THREE LAYERED CRUD API 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

ANSHU SHARMA (191317)

Under the supervision of

DR. TIRATHA RAJ SINGH
&
DR. RAJNI MOHANA
to

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

Certificate
Candidate's Declaration

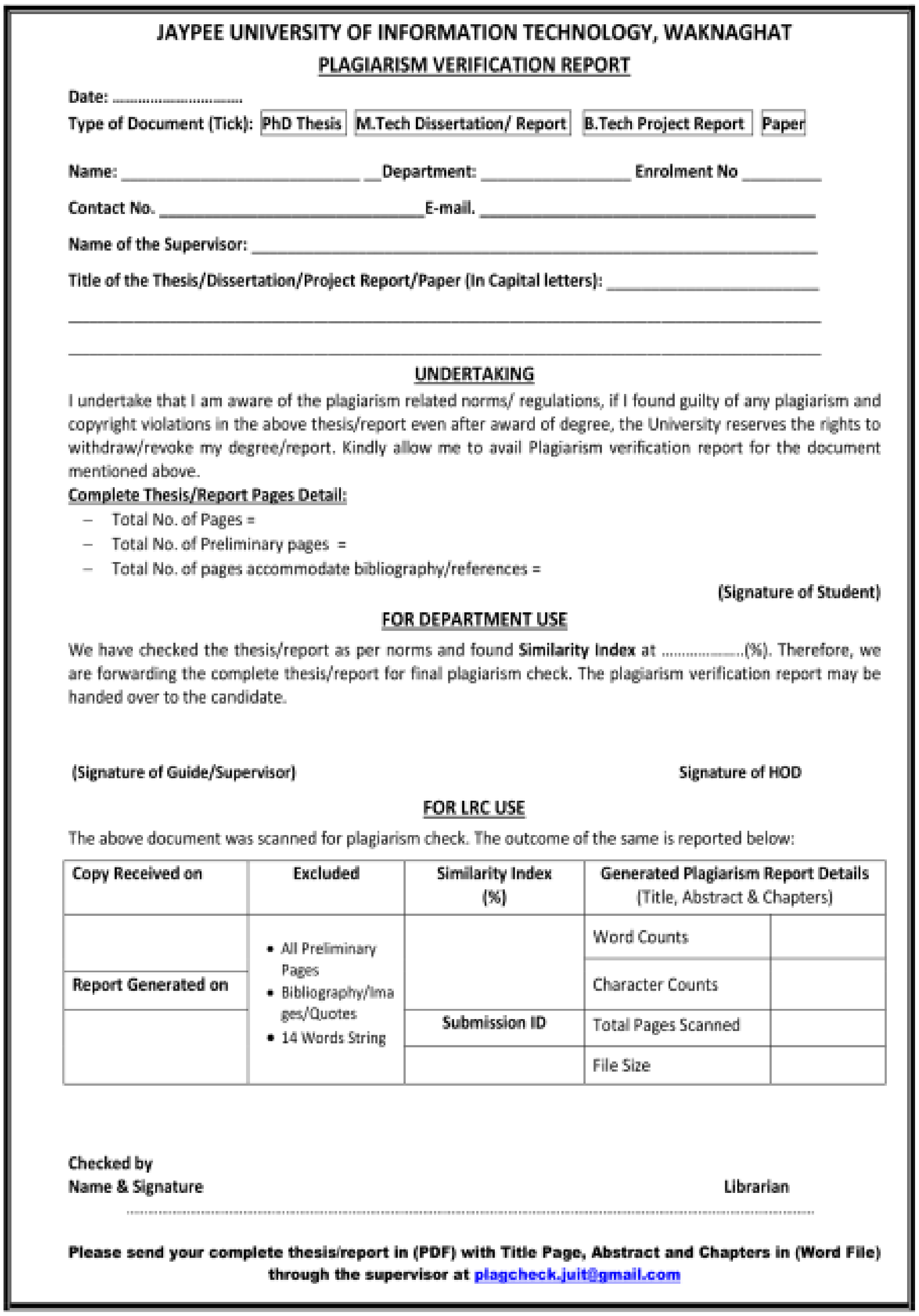
I hereby declare that the work presented in this report entitled **"THREE LAYERED CRUD API ARCHITECTURE"** in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering** submitted in the department of Computer Science & Engineering and Information Technology**,** Jaypee University of Information Technology Waknaghat is an authentic record of my own work carried out over a period from Feb 2023 to April 2023 under the supervision of **DR. TIRATHA RAJ SINGH** Associate Professor , 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)
ANSHU SHARMA 191317

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

(Supervisor Signature)
DR. TIRATHA RAJ SINGH
ASSOCIATE PROFESSOR (BT/BI)
Computer Science and Engineering/Information Technology

Dated:



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

First and foremost, I would want to express my heartfelt thanks and thankfulness to almighty God for his wonderful gifts, which enabled me to successfully finish the project work within the time frame specified.

I would like to express my heartfelt gratitude to my project supervisor, DR. TIRATHA RAJ SINGH, Associate Professor (BT/BI), Department of Computer Science & Engineering, Jaypee University of Information Technology, Waknaghat, India. Our supervisor's extensive experience and genuine interest in the topic of Machine Learning aided me much in completing this research. His unending patience, intellectual leadership, persistent encouragement, constant, vigorous supervision, constructive criticism, excellent counsel, reading many inferior draughts and correcting them at all levels, and reading many inferior draughts and correcting them at all stages enabled us to accomplish this project.

I would also like to express my heartfelt gratitude to everyone who has assisted me, directly or indirectly, in successfully completing this project. In this case, I'd also want to thank the different staff members, both teaching and non-teaching, who have provided handy assistance and assisted my project.

Last but not least, I must express my gratitude for my parents' unwavering support and faith.

ANSHU SHARMA, 191317

TABLE OF CONTENTS

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

LIST OF TABLES

ABSTRACT

The project "CRUD API in Golang using three-layered architecture" aims to offer a modular and scalable API for performing CRUD (provide, Read, Update, Delete) operations on a database. Because of its efficiency and ease of use, Golang is the most popular programming language.

By separating the display layer, business logic layer, and data access layer, the API's three-layered architecture provides greater maintainability and scalability. When users interact with API endpoints, the display layer sends requests to the business logic layer, which processes them and returns results. The data access layer handles CRUD operations and database connectivity.

The project creates a RESTful API that adheres to HTTP method basics and industry best practises. Aside from query parameters for filtering, sorting, and paginating data, the API provides endpoints for adding, receiving, updating, and removing data from the database.

The project is being constructed in accordance with test-driven development (TDD) guidelines, with unit and integration tests covering each important API component. The API is hosted on a cloud platform and containerized using Docker for scalability and availability.

Overall, this project provides a solid basis for creating scalable and maintainable three-layered Golang APIs.

CHAPTER - 1

INTRODUCTION

1.1 INTRODUCTION OF THE COMPANY

ZopSmart Technology is a software solutions company that provides all the resources you need to start an online store. Returning from its range of products, ZopSmart can help you build and run your dream company. It has many products such as Smart Store Eazy and Smart Payment Gateway. Zopsmart produces technology solutions for the retail industry. Its customers range from independent furniture retailers to multinational corporations, and its 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 mission and provides software solutions to some of the best companies.

Zopsmart Technology helps offline businesses looking to expand online by providing the tools and advice needed to build an online store. Their mission is to provide end-to-end products that add value to digitally savvy consumers. They show businesses how to better reach customers, thrive within budget, and get products online as soon as possible. They reduce time to market by helping customers rethink their business, customer interactions, and test analytics through human modeling.

1.2 INTRODUCTION OF THE PROJECT

In recent years, the need to create online applications that provide a good user experience has increased. Building a good API model that interacts with data and performs CRUD (create, read, update, delete) operations is essential for building powerful and capable applications. The performance and simplicity of the popular programming language Golang has captured the attention of the software development community.

The purpose of this project is to provide a CRUD API in Golang with a three-level design based on best practices. The design pattern, known as the three-tier architecture, improves API management and scalability by separating the process from the service and storage process. When interacting with an API endpoint, the processing layer sends requests to the service layer, which processes the requests and generates a response. CRUD operations and database communication are handled by the storage layer.

To assure code quality, we will adhere to test-driven development (TDD) standards, with unit tests and integration tests covering each essential API component. The RESTful API will be created in accordance with HTTP method principles and will include endpoints for adding, removing, updating, and retrieving data from the database.

1.2.1 WHAT IS API?

API stands for "Application Programming Interface". It is a set of guidelines, processes, and tools that facilitates information exchange and communication across different software applications.

Simply described, an API facilitates communication and data sharing by acting as a bridge between several software programmes. There are several methods that APIs may be implemented, including REST APIs, SOAP APIs, and GraphQL APIs. Without APIs, modern software development would be impossible since they enable programmers to build complex, sophisticated applications that interact with other software services and systems.

RestAPI with Go

REpresentational State Exchange, some of the time known as REST, is an engineering plan for conveyed hypermedia frameworks. We can streamline the by and large framework design and improve interaction perceivability by applying the sweeping statement rule to the component interface.

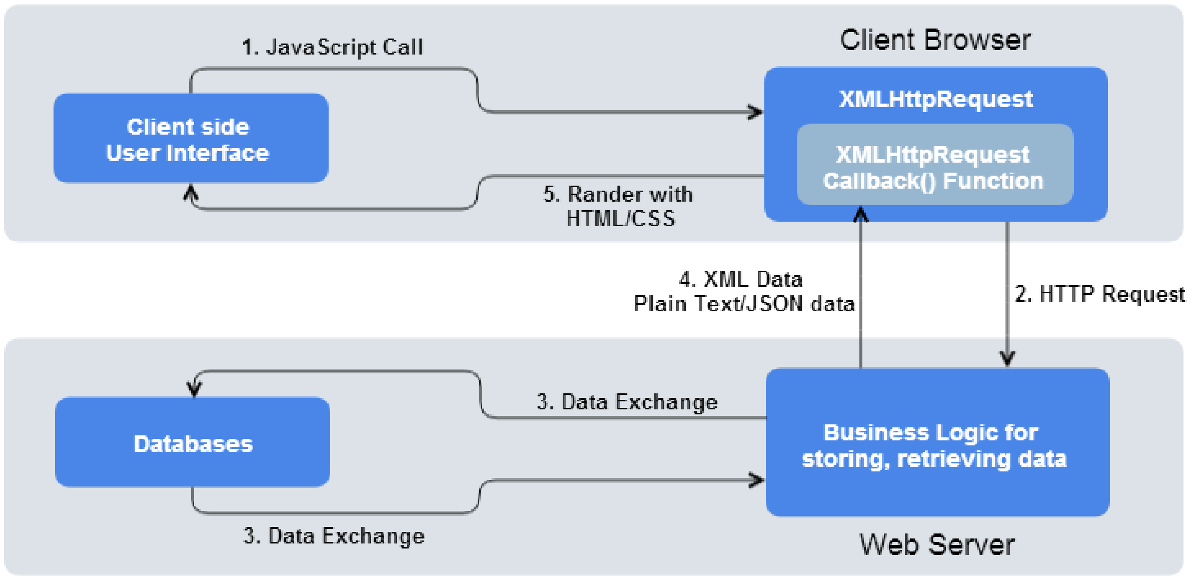
Several structural confinements help in accomplishing a bound together interface and controlling component behavior.

The four constraints listed below can help create a standard REST interface:

1. Each asset utilized within the interaction between the client and the server must be extraordinarily distinguished by the interface.
2. Assets ought to be spoken to reliably within the server reaction to avoid asset control. These representations ought to be utilized by API buyers to change the server's state of the assets.
3. Each asset representation ought to have sufficient subtle elements to clarify how to decipher the message. It ought to moreover detail any additional operations the client may carry out on the asset.
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 ?

Resources like HTML pages may be retrieved via the HTTP protocol. It is a client-server protocol, which means that all requests for data transmission on the Internet are initiated by the recipient, which is frequently the Web browser. A full document is created by combining the numerous sub-documents that are collected, including text, layout descriptions, images, videos, scripts, and more.


Client Browser
1. JavaScript Call
Client side User Interface
XMLHttpRequest
XMLHttpRequest Callback() Function
5. Rander with HTML/CSS
4. XML Data Plain Text/JSON data
2. HTTP Request
Databases
3. Data Exchange
Business Logic for storing, retrieving data
3. Data Exchange
Web Server

(FIGURE - 1.1)

Fetching a page

In contrast to a data stream, individual messages are exchanged between clients and servers during communication. Requests are messages sent by the client, which is often a web browser, and replies are messages delivered by the server in return.

Status Codes

In order to convey the outcome of a client's request, HTTP defines a number of standard status codes. The status codes are divided into five groups.

- **1xx: Informative -** information is sent at the protocol level..
- **2xx: Success -** represents the client's request being accepted with success.
- **3xx: Redirection -** indicates that more steps are necessary for the customer to complete their request.

- **4xx: Client Error -** These error status codes all place the blame on the customers.
- **5xx: Server Error -** Server's fault.

Here are some significant status codes:

S.NO	STATUS CODE	DESCRIPTION
1	200 OK	ACCEPTED
2	201 CREATED	CREATED
3	400 BAD REQUEST	BAD REQUEST
4	401 UNAUTHORIZED	UNAUTHORIZED INFORMATION
5	403 FORBIDDEN	ACCESS DENIED
6	404 NOT FOUND	NOT FOUND
7	500 INTERNAL SERVER ERROR	SERVER ERROR

(TABLE-1.1)

HTTP Status Codes

CRUD OPERATIONS

Make, Perused, Overhaul, and Erase, or CRUD, are the basic information administration exercises utilized in databases and APIs. By performing these activities, database engineers can adjust information in a database by including unused records, recovering ancient ones, overhauling ancient ones, and erasing ancient ones.

1. **Create:** Including a modern record to the database is done utilizing this operation. For occurrence, the data from the frame is spared as a modern record within the database once you yield one to form a unused account.
2. **Read:** Information from the database is recovered utilizing this strategy. For occasion, the item data is pulled from the database and appeared once you see a item page on an e-commerce site.
3. **Update:** This operation adjusts the database's current information. For occasion, the database is upgraded after you make changes to your profile data on a social media site.
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

Go programming is procedural in nature. It was created by Google employees Robert Griesemer, Rob Pike, and Ken Thompson in 2007, although it wasn't published as an open-source programming language until 2009. To handle dependencies efficiently during programme assembly, packages are utilised. This language allows environment adoption patterns just like dynamic languages do.

Go is a concurrent, statically typed, and garbage-collected programming language that was created at Google in 2009. Due to its emphasis on simplicity, efficiency, and speed of comprehension, it is a popular choice for creating command-line tools, web applications, and scalable network services.

Concurrency, or the capacity to carry out a few exercises at once, is bolstered by Go. Go's utilize of Goroutines and Channels, which empower you to make code that can execute numerous activities concurrently, permits for concurrency. Because of this, Go could be a awesome choice for making high-performance, adaptable arrange administrations as well as for settling challenging computational issues.

Key Features of Golang:

- Go is basic to memorize and utilize since of this. Its clear language structure makes it a practical choice for both unpracticed and prepared software engineers.
- Go highlights built-in concurrency back, empowering software engineers to form adaptable and viable code for multicore and dispersed applications.
- Developers are spared of the burden of handling memory allocation and deallocation thanks to Go's automated memory management.
- Go's quick compilation times enable quick iteration throughout development.
- Go can be built to work on a wide run of working frameworks, counting Windows, Linux, and macOS.
- Go may be a statically written dialect, therefore errors are caught at compile time instead of amid runtime.

GO Basics

1. **Packages:** Every Go programme has packages. The programmes launch in the bundle main. A package is a container with several uses to do certain tasks. The math package, for instance, provides the Sqrt() function to determine the square root of a value.
2. **Imports:** When using the import keyword, the desired package is imported from the given directory or, in the absence of a path, from the directory of $GOPATH. Simply copying the selected package from its source directory to the target code—the main program—is all that is required to import a package. Since it brings the packages that are strictly need to run applications, import is essential in Go.

3. **Functions:** In a programme, functions are often a group of instructions or statements that allow the user to reuse previously written code to save memory use, speed up execution, and—most importantly—to enhance readability. A function is essentially a collection of statements that cooperate to carry out a certain job and provide the result to the caller. A function can also do a certain task without ending in a result.
4. **Named Return Values:** Go's return values might be named. If so, they are treated as variables defined at the top of the function.These names ought to be used in the documentation for the return values.A return statement without arguments returns the stated return values. A "naked" return is what is meant by this.Only short functions, such as the one in this example, should make use of naked return statements. They could make longer functions more difficult to read.
5. **Shorthand variable declarations:** Instead of using a var declaration with implicit type inside of a function, the:= short assignment statement may be used. Because every sentence begins with a keyword (var, func, and so on), the:= construct cannot be used outside of a function.
6. **Zero Values:** Factors that are announced without a clear beginning esteem are alloted zero.For numeric sorts, the zero esteem is 0; for boolean sorts, wrong; and for string sorts, "" (the purge string).
7. **Type Inference:** A variable's sort is gathered from the esteem on the correct side when it is pronounced without an unequivocal sort being indicated (either by utilizing the := sentence structure or the var = expression sentence structure).
8. **For is Go's "While":** In Go, we utilize the whereas circle to execute a square of code until a certain condition is met. Not at all like other programming dialects, Go doesn't have a devoted watchword for a whereas circle. In any case, we will utilize the for circle to perform the usefulness of a whereas circle.
9. **Defer:** With a concede articulation, a function's execution is delayed until the encompassing work completes its run.The inputs of the put off call are evaluated

right absent, but the work call isn't carried out until the encompassing work has wrapped up. Work calls that are conceded are pushed into a stack. Conceded calls are handled in last-in, first-out arrange when a work returns.

10. **Slices:** The size of an array is fixed. A slice, on the other hand, provides a flexible, dynamic view of an array's elements. In real use, slices are much more common than arrays.A type []T slice is one having components of type T.

