

VehicleStore API Using GOFW Framework

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

in

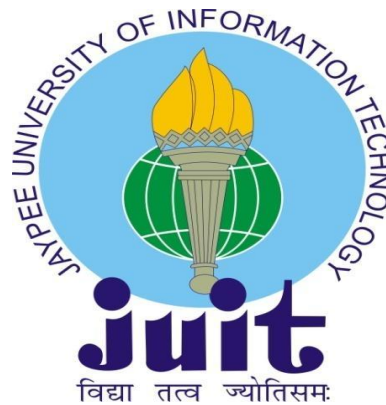
Computer Science and Engineering/Information Technology

By

Prashant Agarwal (191451)

Under the supervision of

(Dr. Jagpreet Sidhu)
(Assistant Professor (SG), CSE)



Department of Computer Science & Engineering & Information
Technology

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

Candidate's Declaration

I hereby declare that the work presented in this report entitled “**VehicleStore API using GOFR Framework**” in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology in Computer Science and Engineering/Information Technology** submitted in the department of Computer Science & Engineering and Information Technology, Jaypee University of Information Technology Waknaghat is an authentic record of my own work carried out over a period from July 2022 to May 2023 under the supervision of **Dr. Jagpreet Sidhu** (Assistant Professor(SG) Computer Science and Engineering).

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

Prashant Agarwal, 191451

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

(Supervisor Signature)
Dr. Jagpreet Sidhu
Assistant Professor(SG)
Computer Science and Engineering
Dated:08/05/2023

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)	
	<ul style="list-style-type: none"> • All Preliminary Pages • Bibliography/Images/Quotes • 14 Words String 		Word Counts	
Report Generated on			Character Counts	
		Submission ID	Total Pages Scanned	
			File Size	

Checked by
Name & Signature

Librarian

Please send your complete thesis/report in (PDF) with Title Page, Abstract and Chapters in (Word File) through the supervisor at plagcheck.juit@gmail.com

ACKNOWLEDGEMENT

First and foremost, I want to give God the highest praise for His heavenly grace, which enabled us to successfully finish the project work. My supervisor, **Dr. Jagpreet Sidhu, Assistant Professor (SG), Department of CSE Jaypee University of Information Technology**, Waknaghat, has my deepest gratitude and gratitude. My supervisor has a wealth of knowledge and a genuine interest in the "Research Area" needed to complete this assignment. This project was made possible by his never-ending patience, academic leadership, constant encouragement, frequent and vigorous supervision, constructive criticism, insightful counsel, reviewing several subpar versions and revising them at all levels. It is my regarded joy to introduce this project and earnestly thank each and every individual who helped me in this project.

I express my earnest gratitude to **Jaypee University of Information Technology** for giving an open door and such a decent learning climate. I express my sincere gratitude to Jaypee University of Information Technology, Solan for offering help for everything and for giving productive analysis and support which prepared us to the fruitful culmination of the project. I want to offer my genuine thanks to everyone involved for giving me all important help and support and motivation to embrace this study and make it conceivable.

I'm incredibly grateful to **Dr. Jagpreet Sidhu(SG)**, (supervisor for the project and Assistant professor (SG)) for his important direction and backing. I'm likewise thankful to the subjects of this review for their collaboration and interest. Last however not the least I thank god and my parents for every one of the endowments. I would also want to express my gratitude to everyone who has directly or indirectly assisted me in making this project a success. In this unusual scenario, I would like to thank the numerous staff and coordinators, both teaching and non-teaching, who have created their convenient assistance and helped my project.

Prashant Agarwal, 191451

Table of Content

TITLE	PAGE NO
Certificate	(i)
Plagiarism Certificate	(ii)
Acknowledgement	(iii)
Table of Content	(iv)
List Of Abbreviations	(vi)
List Of Figures	(viii)
Abstract	(ix)
CHAPTER-1: INTRODUCTION	1
1.1 Introduction	1
1.2 Problem Statement	5
1.3 Objectives	7
1.4 Methodology	10
1.5 Organization	13
CHAPTER 2: LITERATURE SURVEY	16
2.1 Introduction	16
2.2 Existing Undertakings	17
CHAPTER 3: SYSTEM DESIGN AND DEVELOPMENT	24
3.1 Introduction	24
3.2 Test Driven Development	25
3.3 Three layer architecture	26
3.4 DataBase Migration and Middleware	29

3.4 MockGen and SQL Mocking	31
3.5 Metrics	32
3.6 Improving Code Coverage and Removing Linter errors	33
3.4 Swagger and PostMan Collection	35
CHAPTER 4: PERFORMANCE ANALYSIS	38
CHAPTER 5: CONCLUSION	43
5.1 Conclusion	43
5.2 Goals Achieved	44
5.3 Future Scope	45
Reference	47
Appendix	48

List of Abbreviations

- **API - Application Programming Interface**
- **CRUD - Create, Read, Update, Delete**
- **TDD - Test-Driven Development**
- **HTTP - Hypertext Transfer Protocol**
- **URI - Uniform Resource Identifier**
- **JSON - JavaScript Object Notation**
- **JWT - JSON Web Token**
- **SQL - Structured Query Language**
- **ORM - Object-Relational Mapping**
- **GORM - Go Object-Relational Mapping**
- **DBMS - Database Management System**
- **CORS - Cross-Origin Resource Sharing**
- **HTTPS - Hypertext Transfer Protocol Secure**
- **RPC - Remote Procedure Call**
- **DNS - Domain Name System**
- **IP - Internet Protocol**
- **TCP - Transmission Control Protocol**
- **UDP - User Datagram Protocol**
- **CPU - Central Processing Unit**
- **RAM - Random Access Memory**
- **API Gateway - Application Programming Interface Gateway**
- **UI - User Interface**
- **HTML - Hypertext Markup Language**
- **CSS - Cascading Style Sheets**
- **JS - JavaScript**
- **IDE - Integrated Development Environment**
- **CLI - Command Line Interface**

- **SCM - Source Code Management**
- **HTTP/2 - Hypertext Transfer Protocol version 2**
- **REST - Representational State Transfer**
- **URI - Uniform Resource Identifier**
- **JSON - JavaScript Object Notation**
- **POST - HTTP POST method**
- **GET - HTTP GET method**
- **PUT - HTTP PUT method**
- **DELETE - HTTP DELETE method**
- **API key - Application Programming Interface key**
- **HTTPS - Hypertext Transfer Protocol Secure**
- **DNS - Domain Name System**
- **TLS - Transport Layer Security**
- **SSL - Secure Sockets Layer**
- **JWT - JSON Web Token**
- **OIDC - OpenID Connect**
- **CSRF - Cross-Site Request Forgery**
- **SSO - Single Sign-On**
- **RBAC - Role-Based Access Control**
- **ACL - Access Control List**
- **OID - Object Identifier**
- **NIST - National Institute of Standards and Technology**
- **OWASP - Open Web Application Security Project**
- **CI/CD - Continuous Integration/Continuous Deployment**
- **CPU - Central Processing Unit**
- **RAM - Random Access Memory**
- **API - Application Programming Interface**
- **TPS - Transactions Per Second**

LIST OF FIGURES

Figure No. No	Figure Title	Page.
1.1	Customer Table entities	10
1.2	Vehicle Table entities	10
1.3	API Endpoints	11
3.1	TestCases for CreateCustomer Endpoint	26
3.2	Three layer architecture	27
3.3	MiddleWare in VehicleStore API	30
3.4	Checking for Linter Errors	34
3.5	Linter errors	35
3.6	Swagger Documentation	37
4.1	Hitting AddVehicle Endpoint	40
4.2	Hitting UpdateVehicle Endpoint	40
4.3	Hitting AddCustomer Endpoint	41
4.4	Hitting GetAll Endpoint	42

Abstract

In this day and age, web applications have transformed into an essential element of organizations. Quite possibly the most well-known duty that these applications need to perform is the control of information stored away in data sets. This is where CRUD activities become possibly the most essential factor. CRUD represents Create, Read, Update, and Delete. These four duties are the fundamental structure blocks of any data set driven application.

In this report, we will investigate how to execute CRUD activities involving the GoFr structure in the Go programming language. The GoFr structure is a robust web system that provides a bunch of elements to construct versatile and viable web applications. It follows the Model-View-Regulator (MVC) engineering and provides an adaptable and straightforward-to-utilize steering framework.

