	4
LIST OF FIGURES	5
LIST OF TABLES	6
ABSTRACT	7
CHAPTER 1 - INTRODUCTION	8-24
CHAPTER 2 - LITERATURE SURVEY	25
CHAPTER – 3 SYSTEM DESIGN & DEVELOPMENT	26-52
CHAPTER 4 - EXPERIMENTS AND RESULT ANALYSIS	53-58
CHAPTER 5 - CONCLUSIONS	59-60

LIST OF ABBREVIATIONS

- **TDD : Test Driven Development**
- **SQL : Structured Query Language**
- **CRUD: Create, Read, Update, delete**
- **HTTP: Hypertext Transfer Protocol**
- **API: Application Programming Interface**
- **REST: REpresentational State Transfer**
- **IoT: Internet of Things**
- **SOAP: Simple Object Access protocol**
- **URL: Uniform Resource Locator**
- **DB: Database**
- **DBMS: Database Management System**
- **IDE: Integrated Development Environment**

LIST OF FIGURES

1. Chapter- 1

- (FIGURE - 1.1) Fetching a page
- (FIGURE -1.2) Three Layered Architecture
- (FIGURE -1.3) Goland IDE by JetBrains
- (FIGURE -1.4) Creating a New Project in Goland

2. Chapter- 3

- (FIGURE-3.1 - FIGURE 3.36) Code snippets and DB schemas

3. Chapter- 4

- (FIGURE 4.1) Postman
- (FIGURE 4.2) SWAGGER
- (FIGURE 4.3) SWAGGER
- (FIGURE 4.4) POSTMAN ENDPOINTS
- (FIGURE 4.5) POSTMAN ENDPOINTS
- (FIGURE 4.6) POSTMAN ENDPOINTS
- (FIGURE 4.7) POSTMAN ENDPOINTS

LIST OF TABLES

- **Table 1.1 (HTTP STATUS CODES) Chapter -1**
- **Table 3.1 E(HARDWARE REQUIREMENTS) Chapter - 3**
- **Table 3.2 E(SOFTWARE REQUIREMENTS) Chapter - 3**

ABSTRACT

The goal of the project "CRUD API in Golang using three-layered architecture" is to provide a modular and scalable API that executes CRUD (provide, Read, Update, Delete) actions on a database. Golang is the most popular programming language because of its efficiency and ease of use.

The three-layered architecture used in the design of the API ensures improved maintainability and scalability by separating the presentation layer, business logic layer, and data access layer. When interacting with API endpoints, the display layer transmits requests to the business logic layer, which processes them and generates responses. The CRUD activities and database communication are handled by the data access layer.

The project implements a RESTful API that adheres to the fundamentals of HTTP methods and industry best practices. In addition to supporting query parameters to filter, sort, and paginate data, the API offers endpoints for adding, getting, updating, and deleting data from the database.

The project is being built according to test-driven development (TDD) standards, with unit tests and integration tests covering each essential API component. For scalability and availability, the API is deployed on a cloud platform and containerized using Docker.

Overall, this project offers a strong foundation for developing Golang-based, three-layered APIs that are scalable and maintainable.

CHAPTER - 1

INTRODUCTION

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.

1.2 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 WHAT IS 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.

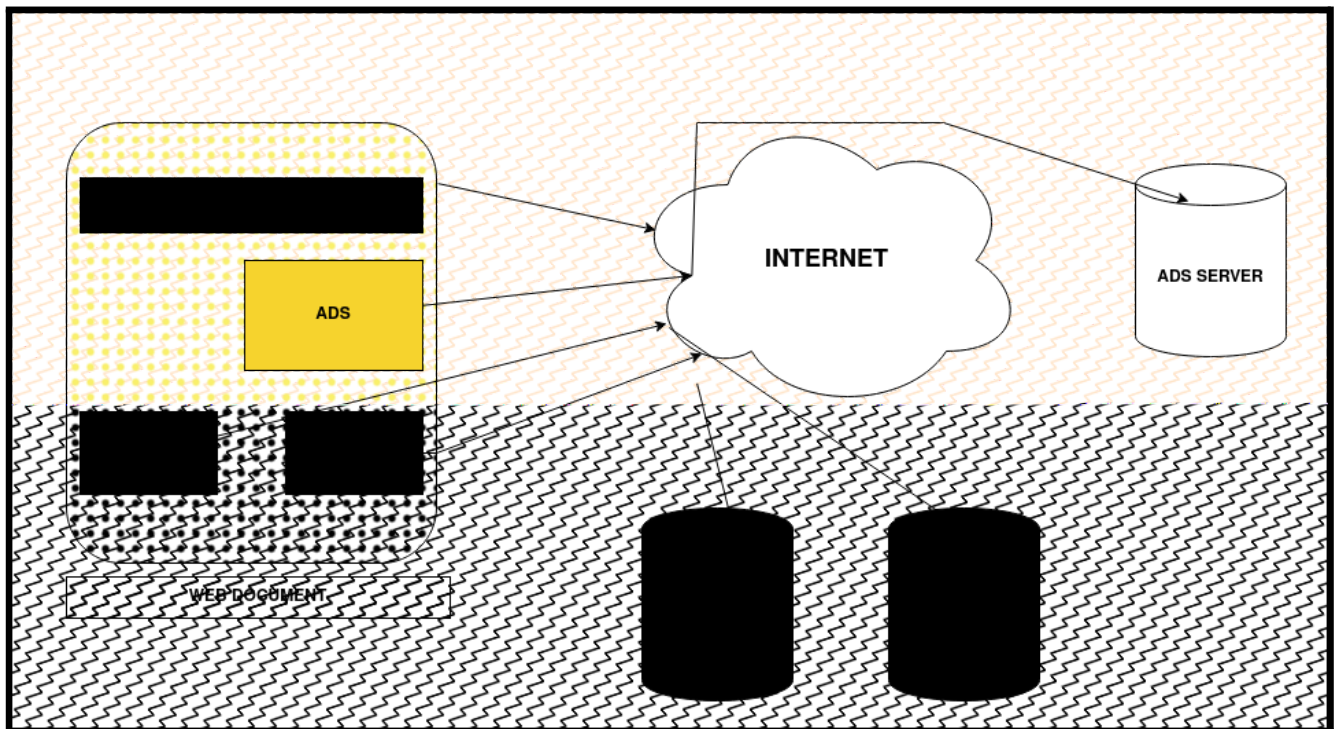
Several architectural restrictions aid in achieving a unified interface and regulating component behavior.

