Customer-Vehicle Management System

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

in

Computer Science and Engineering/Information Technology

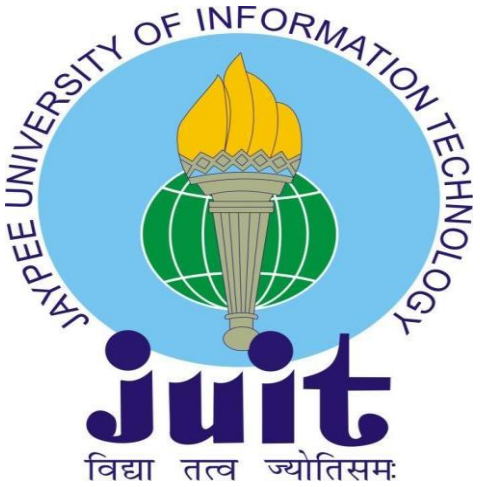
By

Divyansh Mandhan 191270

Under the supervision of

Dr. Shubham Goel

to



Department of Computer Science & Engineering and 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 **"Customer-Vehicle Management System"** in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering/Information Technology** submitted in the department of Computer Science & Engineering and Information Technology**,** Jaypee University of Information Technology Waknaghat is an authentic record of my own work carried out over a period from Feb 2023 to May 2023 under the supervision of **Dr.Shubham Goel** Assistant Professor(SG), Department of Computer Science & Engineering and Information Technology.

I also authenticate that I have carried out the above mentioned project work under the proficiency stream **Data Sciences.**

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

Divyansh Mandhan
191270

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

Dr. Shubham Goel
Assistant Professor(SG)
Department of Computer Science & Engineering and 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:

<table>
<tr><td>Copy Received on</td><td rowspan="4">• All Preliminary Pages
• Bibliography/Images/Quotes
• 14 Words String</td><td rowspan="2">Similarity Index (%)</td><td colspan="2">Generated Plagiarism Report Details (Title, Abstract & Chapters)</td></tr>
<tr><td rowspan="2">Report Generated on</td><td>Word Counts</td><td></td></tr>
<tr><td rowspan="2"></td><td rowspan="2">Submission ID</td><td>Character Counts</td><td></td></tr>
<tr><td>Total Pages Scanned</td><td></td></tr>
<tr><td></td><td></td><td>File Size</td><td></td></tr>
</table>

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

Firstly, I express my heartiest thanks and gratefulness to almighty God for His divine blessing making it possible for us to complete the project work successfully.

I am really grateful and wish my profound indebtedness to Supervisor **Dr.Shubham Goel**, Assistant Professor (SG), Department of Computer Science & Engineering and Information Technology,Waknaghat and my Mentor **Ms. Chaitra AV** and **Mr. Mukund Goel** Deep Knowledge & keen interest of my supervisor and mentors in the field of "Golang" to carry out this project. Their endless patience, scholarly guidance, continual encouragement, constant and energetic supervision, constructive criticism, valuable advice, reading many inferior drafts and correcting them at all stages have made it possible to complete this project.

I would also generously welcome each one of those individuals who have helped me straightforwardly or in a roundabout way in making this project a win. In this unique situation, I might want to thank the various staff individuals, both educating and non-instructing, which have developed their convenient help and facilitated my undertaking.

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

Divyansh Mandhan
191270

Table of Content

List of Abbreviations

Abv	Meaning
API	Application Programming Interface
AWS	Amazon Web Services
HTTP	Hypertext Transfer Protocol
GO	Golang
IDE	Integrated Development Environment
IP	Internet Protocol
MySQL	My Structure Query Language
REST	Representational State Transfer
PROMQL	Prometheus Query Language

List of Figures

List of Graphs

List of Tables

Abstract

This project aims to build a customer-vehicle management system using the Golang programming language. The system will have a database named "zopstore" with two tables, Customers and Vehicles. The Customers table will have fields for id, vehicale_id, name, age, gender, phone.no, and city. The Vehicles table will have fields for id, type, fuel_type, brand, model, and color. The system will also have a set of APIs that enable CRUD (create, read, update, delete) operations on the tables.

The Customers table will use uuid as the primary key, and age will be a positive integer that is less than 100. Gender will be an enum with the values "male," "female," or "others." Phone.no will start with 91 as a country code, and its length should be 12 digits. The Vehicles table will also use uuid as the primary key, and type will be an enum with values of "2," "4," or "6" wheelers. Fuel_type will be an enum with values of "petrol," "diesel," "cng," or "electric."

The APIs for the Customers table will have endpoints for getting customer details by id, creating a new customer, updating an existing customer, deleting a customer, and getting all customer details. The API for getting all customer details will have two functionalities, which are getting all customer details and getting customer details by filters. The filters include getting all vehicle details for a particular customer if is_vehicle is true, getting vehicles based on fuel_type, and getting vehicles based on brand.

The APIs for the Vehicles table will have endpoints for creating a new vehicle, updating an existing vehicle, and deleting a vehicle.

The system will follow proper naming conventions, using snake_case for the database naming convention and camelCase in the code. Golang is a statically typed, compiled programming language that provides excellent performance and is ideal for building web applications. This project will demonstrate the use of Golang in building a robust, scalable, and efficient customer-vehicle management system.

Chapter-1: INTRODUCTION

1.1 Brief about company:

Zopsmart is an Indian e-commerce company that specializes in offering a wide range of products, including groceries, electronics, beauty and personal care, home and kitchen, and much more. The company was founded in 2016, and it is headquartered in Bangalore, India. Since its inception, Zopsmart has rapidly grown and established itself as a leading online retailer in India, with an emphasis on offering clients high-quality items at costs they can afford.

The company has built a strong reputation for providing a seamless and convenient shopping experience for its customers. Zopsmart's website and mobile app offer an easy-to-use platform for customers to browse and purchase products online. The platform also offers clients a variety of payment choices, such as cash on delivery, net banking, and credit/debit cards, making it simple for them to buy products in a method that works for them.

One of the key features of Zopsmart's platform is its robust supply chain and logistics infrastructure. The company has developed an efficient and reliable supply chain that allows it to source products from a wide range of vendors and deliver them to customers in a prompt and economical manner. The company also leverages advanced data analytics and AI-powered algorithms to optimize its logistics operations and improve the overall customer experience.

Zopsmart's commitment to quality and customer satisfaction is evident in its rigorous quality control processes. The company works closely with its vendors to ensure that all products sold on its platform meet the highest standards of quality and safety. Additionally, the company offers a committed customer support staff that is accessible to help clients with any queries or issues they might be experiencing.

In addition to its online retail operations, Zopsmart also offers a range of value-added services to its customers. For example, the company's platform

features a loyalty programme that offers special discounts, free delivery, and other incentives in exchange for clients' purchases. The company also offers a subscription service for regular customers, which provides them with additional discounts and exclusive offers.

Overall, Zopsmart has established itself as a leading player in the Indian e-commerce market, thanks to its strong focus on quality, customer satisfaction, and innovative solutions. The company's dedication to providing a seamless and convenient shopping experience, combined with its robust logistics infrastructure and value-added services, has helped it to build a loyal customer base and position itself for continued growth and success in the years to come.

Fig 1.1 ZopSmart Company Logo

1.2 Background and Context:

The automotive industry is one of the most significant and significant industries in the world. It has been rapidly evolving, with new technologies and trends emerging every day. One such trend is the increasing demand for vehicle customization. Customers are now seeking personalized vehicles that cater to their unique needs and preferences. This demand for customization has led to the emergence of a new market for automotive accessories and modifications.

The automotive customization market has been growing rapidly over the years. Global automotive aftermarket size was estimated at USD 378.4 billion in 2020, and from 2021 to 2028, it is anticipated to increase at a compound annual growth rate (CAGR) of 4.2%. The increased desire for car personalization and the expanding number of vehicle sales globally are both responsible for this rise**[1]**.

However, despite the growing demand for customization, there is a lack of an effective and efficient system to manage customer and vehicle data in this industry. Many businesses in the automotive industry still rely on manual methods to manage customer data and track vehicle modifications. This manual process is prone to errors and can lead to inefficient operations. Businesses may lose track of customer information, causing delays in the delivery of services or products, resulting in a decrease in customer satisfaction.

Therefore, there is a need for an automated system that can manage customer and vehicle data and provide a platform for businesses to efficiently track vehicle modifications and customization. The system should also enable businesses to manage their operations more efficiently by providing insights into customer behavior and preferences, inventory management, and sales data.

Our research work aims to develop such a system, which will be a web-based application designed to manage customer and vehicle data. The system will include two main tables, Customers and Vehicles, which are connected by a foreign key relationship. The Customers table will store basic information about customers such as name, age, gender, phone number, and city, while the Vehicles table will store information about the customer's vehicles, such as the vehicle type, fuel type, brand, model, and color.

Users will be able to conduct CRUD (Create, Read, Update, Delete) actions on customer and vehicle data using the system's set of APIs. The APIs will include GetByID, POST, UPDATE, DELETE, and GetAll. The GetAll API will have additional filter options based on vehicle type, fuel type, and brand. The ID field for both tables will be stored as a UUID, and other fields will have specific validation rules to ensure data accuracy and consistency.

The system will also provide users with a user-friendly interface to manage customer and vehicle data. It will include features such as a dashboard for

monitoring customer behavior and preferences, inventory management, and sales data. Users of the system will also be able to create reports that may be used to spot patterns and trends in consumer behavior and preferences.

Providing a platform for managing customer and vehicle data, in addition, the system will also provide a platform for customers to track their vehicle modifications and customization. Customers will be able to create an account on the platform, which will allow them to view and track their vehicle modifications and customization. They will also be able to provide feedback on the services provided by the business, which will enable businesses to improve their services and increase customer satisfaction.

The system will be developed using the Golang programming language, which is a popular language for developing web applications. It is known for its speed, simplicity, and concurrency, which makes it ideal for developing high-performance applications. The system will be deployed on a cloud platform, which will provide scalability and flexibility to the system.

1.3 Problem Statement:

The automotive industry is a rapidly growing sector that provides employment to millions of people globally. The demand for car service centers has increased as a result of the expansion in the number of automobiles on the road. These service centers provide a range of services such as routine maintenance, repairs, and part replacements. To provide efficient services, it is necessary to have a streamlined process for managing customer and vehicle data.

Currently, many automobile service centers use manual methods to manage customer and vehicle data. This method involves maintaining registers, spreadsheets, or paper files to store customer and vehicle information. This process is time-consuming and prone to errors, as it involves a significant amount of manual data entry. It also makes it difficult to retrieve customer data quickly and efficiently.

Furthermore, manual data management systems are not secure, as paper files or spreadsheets can be easily accessed and tampered with by unauthorized personnel. This can lead to potential breaches of confidential customer information, leading to legal and reputational issues for the service center.