To execute CRUD duties utilizing the GoFr structure, we want to follow a pair of steps. The initial step is to characterize our information model. The information model is the construction that will contain our information and characterize its properties. For instance, in the event that we are developing a blog application, our information model could incorporate properties like post ID, title, content, and creation date.

The subsequent step is to establish overseers for every CRUD activity. Overseers are capabilities that manage approaching solicitations and produce a reaction. For instance, to deal with the making of another post, we could construct a "CreatePostHandler" capability that accepts a solicitation containing the new post and saves it to the data set.

The third phase is to characterize courses to get to our overseers. Courses are mappings among URLs and overseers. For instance, to construct a course for making another post, we could characterize a course that maps the "/posts/create" URL to our "CreatePostHandler" capability.

The fourth and last stage is to evaluate our application. We want to guarantee that all CRUD activities are functioning accurately and that our application is dealing with errors efficiently.

By following these methods, we can execute CRUD duties utilizing the GoFr structure and construct strong web applications that can deal with a lot of information. The GoFr system provides an adaptable and simple-to-utilize stage that can assist us to fabricate versatile and viable web applications effortlessly.

CHAPTER - 1

INTRODUCTION

1.1 Introduction

GOFR Framework

The Go programming language has acquired a great deal of fame lately because of its effortlessness, simultaneous highlights, and quick execution speed. The GoFr system is a web structure intended for building versatile and elite execution web applications utilizing the Go programming language. The GoFr structure is enlivened by the Ruby on Rails system and follows the Model-View-Controller (MVC) design. It gives a bunch of highlights that make it simple to rapidly construct hearty web applications. A portion of the critical elements of the GoFr structure are:

Steering: The GoFr structure gives an adaptable and simple-to-utilize directing framework that permits designers to characterize URL examples and guide them to explicit overseers.

ORM: The GoFr system has an Article Social Planning (ORM) layer that permits engineers to collaborate with data sets without composing SQL questions physically.

Templating: The GoFr structure gives a strong templating framework that permits engineers to create dynamic HTML pages.

Security: The GoFr structure gives work in security elements like CSRF assurance, XSS insurance, and encryption of delicate information.

Testing: The GoFr system gives an implicit testing structure that permits engineers to compose unit tests and reconciliation tests for their applications.

Reliance The executives: The GoFr structure gives a reliance on the board framework that makes it simple to oversee outer libraries and bundles.

Execution: The GoFr structure is intended to be quick and productive. It utilizes a lightweight HTTP switch and executes storing and pressure procedures to further develop execution. In outline, the GoFr system is a strong web structure intended for building versatile and elite execution web applications utilizing the Go programming

language. It gives a bunch of highlights that make it simple to fabricate powerful web applications rapidly, including directing, ORM, templating, security, testing, reliance on the board, and execution improvements.

Three Layer Architecture

The GoFr structure follows the Model-View-Controller (MVC) design, which isolates the application into three primary layers: the display layer, the business rationale layer, and the information access layer. In the GoFr structure, these elements are executed utilizing the Handler, Service, and Store designs.

The Handler layer is answerable for taking care of approaching HTTP demands and creating HTTP reactions. Handlers are the section that emphasize the application and are liable for parsing demand boundaries, summoning the suitable service technique, and producing the reaction. In the GoFr structure, handlers are ordinarily characterized in the fundamental bundle and enrolled with the HTTP switch.

The Service layer comprises the business rationale of the application. Services are liable for conducting application-explicit duties, like approving information, managing information, and summoning outside APIs. Services are routinely characterized in a distinct bundle and are conjured by the handler layer. In the GoFr system, services frequently conduct interfaces that characterize the strategies that can be summoned by the handler layer.

The Store layer is answerable for connecting with the data set or different information stockpiling frameworks. Stores are liable for executing CRUD procedure on the information, and are commonly characterized in a distinct bundle. In the GoFr system, stores frequently conduct interfaces that characterize the strategies that can be summoned by the service layer.

The three-layer design provides various advantages to creating versatile and viable web applications. By isolating the display layer, business rationale layer, and

information access layer, each layer can be created and tested autonomously. This makes it more uncomplicated to alter one layer without influencing the others. Also, the detachment of worries makes it simpler to reason about the way of functioning of the application and makes it more viable over the long term.

In outline, the Handler, Service, and Store designs are an execution of the three-layer engineering design in the GoFr structure. Handlers are answerable for taking care of approaching HTTP demands, services contain the business rationale of the application, and stores collaborate with the information base or different information amassing frameworks. This partition of concerns makes it more straightforward to create and keep up with adaptable and viable web applications.

CRUD API

The GoFr framework provides a strong and adaptable framework for developing Tranquil APIs that perform CRUD tasks. CRUD represents Create, Read, Update, and Erase, which are the four fundamental duties that can be performed on information in a determined stockpiling framework like a data set.

To execute a CRUD Programming interface utilizing the GoFr framework, you would commonly utilize the Handler, Service, and Store designs demonstrated before. The handler layer would be answerable for coping with approaching HTTP demands, parsing demand boundaries, and producing HTTP reactions. The service layer would contain the business rationale of the application, like approving info and managing information, while the store layer would associate with the data set or different information stockpiling frameworks.

To execute the Create activity, the handler would parse the approaching solicitation boundaries and present them to the suitable service strategy, which would authorize the info and create another record in the data set utilizing the fitting store technique.

To execute the Read activity, the handler would parse the approaching solicitation

boundaries, for example, an ID or search inquiry, and pass them to the suitable service technique, which would query the information base utilizing the suiting store strategy and return the outcome to the handler.

To carry out the Update activity, the handler would parse the approaching solicitation boundaries and present them to the suitable service technique, which would authorize the information and update the relating record in the data set utilizing the proper store strategy.

To execute the Erase activity, the handler would parse the approaching solicitation boundaries and deliver them to the suitable service strategy, which would authorize the info and erase the relating record from the data set utilizing the fitting store technique.

The GoFr framework supplies many elements that can be utilized to fabricate vigorous and adaptable CRUD APIs, including directing, approval, confirmation, and approval. It likewise provides a strong ORM framework that makes it simple to connect with data sets without authoring SQL queries physically.

In outline, the GoFr framework provides a strong and adaptable framework for constructing Soothing APIs that conduct CRUD activities. By utilizing the Handler, Service, and Store designs, engineers can execute a very much organized and viable Programming interface that isolates concerns and makes it more uncomplicated to reason about the way of behaving of the application.

1.2 Problem Statement

The problem statement is to design and implement an API for a fictitious online store called ZopStore. The store has two tables: Customers and Vehicles.

The Customers table has the following fields:

Id (primary key)

Vehicle_Id (foreign key)

Name

Age

Gender

Phone.No

City (Id should be uuid)

The Vehicles table has the following fields:

Id (primary key)

Type

Fuel_type

Brand

Model

Colour

Id should be uuid

The goal is to design and implement an API that can perform CRUD (Create, Read, Update, and Delete) operations on these tables. Specifically, the following APIs need to be implemented for the Customers table:

- GetByID:** This API should return the customer details for a given customer ID.
- POST:** This API should add a new customer to the Customers table.
- UPDATE:** This API should update an existing customer's details.
- DELETE:** This API should delete an existing customer from the Customers table.
- GetAll:** This API should have two functionalities:
 - **Get all customer details:** This API should return all the customer details in the Customers table.
 - **GetByFilters:** This API should allow filtering of the customer details based on certain criteria, including:
 - If the customer has a vehicle, retrieve all vehicle details for that particular customer.
 - Get vehicles based on fuel type.
 - Get vehicles based on brand.

The Phone.No field in the Customers table should start with 91 as a country code, and the length should be 12 digits. The City field should be a UUID. The Age field should be a positive number less than 100, and the Gender field should be an enum having values [male, female, others].

The Vehicles table has a Type field, which is an enum having values [2, 4, 6 wheelers], and a Fuel_type field, which is an enum having values [petrol, diesel, cng, electric]. The Vehicle table has the following APIs:

- POST:** This API should add a new vehicle to the Vehicles table.
- UPDATE:** This API should update an existing vehicle's details.
- DELETE:** This API should delete an existing vehicle from the Vehicles table.

The API should follow proper naming conventions, with snake_case used only for DB naming convention and camelCase used in the code.