A uniform REST interface can be achieved by the following four 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

What is HTTP ?

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.



(FIGURE - 1.1)

Fetching a page

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.
- **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 was developed at Google in 2009 and is a statically typed, concurrent, and garbage-collected programming language. It is a well-liked option for developing scalable network services, online applications, and command-line utilities since it is made to be straightforward, effective, and quick to understand.

Concurrency, or the capacity to carry out several activities at once, is supported by Go. Go's use of Goroutines and Channels, which enable you to create code that can execute many actions concurrently, allows for concurrency. Because of this, Go is a great choice for creating high-performance, scalable network services as well as for resolving challenging computational issues.

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":** In Go, we use the while loop to execute a block of code until a certain condition is met. Unlike other programming languages, Go doesn't have a dedicated keyword for a while loop. However, we can use the **for** loop to perform the functionality of a while 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

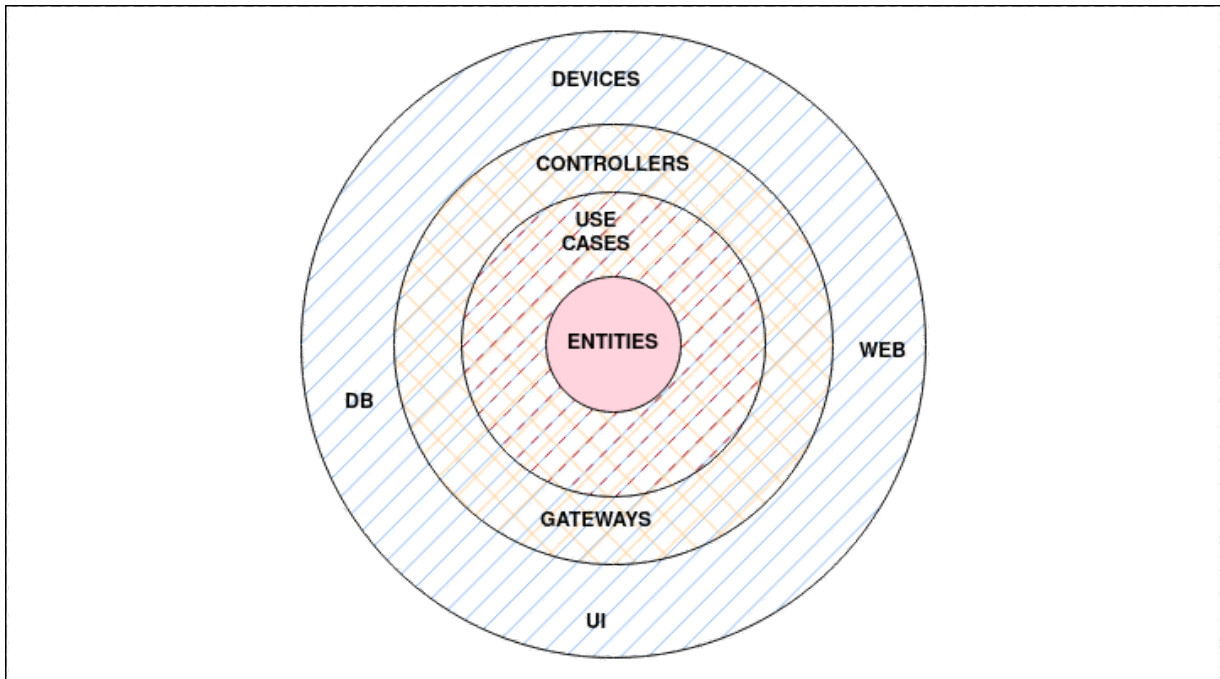
11. Slice as reference to an array: A slice only depicts a portion of the underlying array; it does not actually contain any data. A slice's underlying array's associated elements are changed when the elements of the slice are changed. These modifications will be visible to other slices that use the 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



(FIGURE -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 systematic collection of data is called a database. It might be anything, such as a straightforward grocery list, a photo gallery, or the enormous amount of data in a business network. A database management system, such as MySQL Server, is required to add, access, and process data contained in a computer database. Database management systems, whether used as stand-alone programmes or as a component of other programmes, are essential to computing because computers are excellent at processing vast volumes of data.

Instead of placing all the data in one huge warehouse, a relational database keeps the data in individual tables. Physical files that are optimized for speed contain the database structures. The logical model provides a flexible programming environment with objects like databases, tables, views, rows, and columns. One-to-one, one-to-many, unique, compulsory or optional, and "pointers" between distinct tables are a few examples of the rules you might build up to regulate the relationships

between various data fields. With a well-designed database, your application won't ever encounter inconsistent, duplicate, orphan, out-of-date, or missing data since the database enforces these rules.

"Structured Query Language" is what the SQL portion of "MySQL" stands for. The most popular standard language for accessing databases is SQL. Depending on your programming environment, you might explicitly enter SQL (for example, to generate reports), incorporate SQL statements into other languages' code, or use a language-specific API that obscures the SQL syntax.

1.2 PROBLEM STATEMENT

To Create a CRUD API using three layered architecture in golang. 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 id should be uuid, type should be enum, and fuel_type should be enum too.

For both the tables there are endpoints that need to be executed.