    In Go, there are several ways to create a slice:

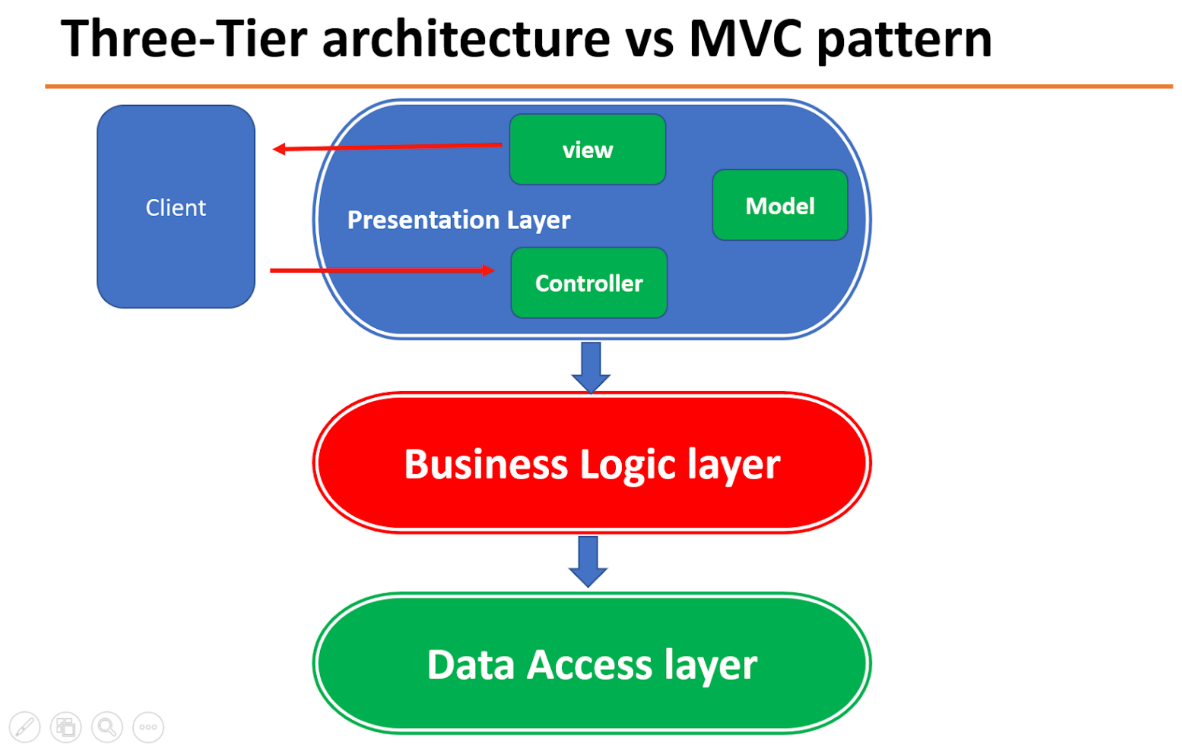
    - The []data type values format may be used to create a slice from an array.
    - It uses the make() function.

11. **Slice as reference to an array:** A cut as it were delineates a parcel of the fundamental cluster; it does not really contain any information. A slice's fundamental array's related components are changed when the components of the cut are changed. These adjustments will be unmistakable to other cuts that utilize the same fundamental cluster.
12. **Length and Capacity:** A powerful and long document. The length of a piece is determined by the number of elements it has. The capacity of the slice calculated from the first element in the slice is all points in the lower row. The len(s) and cap(s) expressions can be used to determine the length and capacity of a part.

THREE LAYERED ARCHITECTURE

The three layers architecture are

- Handler
- Service
- Store



(FIGURE -1.2)

Three Layered Architecture

HANDLER LAYER

The layer is also known as the distribution layer. The request will be received by the delivery process that separates it for the required information. The answer is written to the answer author after reviewing using the document that the call is well established.

SERVICE LAYER

The Use-Case Layer is another name for it. The use case layer is in charge of the application's business logic. The datastore layer will interface with this layer. After obtaining what it needs from the delivery layer, it invokes the datastore layer. Both before and after invoking the datastore layer, the relevant business logic is applied.

STORE LAYER

The Data-store layer is another name for it. The datastore houses the data. You can utilise any kind of data storage device. The use case layer is the only one that communicates with the datastore. Each layer may be separately verified in this way from the others.

The only layer that will change if the application grows to support gRPC is the delivery layer because each layer runs independently from the others. The use case layer and datastore won't change. Even if the datastore changes, the entire programme does not have to be altered. Only the datastore layer will change. Because of this, updating the code, locating and fixing bugs, and growing the programme are all made simple.

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

MySQL

MySQL is simply a database management system.
The systematic collection of information is called a database. It can be anything from a simple shopping list to a photo library or a large database in network marketing. A database management system such as MySQL Server is required to add, access, and manage information contained in computer databases. Whether used as a standalone service or as an integral part of other services, a database management system is essential for computing because computers are so good at processing more information.

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 versatile programming environment is offered by the logical model, which includes objects like databases, tables, views, rows, and columns. You might create rules to govern the relationships between different data fields, such as one-to-one, one-to-many, unique, required or optional, and "pointers" between different tables. Since a well-designed database upholds these constraints, your application won't ever run across inconsistent, duplicate, orphan, out-of-date, or missing data.

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

1.2 PROBLEM STATEMENT

Build CRUD API using three-tier architecture in Golang. Access tables using MySQL connections. The name of the file should be zopstore.

There should be two tables, Customer and Vehicle. The id for the client must be uuid, age must be between 0 and 100, gender must be an enum, phone number must start with code 91 and length must be no more than 12.

There are endpoints that must be executed for both tables.

Customer

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

The following filters are available:

a. If "vehicle" is true, receive all vehicle information for that specific customer;

b. receive cars based on "fuel_type"

c. Purchase cars based on brand

Vehicle

- POST
- UPDATE

Snake_case should only be used for database naming conventions, and camelCase should be used in the code.

1.3 OBJECTIVES

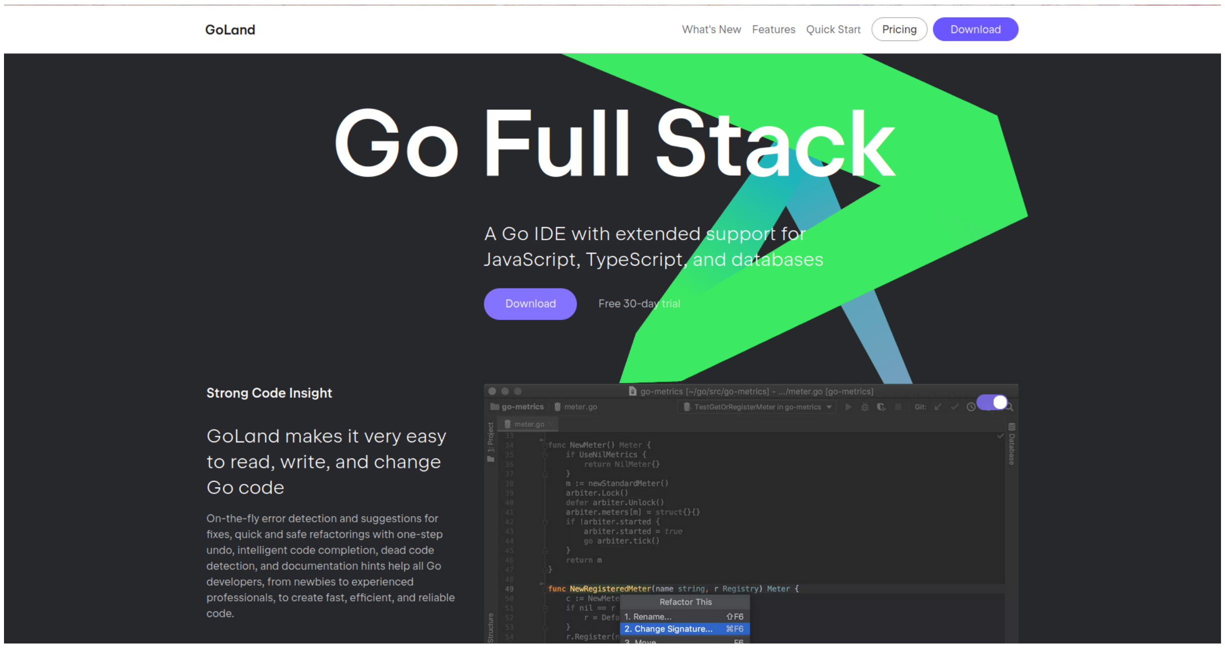
- to use industry best practises to design tested, organised, clean, and maintainable CRUD API.
- Learn basic and advanced concepts of Golang.

1.4 METHODOLOGY

- learn the fundamentals of golang
- learn the fundamentals of connecting to mysql in golang
- In a database, create tables.
- Layers may be created in Goland IDE.
- Utilising Test Driven Development, create test cases for each tier.
- In all the levels, write the function implementation.
- Implement Dependency Injection

About Goland IDE

It is a Golang IDE that JetBrains.com offers. On our local system, it supports the creation of a go programme as well as its running, debugging, and many other functions. Using Goland IDE, we are able to construct Go projects.



Go Full Stack

A Go IDE with extended support for JavaScript, TypeScript, and databases

Strong Code Insight

GoLand makes it very easy to read, write, and change Go code

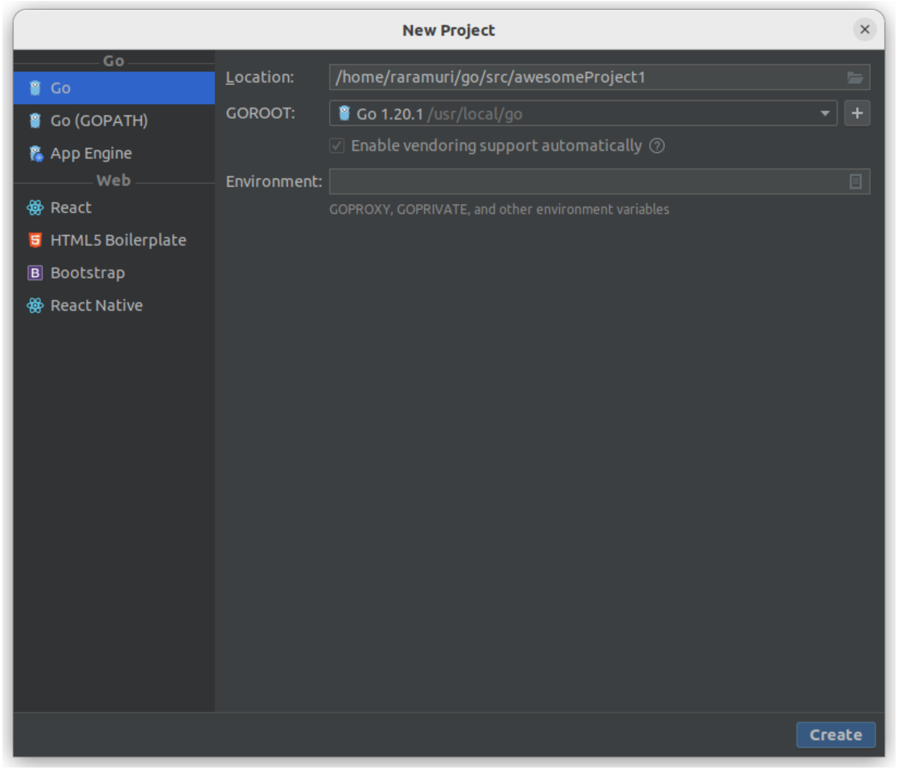
On-the-fly error detection and suggestions for fixes, quick and safe refactorings with one-step undo, intelligent code completion, dead code detection, and documentation hints help all Go developers, from newbies to experienced professionals, to create fast, efficient, and reliable code.

go-metrics [~/go/src/go-metrics] - .../meter.go [go-metrics]

```
func NewMeter() Meter {
    if UseNilMetrics {
        return NilMeter{}
    }
    m := newStandardMeter()
    arbiter.Lock()
    defer arbiter.Unlock()
    arbiter.meters[m] = struct{}{}
    if !arbiter.started {
        arbiter.started = true
        go arbiter.tick()
    }
    return m
}

func NewRegisteredMeter(name string, r Registry) Meter {
    c := NewMete
    if nil == r
        r = Defa
    }
    r.Register(n
```

(FIGURE -1.3)

Goland IDE by JetBrains



Location: /home/raramuri/go/src/awesomeProject1
GOROOT: Go 1.20.1 /usr/local/go
☑ Enable vendoring support automatically
Environment:
GOPROXY, GOPRIVATE, and other environment variables

(FIGURE -1.4)

Creating a New Project in Goland

Test Driven Development(TDD)

In a method called "test-driven development", test cases are written first instead of the code that analyzes them. He relies on recovery to get the job done quickly. One method, called test-driven development, uses automated unit tests to guide design and ensure unrestricted separation of progress.

The TDD approach includes the following steps:

- By developing a test case that completely illustrates the procedure, you may add a test. In order to generate the test cases, the developer must fully understand the features and requirements using user stories and use cases.
- Run each test case to ensure that the new test case is unsuccessful.
- Ensure that the test case is satisfied by your code.
- Put the test scenarios into actions.

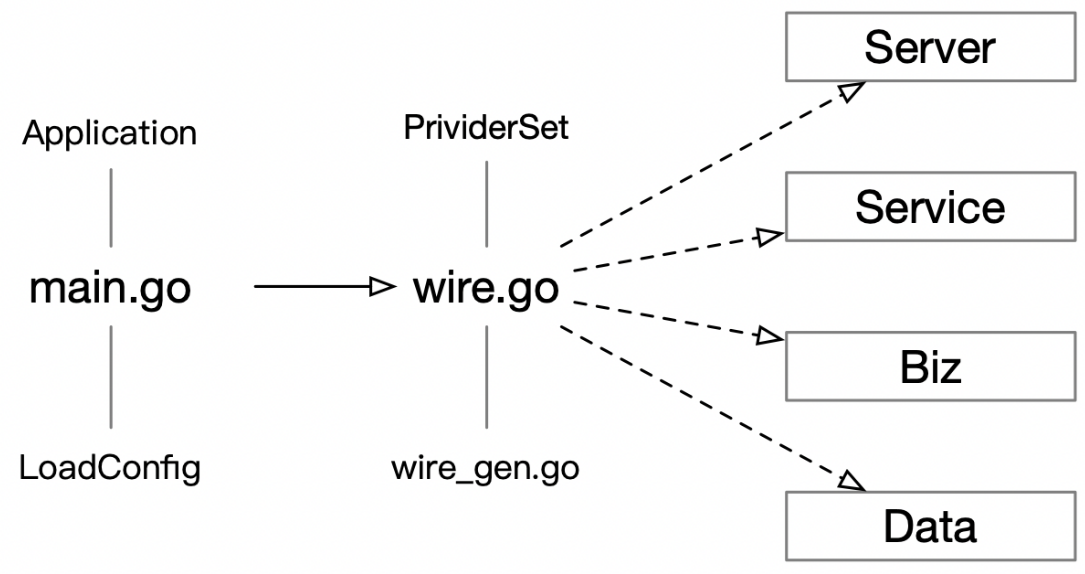
- Refactoring the code gets rid of duplicate code.
- repetition of the previous stages

Advantages of TDD

1. Unit tests offer continuous input on the functions.
2. Enhancing design quality also helps with proper maintenance.
3. Using test-driven development offers protection from bugs.
4. TDD makes ensuring that your application complies with all of the specifications that have been laid down for it.
5. The TDD development lifecycle is rather short.

DEPENDENCY INJECTION

Dependency Injection, sometimes called DI, is a technique for implementing code parsing using some best practices. Sentences that do not make sense together should be alone or separately. Code injection is required to create this dependency. This is a simple explanation of successful shoots.



(FIGURE -1.5)

Dependency injection

A design called Dependency Injection can help you make decisions outside of your application. Applications are often based on external APIs, databases, etc. requires. An application must accept its expectations and use them appropriately; It's not a job to know these details. Consider a scenario where your application contains the following code.

Avoiding injections (patterns) but not injecting abstractions (interfaces) is an important part of injection prevention. It allows you to quickly switch between using specific progressions and transitions from real use to app. The basic unit of measure depends on it.

1.5 ORGANIZATION

There are five chapters in the report. The project's chapters together provide crucial information. The project's fundamental introduction is provided in the first chapter, which covers CRUD OPERATIONS, GOLANG, and THREE LAYERED ARCHITECTURE. The literature review of all the research articles we used and read while creating this project is included in the second chapter.The third chapter contains information on system development and the use of various software programmes, databases, project feasibility studies, limits, etc. The performance analysis is in the fourth chapter. The API project's results and next work are covered in Chapter 5.

CHAPTER - 2

LITERATURE SURVEY

1. **GO Documentation** Go is a statically typed, open source, compiled programming language that was created by R Griesemer, R Pike, and K Thompson while working at Google. Information is available in the Go manual for every topic listed or utilised in golang.
2. **MySQL Documentation** Open-source relational database management system MySQL, also known as My Structured Query Language, enables us to store data, retrieve details, remove a record, etc.
3. **GoMock** Gomock is a mocking framework that works well with the built-in testing package in the GO programming language.
4. **Git and Github Official documentation** It helps you become familiar with the principles of a version control system, such as Git, and how it functions with GitHub.
5. **Three Layered Architecture Industrial Documentation** on three layered architecture used in ZopSmart Technology by Chaitra AV
6. **HTTP Documentation** in regards to the use of status codes, client-server architecture, requests, and answers, etc.
7. **SQL Mocks Documentation**
8. **Dedicated Training and upskilling platform** part of the firm.

CHAPTER - 3

SYSTEM DESIGN & DEVELOPMENT

3.1 ANALYSIS

Feasibility Study

This project aims to provide a fair and efficient way to manage data using an HTTP based API to create, read, modify and delete transactions. The three-tier architectural concept will facilitate maintenance and future growth while ensuring a clear separation of concerns.