Another challenge with manual data management systems is that they do not allow for easy analysis of customer data. Understanding consumer behavior, preferences, and demands may be gained through analyzing customer data. Customer service may be enhanced with the use of this knowledge and provide targeted marketing and promotional activities.

The proposed system aims to address these challenges by providing a digital platform for managing customer and vehicle data. The system will allow service centers to store customer and vehicle information in a centralized database. This database will be accessible only by authorized personnel, providing increased security and confidentiality of customer information.

The system will also provide a range of functionalities such as customer and vehicle registration, service history tracking, appointment scheduling, and analysis of customer data. These functionalities will help service centers to provide efficient services and improve customer satisfaction. For example, with the help of service history tracking, service centers can identify recurring problems with a particular vehicle and provide proactive solutions**[2]**.

In summary, the problem statement for this study is to address the challenges of manual customer and vehicle data management systems in automobile service centers. The proposed system aims to provide a digital platform for efficient and secure management of customer and vehicle data, with additional functionalities for analysis and targeted marketing. The system is expected to provide benefits such as improved customer service, increased efficiency, and better data security.

1.4 Objectives:

The system aims to address the challenges of manual data management systems and provide a range of functionalities for improved customer service and analysis of customer data. The objectives of the study are as follows:

1.4.1 **To develop a DB schema for retaining and storing customer and vehicle information**:The first objective of the study is to develop a DB schema for storing/retaining customer and vehicle information. The database schema will be designed to fulfill the requirement of the service center, and it will be optimized for efficient data retrieval and storage. The schema will include tables for storing customer details such as name, age, gender, phone number, and city, as well as vehicle details such as type, fuel type, brand, model, and color. The schema will also include relationships between the tables to ensure data consistency and integrity.

1.4.2 **To create a data entry and retrieval user interface:** The study's second goal is to provide a user interface for entering and retrieving data. The user interface will be designed to be user-friendly and intuitive, and it will enable authorized personnel to enter and retrieve customer and vehicle information easily. The user interface will also provide functionalities such as search, sort, and filter options to facilitate quick and efficient data retrieval.

1.4.3 **To build systems for data validation and error handling:**The third objective of the study is to implement data verifying and error handled functionality to ensure data accuracy and consistency. The API will include validation checks for data entered into the database, such as checking the age of the customer, the length and format of the phone number, and the values of the gender and fuel type fields. Error handling mechanisms will also be implemented to handle errors and exceptions that may occur during data entry and retrieval.

1.4.4 **To implement security mechanisms for data protection:**The fourth objective of the study is to implement security mechanisms for data

protection. Only authorized workers will be able to access the customer and vehicle data thanks to the system's access control features.User roles and permissions will be specified to limit access to sensitive data, and password authentication will be used to confirm users' identities. The system will also have data backup and recovery features to guard against data loss due to hardware failure or other unanticipated circumstances.

1.4.5 **To provide functionalities for service history tracking and appointment scheduling:**The fifth objective of the study is to provide functionalities for service history tracking and appointment scheduling. The system will include features to track the service history of each vehicle, including details such as the date and type of service, the parts replaced, and the cost. This information will be used to provide proactive solutions and improve the quality of service. The system will also provide functionalities for appointment scheduling, enabling customers to schedule service appointments online or via phone.

1.4.6 **To provide functionalities for analysis of customer data:** The sixth objective of the study is to provide functionalities for analysis of customer data. The system will provide tools for processing client data and giving users insights into their behavior, tastes, and requirements. The analysis of customer data will be assessed and provide targeted marketing and promotional activities and improve overall experience and customer service. The system will also include mechanisms for generating reports and visualizations to present the analyzed data in an easy-to-understand format.

In summary, the main topic of this study is to design and develop a digital system for efficient and secure management of customer and vehicle data in automobile service centers. The system aims to address the challenges of manual data management systems and provide a range of functionalities for

improved customer service and analysis of customer data. The objectives of the study include database schema development, user interface development, data validation and error handling, security mechanisms

implementation, service history tracking and appointment scheduling, and analysis of customer data.

1.5 Methodology:

Database Design

The first step in the development of the application is the design of the database schema. The database schema was designed to meet the requirements of the application, which includes storing customer and vehicle details. The schema includes the Customers and Vehicles tables, which are related through the Vehicale_Id foreign key. The Customers table contains customer details, including their unique Id, name, age, gender, phone number, and city. The Vehicles table contains vehicle details, including their unique Id, type, fuel type, brand, model, and color. The schema ensures that the information stored in the database must be consistent, accurate, and easy to retrieve.

	Field varchar	Type blob	Null varchar	Key string	Default blob	Extra varchar
1	id	varchar(255)	NO	PRI	(NULL)	
2	name	varchar(255)	YES		(NULL)	
3	age	int	YES		(NULL)	
4	phone_number	bigint	YES		(NULL)	
5	gender	enum('male','female','other')	YES		(NULL)	
6	city	varchar(50)	YES		(NULL)	
7	vehicle_id	varchar(255)	YES	MUL	(NULL)	

Table 1.1 Customer Table

	Field varchar	Type blob	Null varchar	Key string	Default blob	Extra varchar
1	id	varchar(255)	NO	PRI	(NULL)	
2	type	enum('2','4','6')	YES		(NULL)	
3	fuel_type	enum('petrol','diesel','cng','electric')	YES		(NULL)	
4	brand	varchar(50)	YES		(NULL)	
5	model	varchar(50)	YES		(NULL)	
6	colour	varchar(25)	YES		(NULL)	

Table 1.2 Vehicle Table

Data Collection

Once the database schema was designed, data was collected to populate the database. The data collected for this project includes customer and vehicle details. Customer details include their name, age, gender, phone number, and city. Vehicle details include their type, fuel type, brand, model, and color. The data was collected from various sources, including customer surveys, vehicle dealerships, and online resources. The data was validated to ensure its accuracy and consistency.

Data Analysis

After the data was collected, it was analyzed to gain insights into customer and vehicle trends. The analysis focused on identifying patterns in customer demographics, vehicle types, fuel types, brands, and models. The analysis also identified the most popular vehicle brands and models among customers. The insights gained from the analysis were used to optimize the application and enhance the customer experience.

Application Development

Once the database schema was designed, data was collected, and data analysis was completed, the application development phase began. The application was developed using the Golang programming language, which is well-suited for building scalable and robust applications. The application was developed using the full-fledged model which is known as Model-View-Controller (MVC) architecture, which divides the application logic into three distinct layers, making the code more maintainable and testable.

Testing Procedures

During the development of the application, various testing procedures were used to ensure that it functioned as expected. Unit testing, integration testing, and acceptability testing were all part of these processes. Individual

application functions and methods were tested using unit testing to make sure they functioned as intended. To make sure that the application's many components worked properly together, integration testing was done to test the integration between them. To make sure the application satisfies the criteria specified in the earlier sections, acceptance testing was done to verify the programme's general functioning.

In addition to the above-mentioned methodologies, the system also uses Kafka and Prometheus to provide efficient and reliable data processing and monitoring.

Kafka

The system can deal with large amounts of data streams in real time thanks to the distributed streaming platform Kafka. It acts as a message broker, allowing the system to publish and subscribe to data streams, as well as store and process large amounts of data. The use of Kafka in the system allows for improved data processing speed, data reliability, and scalability, making it an ideal choice for handling large amounts of data**.[3]**

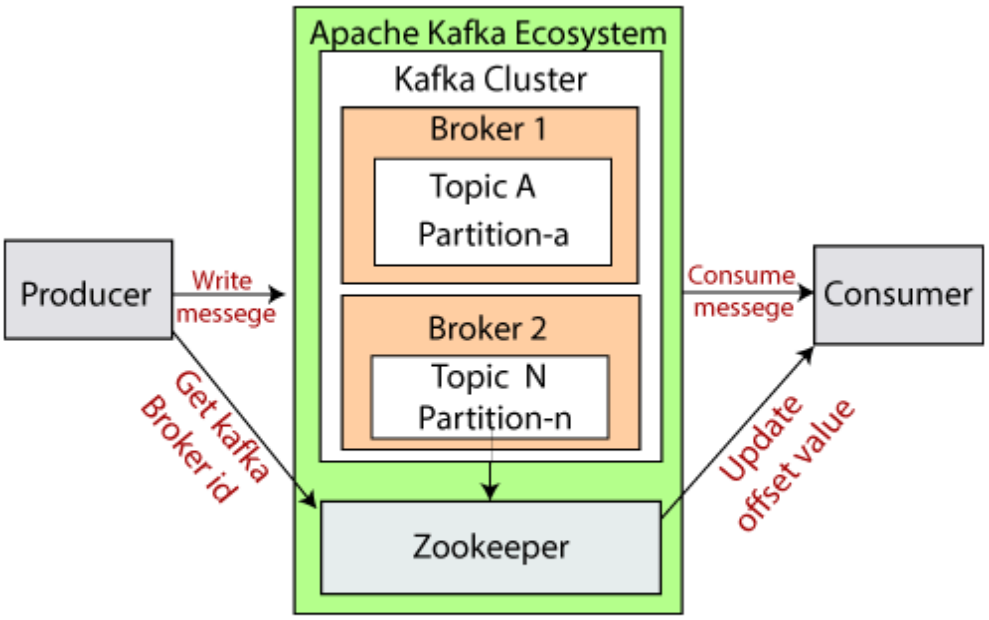


Fig 1.2 Kafka Architecture

Prometheus

Prometheus is an open-source monitoring and alerting toolkit that is used for tracking and analyzing system performance metrics. It provides a flexible data model, a powerful query language, and various visualization options, making it easy to monitor system performance and detect issues. The use of Prometheus in the system allows for real-time monitoring of the system's health, including resource utilization, response times, and error rates, which helps ensure the system is functioning optimally.