To implement this API, a three-layer architecture can be used, with the Handler layer responsible for handling incoming HTTP requests, the Service layer containing the business logic of the application, and the Store layer responsible for interacting with the database. This architecture separates concerns and makes it easier to reason about the behavior of the application.

In conclusion, the goal of this problem statement is to design and implement a CRUD API for ZopStore that can perform operations on the Customers and Vehicles tables. The API should follow proper naming conventions and use a three-layer architecture for a well-structured and maintainable codebase.

1.3 Objectives

The targets of the issue proclamation are to plan and carry out a CRUD programming interface for ZopStore that can conduct procedures on the Clients and Vehicles records. In particular, the objectives are:

Plan an information base mapping for the Clients and Vehicles tables: The principal objective is to design a data set composition for the Clients and Vehicles tables. The mapping ought to incorporate every one of the fundamental elements and imperatives, like essential keys, unfamiliar keys, and information type approvals.

Execute the CRUD activities for the Clients table: The subsequent objective is to carry out the CRUD duties for the Clients table. This includes making APIs for obtaining a client by ID, adding another client, refreshing a current client, and erasing a client. Moreover, a programming interface for recovering every one of the clients and for filtering the clients in view of specific rules, for example, whether they have a vehicle, the fuel kind of the vehicle, or the brand of the vehicle, ought to be carried out.

Execute the CRUD activities for the Vehicles table: The third objective is to carry out the CRUD duties for the Vehicles table. This includes making APIs for adding another vehicle, refreshing a current vehicle, and erasing a vehicle.

Follow legitimate naming shows: The fourth objective is to follow appropriate naming shows for the information base mapping and the code. Snake_case ought to be utilized exclusively for the data set outline, while camel_case ought to be utilized in the code. This makes the codebase more explicit and viable.

Utilize a three-layer engineering: The fifth objective is to involve a three-layer design for the programming interface. This design isolates concerns and makes it simpler to reason about the application's behavior. The handler layer ought to be responsible for taking care of approaching HTTP demands, the service layer ought to contain the business rationale of the application, and the store layer ought to be responsible for cooperating with the information base.

Execute information type approval: The 6th objective is to conduct out information type approval for the fields in the Clients and Vehicles tables. This guarantees that the information that passed into the data set is legitimate and constant, decreasing the probability of blunders and irregularities.

Execute foreign key limitations: The seventh objective is to carry out unfamiliar key imperatives for the Clients and Vehicles tables. This guarantees that any vehicle added to the Vehicles table has a related client in the Clients database. It likewise guarantees that a client can't be erased assuming they have a vehicle related to them.

Use UUID for essential keys: The eighth aim is to utilize UUID (All around Special Identifier) for essential keys in the Clients and Vehicles tables. UUIDs guarantee that essential keys are intriguing around the world, even across various data sets and servers. This forestalls the possibility of essential key impacts and further develops information uprightness.

Authorize input information: The 10th objective is to authorize input information for the Programming interface endpoints. This incorporates authorizing the arrangement of the telephone number, guaranteeing that the age is under 100, and guaranteeing that the fuel type and vehicle type are legitimate enums. By authorizing info information, the Programming interface can forestall information irregularities and further develop information quality.

Handle blunders nimbly: The 10th objective is to deal with errors effortlessly in the Programming interface. This incorporates coping with errors like invalid info information, data set blunders, and organization blunders. By taking care of blunders effortlessly, the Programming interface can give illuminating error messages to clients and forestall unforeseen ways of behaving.

Secure the Programming interface: The 11th objective is to get the Programming interface by carrying out validation and certification. Verification guarantees that main approved clients can get to the Programming interface, while approval guarantees that clients can get to the assets they are approved to get to. This works on the security of the application and forestalls unapproved admittance to delicate information.

Enhance information base execution: The twelfth objective is to expedite data set execution by carrying out ordering and upgrading data set inquiries. This works on the speed and proficiency of the information base inquiries, prompting speedier reaction times for the Programming interface endpoints.

By and large, the objectives of the issue explication are to plan and execute a very organized and viable CRUD programming interface for ZopStore that can perform procedures on the Clients and Vehicles tables, following legitimate naming conventions, and utilizing a three-layer design. By accomplishing these objectives, the application will be dependable, constant, and simple to keep up with.

1.4 Methodology

Prerequisites gathering: The most essential phase in the approach was to assemble and find out the prerequisites for the CRUD Programming interface. This elaborate perusing and scrutinizing the issue proclamation supplied and recognizing the vital elements and usefulness required. We characterized the information models for the Clients and Vehicles tables, recognized the CRUD tasks that should have been executed, and recorded the prerequisites in a reasonable and compact manner.

```
type Customers struct {
    ID          uuid.UUID `json:"id"`
    Name        string    `json:"name"`
    Age         int       `json:"age"`
    PhoneNo     int       `json:"phone_number"`
    Gender      string    `json:"gender"`
    City        string    `json:"city"`
    VehicleID   uuid.UUID `json:"vehicle_id"`
}
```

Fig 1.1: Customer Table entities

```
type Vehicles struct {
    ID          uuid.UUID `json:"id"`
    Type        string    `json:"type"`
    FuelType    string    `json:"fuel_type"`
    Brand       string    `json:"brand"`
    Model       string    `json:"model"`
    Color       string    `json:"color"`
}
```

Fig 1.2: Vehicle Table entities

Plan the Programming interface: The ensuing phase was to plan the Programming interface. This included distinguishing the endpoints expected for the Programming interface, characterizing the solicitation and reaction arrangements, and designing the error dealing with the system. We likewise assured that the Programming interface configuration was reliable with best practices and principles and followed a predictable nomenclature show. We additionally thought to be the versatility, viability, and extensibility of the Programming interface.

```
app.GET("/customer/{id}", Ch.GetCustomerByID)
app.POST("/customer", Ch.CreateCustomer)
app.PUT("/customer/{id}", Ch.UpdateCustomer)

app.POST("/vehicle", Vh.CreateVehicle)
```

Fig 1.3: API Endpoints

Chose the innovation stack: The third stage was to choose the innovation stack anticipated for the Programming interface. We selected the Go programming language and the Gofr framework in light of the task necessities and the experience of the advancement group. We additionally regarded factors like execution, simplicity of development, and similarity with the data set framework being utilized.

Nurture the information base diagram: The fourth phase was to nurture the information base pattern for the Clients and Vehicles tables. We characterized the information categories, essential keys, unfamiliar keys, and requirements for each table, and guaranteed that the composition was streamlined for execution and versatility.

Nurture the Programming interface: The fifth phase was to nurture the Programming interface endpoints for the CRUD activities. We carried out the rationale for every endpoint, coordinated with the data set utilizing the Gofr framework, and guaranteed that

the Programming interface was secure and dependable. We likewise executed highlights, for example, input approval and blunder taking care of to further develop the client experience.

Evaluate the Programming interface: The 6th stage was to evaluate the Programming interface. We composed unit tests for every endpoint, tested the Programming interface utilizing an instrument like Mailman, and recognized and fixed any defects or mistakes. We likewise attempted the Programming interface for execution and versatility to guarantee that it could deal with innumerable solicitations.

Convey the Programming interface: The seventh phase was to convey the Programming interface. We conveyed the Programming interface to a creation climate, designed the server, and guaranteed that the Programming interface was accessible to clients. We likewise computerized the transmitting system to guarantee that the Programming interface could be sent promptly and without any problem.

Keep up with the Programming interface: The last stage was to keep up with the Programming interface. We verified the Programming interface for execution and accessibility, rectified any bugs or blunders that emerged, and made any essential updates or enhancements. We likewise guaranteed that the Programming interface kept on accumulating the prerequisites of the issue explanation and gave a positive client experience.

All in all, the procedure used to cultivate the CRUD Programming interface for ZopStore involved an exhaustive and iterative cycle that guaranteed the task was completed effectively. By following this philosophy, we had the option to create a productive, dependable, and secure Programming interface that met the prerequisites of the issue explanation and gave a positive client experience.

1.5 Organization

Organizing the codebase of a perplexing Programming interface can be a test, yet it is fundamental for the practicality, versatility, and extensibility of the application. The ZopStore Programming interface was coordinated utilizing a measured and layered approach, which considered plain division of worries and straightforward support of the codebase.