When looking for this quality, the following items will be examined:

Technological viability
Operational viability
Financial viability

Technological feasibility: In order to build scalable and useful apps, the project comprises developing a CRUD API in the programming language Golang. Programming language Golang is established and often used. Having built-in support for HTTP servers and clients, Golang is a powerful standard library that is ideal for creating online applications. Because there would be a distinct separation of concerns with the recommended three-layered architecture, it will be easier to manage the codebase and maintain the application over time.

Operational feasibility: The CRUD API's RESTful architecture, which complies with accepted conventions and standards, makes it easy to use and connect with other systems. Using common libraries or API clients, users will be able to rapidly generate HTTP calls for the API to create, read, update, and remove data. Because it is simple to alter single components or add additional levels, the recommended architecture will also make future maintenance and extension easier.

Financial feasibility: A variety of factors, including as development expenses, hosting costs, and income possibilities, will have an impact on the project's financial feasibility. The cost of development will depend on the application's complexity and the team's level of experience.

3.2 REQUIREMENTS

A system need is the capacity of the system to satisfy the conditions desired by the users.

System requirement analysis is completed by separating the requirements into functional and nonfunctional requirements.

Functional requirements

1. **Authentication**
    - The API should provide user authentication and permission.
    - It should be possible for users to create accounts and sign in using them.
    - Roles and permissions for users should be defined and maintained.
2. **Data Management**
    - Data management should be possible using the Create, Read, Update, and Delete actions of the API.
    - Data must be saved in a database or file system.
    - The API should verify data inputs and keep track of consistency.

3. **Data Filtering & Sorting**
    - Data sorting and filtering should be supported through the API using a variety of criteria.
    - The ability for users to query data using specific criteria should be provided.
4. **Error Handling**
    - The ability for users to query data using specific criteria should be provided.
    - Error messages must be truthful.
5. **Testing**
    - All test cases for API methods should pass with 100% coverage.
    - For all layers of unit testing, dependency injection and effective mock use are required.
6. **Documentation**
    - Information must be accurate and validated for all attributes, including appropriate registration requirements.

Non-Functional Requirements

1. ascendable
2. production
3. Readability
4. legibility
5. Usable API
6. justifiable

Technical Requirements

1. GOLand is an IDE for developing clean code.
2. Postman is a platform for API development and consumption.
3. Mysql server provides database management solutions with connection and querying features.
4. Swagger is used for API documentation.
5. VCS Git

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

OS Ubuntu

Coding language GO
Runtime environment GO runtime
Package Manager GO

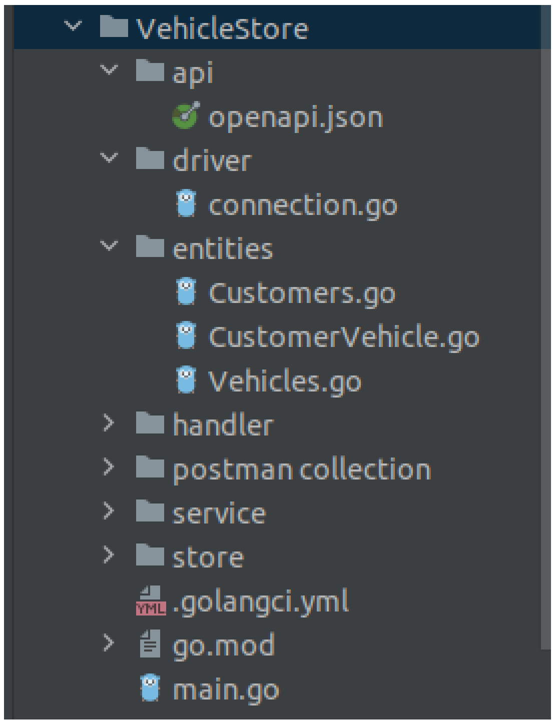
(TABLE -3.2)

Software Configurations

3.3 IMPLEMENTATION

Project structure

1. **Project Folder Name :** VehicleStore
2. **SubFolders**
    - **api:** The documentation for the Swagger API is included.
    - **driver:** Connection file for MySql.
    - **entities:** go files with structure
    - **handler:** Layer files to handle.
    - **service:** Service layer documents
    - **store:** Layer files should be saved.
    - **postmanCollection:** The API Postman Collection
3. **main.go**
4. **go.mod:** dependency

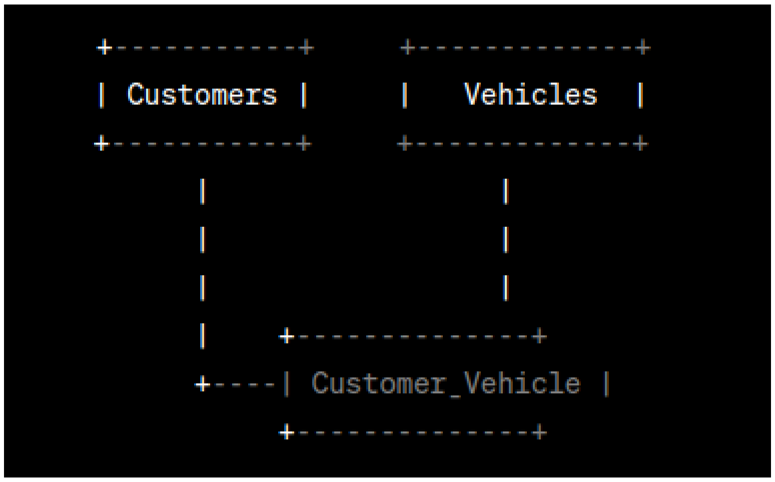


```
VehicleStore
  api
    openapi.json
  driver
    connection.go
  entities
    Customers.go
    CustomerVehicle.go
    Vehicles.go
  handler
  postman collection
  service
  store
  .golangci.yml
  go.mod
  main.go
```

(FIGURE -3.1)

Project Structure

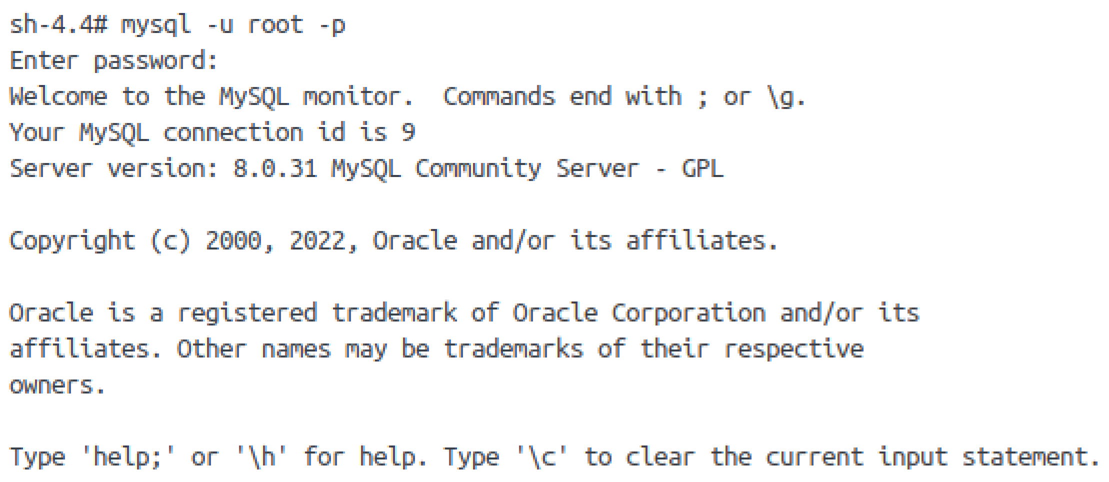
DB SCHEMAS



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

(FIGURE -3.2)

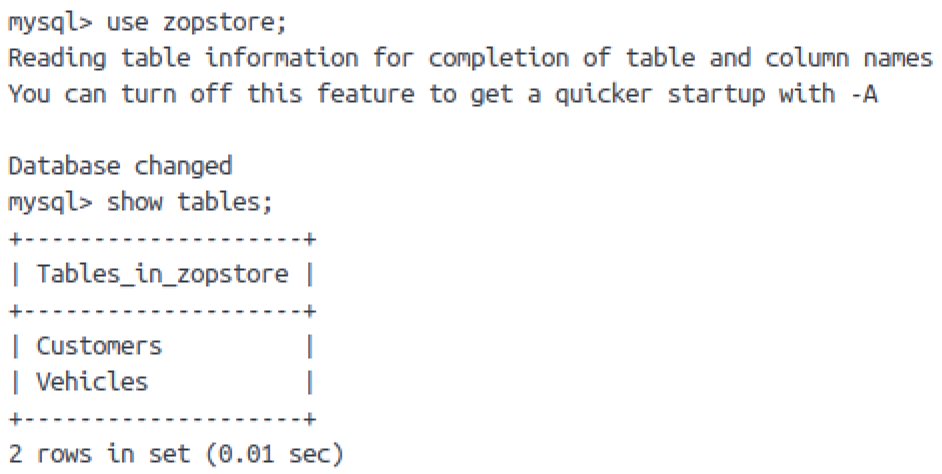
DB SCHEMA



```
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)

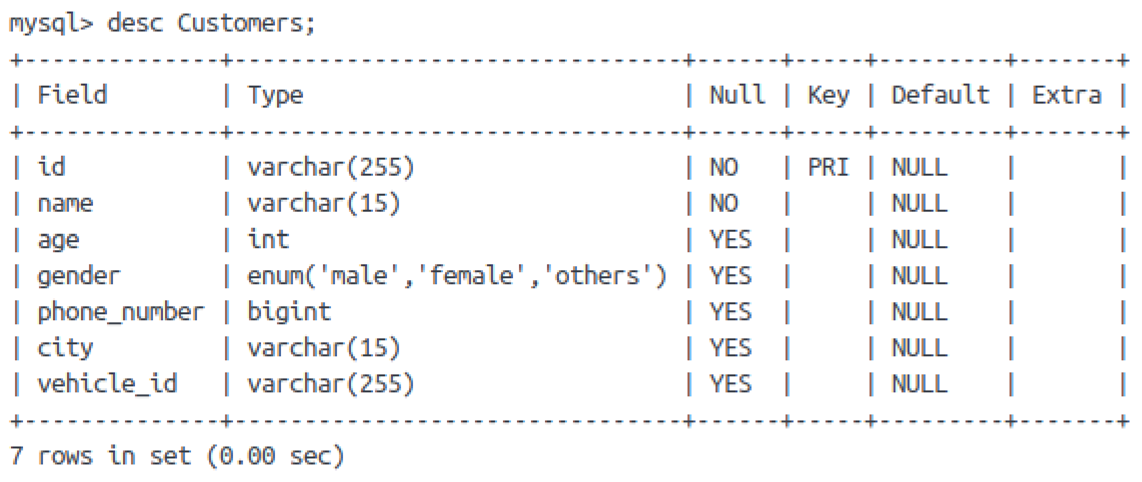
MySQL docker image



```
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)

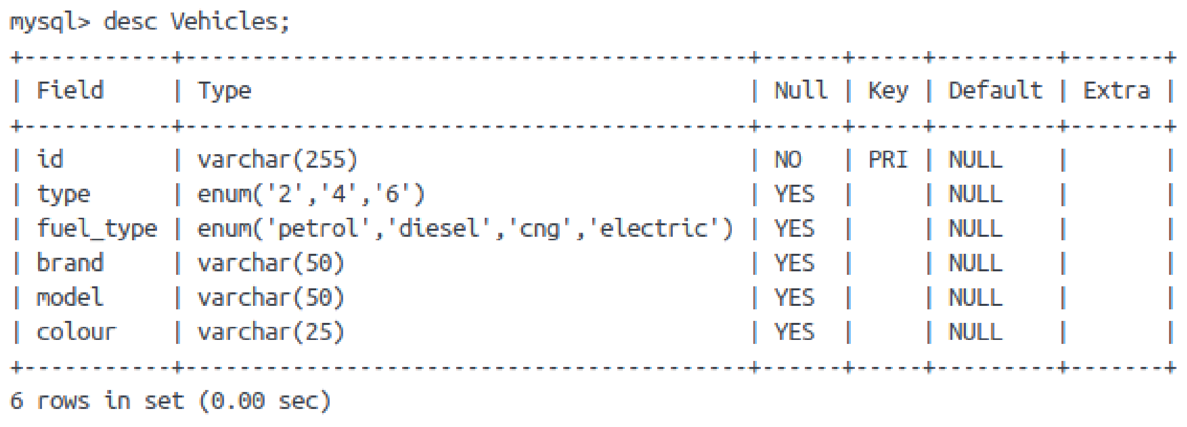
Tables in the Database



```
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)

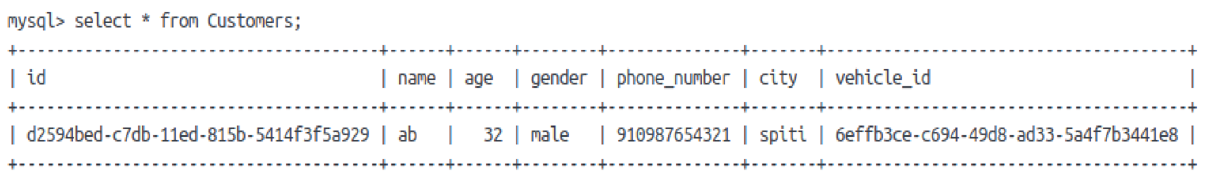
Customer Description



```
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)

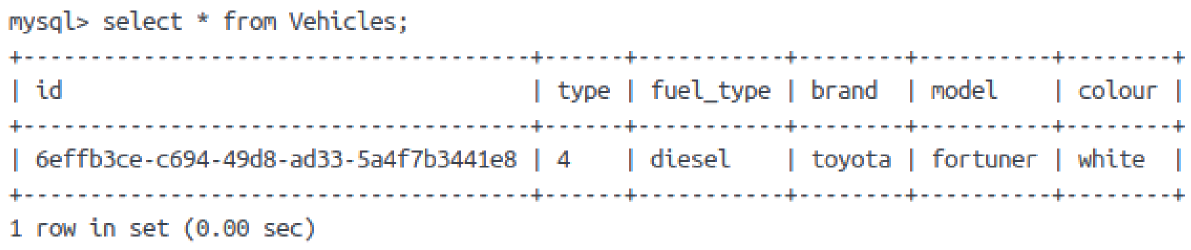
Vehicle Description



```
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

In this image, the "Customers" table contains information about the customers, such as their ID (as a UUID), name, age, gender, phone number, city, and the foreign key for the automobile they own. The "Vehicles" table contains information about the vehicles, including their ID (as a UUID), type (2, 4, or 6 wheelers), fuel type (petrol, diesel, compressed natural gas, or electric), brand, model, and colour.
The "Customer_Vehicle" table, which represents the relationship between the "Customers" and "Vehicles" databases, is a many-to-many relationship. This table contains both the customer ID and the vehicle ID that links them.

The tables can be accessed through the following API endpoints:
API for users

- Using the GET /customers/id> query, you may get a customer by ID.
- Add a new customer.
- Using PUT, you may update an existing client by ID.
- Using DELETE, delete a client by ID.
- GET is used to retrieve customers. Obtain every client.
- GET /client?If vehicle=true is specified, get information about all clients' cars.
- GET /client?Using the fuel_type= fuel_type argument, get all customers with cars that use a specific fuel type.

- GET /client?Using the query brand="brand", get all clients who own vehicles of the specified brand.

API for Vehicles

- Create a new automobile using POST /vehicles.
- PUT /vehicles/ to update a current vehicle by ID.
- Delete a vehicle by entering DELETE /vehicles/.

SYSTEM DESIGN

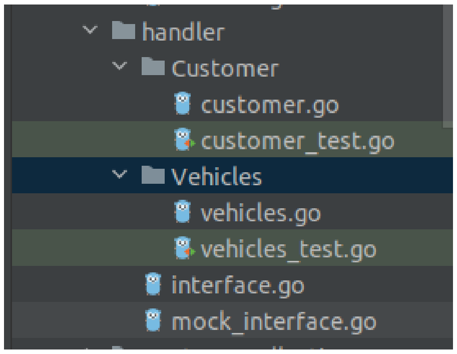
The use of each layer is individual. First of all, the sites and models to be used in the project are created in a separate folder called entity. There are three files in the location folder, Customers.go, Cars. Go to CustomerVehicle.go. Each has different models named Customer, Vehicle, and Customer. The next part is to create an SQL connection to run the database. You should make similar tables in MySQL DB, for example note the Customer and Vehicle schema.

Next is to create the first layer, the renderer layer. In the layer layer there is a file called interface.go which contains an interface that contains all the layers of the next layer. This process is called workers.

- ➔ **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 using the mock gene to generate the model for testing.

WHAT ARE MOCKS?

Mock testing entails separating the code from others while testing it without being distracted by dependencies and other factors such as network difficulties and traffic fluctuations. Mock objects, which replicate real-world behaviour and exhibit genuine properties, are used to replace dependent elements. The guiding idea of mock testing is to prioritise testing above dependencies. We'll discuss the following topics here:

USES OF MOCKS

- It is more useful while performing unit testing.
- when you want to avoid relying on others.
- Despite the fact that you want to employ phantom objects to speed up the testing process.
- Even though it is critical to anticipate how the test will seem.

HOW DOES MOCK TEST WORK?

It is a type of unit testing that allows assertions to be made about how the code driving the test interacts with other system components.

1. During mock testing, dependencies are replaced by objects that replicate the behaviour of the critical ones. It is based on behavior-based verification.
2. The mock object creates a bogus interface to mimic the real object's interface. As a result, it is known as mock.
3. Instead of focusing on the complete code, it emphasises the area that will be checked.
4. The fake object merely reads and responds to test data from a local disc.
5. Mocking requires no modifications to the software.
6. When there are dependencies in the case of constructors and other methods, fake objects are used in place of the inherited class during testing.
7. Unlike normal unit testing, authentication is done by artificially pre-initiating the type of call to be made and the desired behavior.
8. Mocking is used to test protocols, control how APIs should be used and how they will respond when used correctly.

Mock Gen automatically generates mock functions for a particular method in an interface.

The service layer follows, which validates the logic of the methods and functions.

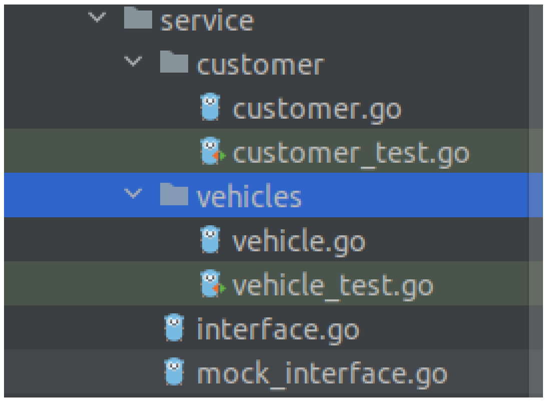
➔ **service**
  - **customer**
    - **customer.go**
    - **customer_test.go**
  - **vehicles**

vehicle.go

vehicle_test.go

interface.go

mock_interface.go



```
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 an interface for the stored procedure called the service layer. Another reason to create a Mock is that we call the service in the handler stored in the service and when the test is executed, this test is also called the similar function, so all these methods will be tested. but since we produce criticism of these works, we will not be able to measure it.