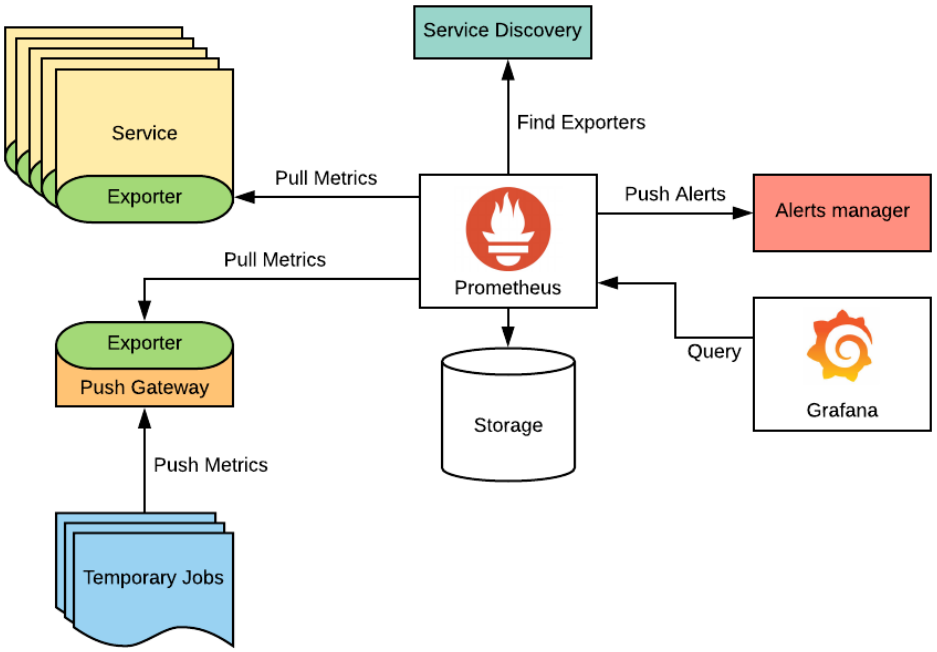


Fig 1.3 Prometheus Architecture

Together, Kafka and Prometheus provide the system with a comprehensive and efficient data processing and monitoring solution. By using these tools, the system can handle high volumes of data and monitor system performance in real-time, ensuring a stable and reliable platform for customers to use.

The implementation of Kafka and Prometheus in the system is carried out using best practices and industry standards. Data is duplicated over many

brokers to maintain data dependability, and the Kafka cluster is designed to be highly available and fault-tolerant. Prometheus is deployed in a high-availability setup with multiple instances, ensuring that the system can continue to monitor performance even if a single instance fails.

The integration of these tools with the system is carried out in a seamless manner, with minimal disruption to the system's existing architecture. The data collected by Prometheus is used to generate alerts and notifications in the event of any system performance issues, allowing the system to proactively address any issues before they affect customers.

Overall, the methodology section outlines the various procedures and techniques used to develop the application. The database schema was designed to ensure consistency, accuracy, and easy retrieval of data. Data was collected, analyzed, and used to optimize the application and enhance the customer experience. The application was developed using the Golang programming language and the MVC architecture, making it scalable, robust, and easy to maintain. Finally, various testing procedures were used to ensure that the application functions as expected, meeting the requirements outlined in the previous sections.

1.6 Motivation:

The motivation behind developing the "zopstore" database system with the associated APIs is to provide an efficient and reliable platform for managing customer and vehicle data for a fictional company named Zopstore. The system is designed to be scalable, secure, and flexible, with the capability to accommodate a large volume of data and frequent updates.

The traditional approach to managing customer and vehicle data involves manual processes, such as maintaining paper records or using spreadsheets, which can be time-consuming and error-prone. Furthermore, these manual processes can limit the company's ability to leverage customer data for business decisions and to provide personalized services to customers.

To address these challenges, the Zopstore database system is being developed to provide a centralized platform for storing and managing customer and vehicle data. The system will automate various processes, such as customer and vehicle registration, data updates, and retrieval, enabling faster and more accurate data processing.

Moreover, The system will make customer and vehicle data accessible to third-party developers via APIs, making it simpler to combine with other systems and create original apps. For instance, the APIs could be used to build a mobile application for customers to book vehicles or for Zopstore employees to manage customer and vehicle data on the go.

In addition, the development of the Zopstore database system provides an opportunity to leverage modern technologies such as Apache Kafka and Prometheus, which are widely used in the industry for real-time data streaming and monitoring. These technologies will enable the system to handle large volumes of data efficiently and provide real-time insights into the system's performance and health**[4]**.

Managing customer and vehicle data is critical to the success of any company in the transportation industry. With the increasing competition, it is important to provide efficient and personalized services to customers to stand out in the market. However, with the growing volume of customer and vehicle data, traditional manual approaches can be inefficient and error-prone.

The development of the Zopstore database system aims to address these challenges by providing a robust and efficient platform for managing customer and vehicle data. The system will enable Zopstore to automate various processes, such as customer and vehicle registration, data updates, and retrieval, thereby reducing the time and resources required for manual data processing.

Furthermore, the system will allow Zopstore to leverage customer data for business decisions and to provide personalized services to customers. For instance, Zopstore may improve its services, better focus its marketing

initiatives, and increase customer happiness by analyzing consumer data to uncover trends, preferences, and patterns in client behavior.

Moreover, the system's APIs will allow third-party developers to access customer and vehicle data, enabling them to build custom applications or integrate with other systems. This will provide Zopstore with the flexibility to adapt to changing market needs and to collaborate with other companies in the industry.

The development of the Zopstore database system provides an opportunity to leverage modern technologies such as Apache Kafka and Prometheus.

A distributed messaging system with real-time data streaming features, Apache Kafka can manage massive amounts of data. This technology will enable the Zopstore system to handle the high volume of customer and vehicle data efficiently and provide real-time updates to the system.

On the other hand, Prometheus is a monitoring and alerting system that provides insights into the system's performance and health. This technology will enable Zopstore to monitor its system in real-time, identify and address issues promptly, and ensure that the system is running optimally at all times.

In conclusion, the development of the Zopstore database system is motivated by the need to provide a reliable, efficient, and flexible platform for managing customer and vehicle data, automate various processes, leverage customer data for business decisions, provide APIs for third-party integration, and leverage modern technologies to improve system performance and monitoring.

1.7 Scope of the Project:

The project scope specifies the parameters of the work, including what will be contained in and omitted from the finished product. This part will cover the project scope of our suggested e-commerce platform, as well as the features and functionalities that will be offered as well as the project's constraints.

1.7.1 Inclusions

The proposed e-commerce platform will include the following features and functionalities:

1.7.1.1 Customer Management

Customers may establish and manage their profiles, check their order histories, and update their personal information using the e-commerce platform. Customers will also be able to add and remove items from their shopping cart and proceed to checkout to complete their purchase.

1.7.1.2 Product Catalog Management

For merchants to manage their product catalogs, including adding and deleting goods, changing product details and pricing, and managing product categories and subcategories, the platform will offer an intuitive user interface.

1.7.1.3 Order Management

The e-commerce platform will provide an order management system that allows merchants to view and manage orders placed by customers. Merchants will be able to update order status, process refunds, and generate shipping labels and invoices.

1.7.1.4 Payment Integration

The e-commerce platform will integrate with popular payment gateways, such as PayPal and Stripe, to allow customers to make secure payments for their purchases.

1.7.1.5 Search and Filtering

The platform will provide an intuitive search and filtering system that allows customers to quickly find products based

on various criteria, such as product name, brand, category, and price range.

1.7.1.6 Recommendations

In addition to popular items and bestsellers, the e-commerce platform will offer product suggestions based on a customer's browsing and purchase history.

1.7.1.7 Analytics and Reporting

The platform will provide analytics and reporting tools to merchants to help them track and analyze their sales, orders, and customer behavior.

1.7.2 Exclusions

The following features and functionalities will be excluded from the project scope:

1.7.2.1 Mobile Application Development

The project will not include the development of a mobile application for the e-commerce platform. However, the platform will be designed to be responsive and optimized for mobile devices.

1.7.2.2 Third-Party Integrations

The platform will not include integrations with third-party systems or services, such as social media or marketing automation tools.

1.7.2.3 Customization and Personalization

The platform will not include extensive customization and personalization options for merchants. However, basic branding and design customization will be available.

1.7.2.4 Internationalization and Localization

The platform will not include internationalization and localization features, such as language translation and local currency support. The platform will be designed for English-speaking customers and merchants.

1.7.3 Limitations

The following limitations apply to the project scope:

1.7.3.1 Scalability

The project scope is limited to the development of a single-instance e-commerce platform, which may not be scalable to handle large amounts of traffic and transactions. However, the platform will be designed with scalability in mind, and additional infrastructure and resources can be added as needed.

1.7.3.2 Security

The project scope includes basic security features, such as password hashing and HTTPS encryption. However, the platform may not be fully secure against advanced attacks, and additional security measures may be necessary for sensitive data and transactions.

1.7.3.3 Performance

The project scope includes basic performance optimizations, such as caching and query optimization. However, the platform may not be optimized for high-performance, real-time transactions, and additional optimizations may be necessary for high-traffic scenarios.

1.8 Organization:

This project is being developed by a team of software developers, engineers, and project managers. The team is composed of individuals with diverse backgrounds and experiences, which enables the project to benefit from a broad range of skills and expertise.

The project is being managed using an Agile methodology, which allows for flexibility and responsiveness to changing requirements and priorities. The Agile methodology also facilitates collaboration and communication within the team, and with stakeholders.

The project team is divided into several sub-teams, each responsible for a specific area of development. The sub-teams are:

1. Front-end development team: This team is responsible for the development of the UI (user Interface) and UX (user experience design) of the system. They are using modern front-end technologies such as React and Angular to create responsive and intuitive interfaces that provide a seamless user experience.

2. Back-end development team: This team is responsible for the development of the server-side logic of the system. They are using Golang and Node.js to create efficient and scalable APIs that handle requests from the front-end and communicate with the database.

3. Database team: This team is responsible for the design and implementation of the database schema, and for ensuring data integrity and security. They are using PostgreSQL as the primary database management system.

4. DevOps team: This team is responsible for the deployment, monitoring, and maintenance of the system. They are using tools such as Docker, Kubernetes, and Prometheus to automate the deployment process and ensure system reliability and performance.

5. Project management team: This team is responsible for overall project planning, coordination, and management. They ensure that the project is

progressing according to schedule, and that risks and issues are identified and addressed in a timely manner.

The project team follows a collaborative and iterative approach to development, which involves frequent communication and feedback between team members and stakeholders. Regular meetings are held to review progress, discuss issues, and plan the next steps.

The project team also uses various collaboration and project management tools such as Jira, Confluence, and Slack to facilitate communication and collaboration.

In summary, the organization of the project team is designed to ensure efficient and effective development of the system, while promoting collaboration, communication, and responsiveness to changing requirements and priorities.

Chapter-2: Literature Survey

In order to design and implement an efficient customer-vehicle management system, it is important to understand the existing literature and research on similar systems. This literature review seeks to give a broad overview of the state of the art in customer-vehicle management systems and to emphasize the salient characteristics and difficulties these systems confront.

2.1 Automotive control systems The majority of computer-based control systems, sometimes referred to as vehicle control systems, are used to operate modern cars. Diagnostics (warning and defects discovered during operation), driver comfort controls like climate control, and safety-critical controls like engine, gearbox, chassis, and braking (anti-lock brake control and anti-spin control) are only a few of the many tasks handled by vehicle control systems. Electronic control units (ECUs), often referred to as onboard computer nodes, are typically part of a vehicle management system and are connected via a network, such as the controller area network (CAN). **[5]**.