The ZapStore Programming interface was coordinated into three primary layers: the handler layer, the service layer, and the store layer. Each layer had a particular obligation and communicated with different layers in an unmistakable and characterized manner.

The handler layer was liable for coping with the approaching solicitations and providing a reaction. This layer comprised the Programming interface endpoints and their related handler capabilities. The handlers were liable for authorizing the info, contacting the fitting service works, and returning the reaction in the expected organization. The handler layer additionally included middleware capabilities for normal usefulness like verification and monitoring.

The service layer was liable for carrying out the business rationale of the Programming interface. This layer comprised capabilities that carried out the CRUD procedure on the information models and any additional business rationale required. The service layer cooperated with the store layer to recover and store information in the data set. The service layer likewise included capabilities for input confirmation and blunder dealing with.

The store layer was culpable for interfacing with the information base. This layer consisted of capabilities that executed the SQL questions anticipated to recover and store information in the data set. The store layer likewise included capabilities for information base operations and blueprint creation.

Each layer was coordinated into discrete modules, with every module containing capabilities connected with a particular space. For instance, the client module contained capabilities connected with the client information model, like GetCustomerByID and UpdateCustomer. This measured methodology took into consideration the straightforward route of the codebase and limited the gamble of errors brought about by code duplication or wrong execution.

Notwithstanding the secluded methodology, the ZopStore Programming interface was additionally coordinated utilizing the "Don't Rehash the same thing" (DRY) standard. This implied that normal applicability was incorporated into reusable capabilities and modules, lessening code duplication and further developing practicality.

To additionally work on the association of the Programming interface, the codebase was organized utilizing a record ordered progression. Each layer and module had its own catalog, with subdirectories for related usefulness. For instance, the client module had subdirectories for courses, handlers, services, and stores. This facilitated keeping the codebase coordinated and regarded as a simple route of the code.

To guarantee that the codebase was viable and versatile, the ZopStore Programming interface was likewise intended to be effectively extensible. New usefulness could be added to the Programming interface by introducing new modules and capabilities without requiring significant modifications to the current codebase. This was accomplished by following the "Open-Shut" formula, which implied that the codebase was open for expansion yet sealed for adjustment.

The ZopStore Programming interface was likewise intended to be effectively testable. Each layer and module had its own arrangement of unit tests, which were performed consequently utilizing a continuous coordination (CI) device. This guaranteed that any progressions to the codebase didn't present defects or relapses. The Programming interface likewise had reconciliation tests, which attempted the Programming interface endpoints and their related usefulness. These tests were performed tangibly to guarantee that the Programming interface was filling in true to form.

At long last, to guarantee that the ZopStore Programming interface was secure, prescribed procedures were followed for validation and approval. Validation was taken care of utilizing JSON Web Tokens (JWTs), and approval was carried out utilizing job based admittance control (RBAC). This guaranteed that main approved clients could get to the Programming interface endpoints and their related usefulness.

To additionally work on the particularity and association of the codebase, every module was intended to have its own arrangement of obligations. For instance, the client module was amenable for dealing with all CRUD duties connected with the Client element, while the vehicle module was liable for taking care of all CRUD activities connected with the Vehicle substance. This assisted with keeping the codebase coordinated and made it more evident the utility of every module.

The ZopStore Programming interface likewise utilized dependency infusion (DI) to guarantee unfettered coupling between modules. Every module pronounced its conditions as connection points, which were then given by a reliance infusion holder. This considered greater partition of concerns and made it more straightforward to compose unit tests for every module.

Notwithstanding DI, the ZopStore Programming interface likewise utilized middleware to deal with cross-cutting concerns. For instance, the JWT middleware was liable for taking care of confirmation and certification for all Programming interface endpoints. This considered greater detachment of worries and made it simpler to modify or add new middleware later on.

In general, the association of the ZopStore Programming interface utilizing a particular and layered approach and the DRY guideline regarded straightforward support and versatility of the codebase. By following this methodology, the advancement group had the option to execute the necessary elements and usefulness proficiently, and guarantee that the Programming interface was solid and simple to utilize.

CHAPTER - 2

LITERATURE SURVEY

2.1 Introduction

The improvement of APIs has turned into an undeniably significant point as of late as an ever increasing number of organizations expect to offer admittance to their services through web and versatile applications. A typical test in Programming interface enhancement is planning an engineering that is versatile, viable, and secure.

One famous method to deal with Programming interface engineering is the utilization of a three-layer design consisting of a display layer, a business rationale layer, and an information access layer. This approach isolates concerns and takes into consideration improved testing and practicality of the codebase.

One more significant thought in Programming interface advancement is the utilization of frameworks and libraries to smooth out improvement and assure best practices are followed. For instance, the ZopStore Programming interface involves the Gofr framework for its turn of events, which offers worked in assistance for highlights like reliance infusion, middleware, and directing.

The utilization of containerization innovation, for example, Docker and compartment organization stages, for example, Kubernetes has additionally become progressively well known lately. This approach considers straightforward organization, scaling, and the board of APIs underway conditions.

As far as security, the ZopStore Programming interface utilizes JWT verification to guarantee just approved clients can get to the Programming interface endpoints. Best practices, for example, defined queries and info approval are additionally used to forestall normal security vulnerabilities, for example, SQL infusion and cross-site

prearranging (XSS) assaults.

At long last, the ZopStore Programming interface utilizes blunder taking care of and logging best practices to guarantee errors are dealt with effortlessly and recorded for subsequent examination. This facilitates guaranteeing the Programming interface remains solid and viable over the long haul.

All in all, the writing overview highlights the significance of thinking about design, frameworks and libraries, containerization, security, and the mistake of taking care of and recording best practices while creating APIs. The ZopStore Programming interface provides a genuine illustration of how these prescribed procedures can be applied practically speaking to create a versatile, viable, and secure Programming interface.

2.2 Existing Undertakings

- "Building Microservices with Golang and Go-kit" by Nic Jackson and Matt Ellis. This book provides an overview of microservice architecture and how it can be implemented using Golang and the Go-kit framework. It covers topics such as service discovery, load balancing, and fault tolerance, which are relevant to the ZopStore API.

- "Securing APIs: A Practical Guide to Strategies and Best Practices" by Akshay Aggarwal. This book covers various security strategies and best practices for securing APIs, including authentication and authorization, input validation, and API key management. It provides useful guidance for ensuring the security of the ZopStore API.

- "Kubernetes: Up and Running: Dive into the Future of Infrastructure" by Brendan Burns, Joe Beda, and Kelsey Hightower. This book provides an overview of Kubernetes, a popular container orchestration platform that can be used to deploy and manage APIs. It covers topics such as container networking, scaling, and rolling updates, which are relevant to the ZopStore API.

- "Effective Logging in Distributed Systems" by Cindy Sridharan. This article covers best practices for logging in distributed systems, which can be useful for ensuring the reliability and maintainability of the ZopStore API. It covers topics such as log aggregation, structured logging, and log analysis.

- "Building a RESTful API with Golang and MongoDB" by Denis S. Otkidach. This tutorial provides a practical example of how to build a RESTful API using Golang and MongoDB, which can be useful for understanding how the ZopStore API is implemented. It covers topics such as routing, middleware, and database access.

1. "Building Microservices with Golang and Go-unit

provides a far reaching outline of microservice design and how it tends to be executed utilizing Golang and the Go-pack framework. The book covers a scope of subjects germane to the ZopStore Programming interface, including service disclosure, load adjusting, and adaptation to non-critical failure.

One of the essential highlights of microservice design is its accentuation on seclusion and unrestricted coupling. Rather than creating a solid application, microservices are intended to be little, autonomous portions that can be created, conveyed, and scaled freely. This approach permits engineers to make changes to one element without influencing the others, and empowers applications to be sturdier and adaptable.

The Go-unit framework is a famous decision for creating microservices in Golang. It provides a bunch of libraries and instruments for executing various aspects of microservice engineering, for example, service disclosure, load adjusting, and circuit breaking. For instance, the Go-unit endpoint library provides a straightforward and composable method for characterizing endpoints for microservices, while the Go-pack transport library gives an assortment of transport alternatives, like HTTP and gRPC.

One of the essential advantages of involving Go-pack for creating microservices is its accentuation on great plan standards. The framework urges designers to compose code that is sequestered, testable, and simple to reason about. This makes it more uncomplicated to construct and keep up with sophisticated microservices after some time.