Next is the storage layer that implements database queries and creates database connections.
The structure of store layer is given as follows:

➔ **store**

customer

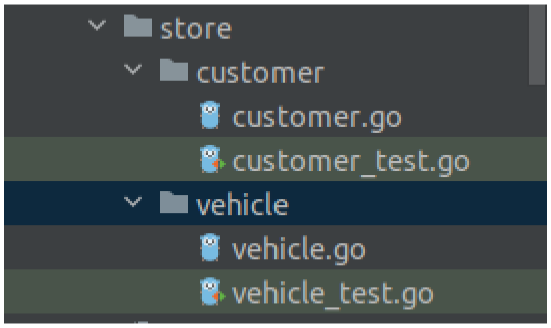
customer.go

customer_test.go

vehicle

vehicle.go

vehicle_test.go



(FIGURE -3.11)

Structure of store layer

The store layer does not employ go mock; instead, it uses SQL mocks to mock the database, preventing modifications from being made to the actual database.

SOL MOCKS

sqlmock is a mock library that makes use of the sql/driver. In testing, any sql driver behaviour may be replicated without requiring a true database connection. It contributes to the correct TDD process.

This library supports multiple connections and concurrency. Mocking and named sql arguments are also supported for go1.8 context-related features. It does not require any changes to your source code. The driver may be used to simulate any sql driver method behaviour.By default, it uses stringent expectation order matching. It does not rely on other parties.

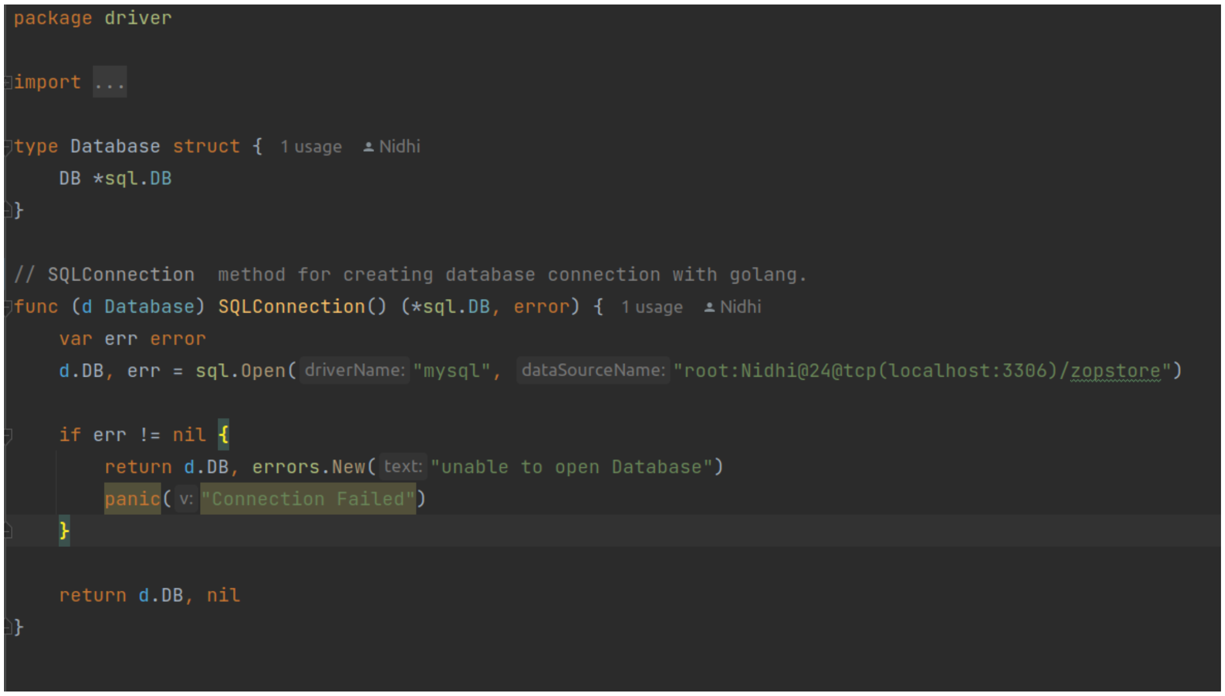
In this layer, we fake the SQL connection and then use these mocks to construct the test cases.

DOCKER

The open platform for building, deploying and running programs is Docker. With the help of Docker, you can separate your application from your development process and ensure fast delivery. You can use Docker to manage your infrastructure the same way you manage applications. By using Docker's approach to quickly deploy, test, and deploy code, you can reduce the time between writing code and running it in production.

We installed a MySQL docker image and ran the operations on the image rather than the actual MySQL table.

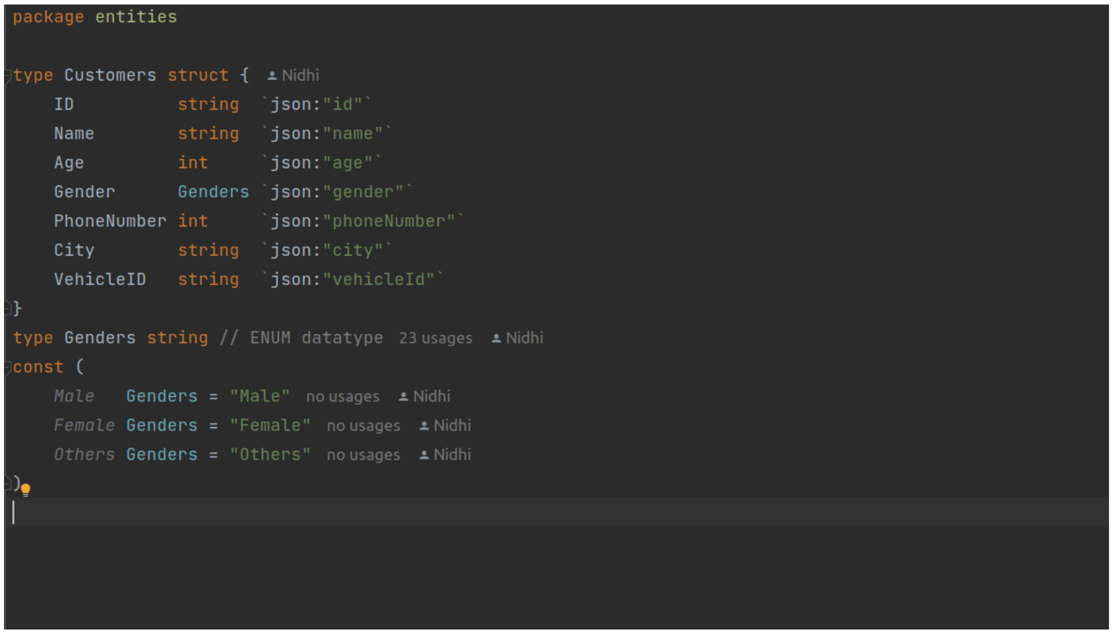
CODE



```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
}
```

(FIGURE-3.12)

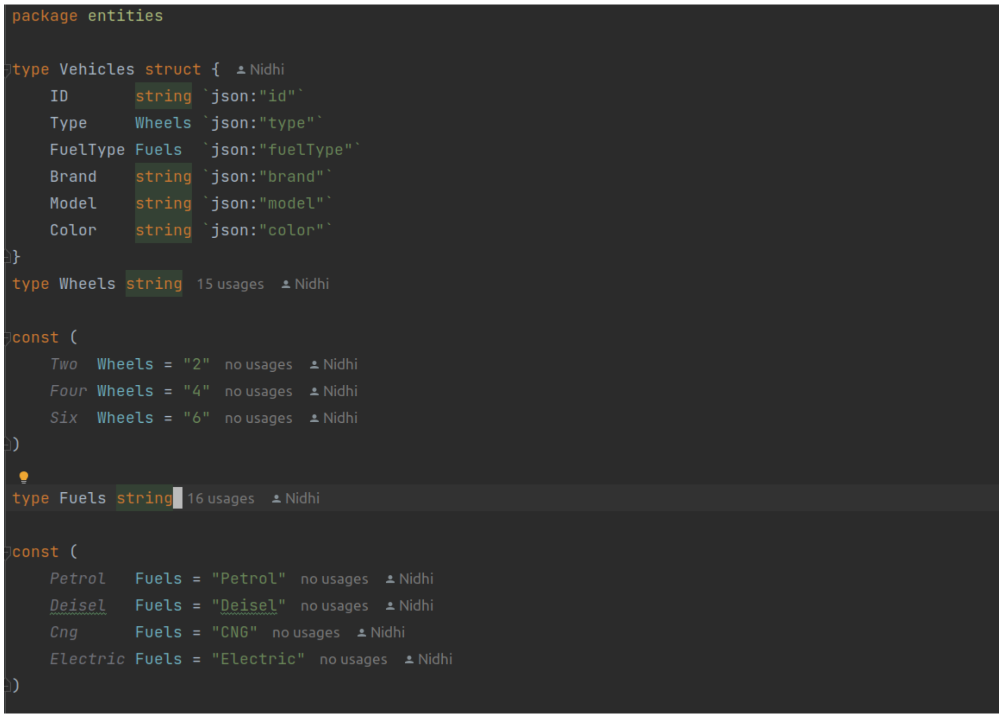
Sql Driver Connection



```
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
)
```

(FIGURE-3.13)

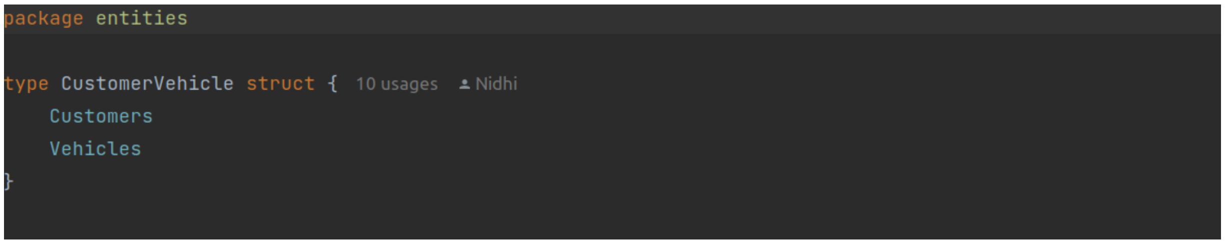
Entities - customers.go (having Customers struct)



```
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
)
```

(FIGURE -3.14)

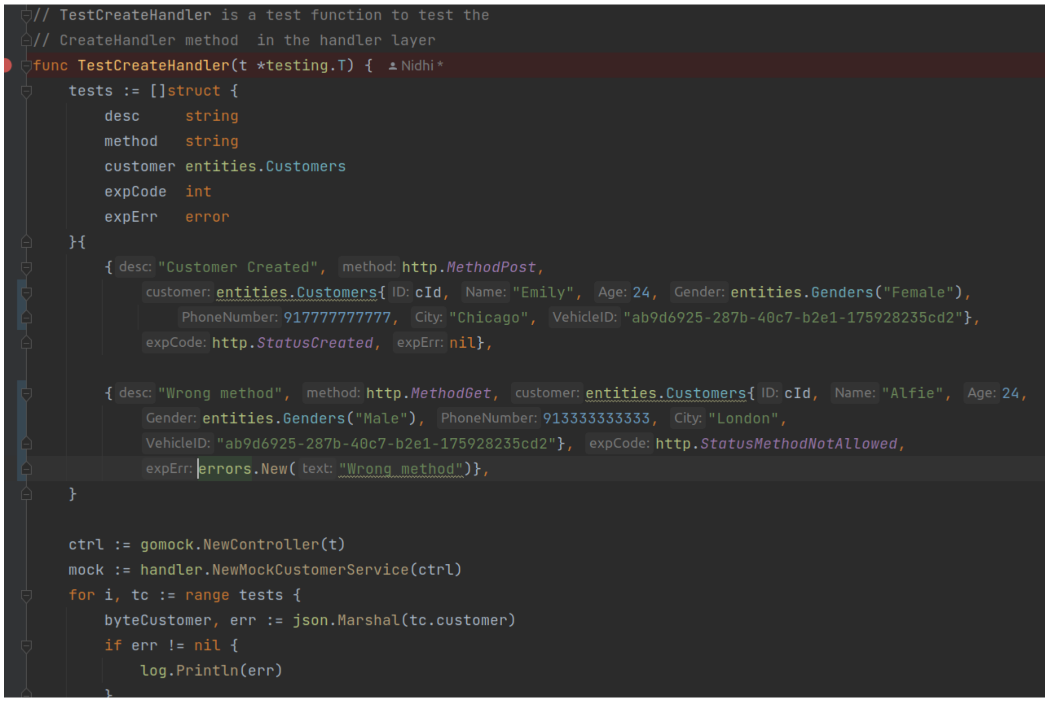
Entities - Vehicles.go(having struct vehicles)



```go
package entities

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

(FIGURE -3.15)

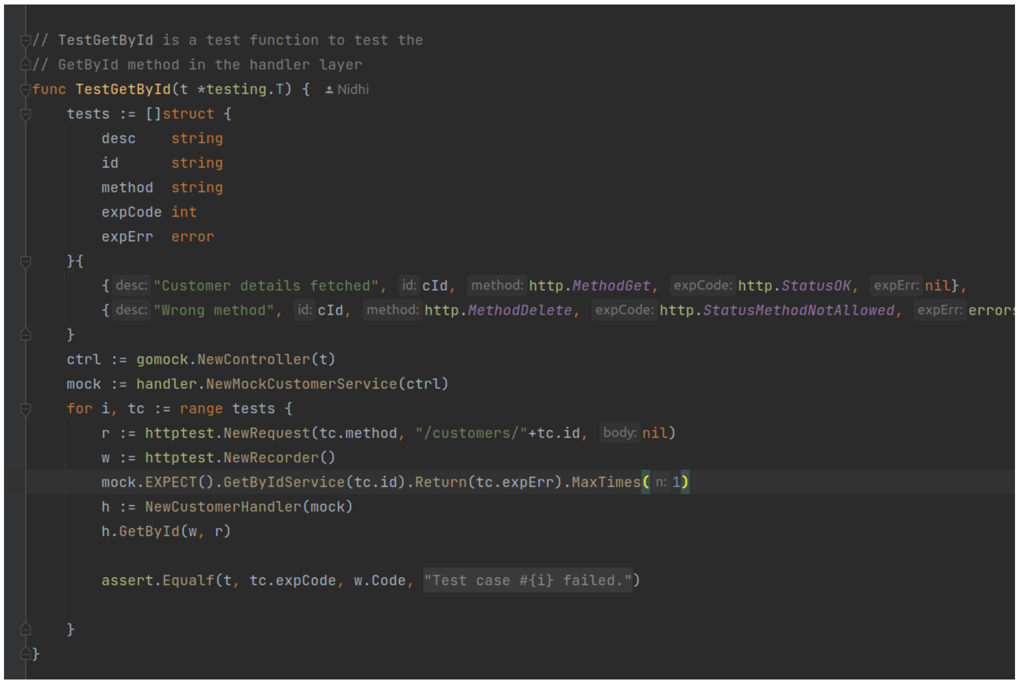
Entities - CustomerVehicle.go(having CustomerVehicle struct)



```go
// 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)
        }
```

(FIGURE - 3.16)

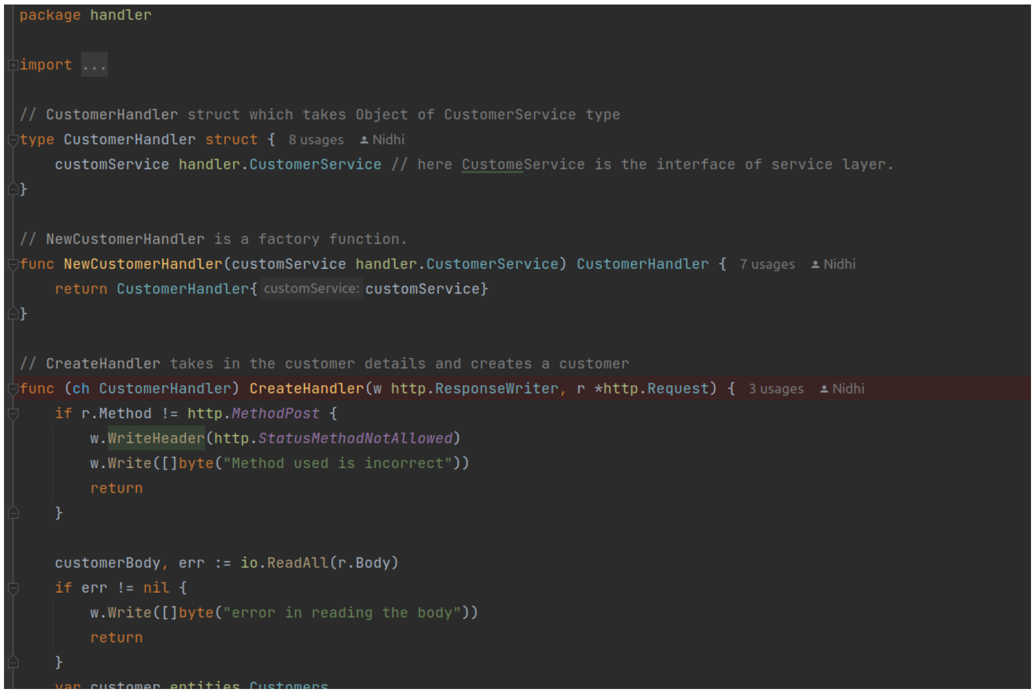
Handler Layer customer_test.go file(test function for creating a customer)



```
// TestGetById is a test function to test the
// GetById method in the handler layer
func TestGetById(t *testing.T) {  Nidhi
    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: error
    }
    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.")