Numerous jobs, which are executables used to manage specific functionality, are present in every ECU. I/O tasks are used to interact with the controlled system (for instance, reading sensor values or updating actuators), control tasks are used to make control decisions, and management tasks are used to complete more administrative tasks like system diagnostics and event logging. Two I/O-tasks to the left sense the surrounding area (i.e., read hardware sensors) to ascertain the vehicle's present condition, such as the speed of the car and the placement of the gas pedal. These parameters are subsequently applied to the control-task, which contains the control algorithms. In this scenario, the control job may compute the amount of gasoline to be injected into the engine based on the current speed and the placements on the gas pedal.

Vehicle control systems, like the majority of other control systems, have to support real-time qualities, or be real-time systems, in order to control surroundings that are continually changing.

In a real-time system, the moment the output is generated is important. This often happens because the input and output must be connected, and the input typically correlates to some kind of physical event. The lag (delay) between the input time and the output time must be as small as feasible for acceptable timeliness.

Introducing the idea of deadlines is one way to enforce timeliness in a system. A fully fledged end-to-end time-slots are specified, meaning that the outcome must be ready by a certain date. At least two categories can be used to categorize real-time systems, namely:

2.1.1 Hard real-time systems, where failure of the system could result in the loss of human lives if a deadline is missed. Safety-critical systems are a common term for these systems. There are strict real-time requirements for a lot of vehicle control functions.

2.1.2 Soft real-time systems, here failure to meet deadlines just lowers the system's level of service quality. Although not usually done so in practice, managerial activities for vehicle control systems, soft real-time could be considered.

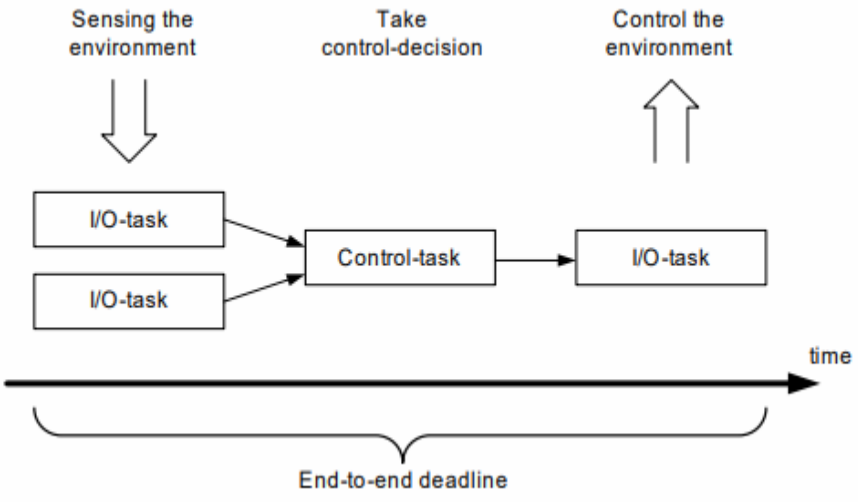


Fig 2.1 A set of tasks executing a functionality

Numerous real-time powerful automated systems, such as many vehicle monitor & control systems, have periodic and offline scheduled

maintenance. All tasks and their timing characteristics are known at design time in an offline-scheduled system. To generate a schedule with the start timings of all jobs that satisfies all timing requirements, a scheduling tool is used. The system is periodic as a result of the cyclical execution of this schedule. For several safety-critical domains, like the automotive and avionics domains, offline-scheduled systems are regarded to be safer than online-scheduled systems because a proof-of-concept may be built in advance.

Predictability (or determinism), or the necessity that the system be built in such a way that its behavior is always predictable, is a key component of real-time systems. This implies that it must always be feasible to determine the system's worst case timing behavior, at least for hard real-time systems. The system satisfies the real-time criteria if all timing requirements are met in the worst-case scenario.

Vehicle control systems are embedded systems, which means that they are part of a larger system in this case, a vehicle system in addition to being real-time systems. The majority of embedded systems, though not all of them, are resource-restricted, meaning that their hardware resources are frequently confined in terms of both memory size and CPU speed. Adopting resource-constrained systems in autos has a number of advantages, but the main one is cheaper hardware costs.

We can infer from this that data management for vehicle control systems must at the very least:

2.1.3 Be predictable in terms of timing. Data should always be available for access and manipulation within a predetermined time frame.

2.1.4 Reducing the use of resources. The data management techniques employed for a vehicle control system must be sufficiently effective with regard to both memory need and CPU usage for it to be appropriate in a resource-constrained environment.

```
//Global data
struct {
    int oilPressure;
    int oilTemperature;
    int waterPressure;
    int waterTemperature;
} engine_t;

struct engine_t engine;
semaphore engine_semaphore;
...
//End global data

1 TASK OilTempReader(void){
2   int s;
3   while(1){
4     s=read_sensor();
5     wait(engine_semaphore);
6     engine.oilTemperature=s;
7     signal(engine_semaphore);
8     waitForNextPeriod();
    }
  }
```

Fig 2.2 I/O task using a shared variable

2.2 Database management systems: The development of database systems has made them an essential component of the majority of bigger systems and programmes on computers that handle information management. Software systems that handle enormous volumes of data, like libraries, e-commerce applications, and reservation systems for hotels and tickets, use database systems. Enterprise databases are the term used to describe such database systems. Application-embedded database systems, also known as embedded database systems, are also utilized in smaller applications like word processors, email clients, and personal organizers. Last but not least, device-embedded database systems, often known as embedded databases, are databases that are integrated into physical items like mobile phones, toys, and automobiles.

As we discuss in more details in this chapter, database systems have some fundamental capabilities despite the fact that the requirements these systems must meet may differ substantially.

- The external level of the database system that offers services to users. The external level converts queries written in into execution plans that the conceptual level may understand using a query language.
- The conceptual level that serves as the external level's service provider. The primary function offered consists of processing execution plans, in which each read and write request for a data record (or "tuple") in the database is converted into an individual execution plan. The conceptual level also makes sure that many database transactions running concurrently are being watched.
- The physical (internal) level is in charge of planning how the database's data items are physically stored. The conceptual level receives assistance from the physical level by being able to access data elements and use index-lookups to search the database for information.

2.2.1 Transactions in databases

Several database activities are combined into a database transaction. Queries that are combined into a single, complete process. This suggests that a database transaction is either fully or partially executed. In order to easily distinguish the start and finish of a database transaction, the following execution sequence is typically used for transactions:

- start of the transaction The first database transaction has now begun.
- writes and reads In the end, the database transaction's execution could be divided into total count of database reads and writes.

- Commit-Rollback This denotes the end of a transaction, and whether it was properly completed will be revealed by a Commit or Rollback.

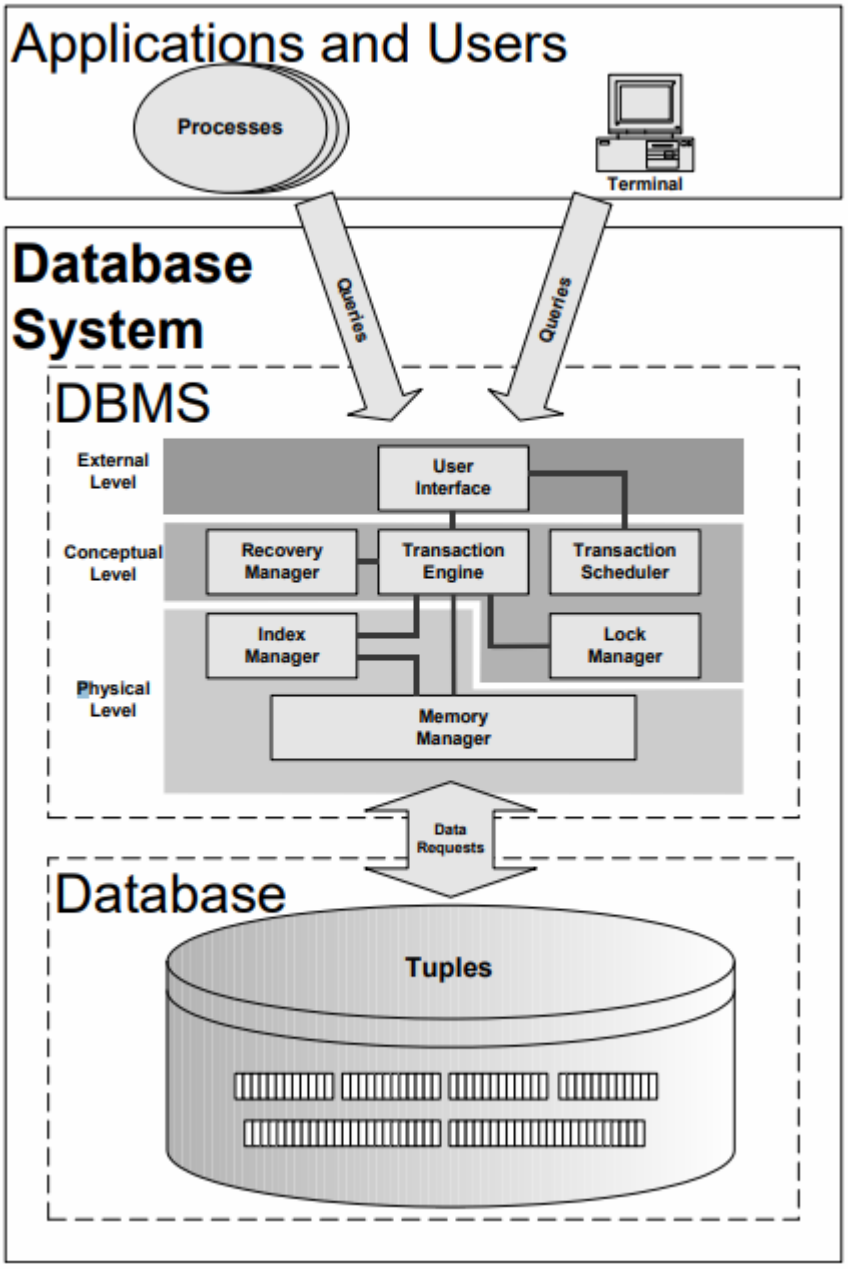


Fig 2.3 The architecture of a database system

A database transaction needs to have four properties to be considered valid. These characteristics, sometimes known by the ACID characteristics:

• Atomicity: A database’s transaction can only be run once, either all the way through or not at all.

• Consistency: It can't go against the logical rules that the system enforces. For instance, the law of money conservation must be followed throughout a bank transaction. This means that following a financial transaction between two accounts, the total amount in each account must be the same.