Customer

- GETByID
- POST
- UPDATE
- DELETE
- GETALL [it has two functionalities, one is to get all customer details, and next is getByFilters]

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

Vehicle

- POST

- UPDATE

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

1.3 OBJECTIVES

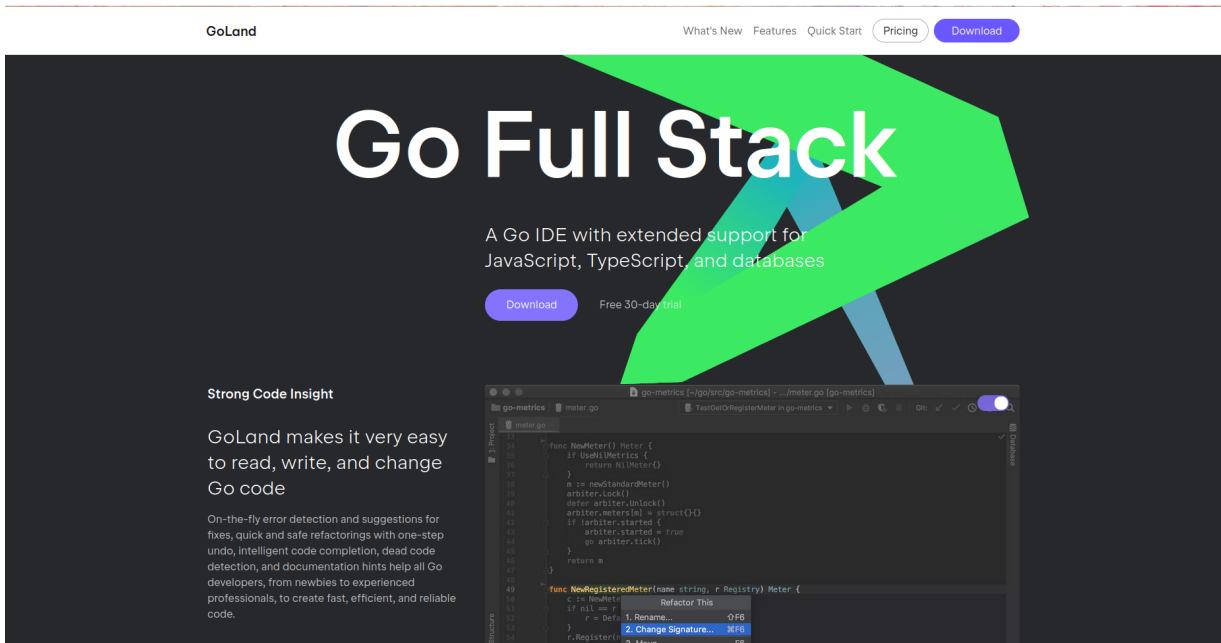
- To create testable, structured, clean and maintainable CRUD API by using industrial best practices.
- To understand Golang basic and advanced concepts.

1.4 METHODOLOGY

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

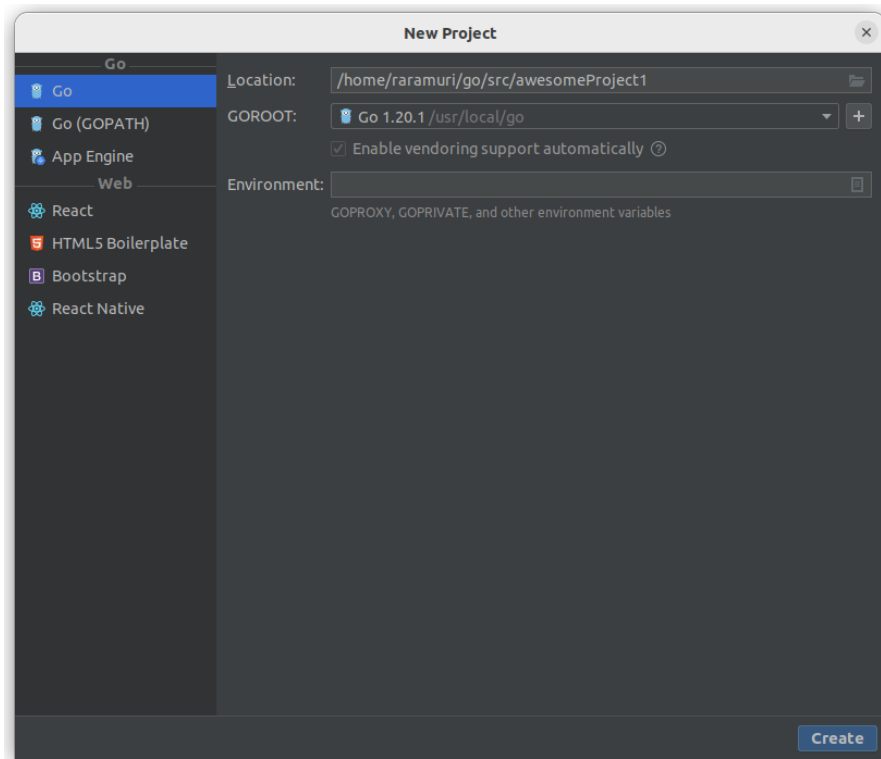
About Goland IDE

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.



(FIGURE -1.3)

GoLand IDE by JetBrains



(FIGURE -1.4)

Creating a New Project in GoLand

Test Driven Development(TDD)

In the method known as "test driven development," test cases rather than the code that verifies them are written first. It relies on repeating a quick development cycle. A method known as test driven development uses automated unit tests to direct 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