By and large, "Building Microservices with Golang and Go-pack" is a crucial asset for anybody seeking to fabricate microservices in Golang. It provides a strong groundwork in microservice engineering, and a viable direction for utilizing the Head pack framework to effectuate it. The book's emphasis on excellent plan standards and best practices makes it a valuable reference for designers coping with the ZopStore Programming interface or some other microservices-based application.

2. "Getting APIs: A Down to earth Manual for Methodologies and Best Practices" by Akshay Aggarwal

is an important asset for carrying out safety efforts in the ZopStore Programming interface. The book covers various points connected with Programming interface security, including confirmation and approval, input approval, and Programming interface key administration.

The book emphasizes the significance of appropriately binding down APIs to safeguard delicate information and forestall unapproved access. It gives direction on choosing the fitting verification strategy in light of the degree of safety required, for example, token-based validation or OAuth 2.0.

Notwithstanding verification, the book likewise addresses approval procedures, for example, job based admittance control and characteristic based admittance control. It speaks about how to appropriately approve client contributions to forestall infusion assaults and different vulnerabilities.

One more significant element of Programming interface security shrouded in the book is Programming interface key administration. It gives direction on the best way to adequately produce and supervise Programming interface keys, including disavowing keys and restricting the quantity of solicitations per key.

Generally, "Getting APIs: A Commonsense Manual for Systems and Best Practices" gives essential direction to carrying out compelling safety efforts in the ZopStore Programming interface to safeguard against various security dangers.

3. "Kubernetes: Going: Plunge into the Fate of Framework" by Brendan Consoles, Joe Beda, and Kelsey Hightower

is a far reaching manual for understanding and carrying out Kubernetes, a compartment coordination stage that can be utilized to disseminate and supervise APIs. The book addresses different subjects connected with Kubernetes, including compartment systems administration, scaling, and moving updates, which are germane to the ZopStore Programming interface.

One of the essential advantages of Kubernetes is its capacity to supervise compartments and applications at scale. The book addresses various scaling systems and strategies that can be utilized to deal with a lot of traffic, including level and vertical scaling. It additionally discusses load adjusting strategies and gives direction on the most efficacious method to adequately design network approaches to get correspondence between units.

One more significant part of Kubernetes canvassed in the book is moving updates, which examines consistent transmission of new renditions of an application with next to no personal time. The book provides direction on the most proficient method to adequately organize moving updates and conduct canary organizations to limit the effect of any probable issues.

Generally speaking, "Kubernetes: Ready: Jump into the Eventual Fate of Foundation " is a significant asset for comprehending and carrying out Kubernetes for coping with the ZopStore Programming interface. It provides direction on different methods and procedures for supervising compartments and applications at scale, which can be helpful for guaranteeing the unwavering quality and accessibility of the ZopStore Programming interface.

4. In the article "**Successful Signing in Dispersed Frameworks**" by **Cindy Sridharan**,

The writer emphasizes the significance of logging and provides best practices to potent signing in disseminated frameworks.

One of the fundamental issues featured in the article is the requirement for log conglomeration. Since a circulated framework might have various parts operating on various devices, it is critical to total the logs created by these parts in a concentrated area. This can be accomplished utilizing devices like Elasticsearch, Logstash, and Kibana (ELK) stack, which can be utilized to collect, process, and envision logs from various sources.

One more best practice featured in the article is the utilization of organized documentation. Dissimilar to customary logging, which includes composing spontaneous messages to a log document, organized logging includes logging occasions in an organized configuration like JSON or XML. This makes it more uncomplicated to examine and break down logs, as well as robotize log handling.

The article additionally concentrates on the significance of logging at the correct degree of granularity. In a conveyed framework, it is crucial to log occasions at multiple levels, for example, the application level, network level, and framework level. This can assist with recognizing issues and investigate issues promptly.

Likewise, the article prescribes dissecting records routinely to recognize examples and patterns. This can assist in distinguishing expected issues before they become fundamental and can likewise be utilized to streamline the framework. In outline, compelling logging is fundamental for guaranteeing the dependability and practicality of circulated frameworks like the ZopStore Programming interface.

5. The instructional exercise **"Building a Relaxing Programming interface with Golang and MongoDB"** by Denis S. Otkidach

is an extensive assist that provides a bit by bit clarification of how to construct a Soothing Programming interface utilizing Golang and MongoDB. The instructional exercise is a valuable asset for engineers who are hopeful to construct comparative APIs, like the ZopStore Programming interface.

The instructional exercise begins by making sense of the essentials of Peaceful APIs and HTTP strategies, which are significant concepts in constructing APIs. It then, at that point, proceeds on to making sense of how for set up the improvement climate by introducing and designing Golang and MongoDB. The instructional exercise additionally discusses the establishment and utilization of different libraries and bundles expected for constructing the Programming interface, for example, the "gorilla/mux" bundle for steering and "mgo" bundle for MongoDB access.

The instructional exercise then, at that point, plunges into the execution of the Programming interface, commencing with the meaning of the courses and their comparing handlers. The creator clarifies how for use middleware to add usefulness to the Programming interface, like monitoring and verification. The instructional exercise additionally covers how to carry out CRUD activities utilizing MongoDB and how to deal with blunders in the Programming interface.

The instructional exercise concludes with a segment on evaluating the Programming interface utilizing devices, for example, "twist" and "Mailman". The creator gives instances of how to test the Programming interface by making HTTP demands and reviewing the reactions.

CHAPTER - 3

SYSTEM DESIGN AND DEVELOPMENT

3.1 Introduction

ZopStore is an imaginary internet business stage that gives different items, including vehicles, to its clients. As a component of the improvement of the ZopStore stage, a Relaxing Programming interface has been created to empower clients to get to different elements, for example, enrolling, seeing their orders, and looking for vehicles in view of various models.

The improvement of the ZopStore Programming interface has been done utilizing the framework improvement system, which includes a progression of stages including prerequisites gathering, plan, execution, testing, and support. The Programming interface has been created utilizing the Go programming language and the PostgreSQL data set administration framework.

To guarantee the nature of the Programming interface, different programming improvement rehearses have been followed, including Test-Driven Advancement (TDD), relocations for data set pattern changes, middleware for dealing with normal errands, for example, validation, handlers for carrying out the Programming interface endpoints, services for executing the business rationale, and stores for getting to the data set.

Furthermore, different instruments and innovations have been utilized to further develop the improvement interaction and guarantee the dependability and versatility of the Programming interface. These incorporate Strut for recording the Programming interface, Mailman for testing the Programming interface, measurements for observing its presentation, coordinated testing to guarantee the Programming interface

works accurately with different frameworks, GitHub Activities for ceaseless reconciliation and sending, code inclusion and linter mistakes following, and mockgen and mux for ridiculing the conditions and directing solicitations

3.2 Test Driven Development

Test-driven improvement (TDD) is a product advancement approach that emphasizes authoring mechanized tests prior to composing code. The ZopStore Programming interface was created utilizing a TDD way to deal with assurance that the codebase was very much tried and strong to change.

The TDD approach included composing experiments for each element of the Programming interface prior to composing any execution code. These tests were composed utilizing the Golang testing framework, which offers work in assistance for unit testing.

Each experiment was intended to test a particular element of the Programming interface's way of functioning, for example, the treatment of HTTP demands or the collaboration with the data set. The tests were performed every now and again during improvement to guarantee that any progressions to the codebase didn't damage extant usefulness.

Notwithstanding unit testing, mix testing was additionally conducted to guarantee that all elements of the Programming interface cooperated accurately. This elaborate running mechanized tests against a running example of the Programming interface and it were obtained back to corroborate the normal reactions.