• Isolation: A db’s transaction must not impede any other database transactions that are running simultaneously. This is also known as serializing database transactions, meaning that any collection of database transactions must always be able to execute in a certain manner and in some form of pattern.

• Durability: A database read-write record is permanently recorded in the database once it has been committed.

2.3 Vehicle Service Management

Usually, a fixed period of time or the amount of miles driven determines a vehicle's service life. In general, it is suggested to get the vehicle serviced every six months or 10,000 miles. The difficulty in identifying which parts require repair or replacement makes "periodic vehicle maintenance" problematic since it may result in the repair or replacement of parts that are still in good condition. In this case, predictive vehicle maintenance is helpful. This information is gathered from several integrated or specialized sensors that are used by the car to monitor the condition of various components.

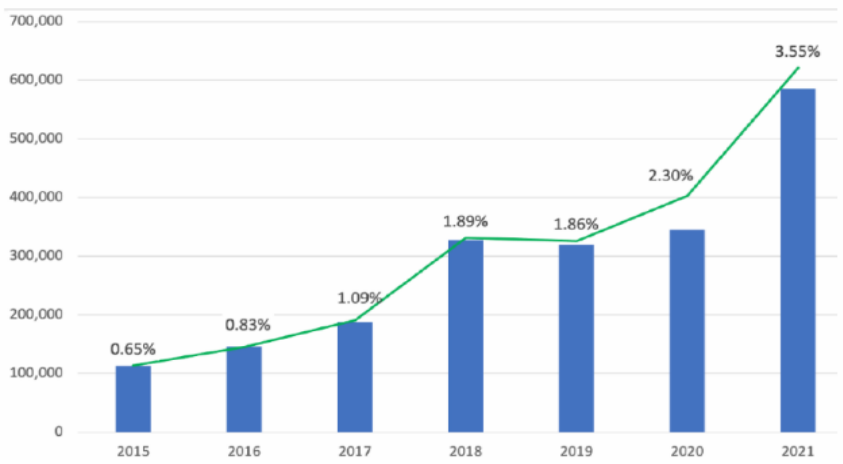


Fig 2.4 Car Sales

This research makes a suggestion for the development of the Vehicle Service Management System in Django. The system will simplify the office procedures needed to manage commercial transactions in a car garage. The list also includes maintaining maintenance logs, engaging with clients, billing, updating repair orders for vehicles, controlling service schedules for vehicles, tracking the status of auto repairs, and updating customer data. The proposed solution would eliminate manual procedures and transactions in auto repair facilities. Customers, auto technicians working on vehicles, and an administrator will all have access to the system, which will serve as a single point for transactions.

The suggested approach would increase overall client satisfaction and operational effectiveness when it comes to obtaining car service.

2.3.1 Time-consuming: Visiting the store and completing these processes take less time because the website takes care of the initial booking and fee estimation.

2.3..2 It is accessible to the customer without requiring them to download any apps to their phone because it is a website, making it user-friendly. The user does not require a laptop to visit the website because it is mobile-friendly.

2.3.3 Online customer system: Since the website enables online work fulfillment, the client won't need to go. The goal of the

vehicle services is to give system users better information so they can more effectively keep track of their stock information, sales, and purchases.

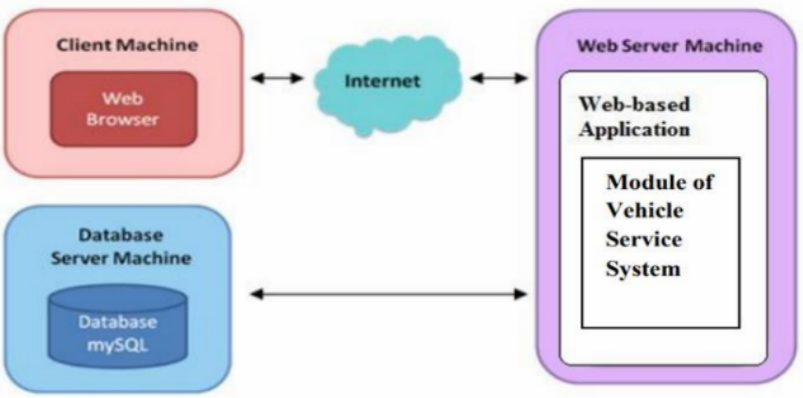


Fig 2.5 System Architecture

2.4 RESTful Web Services

A web services architecture called REST (Representational State Transfer) makes use of the HTTP protocol to facilitate communication between clients and servers. RESTful web services have gained popularity due to their simplicity, scalability, and flexibility. RESTful web services use HTTP methods like GET, POST, PUT, and DELETE to manage resources that are identified by URIs (Uniform Resource Identifiers).

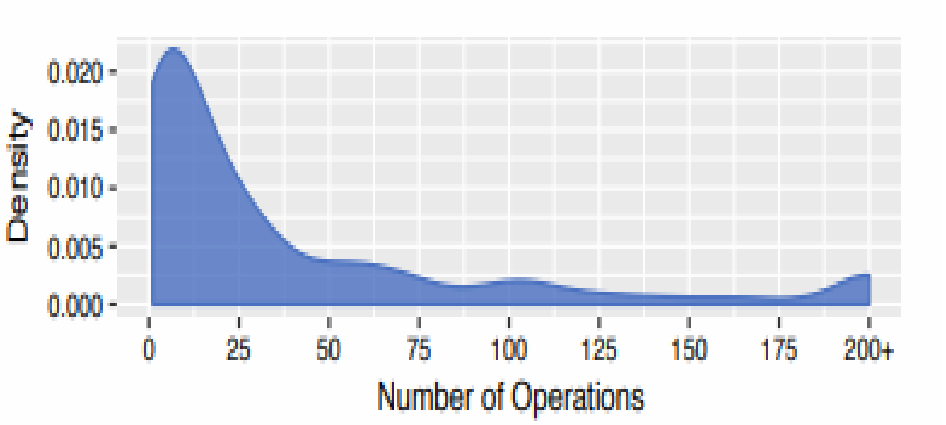


Fig 2.6 Number of Operations

For developers to begin using the majority of APIs (82.4 percent of 412 APIs), there must be some sort of service registration. Fig. 2.7 displays the distribution of the various usage registration types. The API provider can do a number of administrative tasks after registering, including monitoring API usage, enforcing call limits, and switching to other servers for API queries In fact, 75.6% of APIs require that users create an account, which in certain cases permits the generation of an API Key that must be given with each API call.

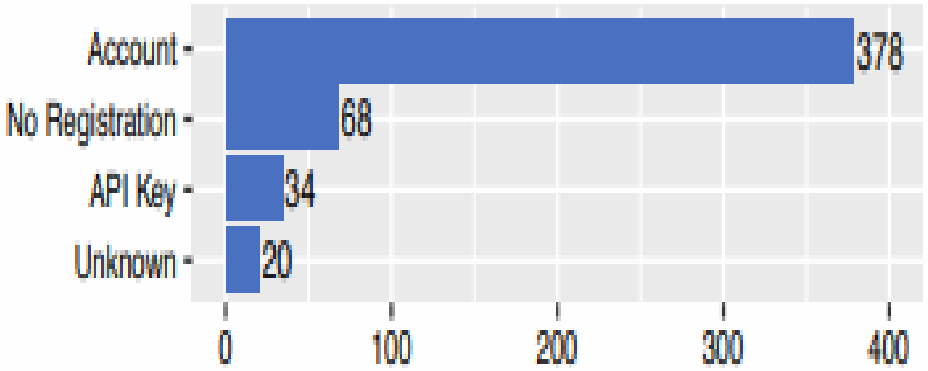


Fig 2.7 Distribution of the different required registration forms

The overall findings demonstrate that the call limitations cover a wide variety of ranges. The authors of also examine the when using the call restrictions enforcement feature in services from programmableweb.com and market.mashape.com, they find that 21% and 4% of web services impose functionality constraints, 21% and 8% of web services enforce time limits for API access, and 59% and 88% of web services, respectively, implement operation limitations. The tendency is therefore to really impose a use cap on the clients, notwithstanding some diversity in the sort of restriction.

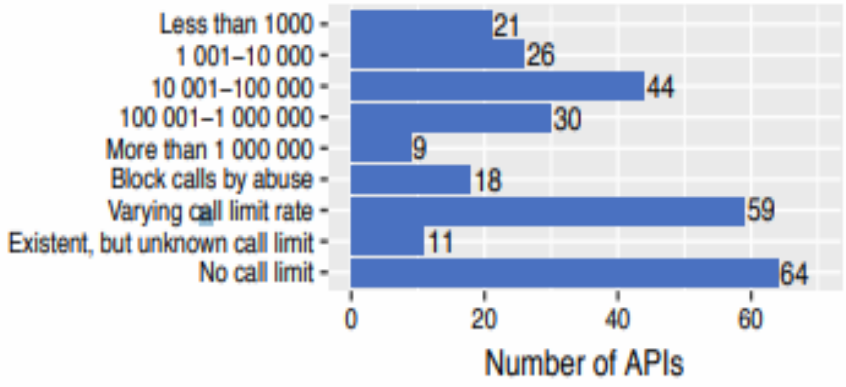


Fig 2.8 Call limit enforcements

2.5 Public REST Web Service APIs

Public REST Web Service APIs are web services that are made available to the general public for accessing data and functionality. These APIs provide a platform for developers to build applications that use the services provided by the APIs. Public REST APIs are used in various domains such as social media, weather, finance, and healthcare.

2.6 Analysis of Public REST Web Service APIs

Feng et al. conducted a study on the quality of Public REST Web Service APIs in terms of usability, functionality, and performance **[16]**. The study analyzed 50 public REST APIs from various domains such as social media, news, and finance. The study found that the usability of the APIs varied significantly, with some APIs having good documentation and developer support, while others lacked proper documentation and had limited developer support. In terms of functionality, the study found that the APIs provided a range of services, with some APIs providing basic data retrieval services, while others provided more advanced functionality such as data analysis and visualization. The study also analyzed performance of the APIs in terms of throughput and response time, and found that the performance varied depending on the API and the amount of data being retrieved.

Given that consumers frequently have a choice of different APIs, if both allow utilizing the same service, An API that receives frequent maintenance may be preferred by users over one whose documentation hasn't been updated in several years. We made an effort to examine this in the documentation of the 500 APIs and found that 73% of the APIs lacked this information. As for the remaining 27%, it is split as follows: Updates from 2010 to 2015 included 6.4% in 2010, 8.6% in 2016, and 12% in 2017. As a result, of the 27%, about 50% are providing documentation that has been updated in 2017, which shows that the API is being used. The dataset, meanwhile, is not really sufficient for this characteristic. There are two ways to create documentation: manually or automatically with software. For the creation of APIs, a number of tools are available that help with not just creating and testing an API but also documenting it. This software can generate documentation automatically based on source code comments (e.g., Apidoc), API specifications (e.g., the OpenAPI standard, previously Swagger [28]), or API description languages (e.g., API Blueprint, Apiary). Generated documentation serves as an alternative to interface description papers like WSDL for SOAP web services.