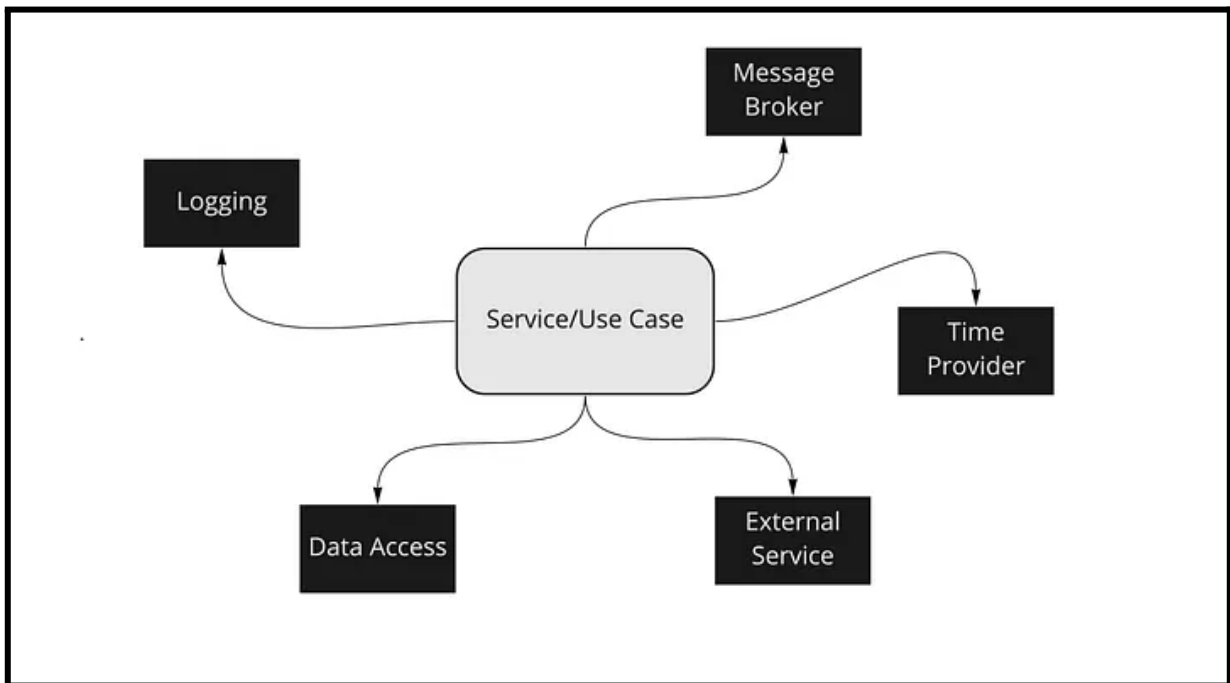
Advantages of TDD

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.



(FIGURE -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 - 2

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** 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 - 3

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

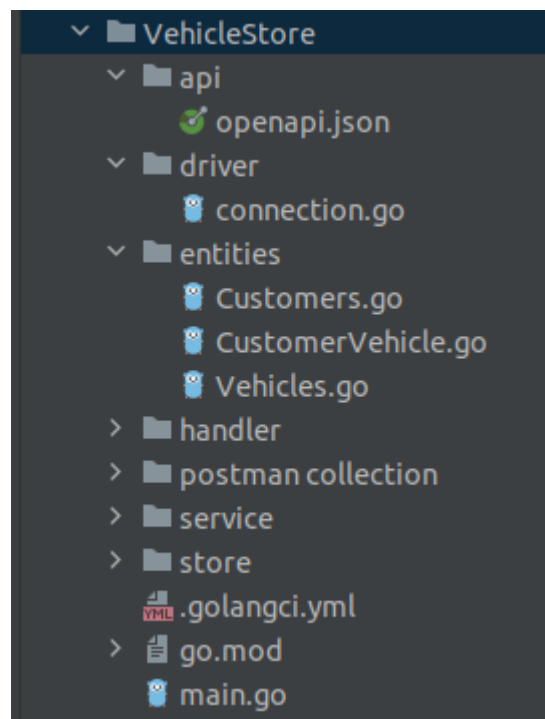
(TABLE -3.2)

Software Configurations

3.3 IMPLEMENTATION

Project structure

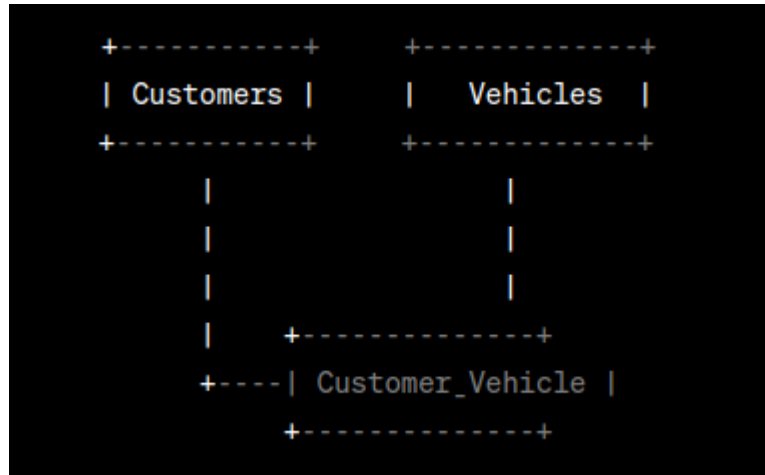
1. **Project Folder Name** : VehicleStore
2. **SubFolders**
 - **api**: Contains the swagger API documentation
 - **driver**: MySql connection file.
 - **entities**: go files having structures
 - **handler**: Handler layer files.
 - **service**: Service layer files
 - **store**: Store layer files
 -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



(FIGURE -3.1)

Project Structure

DB SCHEMAS



```
+-----+ +-----+
| Customers | | Vehicles |
+-----+ +-----+
|           | |           |
|           | |           |
|           | |           |
|           | |           |
|           | |           |
|           | |           |
|           | |           |
+-----+ +-----+
+----+ | Customer_Vehicle |
+-----+
```

(FIGURE -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.
```

(FIGURE -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)

```

(FIGURE -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)

```

(FIGURE -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)
```

(FIGURE -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 |
+-----+-----+-----+-----+-----+-----+-----+
```

(FIGURE -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)
```

(FIGURE -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.

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.

Next was 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

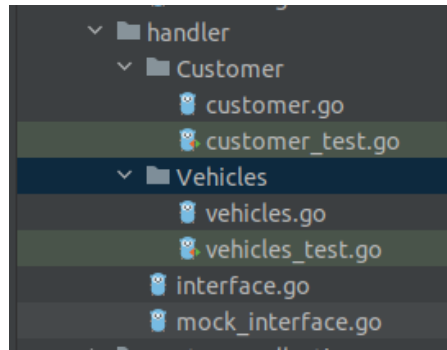
vehicles

vehicles.go

vehicles_test.go

interface.go

mock_interface.go



(FIGURE - 3.9)

Structure of handler layer

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

WHAT ARE MOCKS?