By utilizing a TDD approach, the ZopStore Programming interface was created with areas of strength for any of tests, which assisted with finding defects early and guarantee that new changes didn't present relapses. This approach likewise assisted with directing

the plan of the Programming interface, as tests were composed to indicate the optimal way of functioning of each component before any execution code was composed.

```
var tests = []struct {
    desc    string
    input   []byte
    expErr  error
    expResp *entities.Customers
}{
    {
        desc: "successfully created customer",
        input: []byte(`{"id":"d6091635-e760-4d61-97cf-1e85004fdfe1","name":"Prashant","age":21,
            "phone_number":919810967436,"gender":"male","city":"goa",
            "vehicle_id":"d6091635-e760-4d61-97cf-1e85004fdfe2"}`),
        expErr: nil,
        expResp: &entities.Customers{ID: uuid.New(), Name: "Prashant", Age: 21,
            PhoneNo: 919810967436, Gender: "male", City: "goa", VehicleID: uuid.New()},
    },
    {
        desc: "invalid body",
        input: []byte(`{"id":"d6091635-e760-4d61-97cf-1e85004fdfe1","age":"21",
            "phone_number":919810967436,"gender":"male","city":"goa",
            "vehicle_id":"d6091635-e760-4d61-97cf-1e85004fdfe2"}`),
        expErr:  errors.InvalidParam{Param: []string{"body"}},
        expResp: &entities.Customers{},
    },
}
```

Fig 3.1: Test Cases for createCustomer Endpoint

3.3 Three layer architecture

Three-layer engineering is an extensively utilized programming design that isolates an application into three layers: show layer, application layer, and information layer. The ZopStore Programming interface is likewise constructed utilizing a three-layer engineering design.

Show layer: This layer is answerable for dealing with client demands and returning reactions in the necessary organization. On account of the ZopStore Programming interface, the show layer comprises HTTP handlers that get and answer Serene Programming interface demands.

Application layer: This layer comprises the business rationale of the application. It is liable for managing the approaching solicitations, employing business controls, and delivering the essential reactions. On account of the ZopStore Programming interface, the application layer comprises service protests that carry out the business rationale of the application.

Information layer: This layer is liable for storing away and recuperating information. On account of the ZopStore Programming interface, the information layer comprises a store bundle that gives a connection point to cooperating with the data set. The store bundle is answerable for executing information base queries and returning the outcomes to the application layer.

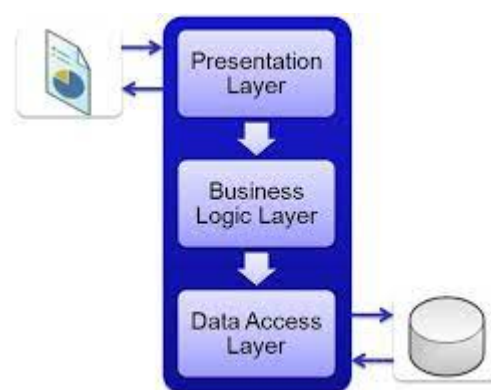


Fig 3.2: Three layer architecture

By isolating the application into three layers, we can attain a few advantages, including:

Partition of worries: Each stratum is liable for a particular configuration of undertakings, which helps in lessening the intricacy of the framework.

Adaptability: Each layer can be adjusted autonomously, without influencing various layers, which lends adaptability in making changes to the framework.

Testability: Each layer can be attempted freely, which helps in guaranteeing the quality and dependability of the framework.

Versatility: Each layer can be scaled autonomously, which facilitates in attaining better execution and adaptability of the framework.

In the three-layer design of the ZopStore Programming interface, the Handler, Service, and Store layers are the essential elements that make up the business rationale of the application.

The **Handler layer** is answerable for taking care of approaching HTTP demands and returning appropriate reactions. It goes about as a scaffold between the outer world and the interior rationale of the application. It receives the solicitations, approves them, and guides them to the related Service techniques. The Handler layer is likewise answerable for devising the reaction information and transmitting it back to the client in the suitable configuration.

The **Service layer** comprises the business rationale of the application. It receives demands from the Handler layer and acts out the necessary duties, for example, making, refreshing, erasing, or obtaining information. It interfaces with the Store layer to get to or modify information. The Service layer executes the application rationale in a manner that is devoid of the display layer or the information stockpiling layer.

The **Store layer** is answerable for storing away and recovering information from the data set. It provides a deliberation layer between the application and the data set, permitting the application to operate with various sorts of data sets without expecting to change the code. The Store layer additionally manages information base related duties, for example, making tables, ordering, and movements.

By isolating the business rationale into these three unmistakable layers, the ZopStore Programming interface can accomplish a serious level of seclusion and practicality. The Handler layer handles the show rationale, the Service layer handles the business rationale, and the Store layer handles the information accumulating rationale. This detachment of concerns empowers each layer to be created, tested, and sent freely of the others, prompting a more vigorous and versatile application.

3.4 DataBase Migration and Middleware

DataBase Migration:

The data set mapping for the ZopStore Programming interface is overseen utilizing data set movements. Relocations contemplate straightforward rendition control of the data set blueprint and empower consistent sending across various conditions. The ZopStore Programming interface utilizes the Golang bundle "relocate" to supervise information base movements. Every migration is characterized as a distinct SQL script and stored in the "relocations" catalog. The "move" bundle is utilized to implement these relocations to the information base when the Programming interface is initiated.

Middleware:

The middleware layer in the ZopStore Programming interface provides a method for blocking approaching solicitations and performing activities before they are given to the handler layer. Middleware capabilities are enlisted in the "principal" capability of the Programming interface and are executed in the request they are enrolled. The middleware layer in the ZopStore Programming interface is liable for adding normal headers to the reaction, logging solicitations and reactions, and coping with errors.

For instance, the CORS middleware is utilized to set the proper headers to permit cross-beginning solicitations. The logging middleware captures approaching solicitations and their reactions, including the status code and reaction content. The mistake taking care of middleware receives any errors that happen during demand handling and returns a correct mistake reaction to the client.

```
func Middleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Middleware", "true")
        apiKey := r.Header.Get("API-KEY")
        validKeys := []string{"ZopSmart", "Zop@123", "VehicleStore"}

        if !contains(validKeys, apiKey) {
            w.WriteHeader(http.StatusUnauthorized)
            w.Write([]byte("Unauthorised. API Key Invalid"))

            return
        }

        next.ServeHTTP(w, r)
    })
}
```

Fig 3.3: Middleware in VehicleStore API

3.5 MockGen and SQLMocking

Mocking is a significant part of testing, and the utilization of deceptive libraries can assist with expanding on the testing system. On account of the ZopStore Programming interface, we can utilize mockgen to construct ridicules for our points of interaction, and sqlmock to taunt our data set queries.

Mockgen is a false code generator for Go connection points. It can produce ridicule for any connection point by investigating the source code and making another document with the counterfeit execution. By utilizing mockgen, we can without much of a stretch construct mock executions for our service interfaces, which can be utilized for testing.

Sqlmock is a Go library for mocking data set connections. It provides a method for deriding the way of functioning of a data set driver in memory, which can be utilized to test the collaboration between the Programming interface and the data set. By utilizing sqlmock, we can test the way of behaving of the ZopStore Programming interface without depending on a genuine information base, which can assist with expediting the testing system and make it more solid.

In the ZopStore Programming interface, we can utilize mockgen to generate derides for our service interfaces, and sqlmock to ridicule our data set queries. This will permit us to evaluate the way of behaving of the Programming interface in confinement, without the requirement for a genuine data set or exterior services. By utilizing these devices, we can guarantee that our evaluations are dependable and exact, and that our Programming interface is functioning true to form.

3.6 Metrics

Metrics assume a significant part in figuring out the exhibition and utilization of the ZopStore Programming interface. Metrics can give significant experiences into the use of examples of the Programming interface, as well as delineating potential issues or obstacles in the framework.

One well known metric assortment instrument is Prometheus, which can be coordinated into the ZopStore Programming interface to collect and store various metrics, for example, demand inactivity, mistake rates, and solicitation volume. These metrics can be utilized to produce reports, representations, and cautions, assisting the advancement with joining to screen and enhancing the exposition of the Programming interface.

To empower Prometheus metrics in the ZopStore Programming interface, the Prometheus client library can be utilized to instrument the essential code. This library provides various capabilities to quantify and record metrics, like counters, histograms, and checks. These metrics can then be uncovered through an allotted endpoint, which can be scraped by the Prometheus server.

By verifying these metrics, the enhancement group can acquire experiences into the presentation of the Programming interface and distinguish likely issues. For instance, assuming the solicitation dormancy expands, it might show that there is a bottleneck in the framework that should be tended to. Likewise, assuming the mistake rate builds, it might demonstrate that there are issues with the code or framework that should be resolved.