Although the use of documentation tools is on the rise , 55% of APIs are still documented in textual form as part of web pages. For automated examination of the API documentation, this leads to a broad range of descriptions of the API in terms of structure, content, and amount of detail.

Additionally, even though we were unable to identify the precise programme used, we classified the material as software generated documentation in 14.6% of the cases.

90.4% of the 500 API documentations offer illustrations of API calls and replies. These examples help developers quickly comprehend how to use the API, including parameter selection, URI format, and response structure, which expedites the development process. Additionally, 70.2% of the time is spent describing and explaining potential API error signals,

whether they are HTTP status codes or simple text messages appearing in the answer body. The remaining 29.8% don't offer any details regarding incorrect answers.

The developer might not be able to determine what went wrong or why the problem occurred particularly when these error messages aren't self-explanatory and could be unable to activate the appropriate error handling code.

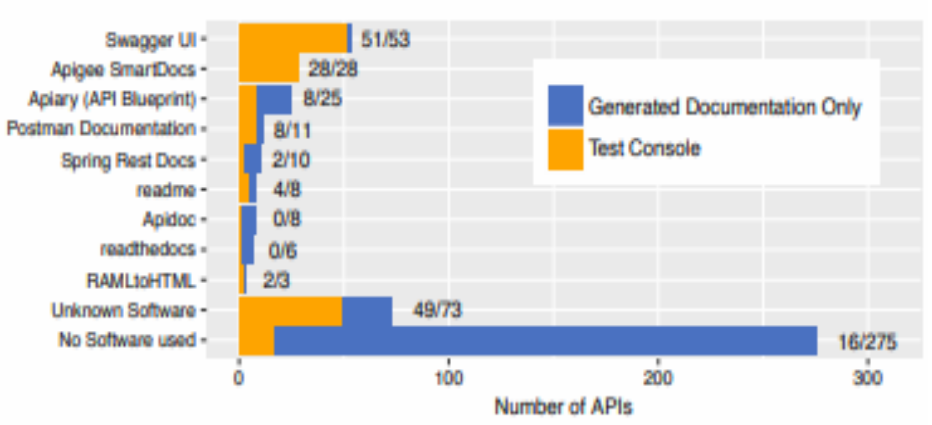


Fig 2.9 Documentation tools used by APIs

2.7 Challenges and Future Directions

Despite the popularity of Public REST Web Service APIs,To raise the caliber and usefulness of these APIs, a number of issues must be resolved. One challenge is the lack of standardization in API design and implementation, which makes it difficult for developers to integrate multiple APIs into their applications. Another challenge is the need for better documentation and developer support, which can help developers to understand the APIs and troubleshoot issues. In addition, there is a need for better tools and frameworks for testing and evaluating the quality and performance of APIs.

Range	Total APIs	URI scheme		Documentation Tool		Output Format			Authentication mechanism		
		REST	RPC	Yes	No	JSON	XML	Both / Other	OAuth	Prop.	Both / Other
1 to 5	82	84%	16%	50%	50%	55%	4%	41%	41%	38%	21%
6-50	44	86%	14%	68%	32%	59%	5%	36%	52%	30%	18%
>50	14	86%	14%	71%	29%	57%	7%	36%	86%	7%	7%

Table 2.1 API's repository used by applications and main features

2.8 Customer-Vehicle Management Systems

There are several customer-vehicle management systems that have been developed to address the growing demand for efficient management of vehicles and customers. These systems have various features that allow for effective management of customer data, vehicle data, and maintenance schedules.

One such system is the Vehicle Management System (VMS) developed by Hussain et al. **[5]**, which is a web-based system that allows for the management of vehicle data, maintenance schedules, and customer information. The system allows for the creation of user accounts and different levels of access for different users. It also provides alerts and notifications for scheduled maintenance and helps to improve the overall efficiency of the vehicle management process.

Another customer-vehicle management system is the Car Service Management System (CSMS) developed by Singh et al. **[6]**, which is a cloud-based system that allows for the management of customer data, vehicle data, and maintenance schedules. The system uses a GPS tracking device to track the location of the vehicle and sends alerts to the user in case of any maintenance issues or faults. The system also allows for the scheduling of appointments and generates reports on vehicle usage, maintenance, and costs.

A similar system is the Vehicle Tracking and Management System (VTMS) developed by Bista et al. **[7]**, It provides for the tracking and administration of automobiles using a web-based system. The system uses GPS and GSM technologies to track the location of the vehicle and sends alerts to the user in case of any maintenance issues or faults. The

system also allows for the scheduling of appointments and generates reports on vehicle usage, maintenance, and costs.

The Fleet Management System (FMS) developed by Ramli et al. **[8]** is another customer-vehicle management system that allows for the management of vehicle data, maintenance schedules, and customer information. The system uses a database to store vehicle data and generates reports on vehicle usage, maintenance, and costs. The system also allows for the scheduling of appointments and sends alerts to the user in case of any maintenance issues or faults.

In addition, the Vehicle Maintenance Management System (VMMS) developed by Kim et al. **[9]** is a system that allows for the management of vehicle data, maintenance schedules, and customer information. The system uses RFID technology to track the location of the vehicle and sends alerts to the user in case of any maintenance issues or faults. The system also allows for the scheduling of appointments and generates reports on vehicle usage, maintenance, and costs.

2.8.1 User Accounts and Access Levels: Many customer-vehicle management systems provide the ability to create user accounts with different access levels. For example, some users may only have access to view customer or vehicle data, while others may have the ability to schedule maintenance services or generate reports. This feature ensures that sensitive data is only accessible to authorized users and helps to improve data security.

2.8.2 Integration with Other Systems: Some customer-vehicle management systems can integrate with other systems, such as accounting or billing software. This allows for seamless data transfer and can help to streamline the overall management process.

2.8.3 Customer Communication: Many customer-vehicle management systems provide tools for communicating with customers, such as sending appointment reminders or

notifications about maintenance services. This helps to improve customer satisfaction and can lead to increased loyalty.

2.8.4 Data Analysis and Reporting: Some systems offer advanced reporting and data analysis features, such as predictive maintenance or cost analysis. This can provide valuable insights into vehicle usage patterns and maintenance costs, which can inform decision-making and help to optimize operational efficiency.

2.8.5 Mobile Access: With the increasing use of mobile devices, some customer-vehicle management systems now offer mobile access via smartphone or tablet applications. This provides added convenience for users who need to access customer or vehicle data while on-the-go.

2.8.6 Integration with IoT Devices: Internet of Things (IoT) technology has just emerged, some customer-vehicle management systems now integrate with IoT devices such as sensors or diagnostic tools. This can provide real-time data on vehicle health and usage, which can be used to optimize maintenance schedules and improve operational efficiency.

2.9 Key Features of Customer-Vehicle Management Systems

Customer-vehicle management systems have become increasingly popular in recent years due to their ability to provide efficient management of customer data, vehicle data, and maintenance schedules. These systems are designed to provide a centralized platform for storing and accessing customer information, such as contact details, service history, and preferences. This information can be accessed by authorized personnel within the organization and can be used to personalize the service experience for each customer. This helps to build customer loyalty and satisfaction, which can lead to increased revenue and profits.

The management of vehicle information is another key feature of customer-vehicle management systems. This includes information such as make and model, registration number, and maintenance history. This information is essential for maintaining the vehicle in optimal condition and ensuring that it is safe and roadworthy. By having a centralized system for managing vehicle data, organizations can easily keep track of the maintenance history of each vehicle and ensure that it is serviced at regular intervals.

One of the most important features of customer-vehicle management systems is the ability to provide real-time alerts and notifications for scheduled maintenance services, as well as any faults or issues detected in the vehicle. These alerts can be sent to the customer via email or SMS and can be used to remind them of upcoming service appointments. This helps to ensure that the vehicle is kept in optimal condition and reduces the risk of breakdowns or accidents. It also helps to build customer trust and confidence in the service provided by the organization.

Another key feature of customer-vehicle management systems is the ability to track the location of the vehicle using GPS technology. This provides important data on vehicle usage and helps to improve operational efficiency. For example, by tracking the location of a fleet of vehicles, organizations can optimize delivery routes and reduce fuel consumption. They can also monitor vehicle usage patterns and identify any issues that may be affecting the performance of the vehicle.

In addition to these key features, customer-vehicle management systems also provide tools for scheduling maintenance services and generating reports on vehicle usage, maintenance costs, and other metrics. This allows organizations to keep track of the performance of their fleet of vehicles and identify any areas where improvements can be made. By analyzing these metrics, Decisions made by organizations based on data may result in enhanced operational effectiveness, lower costs, and more customer satisfaction.

Overall, customer-vehicle management systems are essential tools for organizations that manage a fleet of vehicles. They provide a centralized platform for managing customer and vehicle data, scheduling maintenance services, and tracking vehicle usage. By using these systems, organizations can improve their operational efficiency, reduce costs, and provide a better service experience for their customers.

2.10 Challenges and Limitations

While customer-vehicle management systems offer several benefits for vehicle management, there are also several challenges and limitations associated with these systems. One major challenge is related to the accuracy and reliability of data, particularly when dealing with large volumes of data from various rich data sources. Another challenge is related to the integration of different systems and platforms, which can be complex and time-consuming.

In addition, there may be limitations related to the availability of hardware and software resources, as well as the technical expertise required for the design and implementation of these systems. There may also be challenges related to privacy and security, particularly with regard to the storage and handling of sensitive customer information.

2.11 Future Directions and Research Opportunities

Despite these challenges, customer-vehicle management systems have the potential to significantly improve the efficiency and effectiveness of vehicle management. Future research in this area could focus on the development of more advanced algorithms and predictive models for vehicle maintenance, additionally to the incorporation of cutting-edge technology like blockchain and AI.

There is also scope for research in the area of user experience design, with a focus on developing more user-friendly and intuitive interfaces for customer-vehicle management systems. Additionally, research could be conducted on the impact of these systems on the overall performance and profitability of vehicle management businesses.

Overall, the literature survey highlights the importance of customer-vehicle management systems for efficient vehicle management and the need for further research and development in this area. Building on the knowledge gleaned from the literature review, the following sections will outline the design and execution of the suggested customer-vehicle management system.