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

- 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 while it's important to understand how the test will look in advance

HOW DOES MOCK TEST WORK?

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.

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 :

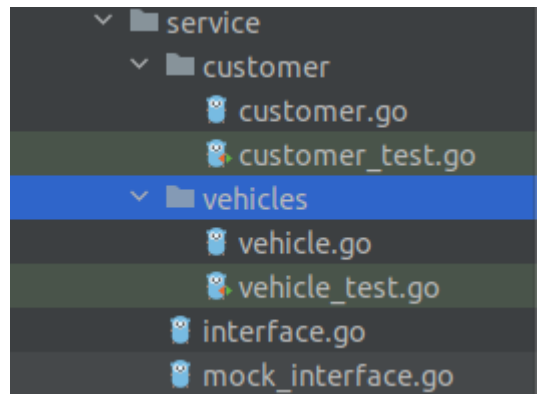
→ **service**

customer

customer.go

customer_test.go

vehicles
 vehicle.go
 vehicle_test.go
interface.go
mock_interface.go



(FIGURE -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.

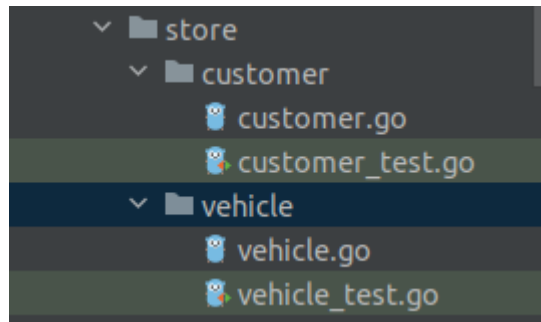
Next is the store layer where db queries are used and the database connection is established.

The structure of store layer is given as follows:

→ store

customer
customer.go
customer_test.go

vehicle
vehicle.go
vehicle_test.go



(FIGURE -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

An open platform for creating, distributing, and running programmes is Docker. You may divide your apps from your infrastructure with the help of Docker, allowing for rapid software delivery. You can manage your infrastructure using Docker in the same manner that you manage your applications. You may drastically shorten the time between writing code and executing it in production by utilizing Docker's methodology for shipping, testing, and deploying code quickly.

We installed docker image of MySQL and performed the operations on the image not the actual MySQL table.

CODE

```
package driver

import ...

type Database struct {
    DB *sql.DB
}

// SQLConnection method for creating database connection with golang.
func (d Database) SQLConnection() (*sql.DB, error) {
    var err error
    d.DB, err = sql.Open(driverName: "mysql", dataSourceName: "root:Nidhi@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
}
```

(FIGURE-3.12)

Sql Driver Connection

```
package entities

type Customers struct {
    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
const (
    Male   Genders = "Male"   no usages
    Female Genders = "Female" no usages
    Others Genders = "Others" no usages
)
```

(FIGURE-3.13)

Entities - customers.go (having Customers struct)

```
package entities

type Vehicles struct {
    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
const (
    Two   Wheels = "2"   no usages
    Four  Wheels = "4"   no usages
    Six   Wheels = "6"   no usages
)

type Fuels string // 16 usages
const (
    Petrol   Fuels = "Petrol" no usages
    Diesel   Fuels = "Diesel" no usages
    Cng      Fuels = "CNG"   no usages
    Electric Fuels = "Electric" no usages
)
```

(FIGURE -3.14)

Entities - Vehicles.go(having struct vehicles)

```
package entities

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

(FIGURE -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.StatusOK, 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("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)
        }
    }
}
```

(FIGURE - 3.16)

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

```

// 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.New("Method not allowed")},
    }

    ctrl := gomock.NewController(t)
    mock := handler.NewMockCustomerService(ctrl)
    for i, tc := range tests {
        r := http.NewRequest(tc.method, "/customers/"+tc.id, body: nil)
        w := http.NewRecorder()
        mock.EXPECT().GetByIdService(tc.id).Return(tc.expErr).MaxTimes(1)
        h := NewCustomerHandler(mock)
        h.GetById(w, r)

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

```

(FIGURE -3.17)

Handler test function for get by Id

```

package handler

import "net/http"

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

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

// CreateHandler takes in the customer details and creates a customer
func (ch CustomerHandler) CreateHandler(w http.ResponseWriter, r *http.Request) {
    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
    }
}

```

(FIGURE -3.18)

Handler Layer customer.go for create customer method

```

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
}

```

(FIGURE -3.19)

Get BY ID method in handler layer

```

// 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)
}

```

(FIGURE -3.20)

Create vehicle in vehicle.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 {

```

(FIGURE -3.21)

Update Vehicle in vehicle.go for handler layer

```

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("Two")}, expCode: http.StatusOK, expErr: nil},
        {desc: "Wrong method", method: http.MethodGet, vehicle: entities.Vehicles{ID: vId, Type: entities.Wheels("Two")}, expCode: http.StatusMethodNotAllowed, expErr: nil},
    }
    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(1)
        h := NewVehicleHandler(mock)
        h.PostHandler(w, r)
        assert.Equalf(t, tc.expCode, w.Code, "Test case #{i} failed.")
    }
}

```

(FIGURE -3.22)

Test function for create vehicle in vehicle_test.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
}
```

(FIGURE- 3.23)

Interface.go in handler layer

```
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)
}
```

(FIGURE- 3.24)