    }
}
```

(FIGURE -3.17)

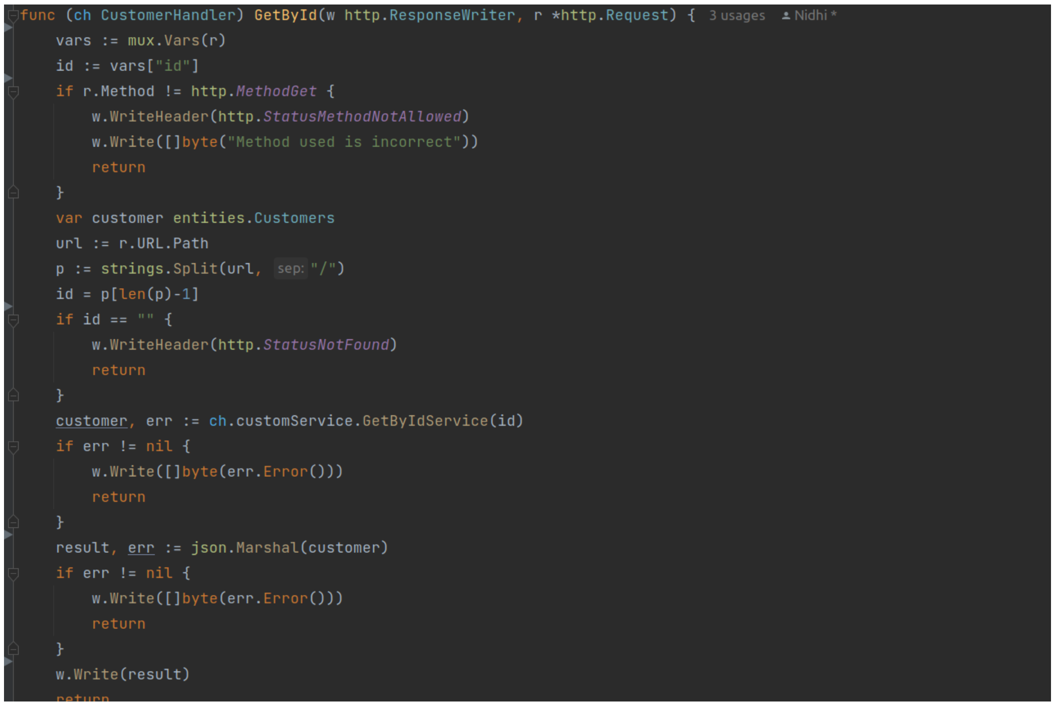
Handler test function for get by Id



```
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
```

(FIGURE -3.18)

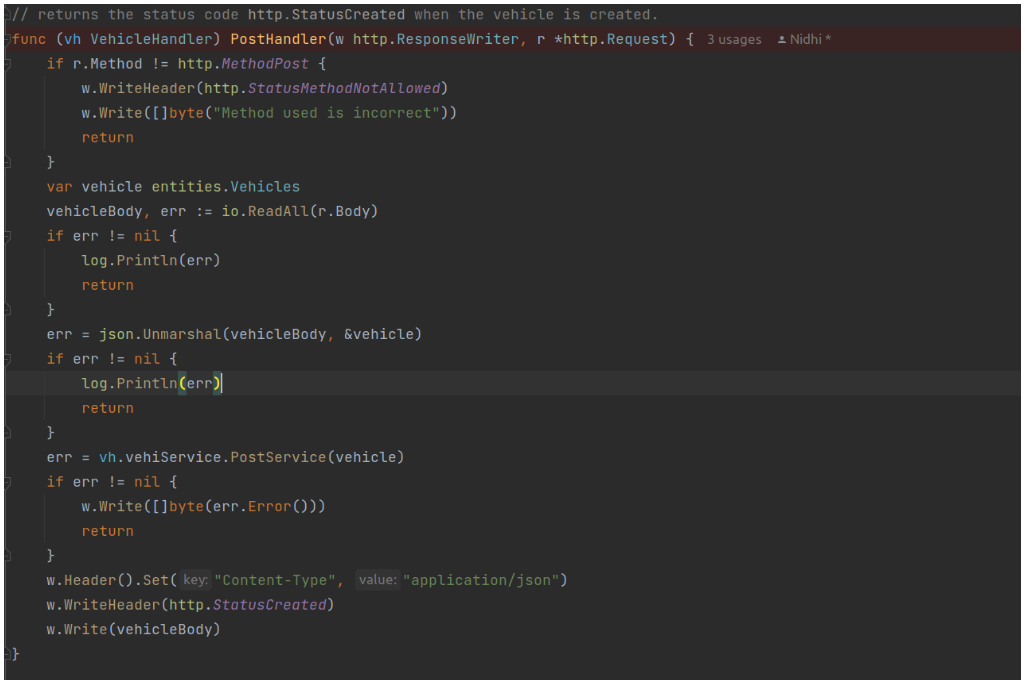
Handler Layer customer.go for create customer method



```
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)
```

(FIGURE -3.19)

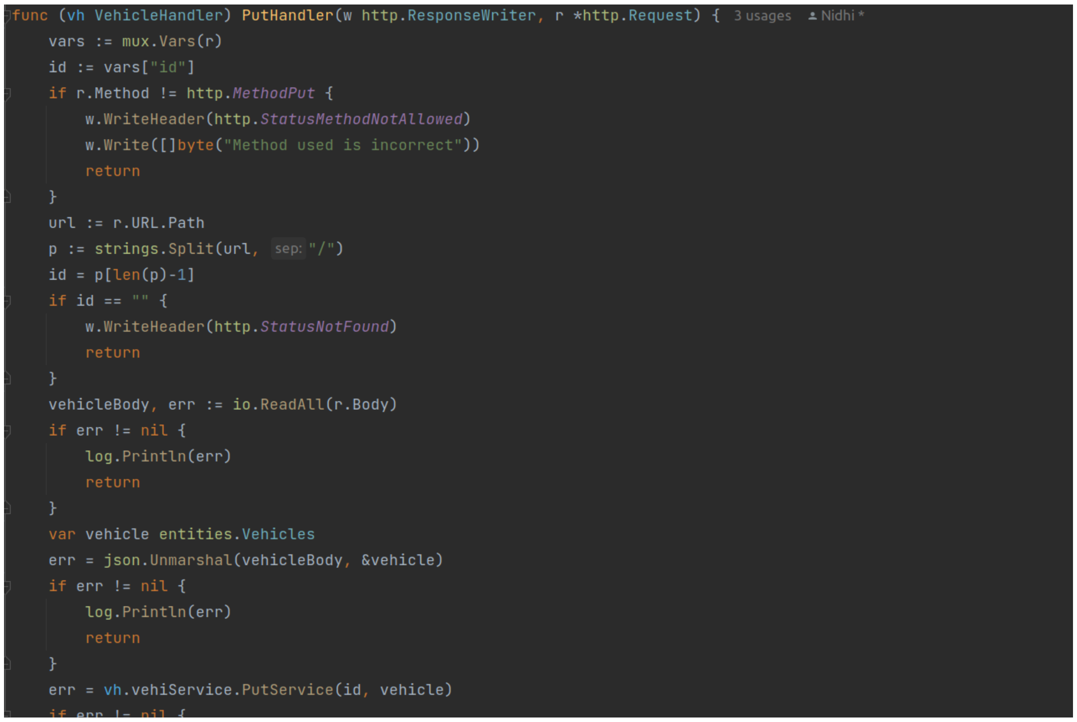
Get BY ID method in handler layer



```
// 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)

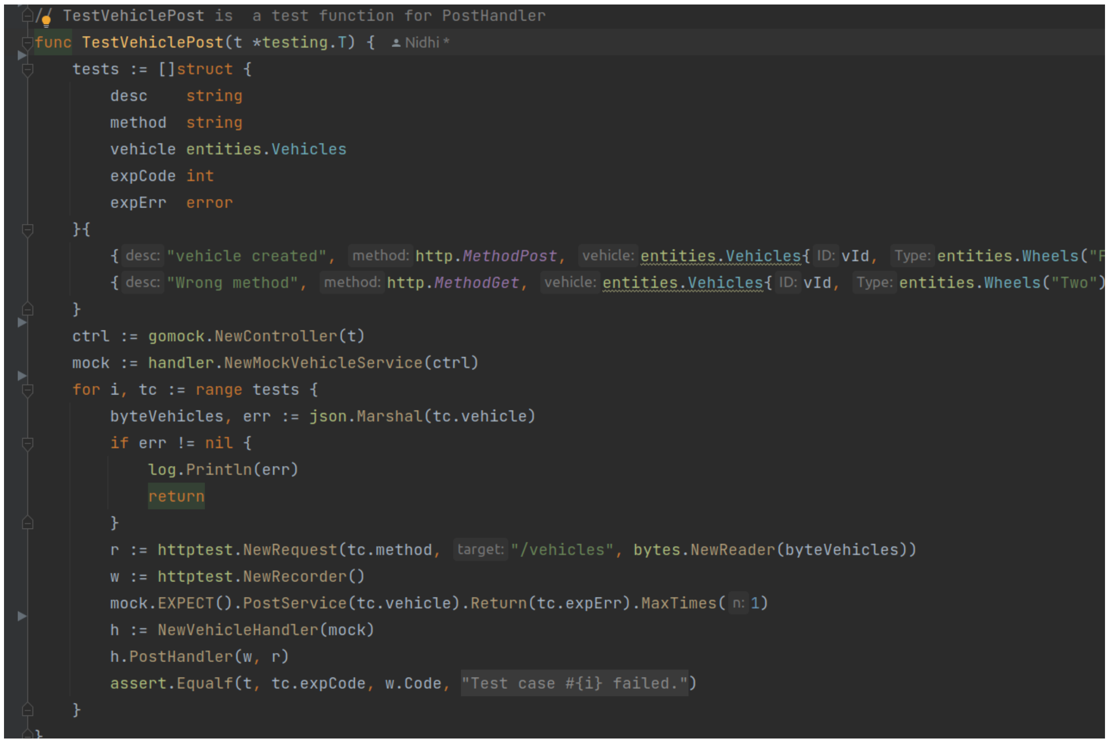
Create vehicle in vehicle.go



```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)

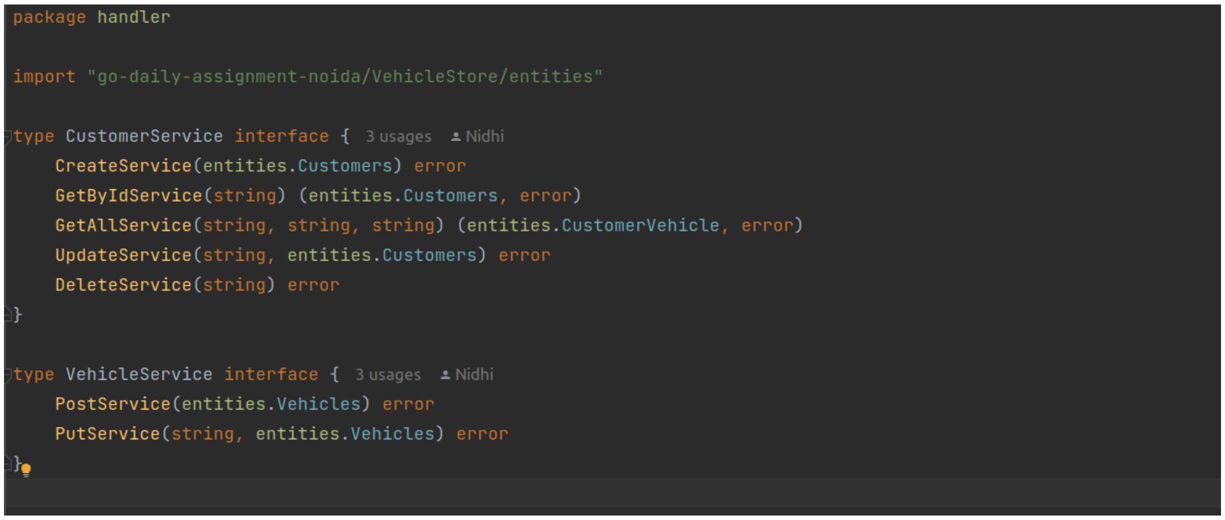
Update Vehicle in vehicle.go for handler layer



```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.")
    }
}
```

(FIGURE -3.22)

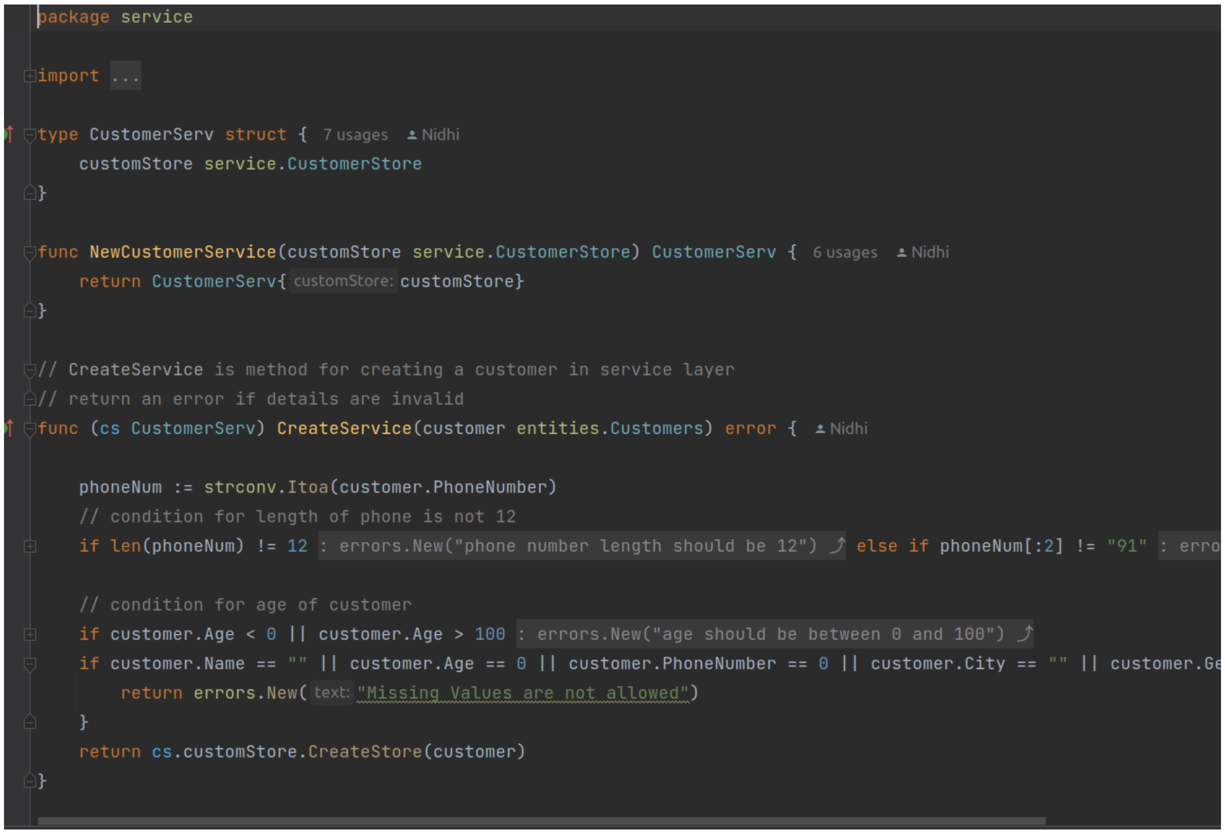
Test function for create vehicle in vehicle_test.go



```
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)