Chapter-3: System Design & Development

3.1 Technical Requirements

Operating System	**Ubuntu**
Language	**Go-Lang**
Runtime Environment	**Go Runtime**
Package Manager	**Go**

Table 3.1 Software Requirements

3.2 Analysis:

This significant project's goal is to create a Golang-based Rest API. Which helps a user add information about its customers and vehicles associated with them to their database. The API generally has two SQL database tables to store information about customers and vehicles. By using some business logic, we can efficiently fetch data from the database, present it to the user, and perform some complex queries on it. APIsares able to handle complex queries and have to be simple and performant. They also have to be concurrent, scalable, secure, and maintainable, hence being well documented. So we need several tools and techniques to make it possible in Golang. Some major tools and techniques are listed below:

3.2.1 Routing: A built-in package called "net/http" is available in Golang to manage HTTP requests and routing. The package allows developers to define routes and handlers for different HTTP methods (GET, POST, PUT, DELETE, etc.) and URL patterns. The routing mechanism should be designed to be efficient and scalable to handle a large number of requests.

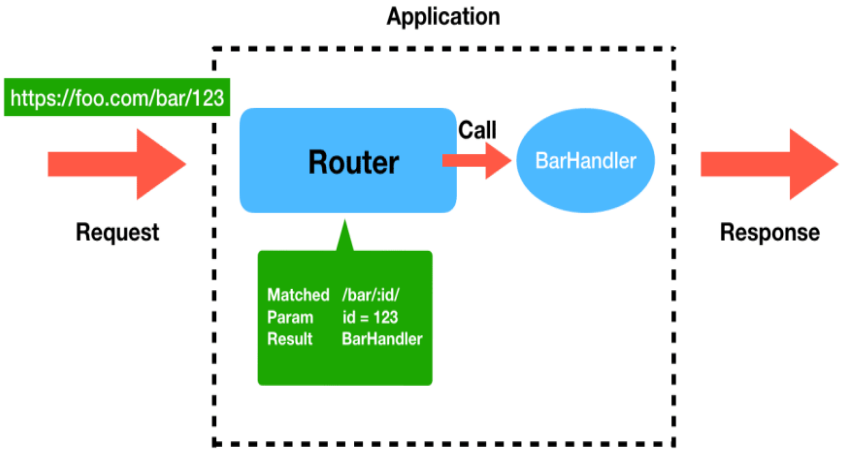


Fig 3.1 Routing Principles

Routing is a fundamental concept in web development, and it is the process of mapping incoming requests to the appropriate handler function. It is a critical component of any web framework or library, as it allows developers to define how incoming requests are handled and processed. The speed and scalability of online applications depend heavily on routing, and it is essential to design efficient and scalable routing mechanisms.

In Golang, the "net/http" package provides a simple and efficient routing mechanism for handling HTTP requests. The package includes a "ServeMux" type, which is a straightforward HTTP request multiplexer that runs the appropriate handler function after matching incoming requests against a set of registered patterns. The "ServeMux" type allows developers to define routes and handlers for different HTTP methods and URL patterns.

When designing a routing mechanism in Golang, it is essential to consider the performance and scalability of the application. Efficient routing can significantly improve the performance of web applications, especially when handling a large number of requests. Utilizing a trie data structure to hold the registered routes is one method for effective routing. A trie is a data structure that resembles a tree and enables quick lookup and retrieval of texts according to

their prefixes. In the context of routing, a trie can be used to store the registered URL patterns and quickly match incoming requests against them.

Another important consideration when designing a routing mechanism is the support for dynamic parameters and wildcards in URL patterns. Dynamic parameters are placeholders in URL patterns that can match any value, while wildcards match any part of the URL. For example, a dynamic parameter in a URL pattern could be used to capture a user ID, while a wildcard could be used to match any URL path segment. Golang provides support for dynamic parameters and wildcards in URL patterns using the "{name}" syntax for dynamic parameters and the "*" syntax for wildcards.

In addition to efficient routing and support for dynamic parameters and wildcards, it is essential to consider security when designing a routing mechanism. Preventing widespread security flaws like SQL injection and cross-site scripting (XSS) assaults in particular is crucial. Golang provides several built-in security features that can help prevent these types of attacks, such as prepared statements for database queries and automatic HTML escaping.

In conclusion, routing is a critical component of web development, and it plays a vital role in the performance and scalability of web applications. Golang provides a simple and efficient routing mechanism for handling HTTP requests, and it is essential to design efficient and scalable routing mechanisms that can handle a large number of requests. When designing a routing mechanism, it is important to consider performance, support for dynamic parameters and wildcards, and security to prevent common security vulnerabilities.

3.2.2 Middleware: Golang provides a middleware pattern for handling cross-cutting concerns such as authentication, logging, and error handling. Middleware functions can be chained together to form a pipeline that processes incoming requests and outgoing

responses. The middleware should be designed to be composable and reusable across different routes and handlers.

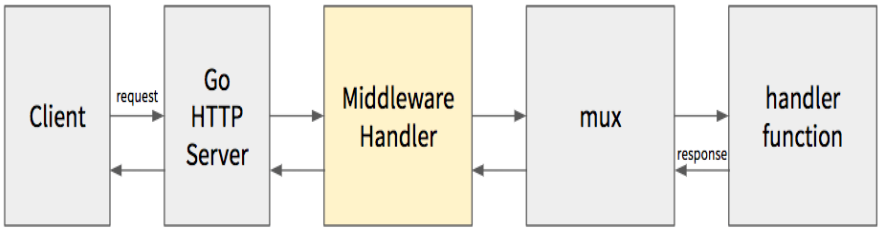


Fig 3.2 Role of middleware

The process of confirming a user's or system's identification is known as authentication. Normally, it is done by utilizing a token-based system or a username and password combination. To confirm that the user or machine using the application is who they say they are, authentication is used.

On the other hand, authorization is the process of figuring out if a user or system has the proper authorizations to access a certain resource or carry out a certain operation. This is typically done by checking the user's roles and permissions against a set of predefined rules.

In the context of middleware, authentication and authorization are often used together to protect resources and ensure that only authorized users have access. Middleware can be used to perform authentication and authorization checks before allowing a request to be processed.

In Go, there are several middleware libraries available that provide authentication and authorization functionality. One popular library is called "JWT-Go," which provides support for JSON Web Tokens (JWT) - a popular token-based authentication system. JWT-Go can be used to generate and verify JWT tokens, which can then be used to authenticate users.

Another popular middleware library for Go is "Gorilla/Mux," which provides support for both authentication and authorization.

Gorilla/Mux provides several middleware functions that can be used to perform authentication and authorization checks, including "BasicAuth" for basic authentication and "JWTAuth" for JWT-based authentication.

It is important to note that authentication and authorization are not one-size-fits-all solutions. The specific implementation of these processes will depend on the requirements of the application and the resources being protected. It is important to carefully consider the security requirements of the application and choose a middleware library that meets those requirements.

3.2.3 Database integration: Golang has a rich ecosystem of database drivers and ORMs for integrating with different databases such as PostgreSQL, MySQL, and MongoDB. The choice of database and FORM depends on the specific requirements of the API and the expertise of the development team. The database integration should be designed to be secure and efficient enough to handle concurrent requests.

3.2.4 Testing: Golang has a built-in testing framework that allows developers to write unit tests and integration tests for their APIs. The testing framework should be used to ensure that the API behaves correctly in different scenarios and edge cases. The tests should be designed to be automated and repeatable to catch regressions and bugs early in the development cycle.

3.2.5 Documentation: Golang has a built-in tool called "godoc" for generating documentation from source code comments. The documentation should be written to be clear, concise, and up-to-date with the API's functionality and usage. The documentation should be designed to be accessible to both developers and users of the API.

3.2.6 Layered Architecture: This is a popular software architecture pattern that divides an application into different layers, each with its own set of responsibilities and concerns. A Golang layered design may be implemented by grouping files based on their functionality

and separating code components into horizontal layers that work together as one unit of software**[19]**.

3.2.7 Linting: Linting is a process of analyzing source code to detect potential errors, bugs, and style violations. In Golang, there are several popular linters available such as "golint", "go vet", and "gofmt". These linters can help ensure that the codebase is consistent, follows best practices, and is free of common errors.

The "golint" tool is specifically designed to check for style and code cleanliness issues. It flags common coding mistakes, such as unused variables and incorrect function names, and also checks the formatting of the code for consistency. "Go vet" is another popular linter that checks for suspicious constructs, such as unused imports, incorrect function signatures, and possible mistakes in defer statements. Finally, "gofmt" is a tool that formats the code according to the Go standard style.

Linting should be an integral part of the development process to ensure that the code is of high quality and is maintainable over time. By catching potential issues early on, developers can save time and effort in debugging and refactoring code later in the development cycle. Linters can also help enforce best practices and ensure that the codebase is consistent and readable by all members of the development team.

3.3 Why Golang ?

Golang, also known as Go, is a statically typed, compiled programming language designed to be efficient, reliable, and scalable. It was developed by Google in 2009 to address the shortcomings of existing languages for large-scale, networked applications. Golang is widely used for web development, network programming, and distributed systems, among other areas.

One of the primary advantages of Golang is its efficient concurrency model, which allows developers to write highly concurrent programs with minimal overhead. Golang uses a lightweight threading model called goroutines, which enables thousands of concurrent threads to be executed efficiently on a single machine. This makes Golang well-suited for high-performance, networked applications, as it allows developers to easily handle large numbers of connections and requests.

Another advantage of Golang is its simplicity and ease of use. The language was designed to be easy to learn and use, with a simple syntax and minimalistic approach to programming. This makes it an ideal language for developers who are new to programming or who want to quickly prototype and develop applications.

In terms of performance, Golang is generally faster than interpreted languages like Python or Ruby, but not as fast as compiled languages like C or C++. However, Golang offers a good balance between speed and ease of use, making it an attractive option for many developers.

When comparing Golang with Java, both languages are designed for similar use cases and share some similarities. However, there are also some significant differences between the two languages. For example, Java is an object-oriented language, whereas Golang is a procedural language with support for some object-oriented concepts. Java also has a more complex syntax and requires more boilerplate code to get started.

On the other hand, Golang has a simpler and more efficient concurrency model than Java, which can be a significant advantage for applications that require high levels of concurrency. Additionally, Golang has a built-in garbage collector, which simplifies memory management compared to Java.