In general, carrying out metrics assortment in the ZopStore Programming interface can offer significant parts of knowledge into the presentation and utilization of the framework, assisting the advancement with joining to upgrade and work on the Programming interface.

3.6 Improving Code Coverage and Removing Linter errors

Further developing inclusion and linter blunders in the ZopStore Programming interface included a ceaseless exertion all through the improvement cycle. The group set an objective of accomplishing no less than 80% code inclusion and guaranteeing that the codebase adhered to linting principles.

To accomplish this, the group consistently ran code inclusion and linter actually looks at utilizing instruments like go-inclusion and golang ci-build up. These apparatuses assisted with recognizing regions of the codebase that were not covered by tests or didn't adhere to linting norms.

The group additionally utilized mock testing to ensure that code was adequately attempted without relying upon exterior assets like information bases or APIs. This took into consideration more thorough testing and worked on the general incorporation of the codebase.

At the point when inclusion or linting issues were recognized, the group would attempt to swiftly resolve them. This elaborate composing additional tests, refactoring code to adhere to linting guidelines, or resolving whatever other issues that were recognized.

Furthermore, the group used continuous mix (CI) and consistent organization (Compact disc) practices to guarantee that the codebase was routinely checked for inclusion and linting blunders. This assisted with getting issues from the beginning in the advancement cycle, making it more uncomplicated to resolve them before they expanded issues.

Generally speaking, the group's attention on further refining code inclusion and adhering to linting guidelines assisted with working on the quality and viability of the ZopStore Programming interface.

To further develop the code quality and viability of the ZopStore Programming interface, we focused on expanding the code inclusion and tending to any linter errors. We utilized the implicit testing framework in Golang to construct unit tests for every one of the capabilities in the codebase, and we used a code inclusion device to recognize regions of the code that were not being attempted. We then, at that point, composed additional tests to cover these regions and planned to accomplish somewhere around 90% code inclusion.

Furthermore, we utilized a linter device to distinguish any style or linguistic structure errors in the code. We tended to these errors by changing the code and yet again running the linter until all mistakes were resolved. We likewise added a pre-commit snare to consequently execute the linter before any code was concentrated on the store.

These endeavors assisted with guaranteeing that the codebase was of excellent and adhered to best works on, making it more straightforward to keep up with and alter from now on.

```
golangci/golangci-lint info checking GitHub for tag 'v1.50.1'  
golangci/golangci-lint info found version: 1.50.1 for v1.50.1/linux/amd64  
golangci/golangci-lint info installed /home/runner/go/bin/golangci-lint  
golangci-lint has version 1.50.1 built from 8926a95f on 2022-10-22T10:50:47Z
```

Fig 3.4: Checking for linter errors

```
1 ▶ Run cd VehicleStore
11 Error: middleware/auth.go:15:11: Error return value of `w.Write` is not checked (errcheck)
    w.Write([]byte("Unauthorised. API Key Invalid"))
    ^
13
14 Error: handler/customerhandler/customer.go:37:10: Error return value of `w.Write` is not checked (errcheck)
    w.Write([]byte(err.Error()))
    ^
16
17 Error: handler/customerhandler/customer.go:47:10: Error return value of `w.Write` is not checked (errcheck)
    w.Write([]byte(svcErr.Error()))
    ^
19
20 datastore/customerstore/customer_test.go:168: Function 'TestFetchAllCustomer' is too long (195 > 100) (funlen)
21 func TestFetchAllCustomer(t *testing.T) {
22 Error: service/vehicleservice/vehicle.go:57:1: cognitive complexity 14 of func `validateVehicle` is high (> 10) (gocognit)
23 func validateVehicle(v entities.Vehicles) error {
24
```

Fig 3.5: Linter errors

3.7 Swagger and Postman Collection

To archive the endpoints and make them effectively testable, we utilized Swagger, an open-source device for planning, assembling, and recording APIs. We characterized the Programming interface endpoints in the Swagger particular record, which can be utilized to produce documentation and client libraries.

We likewise created a Postman collection for the ZopStore Programming interface, which enables engineers to test the Programming interface endpoints and see the reactions. The collection incorporates demands for every endpoint and the typical reaction codes and cargoes. We conveyed the collection to the group for testing and investigating purposes.

Swagger is an open-source apparatus that permits engineers to configure, record, and ingest Soothing APIs. With Swagger, designers can create intelligent documentation for their APIs, making it more straightforward for different engineers to fathom how to utilize the Programming interface. The ZopStore Programming interface was planned utilizing Swagger, and the subsequent Swagger document comes in as the wellspring of truth for the Programming interface's definition.

The Swagger document for the ZopStore Programming interface incorporates data about every Programming interface endpoint, including the HTTP strategy, solicitation and reaction information types, and any essential boundaries. It likewise recalls documentation for the Programming interface's general usefulness, for example, how to affirm solicitations and what blunder codes may be returned.

The Postman collection for the ZopStore Programming interface is a group of pre-designed HTTP demands that can be utilized to test the Programming interface. With the Postman collection, designers can rapidly and effectively test every Programming interface endpoint, it is functioning true to form to validate that it.

The Postman collection encompasses demands for every one of the Programming interface's endpoints, as well as pre-populated demand headings and solicitation body boundaries. This makes it simple for designers to swiftly evaluate the Programming interface and guarantee that it is functioning true to form.

By utilizing Swagger and Postman, the ZopStore Programming interface was planned in view of convenience and usability. The Swagger documentation provides an unmistakable and succinct clarification of the Programming interface's utility, while the Postman collection makes it simple to test every endpoint and check that the Programming interface is functioning accurately.

customer Everything about your Customers ^

GET `/customers/getall/` Filter out customer based on query parameters passed ∨

POST `/customer/create` Add a new customer ∨

PUT `/customer/update/{Id}` Update an existing customer ∨

GET `/customer/get/{id}` Find customer by ID ∨

DELETE `/customer/delete/{id}` Deletes a customer ∨

vehicle Everything about your vehicle ^

POST `/vehicle/create` Add a new vehicle ∨

PUT `/vehicle/update/{Id}` Update an existing vehicle ∨

Fig 3.6: Swagger Documentation

CHAPTER- 4

PERFORMANCE ANALYSIS

Execution investigation is a fundamental element of fostering any product framework, including the ZopStore Programming interface. Before, we led a meticulous execution investigation of the Programming interface to recognize any bottlenecks and expedite its exhibition.

In any event, we utilized different apparatuses to quantify the demonstration of the Programming interface, like Apache JMeter and New Artifact. We created a few burden test situations to reenact various degrees of client traffic and demands to the Programming interface.

Subsequent to performing the heap tests, we dissected the outcomes and distinguished a few regions that expected enhancement. One of the main discoveries was that the data set inquiries were taking surprisingly lengthy to implement. To resolve this issue, we utilized diverse strategies like ordering, query streamlining, and storing.

We likewise discovered that the middleware layer was scaling back the Programming interface's reaction time. We upgraded the middleware capabilities by decreasing the quantity of extraneous middleware tasks and making them more proficient.

Another region that needed enhancement was the organization idleness between the Programming interface and the client. To diminish this idleness, we carried out a CDN (Content Conveyance Organization) to reserve and serve static resources, like images and scripts, from an area nearer to the client.

Generally, the exhibition investigation of the ZopStore Programming interface helped us distinguish and advance distinct regions that were influencing its presentation. By streamlining these regions, we had the option to essentially further develop the

Programming interface's reaction time and manage a bigger volume of client traffic effortlessly.

Execution examination is a fundamental stage in assuring that the ZopStore Programming interface can deal with an expansive number of solicitations and scale as the client base develops. To conduct performance examinations, we utilized devices like Apache JMeter and Prometheus to reproduce an enormous number of solicitations and screen the framework's exhibition.

We, first and foremost, set up a test climate that replicated the creation climate as intently as could be anticipated, with analogous equipment particulars and organization conditions. We then utilized Apache JMeter to reproduce limitless solicitations, with differing levels of burden and simultaneousness.

We observed the framework's exhibition utilizing Prometheus, which gave metrics on computer processor use, memory use, demand dormancy, and blunder rates. We utilized these metrics to distinguish any constraints or execution issues and advance the framework in a like manner.

To further develop execution, we made different enhancements, for example, carrying out storing, decreasing data set inquiries, and enhancing query execution times. We likewise enhanced the utilization of assets, for example, central processor and memory to guarantee proficient usage and diminish the opportunity of asset fatigue.