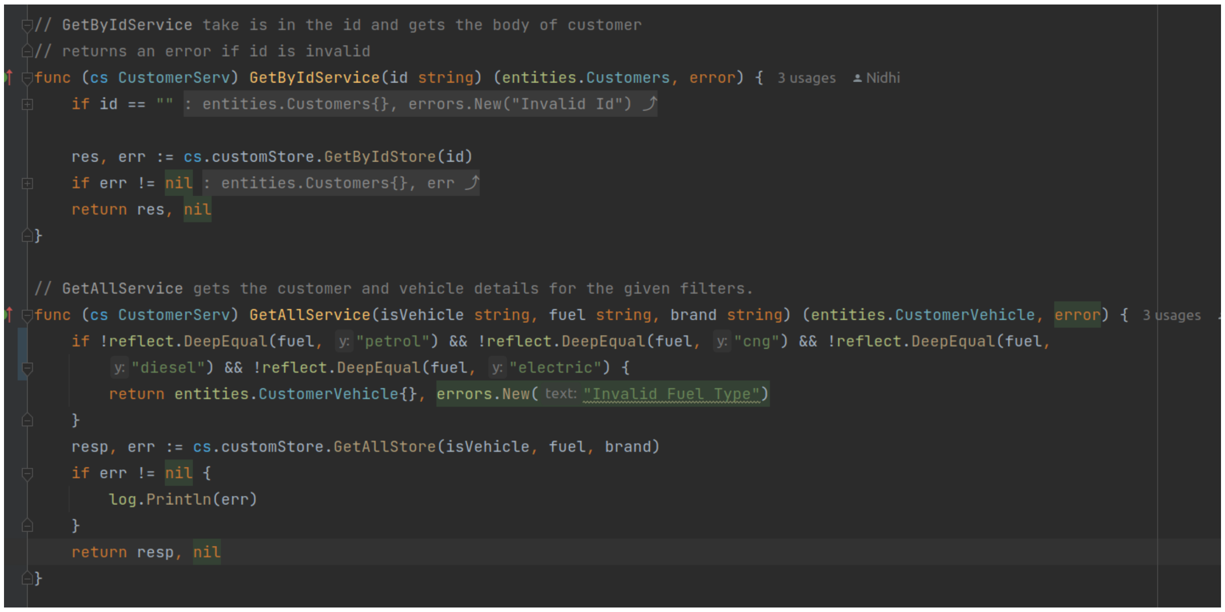
Interface.go in handler layer



```
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)

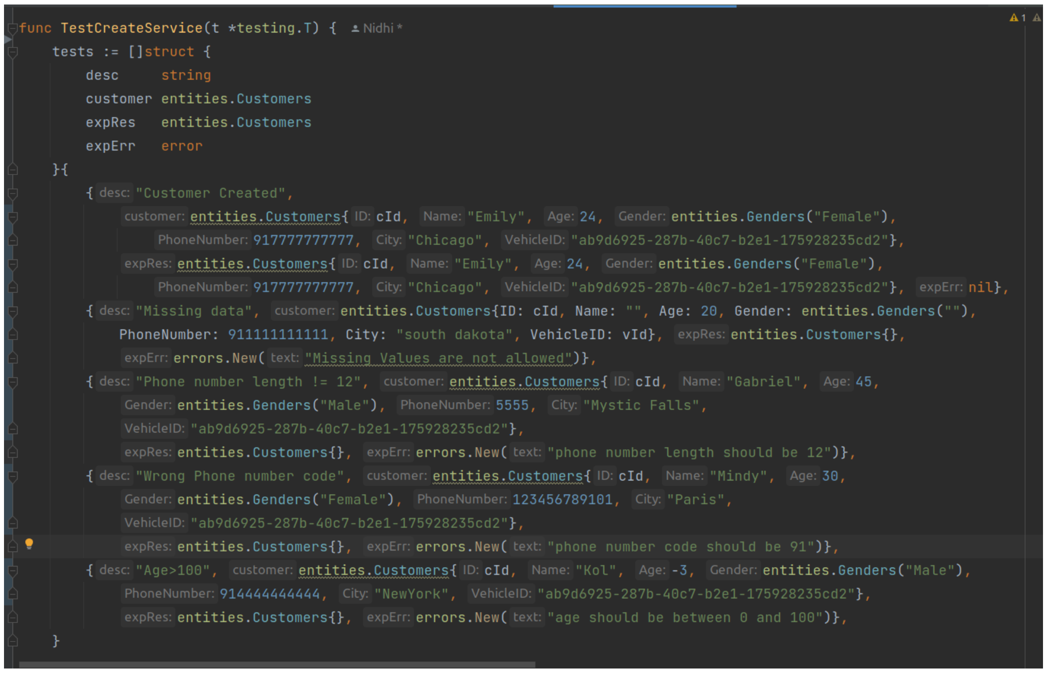
Create customer method in service layer



```go
// 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)

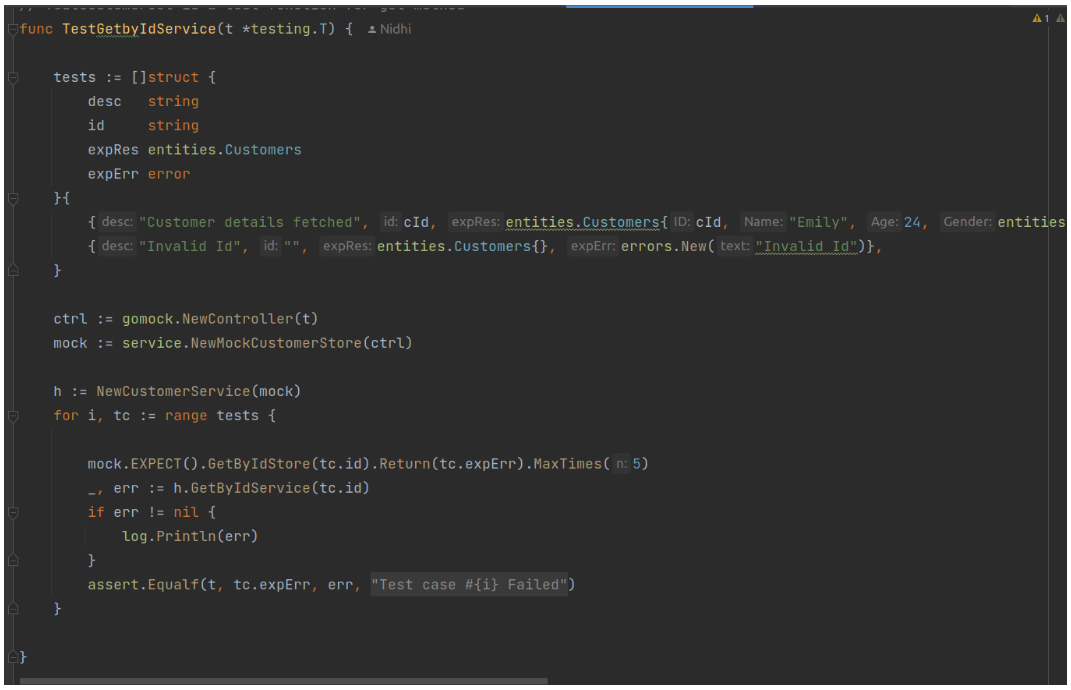
GetById and get All method in service layer



```go
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: 917777777777,  City: "Chicago",  VehicleID: "ab9d6925-287b-40c7-b2e1-175928235cd2"},
            expRes: entities.Customers{ ID: cId,  Name: "Emily",  Age: 24,  Gender: entities.Genders("Female"),
                PhoneNumber: 917777777777,  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: 911111111111, 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)

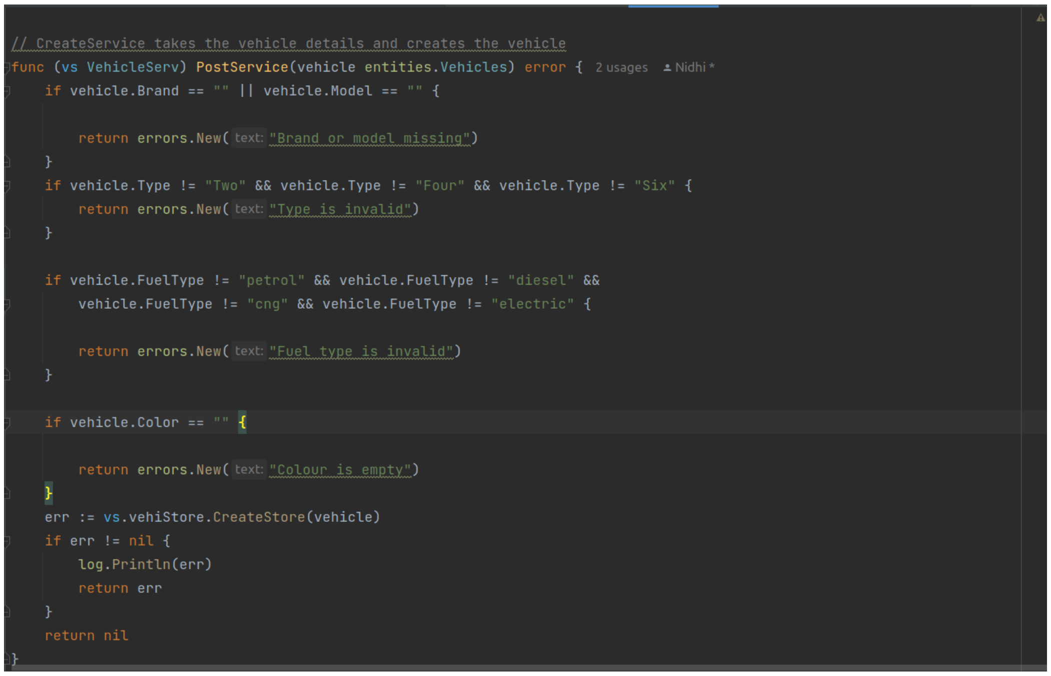
Test cases for Create customer in service layer



```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")
    }

}
```

(FIGURE -3.27)

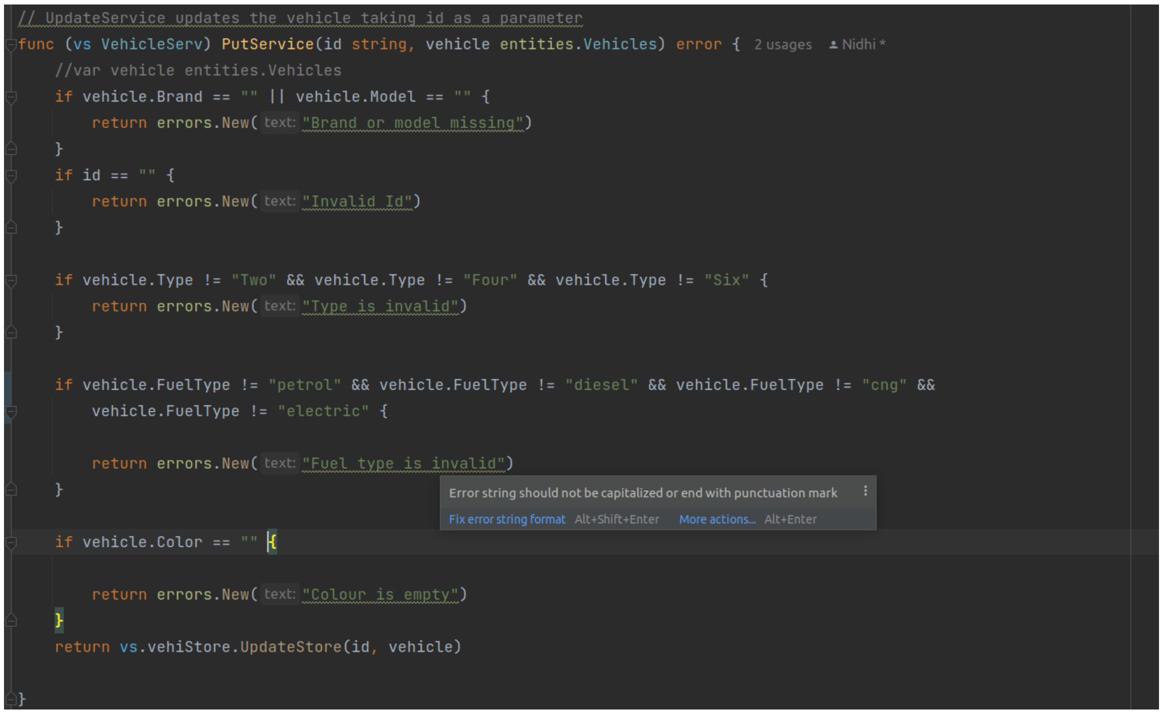
Test function for GetByID in service layer



```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
}
```

(FIGURE -3.28)

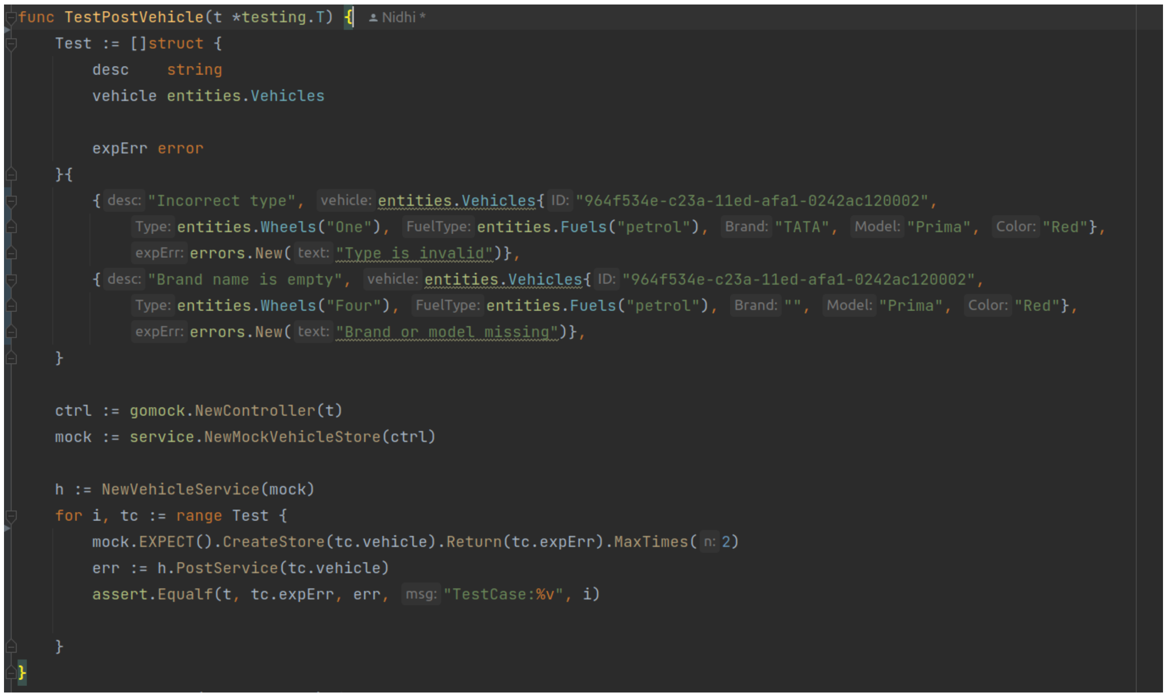
Method create vehicle in service layer



```
// 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")
    }
                                    Error string should not be capitalized or end with punctuation mark
                                    Fix error string format  Alt+Shift+Enter   More actions...  Alt+Enter
    if vehicle.Color == "" {

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

}
```

(FIGURE-3.29)

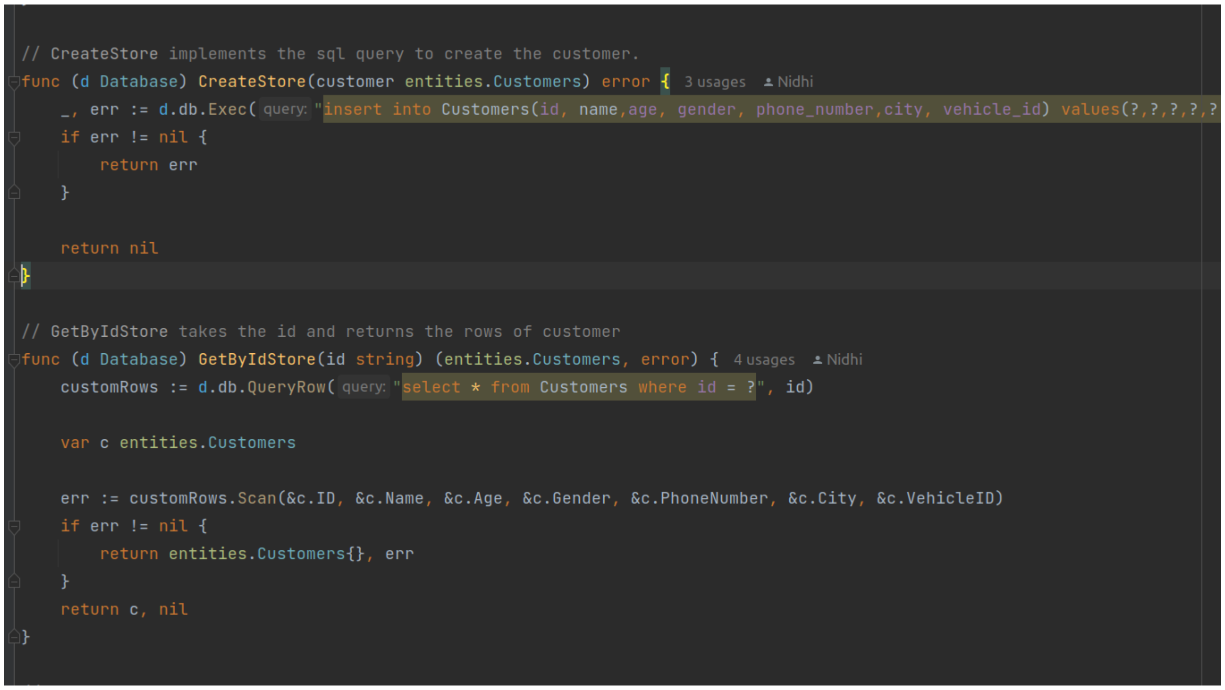
Update Vehicle Method in service layer



```
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.Equalf(t, tc.expErr, err,  msg: "TestCase:%v", i)