Create customer method in service layer

```

// GetByIdService take is in the id and gets the body of customer
// returns an error if id is invalid
func (cs CustomerServ) GetByIdService(id string) (entities.Customers, error) { 3 usages  ▲ Nidhi
    if id == "" : entities.Customers{}, errors.New("Invalid Id") ↗

    res, err := cs.customStore.GetByIdStore(id)
    if err != nil : entities.Customers{}, err ↗
    return res, nil
}

// GetAllService gets the customer and vehicle details for the given filters.
func (cs CustomerServ) GetAllService(isVehicle string, fuel string, brand string) (entities.CustomerVehicle, error) { 3 usages
    if !reflect.DeepEqual(fuel, y:"petrol") && !reflect.DeepEqual(fuel, y:"cng") && !reflect.DeepEqual(fuel,
        y:"diesel") && !reflect.DeepEqual(fuel, y:"electric") {
        return entities.CustomerVehicle{}, errors.New(text: "Invalid Fuel Type")
    }
    resp, err := cs.customStore.GetAllStore(isVehicle, fuel, brand)
    if err != nil {
        log.Println(err)
    }
    return resp, nil
}

```

(FIGURE - 3.25)

GetById and get All method in service layer

```

func TestCreateService(t *testing.T) { ▲ Nidhi*
    tests := []struct {
        desc      string
        customer  entities.Customers
        expRes    entities.Customers
        expErr    error
    }{
        { desc: "Customer Created",
          customer: entities.Customers{ ID: cId, Name: "Emily", Age: 24, Gender: entities.Genders("Female"),
            PhoneNumber: 91777777777, City: "Chicago", VehicleID: "ab9d6925-287b-40c7-b2e1-175928235cd2"},
          expRes: entities.Customers{ ID: cId, Name: "Emily", Age: 24, Gender: entities.Genders("Female"),
            PhoneNumber: 91777777777, City: "Chicago", VehicleID: "ab9d6925-287b-40c7-b2e1-175928235cd2"}, expErr: nil},
        { desc: "Missing data", customer: entities.Customers{ ID: cId, Name: "", Age: 20, Gender: entities.Genders(""),
            PhoneNumber: 91111111111, City: "south dakota", VehicleID: vId}, expRes: entities.Customers{},
          expErr: errors.New(text: "Missing Values are not allowed")},
        { desc: "Phone number length != 12", customer: entities.Customers{ ID: cId, Name: "Gabriel", Age: 45,
            Gender: entities.Genders("Male"), PhoneNumber: 5555, City: "Mystic Falls",
            VehicleID: "ab9d6925-287b-40c7-b2e1-175928235cd2"},
          expRes: entities.Customers{}, expErr: errors.New(text: "phone number length should be 12")},
        { desc: "Wrong Phone number code", customer: entities.Customers{ ID: cId, Name: "Mindy", Age: 30,
            Gender: entities.Genders("Female"), PhoneNumber: 123456789101, City: "Paris",
            VehicleID: "ab9d6925-287b-40c7-b2e1-175928235cd2"},
          expRes: entities.Customers{}, expErr: errors.New(text: "phone number code should be 91")},
        { desc: "Age>100", customer: entities.Customers{ ID: cId, Name: "KoL", Age: -3, Gender: entities.Genders("Male"),
            PhoneNumber: 914444444444, City: "NewYork", VehicleID: "ab9d6925-287b-40c7-b2e1-175928235cd2"},
          expRes: entities.Customers{}, expErr: errors.New(text: "age should be between 0 and 100")},
    }
}

```

(FIGURE - 3.26)

Test cases for Create customer in service layer


```

func TestGetByIdService(t *testing.T) {
    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.GenderMale}, expErr: nil},
        {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.expRes).MaxTimes(n: 5)
        _, err := h.GetByIdService(tc.id)
        if err != nil {
            log.Println(err)
        }
        assert.Equalf(t, tc.expErr, err, "Test case #{i} Failed")
    }
}

```

(FIGURE -3.27)

Test function for GetById in service layer

```

// CreateService takes the vehicle details and creates the vehicle
func (vs VehicleServ) PostService(vehicle entities.Vehicles) error {
    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
}

```

(FIGURE -3.28)

Method create vehicle in service layer

```

// 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.vehStore.UpdateStore(id, vehicle)
}

```

(FIGURE-3.29)

Update Vehicle Method in service layer

```

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.Equal(t, tc.expErr, err, msg: "TestCase:%v", i)
    }
}

```

(FIGURE - 3.30)

Test function for create vehicle in service layer

```

// 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
}

```

(FIGURE -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 id = ?")
    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
    }
}

```

(FIGURE -3.32)

Update customer and Delete customer in store layer

```

func TestCreateStore(t *testing.T) {
    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(1, rowsAffected: 1))
        err = d.CreateStore(c)
        assert.Equalf(t, err, tc.expErr, msg: "Test case %v failed", i)
    }
}

```

(FIGURE - 3.33)

Test Function for create customer in store layer

```

func TestGetCustomer(t *testing.T) {
    // 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 fetched", id: "2a43cf1e-d6ae-4710-a109-ec7e627e0c26", rows: sqlmock.NewRows([]string{"Id", "Name", "Age", "Gender", "City", "VehicleID"}).AddRow(values: "2a43cf1e-d6ae-4710-a109-ec7e627e0c26", "Tony", 50, "Male", 918888888888, "New York", "964f534e-c23a-11ed-afa1-0242ac120002")},
        { desc: "Invalid Id", id: "", rows: sqlmock.NewRows([]string{}).AddRow(), expRes: entities.Customers{}, expErr: errors.New("Invalid Id")},
    }
    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).WillReturnError(tc.expErr)
        _, err = d.GetByIdStore(tc.id)
        assert.Equalf(t, err, tc.expErr, "Test case #{i} Failed")
    }
}

```

(FIGURE -3.34)

Test Function for GET customer in store layer

```

import ...

type Database struct { 5 usages 1 Nidhi
    db *sql.DB
}

func NewVehicleStore(db *sql.DB) Database { 3 usages 1 Nidhi
    return Database{db: db}
}

// CreateStore implements the insert query in vehicles table.
func (d Database) CreateStore(vehicle entities.Vehicles) error { 3 usages 1 Nidhi

    result, err := d.db.Exec(query: "insert into Vehicles( id,type,fuel_type, brand, model,colour) values(?,?,?,?,,?)", ve
    if err != nil {
        log.Println(err)
        return nil
    }
    rowAffected, err := result.RowsAffected()
    if err != nil : err

    if rowAffected == 0 {
        log.Println(err)
        return nil
    }
    return nil
}

```

(FIGURE -3.35)

Create vehicle in store layer