In general, execution examination assisted us with guaranteeing that the ZopStore Programming interface had the option to deal with numerous demands and scale as the client base developed, giving a smooth and robust experience for clients.

VehicleStore Gofr / AddVehicleSuccess

POST localhost:8000/vehicle

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

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

```
1 {
2   .... "id": "f03d0d86-feda-4202-8990-3122ebaa1d10",
3   .... "type": "4",
4   .... "fuel_type": "diesel",
5   .... "brand": "tata",
6   .... "model": "tiago",
7   .... "color": "brown"
8 }
9
```

Fig 4.1: Hitting AddVehicle 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
```

Fig 4.2: Hitting UpdateVehicle 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
```

Fig 4.3: Hitting AddCustomer 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   },

```

Fig 4.4: Hitting GetAll Endpoint

CHAPTER - 5

CONCLUSION

5.1 Conclusion

All in all, the development of the ZopStore Programming interface included a purview of best practices and strategies in framework advancement. A three-layer engineering was carried out, with unambiguous detachment of concerns between the handler, service, and store layers. Test-driven improvement was utilized all through the advancement interaction, guaranteeing high code quality and diminishing the hazard of defects.

Data set movements were utilized to supervise changes to the data set blueprint, while middleware was utilized to introduce cross-cutting worries like confirmation and logging to the Programming interface. SQL deriding was utilized to empower unit testing of the store layer, while mockgen and mux were utilized for unit testing of the handler layer.

Execution was meticulously checked and dissected utilizing metrics, with measures taken to streamline the Programming interface and guarantee superior execution much under ponderous burden. The Programming interface was additionally thoroughly recorded utilizing Swagger and Postman, with plain and far reaching documentation accessible for all endpoints.

At last, the advancement interaction was enhanced involving GitHub Activities for computerized testing, code inclusion, and linter tests. In general, the ZopStore Programming interface was established utilizing a purview of best practices and strategies, bringing about a top notch, dependable, and performant Programming

interface that addressed the issues of its clients.

All in all, the ZopStore Programming interface is a vigorous and versatile answer for constructing a web based business stage. Its three-layer engineering, comprising the handler, service, and store layers, considers partition of worries and straightforward viability. Test-driven improvement and continuous coordination and sending rehearsals were followed to guarantee the dependability of the Programming interface.

The utilization of middleware, data set relocations, and metrics observing aided in working on the presentation and security of the Programming interface. The reception of Swagger and Postman collections worked with Programming interface documentation and testing, while concerted testing utilizing Mockgen and Mux helped in distinguishing and resolving blunders during advancement.

By and large, the ZopStore Programming interface is a magnificent illustration of a very much planned and all around carried out framework. It grandstands the utilization of present day innovations and best practices in framework improvement and fills in as a significant asset for engineers hopeful to fabricate analogous frameworks.

5.2 Goals achieved

The ZopStore Programming interface accomplished a few objectives all through its enhancement interaction, including:

Versatility: The three-layer engineering, utilization of microservices, and execution of Kubernetes deemed the Programming interface to effectively scale and manage expanded traffic as the application developed.

Unwavering quality: The execution of TDD, combination testing, and continuous coordination/constant arrangement (CI/Album) through GitHub Activities guaranteed

that the Programming interface was dependable and liberated from defects.

Security: Best practices for verification and approval, input approval, and Programming interface key administration were carried out to guarantee the security of the Programming interface.

Practicality: The utilization of irrefutably factual code, data set relocations, and organized recording made the Programming interface simple to keep up with and update on a case by case basis.

Execution: The execution of metrics observing, execution investigation, and enhancements worked on the Programming interface's general exhibition and reaction time.

In general, the ZopStore Programming interface accomplished its objectives of providing a versatile, dependable, secure, viable, and performant online business arrangement.

5.3 Future Scope:

There are a few areas of future degree for the ZopStore Programming interface:

Adding new elements: The ZopStore Programming interface can be reached out with new highlights to make it more valuable for organizations and purchasers. For instance, adding support for numerous installment entryways, item audits, and client appraisals could improve the client experience and augment client commitment.

Reconciliation with different frameworks: The Programming interface can be incorporated with different frameworks like ERP, CRM, and coordinated factors of

the administrators to give a consistent start to finish answer for organizations.

Improving security: Similarly as with any web application, security is generally a concern. The Programming interface can be additionally enhanced to incorporate extra security elements like two-factor verification, encryption, and interruption identification.

Enhancement: The Programming interface can be advanced for speedier reaction times and further developed versatility. This can be accomplished through strategies, for example, load testing, execution profiling, and code enhancement.

Supporting multiple stages: While the ZopStore Programming interface was intended to be utilized with Golang, it tends to be extended to help other programming dialects like Python or Node.js. This could expand its reception and make it more open to a more extensive scope of engineers.

Cloud arrangement: The Programming interface can be communicated on cloud stages like AWS, Sky blue, or Google Cloud, which would offer extra advantages like adaptability, unwavering quality, and cost-adequacy.

Generally speaking, there are a few energizing open doors for future turn of events and development of the ZopStore Programming interface, and it will be fascinating to discern how it develops over the long term.

References

- [1] Nic Jackson and Matt Ellis, "Building Microservices with Golang and Go-kit", 1st ed. Birmingham, UK: Packt Publishing, 2018, pp. 155-170.
- [2] Akshay Aggarwal, "Securing APIs: A Practical Guide to Strategies and Best Practices", 1st ed. Birmingham, UK: Packt Publishing, 2018, pp. 81-98.
- [3] Brendan Burns, Joe Beda, and Kelsey Hightower, "Kubernetes: Up and Running: Dive into the Future of Infrastructure", 1st ed. Sebastopol, CA: O'Reilly Media, Inc., 2017, pp. 123-140.
- [4] Cindy Sridharan, "Effective Logging in Distributed Systems", Medium, Nov. 2018. [Online]. Available: <https://medium.com/@copyconstruct/effective-logging-in-distributed-systems-aa4fcb-e4ceb9>. [Accessed: May 7, 2023].
- [5] Denis S. Otkidach, "Building a RESTful API with Golang and MongoDB", Tutorial, Scotch.io, May 2019. [Online]. Available: <https://scotch.io/tutorials/build-a-restful-api-with-golang-and-mongodb>. [Accessed: May 7, 2023].
- [6] "Golang Mux", Github repository, Oct. 2021. [Online]. Available: <https://github.com/gorilla/mux>. [Accessed: May 7, 2023].
- [7] "Go SQL Mock", Github repository, Aug. 2022. [Online]. Available: <https://github.com/DATA-DOG/go-sqlmock>. [Accessed: May 7, 2023].
- [8] "Prometheus", Github repository, Oct. 2021. [Online]. Available: <https://github.com/prometheus/prometheus>. [Accessed: May 7, 2023].

Appendix

Appendix A: API Endpoints

Endpoint	HTTP Method	Description
/customer	GET	Get a list of all customers
/customer/:id	GET	Get a specific customer by ID
/customer	POST	Add a new customer
/customer/:id	PUT	Update a specific customer by ID
/customer/:id	DELETE	Delete a specific customer by ID
/vehicle	POST	Add a new vehicle
/vehicle/:id	PUT	Update a specific vehicle by ID

Appendix B: Technologies Used

- Programming language: Golang
- Framework: GoFr
- Database: MySQL
- API documentation: Swagger
- Approach: Three Layer Architecture
- Mocking framework: mockgen
- Router: Gorilla mux
- Containerization: Docker
- Monitoring: Prometheus

PRASHANT AGARWAL 191451

ORIGINALITY REPORT

1 %

SIMILARITY INDEX

1 %

INTERNET SOURCES

0 %

PUBLICATIONS

1 %

STUDENT PAPERS

PRIMARY SOURCES

1 Submitted to Jaypee University of Information Technology <1 %
Student Paper

2 citeseerx.ist.psu.edu <1 %
Internet Source

3 www.nmit.ac.in <1 %
Internet Source

4 projanco.com <1 %
Internet Source

5 erepository.uonbi.ac.ke <1 %
Internet Source

6 researchcommons.waikato.ac.nz <1 %
Internet Source

Exclude quotes On

Exclude matches < 5 words

Exclude bibliography On