    }
}
```

(FIGURE - 3.30)

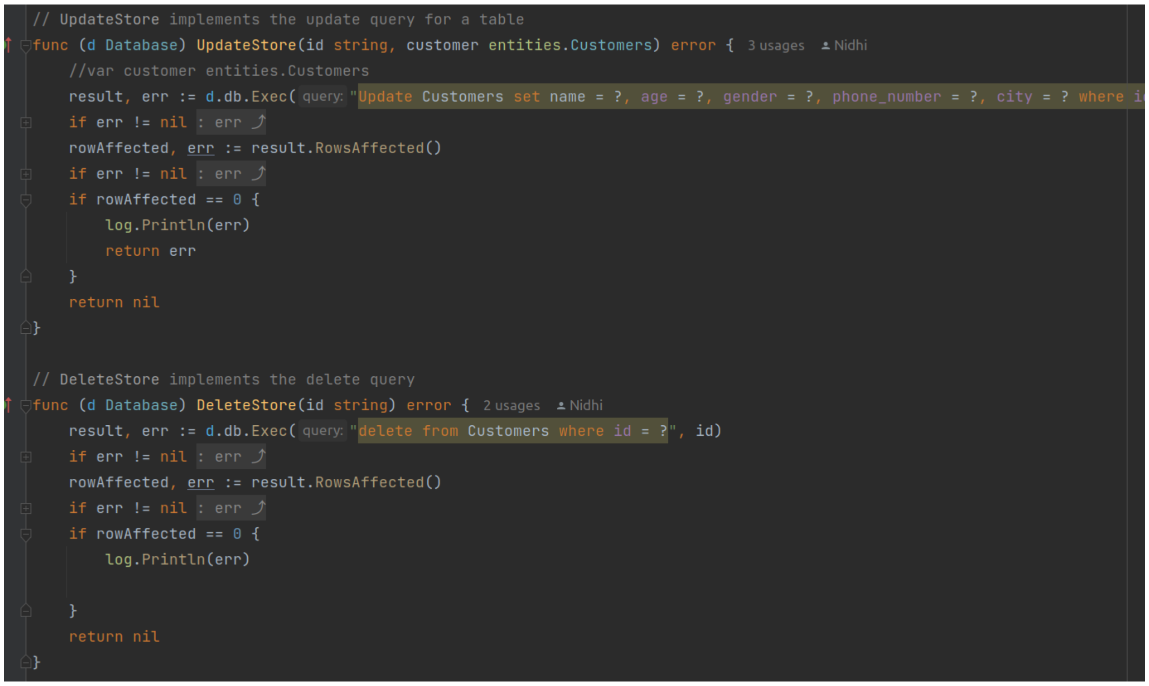
Test function for create vehicle in service layer



```go
// 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)

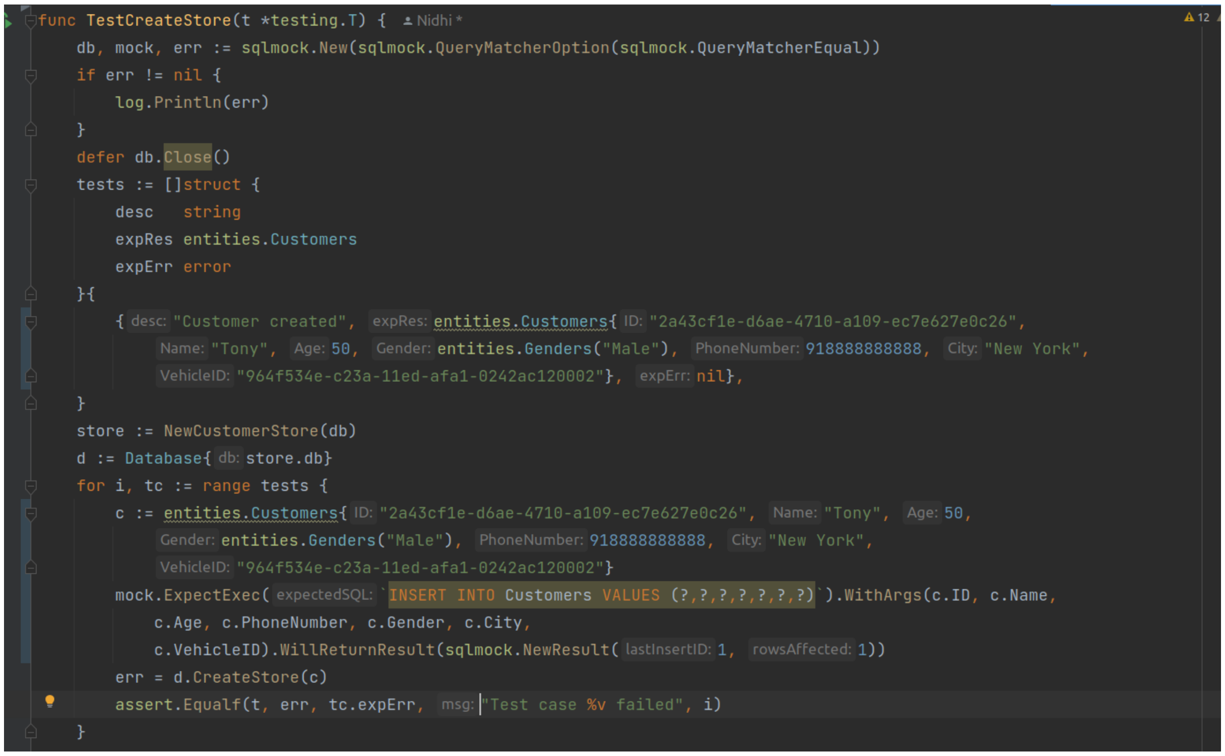
Create Customer and GetByIdStore in store layer



```go
// 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
}
```

(FIGURE -3.32)

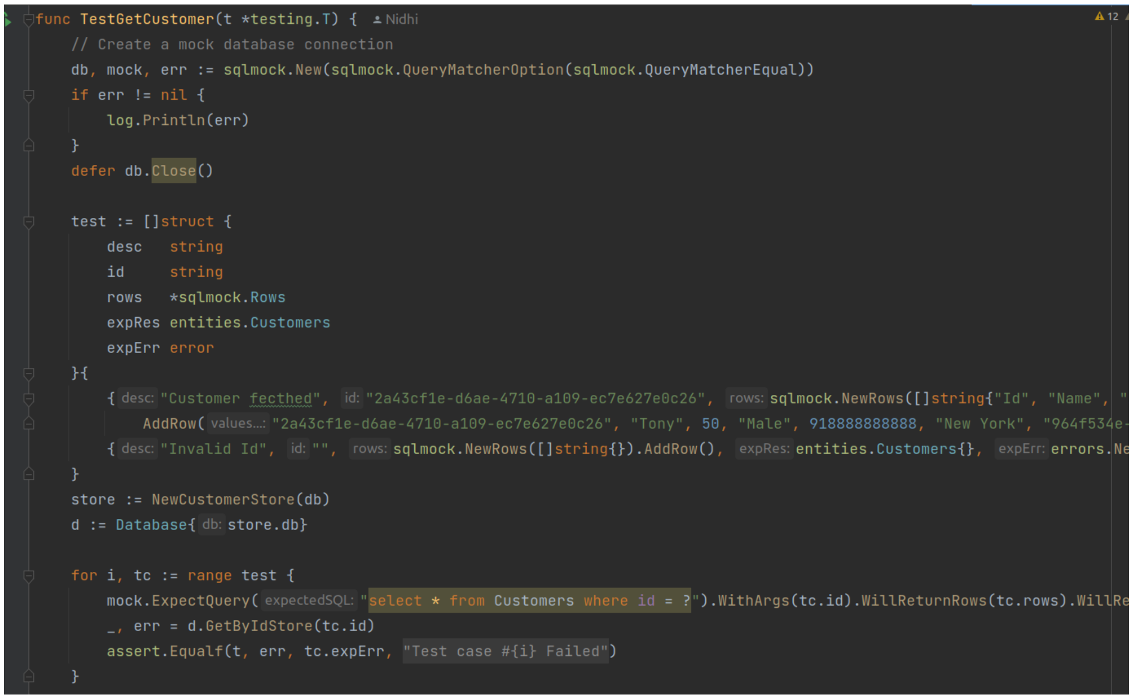
Update customer and Delete customer in store layer



```
func TestCreateStore(t *testing.T) {  Nidhi *
    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)
    }
```

(FIGURE - 3.33)

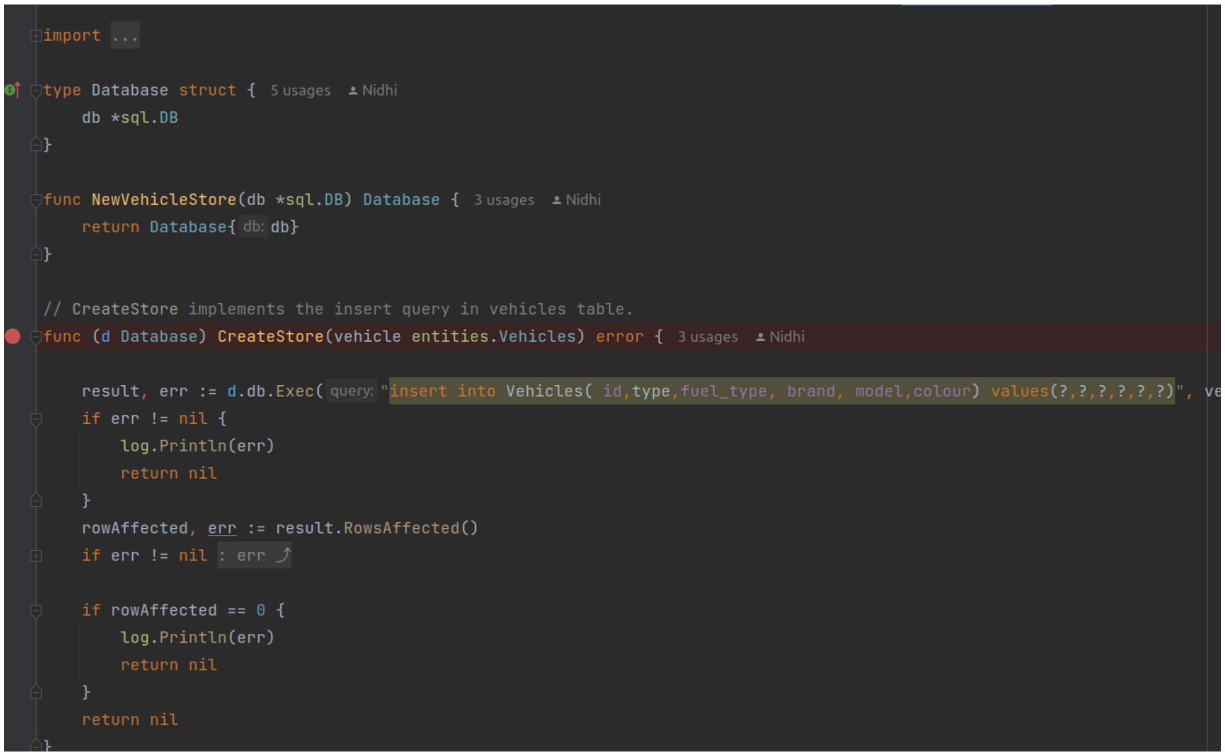
Test Function for create customer in store layer



```
func TestGetCustomer(t *testing.T) {  Nidhi
    // 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")
    }
```

(FIGURE -3.34)

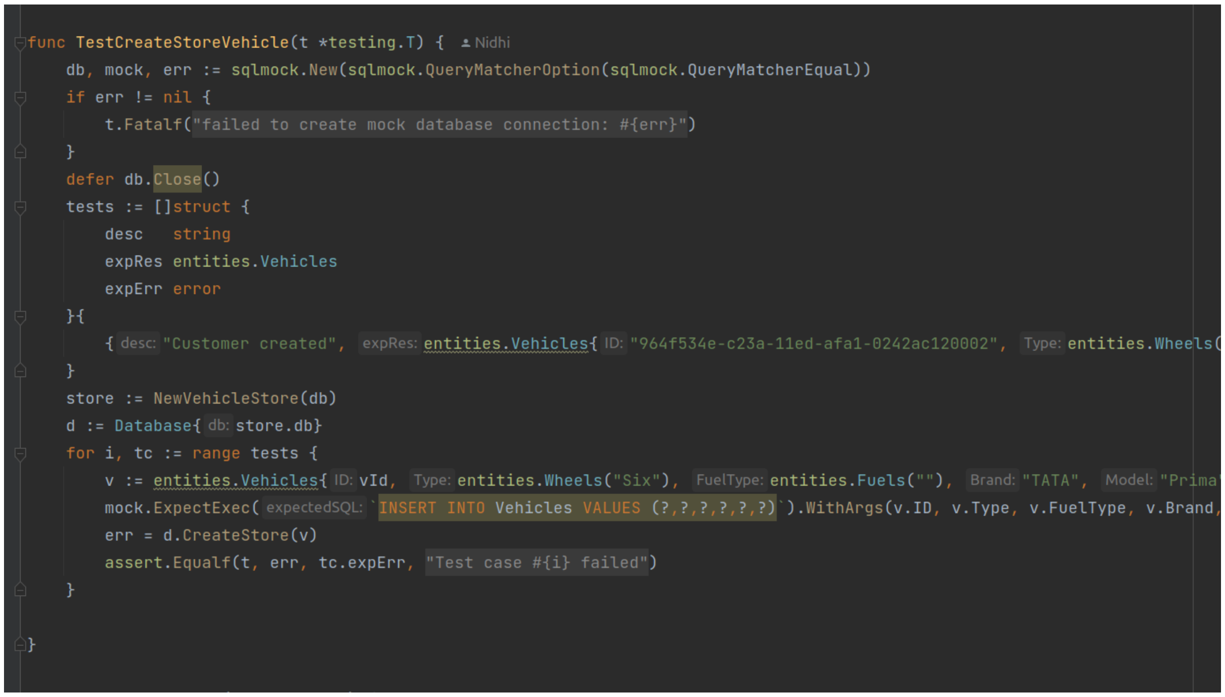
Test Function for GET customer in store layer



```go
import ...

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

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

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



```go
func TestCreateStoreVehicle(t *testing.T) {  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.Equalf(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. Every test function in every tier passed with 100% coverage.
2. All of the endpoints are functioning normally.
3. The status codes are all right.

Postman is used for end point testing.

POSTMAN

Postman is an API platform for developing and deploying APIs. Postman increases collaboration and streamlines every stage of the API lifecycle to help you develop better APIs quicker.

(FIGURE- 4.1)

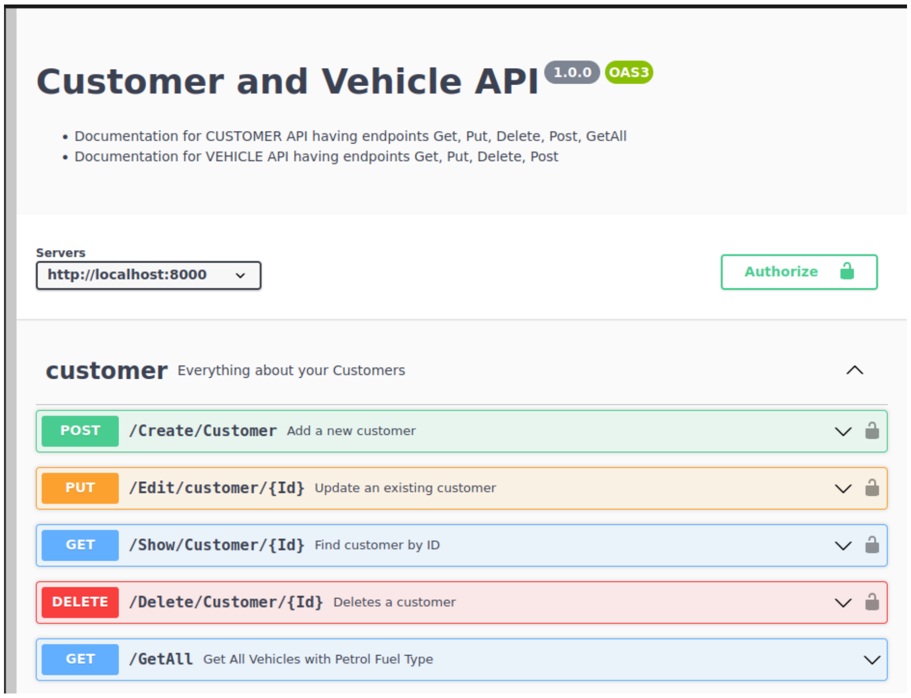
Postman

Postman can store and manage everything related to APIs, such as API specs, documentation, workflow recipes, test cases and results, metrics, and more.

The API documentation is created with the swagger API tool.

SWAGGER

You define the internal structure of your API in Swagger so that computers can understand it. Central to all the benefits in Swagger is the API's ability to define its own structure. Why is this good? However, we can create a perfectly interactive API document by reading your API model. We can also review other options such as technical evaluation and start building libraries for your API in different languages. Swagger does this by requesting a YAML or JSON response from your API that provides a general description of your API as a whole.