```

func TestCreateStoreVehicle(t *testing.T) { 1 Nidhi
    db, mock, err := sqlmock.New(sqlmock.QueryMatcherOption(sqlmock.QueryMatcherEqual))
    if err != nil {
        t.Fatalf("failed to create mock database connection: #{err}")
    }
    defer db.Close()
    tests := []struct {
        desc string
        expRes entities.Vehicles
        expErr error
    }{
        {desc: "Customer created", expRes: entities.Vehicles{ ID: "964f534e-c23a-11ed-afa1-0242ac120002", Type: entities.Wheels(
    }
    store := NewVehicleStore(db)
    d := Database{db: store.db}
    for i, tc := range tests {
        v := entities.Vehicles{ ID: vID, Type: entities.Wheels("Six"), FuelType: entities.Fuels(""), Brand: "TATA", Model: "Prima
        mock.ExpectExec(expectedSQL: "INSERT INTO Vehicles VALUES (?,?,?,?,,?)").WithArgs(v.ID, v.Type, v.FuelType, v.Brand,
        err = d.CreateStore(v)
        assert.Equal(t, err, tc.expErr, "Test case #{i} failed")
    }
}

```

(FIGURE -3.36)

Test function for create vehicle in store layer

CHAPTER - 4

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

An API platform for creating and utilizing APIs is called Postman. To help you design better APIs faster, Postman improves collaboration and simplifies every stage of the API lifecycle.



(FIGURE- 4.1)

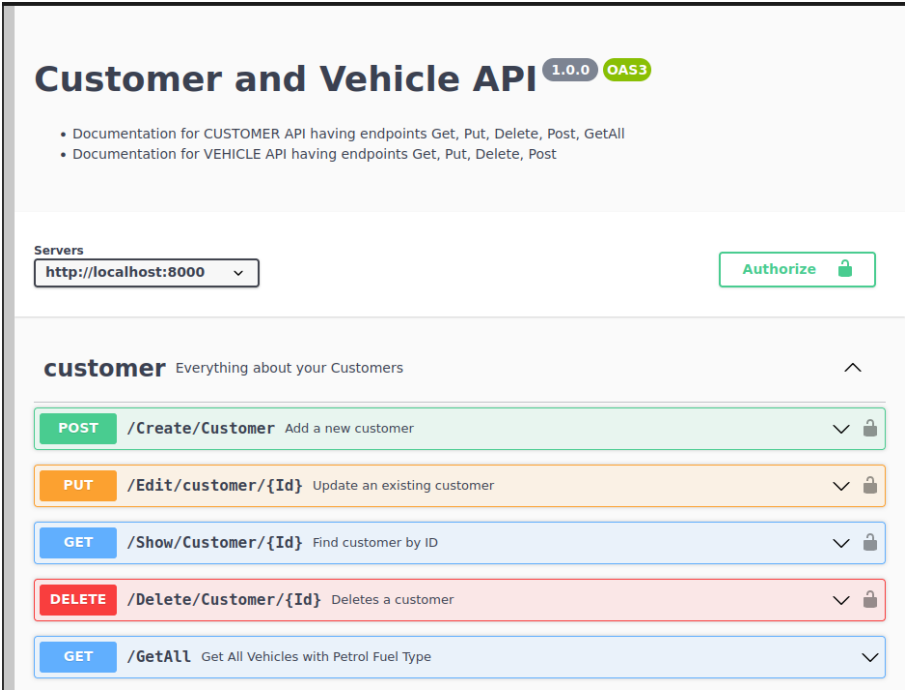
Postman

Everything connected to APIs, including API specifications, documentation, workflow recipes, test cases and results, metrics, and more, may be stored and managed by Postman.

The API documentation is made using swagger API tool.

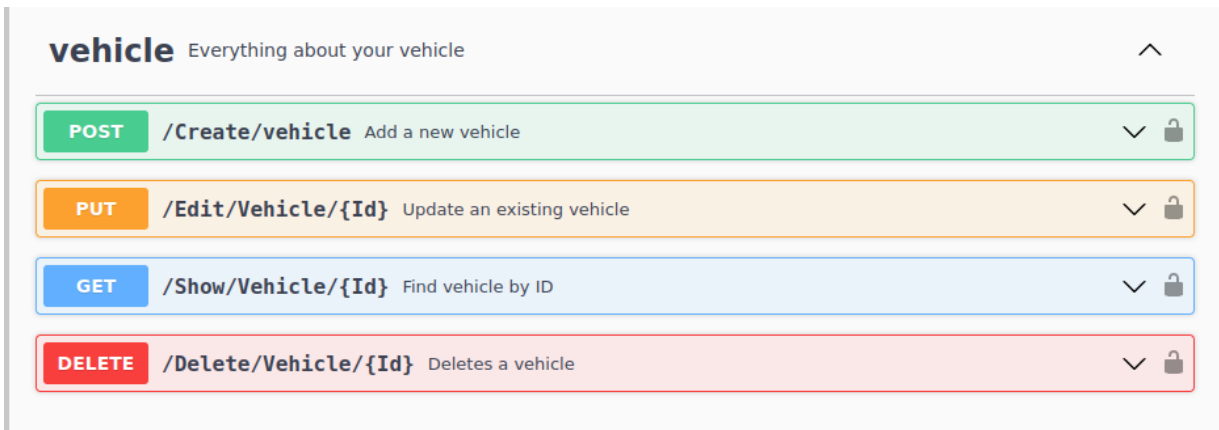
SWAGGER

You can define your APIs' internal structure in Swagger so that computers can understand it. The core of all goodness in Swagger is the ability of APIs to describe their own structures. Why is it so fantastic? We can, however, automatically create stunning and interactive API documentation by reading the structure of your API. We can also examine other options, such automated testing, and automatically produce client libraries for your API in a variety of languages. Swagger accomplishes this by requesting a YAML or JSON response from your API that offers a thorough description of your whole API.



(FIGURE- 4.2)

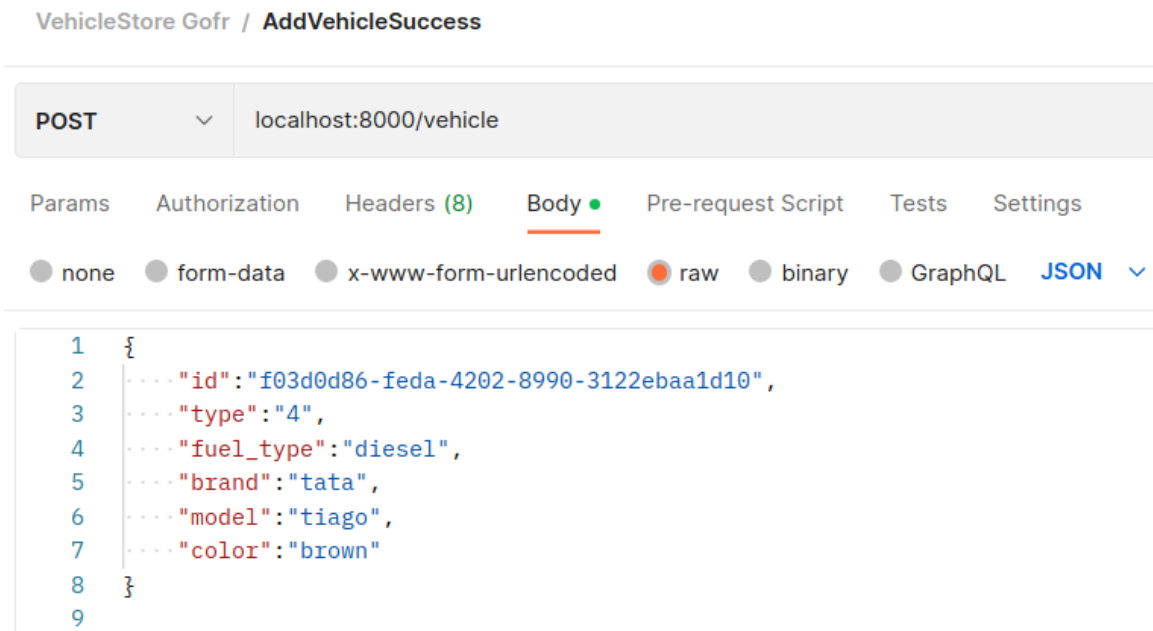
Swagger -API documentation for customer



(FIGURE- 4.3)

Swagger -API documentation for vehicle

ENDPOINTS RESULT FROM POSTMAN



(FIGURE- 4.4)

Add vehicle Endpoint

VehicleStoreAPI / UpdateVehicle

PUT localhost:8080/vehicle/786261f2-b75d-481b-81eb-530f0b3c525d

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

none form-data x-www-form-urlencoded raw binary GraphQL **JSON** ▾