Customer and Vehicle API 1.0.0 OAS3

- Documentation for CUSTOMER API having endpoints Get, Put, Delete, Post, GetAll
- Documentation for VEHICLE API having endpoints Get, Put, Delete, Post

Servers: http://localhost:8000

Authorize

customer Everything about your Customers

POST /Create/Customer Add a new customer

PUT /Edit/customer/{Id} Update an existing customer

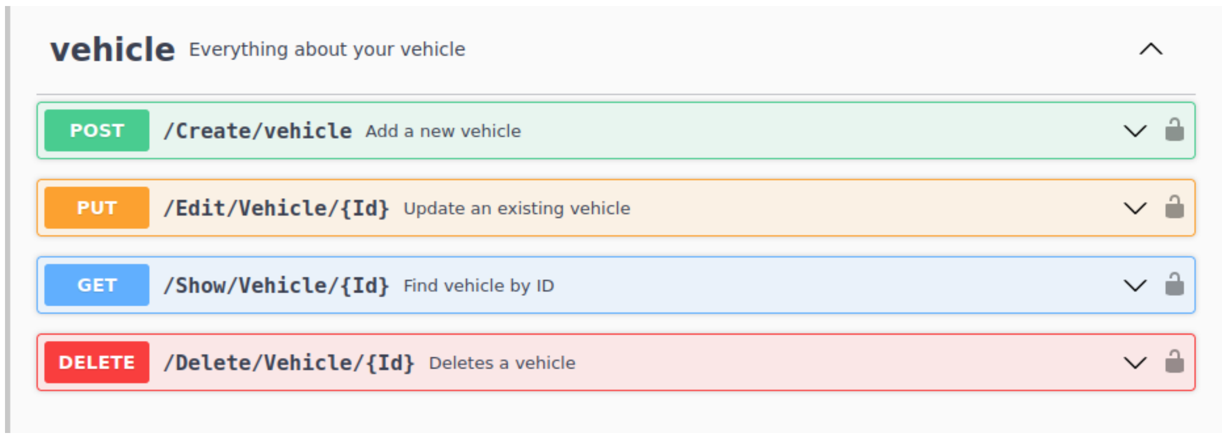
GET /Show/Customer/{Id} Find customer by ID

DELETE /Delete/Customer/{Id} Deletes a customer

GET /GetAll Get All Vehicles with Petrol Fuel Type

(FIGURE- 4.2)

Swagger -API documentation for customer



vehicle Everything about your vehicle

POST /Create/vehicle Add a new vehicle

PUT /Edit/Vehicle/{Id} Update an existing vehicle

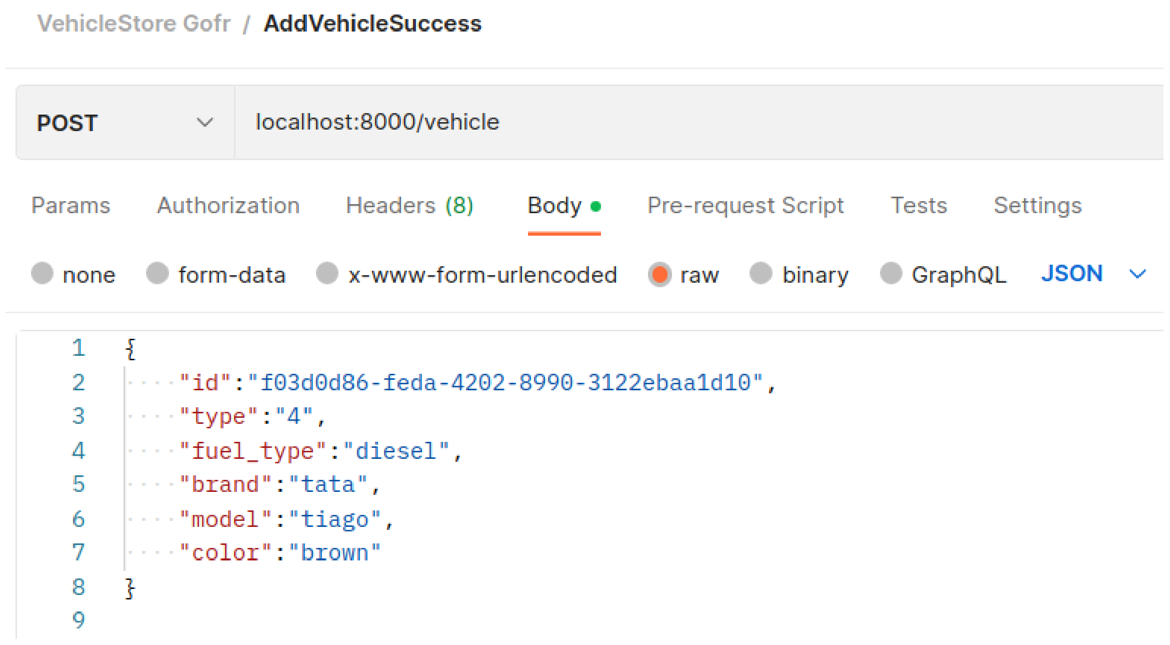
GET /Show/Vehicle/{Id} Find vehicle by ID

DELETE /Delete/Vehicle/{Id} Deletes a vehicle

(FIGURE- 4.3)

Swagger -API documentation for vehicle

ENDPOINTS RESULT FROM POSTMAN



VehicleStore Gofr / AddVehicleSuccess

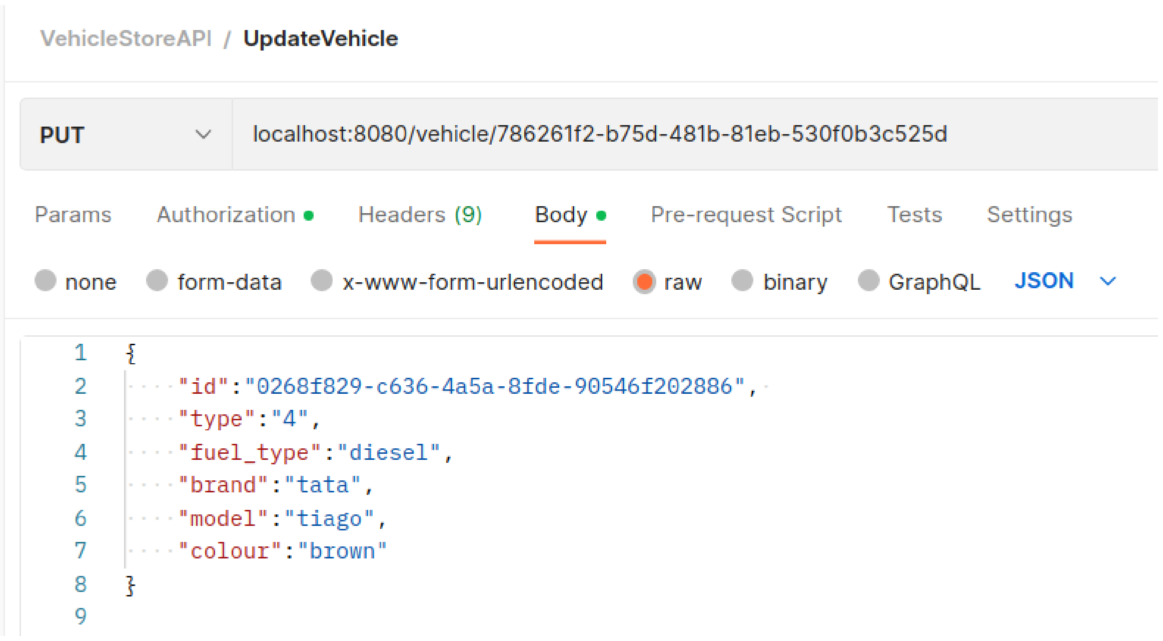
POST localhost:8000/vehicle

Body: [✓ raw] JSON

```
{
    "id":"f03d0d86-feda-4202-8990-3122ebaa1d10",
    "type":"4",
    "fuel_type":"diesel",
    "brand":"tata",
    "model":"tiago",
    "color":"brown"
}
```

(FIGURE- 4.4)

Add vehicle Endpoint

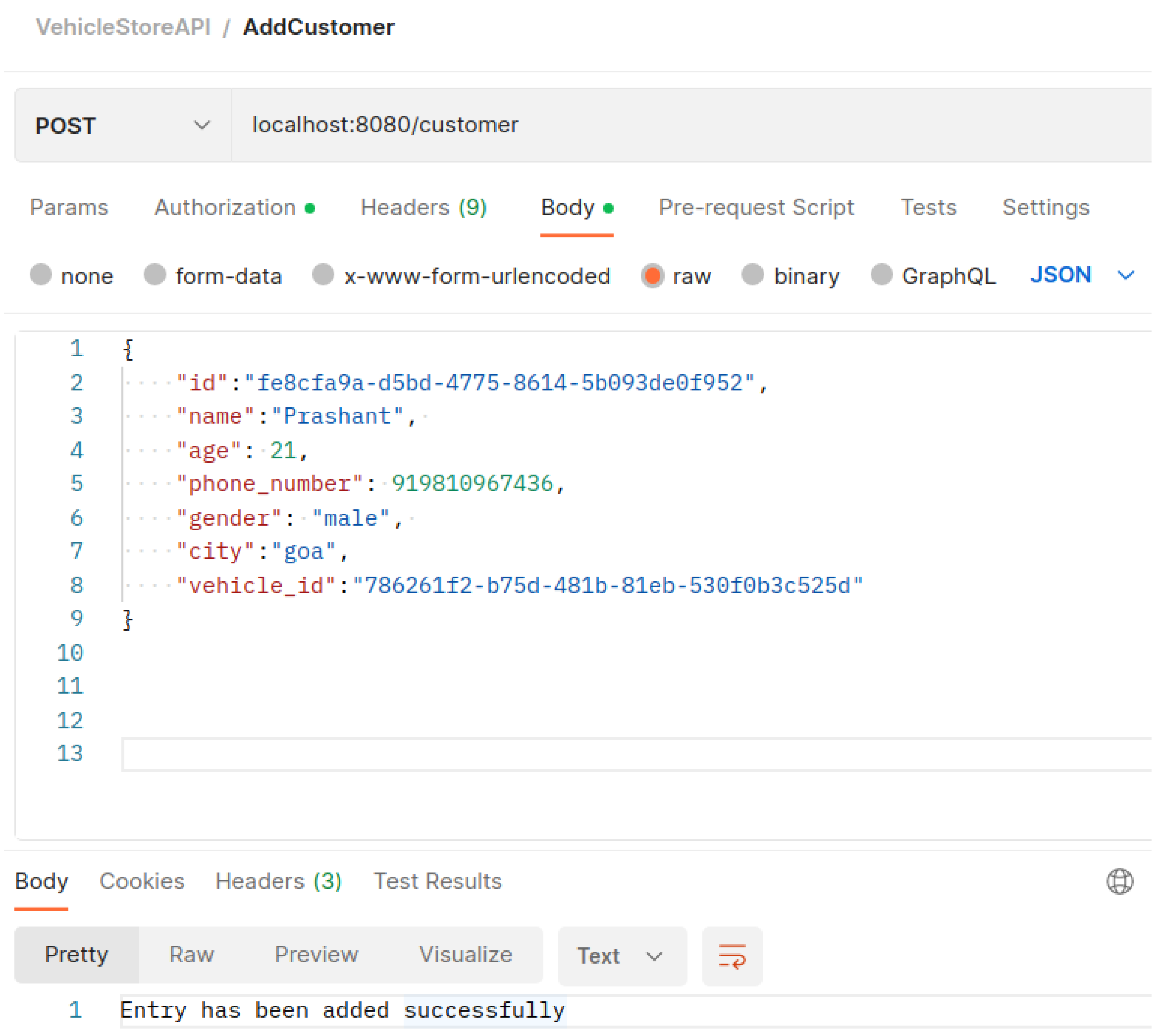

VehicleStoreAPI / UpdateVehicle

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

```
{
    "id":"0268f829-c636-4a5a-8fde-90546f202886",
    "type":"4",
    "fuel_type":"diesel",
    "brand":"tata",
    "model":"tiago",
    "colour":"brown"
}
```

(FIGURE- 4.5)

Update vehicle endpoint



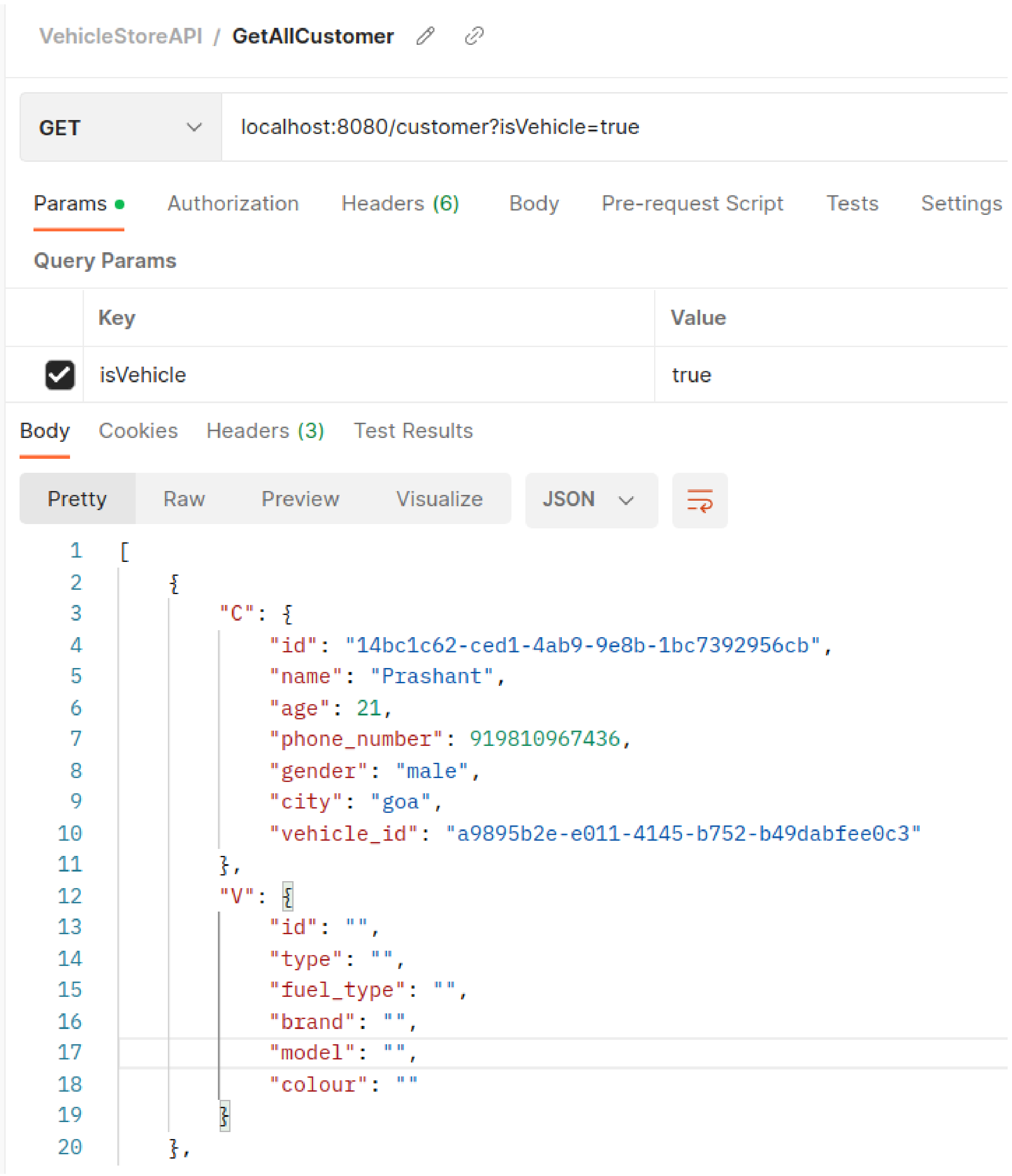
POST localhost:8080/customer

```
{
    "id":"fe8cfa9a-d5bd-4775-8614-5b093de0f952",
    "name":"Prashant",
    "age": 21,
    "phone_number": 919810967436,
    "gender": "male",
    "city":"goa",
    "vehicle_id":"786261f2-b75d-481b-81eb-530f0b3c525d"
}
```

```
Entry has been added successfully
```

(FIGURE- 4.6)

Add Customer endpoint



```
localhost:8080/customer?isVehicle=true
```

Key	Value
isVehicle	true

```
[
    {
        "C": {
            "id": "14bc1c62-ced1-4ab9-9e8b-1bc7392956cb",
            "name": "Prashant",
            "age": 21,
            "phone_number": 919810967436,
            "gender": "male",
            "city": "goa",
            "vehicle_id": "a9895b2e-e011-4145-b752-b49dabfee0c3"
        },
        "V": {
            "id": "",
            "type": "",
            "fuel_type": "",
            "brand": "",
            "model": "",
            "colour": ""
        }
    },
```

(FIGURE- 4.7)

GetAll endpoint

CHAPTER - 5

5.1 CONCLUSIONS

As a result, best practices and practices were taken into account when creating the ZopStore programming interface. Handlers, services, and stored procedures are clearly separated from each other in a three-tier engineering process. Use test-driven development throughout the development process to ensure code quality and reduce the risk of errors.

Middleware is used to provide common concerns such as availability and access to interactions, while data carriers are used to monitor changes in dataset plans. Unit testing of the storage tier is used by SQL derivation, while unit testing of the tier is used by mockgen and mux.

Metrics are used to carefully monitor and analyze performance, and measurements are made to simplify the programming interface and ensure performance even under harsh conditions. Additionally, the programmatic interface is written entirely using Swagger and Postman, with clear and concise documentation for each endpoint.

Finally, GitHub activities for automated testing, code injection, and test linter improve connectivity between the development team. Overall, the ZopStore programming interface has been designed using best practices and methods that lead to world-class, reliable and efficient communication that solves customer's problems.

5.2 OBJECTIVES ACHIEVED

1. Flexible Programs.

2. Simple to carry out
3. Safe
4. Technical usable
5. Better Quality

5.3 FUTURE WORK

Frameworks can be used to generate this API. Frameworks improve code quality and aid in the development of a better API. At ZopSmart Technologies, we developed our own Go Framework called gofr. This API is compatible with the gofr framework.

The programming interface may be upgraded for faster reaction times and greater diversity. This may be accomplished through the use of techniques such as load testing, execution profiling, and code enhancement.

The programming interface may interact on cloud platforms such as AWS, Sky Blue, or Google Cloud, providing additional benefits such as flexibility, consistent quality, and cost-effectiveness.

In overall, the ZopStore Programming interface offers a few great open doors for future events and growth, and it will be interesting to see how it evolves over time.

References

All references are collected from the online go documentation and the in-house learning platform provided 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/

BTech Thesis

ORIGINALITY REPORT

7%	4%	0%	5%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to American University of Nigeria Student Paper	1%
2	tip.golang.org Internet Source	1%
3	Submitted to University of Wales Institute, Cardiff Student Paper	1%
4	Submitted to University of Derby Student Paper	1%
5	Submitted to University of Makati Student Paper	1%
6	www.fatalerrors.org Internet Source	<1%
7	Submitted to Liverpool John Moores University Student Paper	<1%
8	Submitted to Asia Pacific Instutute of Information Technology Student Paper	<1%

9	Submitted to Hacettepe University Student Paper	<1 %
10	www.coursehero.com Internet Source	<1 %
11	rajabishek.com Internet Source	<1 %
12	Submitted to National School of Business Management NSBM, Sri Lanka Student Paper	<1 %
13	Submitted to University of Leeds Student Paper	<1 %

Exclude quotes	On	Exclude matches	< 14 words
Exclude bibliography	On		