```
1 {
2   .... "id": "0268f829-c636-4a5a-8fde-90546f202886",
3   .... "type": "4",
4   .... "fuel_type": "diesel",
5   .... "brand": "tata",
6   .... "model": "tiago",
7   .... "colour": "brown"
8 }
9
```

(FIGURE- 4.5)
Update vehicle endpoint

VehicleStoreAPI / AddCustomer


POST localhost:8080/customer


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

● none ● form-data ● x-www-form-urlencoded ● **raw** ● binary ● GraphQL **JSON** ▾

```
1  {
2    .... "id": "fe8cfa9a-d5bd-4775-8614-5b093de0f952",
3    .... "name": "Prashant",
4    .... "age": 21,
5    .... "phone_number": 919810967436,
6    .... "gender": "male",
7    .... "city": "goa",
8    .... "vehicle_id": "786261f2-b75d-481b-81eb-530f0b3c525d"
9  }
```



10
11
12
13

Body Cookies Headers (3) Test Results 

Pretty Raw Preview Visualize Text ▾ 

```
1 Entry has been added successfully
```

(FIGURE- 4.6)
Add Customer endpoint

VehicleStoreAPI / GetAllCustomer  


GET localhost:8080/customer?isVehicle=true

Params **●** Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

	Key	Value
<input checked="" type="checkbox"/>	isVehicle	true

Body Cookies Headers (3) Test Results

Pretty Raw Preview Visualize JSON 

```

1  [
2    {
3      "C": {
4        "id": "14bc1c62-ced1-4ab9-9e8b-1bc7392956cb",
5        "name": "Prashant",
6        "age": 21,
7        "phone_number": 919810967436,
8        "gender": "male",
9        "city": "goa",
10       "vehicle_id": "a9895b2e-e011-4145-b752-b49dabfee0c3"
11     },
12     "V": {
13       "id": "",
14       "type": "",
15       "fuel_type": "",
16       "brand": "",
17       "model": "",
18       "colour": ""
19     }
20   },

```

(FIGURE- 4.7)

GetAll endpoint

CHAPTER - 5

5.1 CONCLUSIONS

All in all, best practices and methods were considered during the creation of the ZopStore Programming interface. The handler, service, and storage layers were clearly separated from one another in a three-layer engineering process. The use of test-driven development throughout the development process ensured good code quality and reduced the risk of bugs.

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, GitHub Activities for automated testing, code inclusion, and linter tests improved the connection between development teams. Generally speaking, the ZopStore Programming interface was built using a range of best practices and methodologies, resulting in a top-notch, dependable, and performant Programming interface that solved the challenges of 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 inhouse 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 inhouse learning platform offered by Zopsmart technologies.

- [1] <https://go.dev/doc/>
- [2] <https://github.com/golang/mock>
- [3] <https://github.com/DATA-DOG/go-sqlmock>
- [4] <https://github.com/gorilla/mux>
- [5] <https://dev.mysql.com/doc/>
- [6] <https://www.linux.org/>
- [7] <https://docs.docker.com/>
- [8] <https://kubernetes.io/docs/home/>
- [9] <https://ngdocs.harness.io/>
- [10] <https://prometheus.io/docs/introduction/overview/>