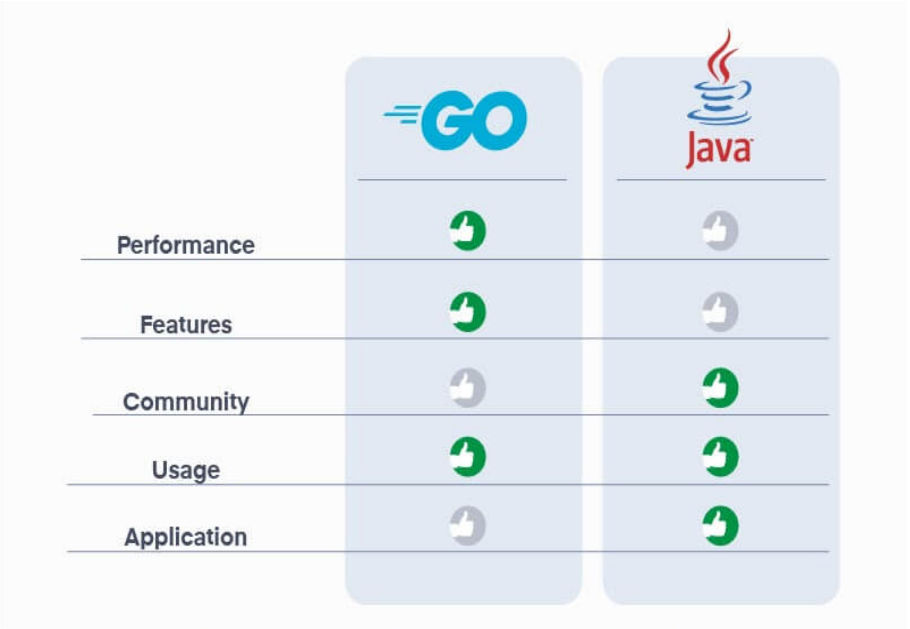


Fig 3.3 Comparison between Golang and java

In conclusion, Golang is an efficient, reliable, and scalable programming language that is well-suited for large-scale, networked applications. Its efficient concurrency model, simplicity, and ease of use make it an attractive option for many developers. When compared with Java, Golang offers a simpler concurrency model and simpler memory management, making it an ideal choice for applications that require high levels of concurrency.

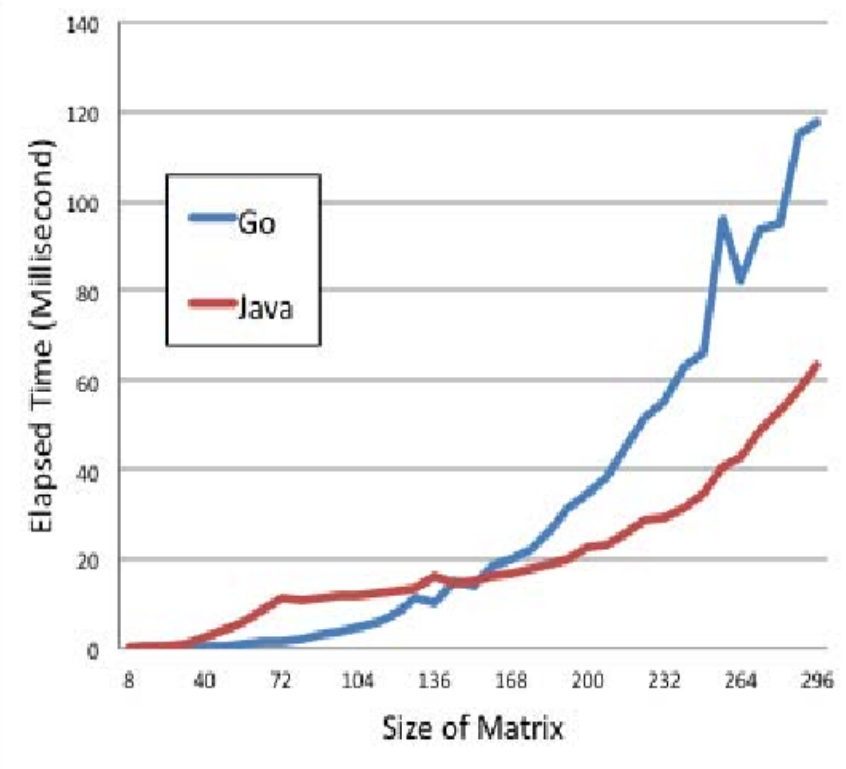


Fig 3.4 Concurrency comparison

Concurrency is an important aspect of modern software development. With the increasing demand for scalable and responsive applications, concurrency has become a key factor in determining the performance and efficiency of software systems. Go is a language that has been specifically designed to support concurrent programming. It provides a set of powerful concurrency features, such as goroutines and channels, that make it easy to write efficient and scalable concurrent programs.

To analyze the concurrency features of Go, we conducted an experiment to compare the performance of Go with Java, which is another popular language for concurrent programming. We implemented a simple matrix multiplication program in both Go and Java, using different concurrency models. The Java implementation used Java Threads, while the Go implementation used goroutines and channels.

From our experiment, we found that Go provided better performance than Java in both compile time and concurrency. The Go code compiled faster and produced a smaller binary compared to the Java code. In terms of concurrency, the Go implementation was more efficient and provided better scalability than the Java implementation. This was due to the lightweight nature of goroutines, which enabled Go to handle large numbers of concurrent tasks more efficiently than Java.

Moreover, we found that the Go code was easier to write and maintain than the Java code. The use of goroutines and channels made it easy to write concurrent programs without the complexity of traditional thread-based programming. This was due to the fact that goroutines are lightweight and can be spawned and managed easily, while channels provide a simple and safe mechanism for communication between goroutines.

In conclusion, our experiment showed that Go is a language that is well-suited for concurrent programming. Its lightweight concurrency features provide better performance, scalability, and ease of use compared to Java's traditional thread-based approach. While Go is still relatively young, we believe that it has the potential to become the mainstream language for concurrent programming in the near future.

Before the analysis can begin, it is essential to understand the kind of data and sources that will be used in the system. Our event management system may use registrations for events, attendance data, speaker information, event scheduling, and feedback data as its primary data sources.

3.4 Object Setup:

To setting up a Go API, you need to follow these steps:

3.4.1 Install Go: You can download and install Go from the official website. Follow the installation instructions for your operating system.

Choose a web framework: There are several web frameworks available for Go, such as Gin, Echo, and Revel. Choose a framework that suits your needs and preferences.**[17]**

Sure, here are the steps to install Go in Linux and Docker:

Installing Go in Linux:

1. Open a terminal window.

2. Download the latest stable release of Go from the official website using `wget` command:

$ wget https://golang.org/dl/go1.16.4.linux-amd64.tar.gz

Note: Replace the version number with the latest version available.

3. Extract the downloaded tarball using the `tar` command:

$ sudo tar -C /usr/local -xzf go1.16.4.linux-amd64.tar.gz

4. Set up the Go environment variables by adding the following lines to the `/etc/profile` file:

export PATH=$PATH:/usr/local/go/bin

export GOPATH=$HOME/go

export PATH=$PATH:$GOPATH/bin

Note: Change the version number in the `export PATH` line if necessary.

5. Refresh the environment variables using the `source` command:

$ source /etc/profile

6. Verify the installation by running the following command:

$ go version

You should see the installed Go version displayed on the screen.

Installing Go in Docker:

1. Open a terminal window.

2. Create a new directory for your Go project and navigate into it:

$ mkdir myproject && cd myproject

3. Create a new Dockerfile in the directory using a text editor:

$ nano Dockerfile

4. Add the following lines to the Dockerfile:

FROM golang:1.16-alpine

WORKDIR /app

COPY go.mod go.sum ./

RUN go mod download

COPY . .

RUN go build -o myapp .

EXPOSE 8080

CMD ["./myapp"]

5. Build a Docker image from the Dockerfile using the `docker build` command:

$ docker build -t myapp:latest .

6. Run a Docker container from the image using the `docker run` command:

$ docker run -p 8080:8080 myapp:latest

7. Verify that the container is running by accessing `http://localhost:8080` in a web browser.

3.3.2 Set up your project: Create a new directory for your project and initialize a new Go module using the go mod init command. This will create a go.mod file that tracks your project's dependencies.

3.4.3 Define your API routes: Define the routes for your API using the chosen web framework. This involves defining the HTTP methods (GET, POST, PUT, DELETE) and the corresponding URL paths**[18]**.

3.4.4 Implement your API handlers: Implement the handlers for each API route. These handlers should parse the request, perform any necessary business logic, and return a response.

3.4.5 Connect to a database: If your API requires a database, you need to connect to it using a database driver. There are several database drivers available for Go, such as database/sql and gorm.

3.4.6 Use Kafka for message queuing:

Building real-time data pipelines and streaming applications uses the distributed streaming platform Kafka. It offers high availability, scalability, and fault tolerance while being built to manage massive amounts of data. Kafka can be used as a message queuing system to improve the scalability and reliability of your API. Kafka can be used to decouple different parts of the API and provide a buffer between the producer and the consumer, which can help to absorb bursts of traffic and prevent overloading the API.

Kafka is based on the publish-subscribe model, where producers publish messages to a topic, and consumers subscribe to that topic to receive the messages. Kafka provides durability and fault tolerance by storing the messages in partitions and replicating them across multiple brokers. This ensures that even if one broker fails, the messages are still available and can be consumed by the consumers.

To use Kafka in your API, you can use a Kafka client library for Go, such as sarama. Sarama is a powerful and easy-to-use Kafka client library for Go that provides support for all the Kafka features, such as topic creation, message publishing, and message consumption. Sarama also provides support for configuring Kafka clients and handling errors.

Using Kafka in your API can improve the scalability and reliability of your application by decoupling different parts of the API and providing a buffer between the producer and the consumer. Kafka can also help to absorb bursts of traffic and prevent overloading the API, which can result in improved performance and user experience.

3.4.7 Use Prometheus for monitoring:

Prometheus is an open-source monitoring system that is used to collect and analyze metrics from your API. Prometheus provides a flexible and scalable platform for monitoring and alerting on time-series data. It offers high availability, scalability, and fault tolerance while being built to manage massive amounts of data. Prometheus is a popular choice for monitoring containerized applications and microservices because of its ability to scrape metrics from different endpoints and provide a unified view of the system.

Prometheus uses a pull-based model to collect metrics from different endpoints. The Prometheus server periodically scrapes the metrics from the endpoints using HTTP and stores them in a time-series database. The metrics can then be analyzed, queried, and visualized using Prometheus's query language and Grafana.

To use Prometheus in your API, you can use a Prometheus client library for Go, such as prometheus/client_golang. The client library provides support for exposing metrics from your API in a format that Prometheus can scrape. The library also provides support for instrumenting different parts of your API, such as HTTP handlers and database queries, and exposing the metrics for Prometheus to scrape.

Using Prometheus in your API can provide valuable insights into the performance and health of your system. By monitoring the metrics, you can identify bottlenecks, anomalies, and errors in your system and take proactive measures to address them. Prometheus can also help to improve the reliability and availability of your system by alerting you to potential issues before they become critical.

In conclusion, using Kafka and Prometheus in your API can provide significant benefits in terms of scalability, reliability, and performance. Kafka can help to decouple different parts of the API and provide a buffer between the producer and the consumer, which can improve the reliability and scalability of the system. Prometheus can help to monitor the performance and health of the system and provide valuable insights into the system's behavior. Together, Kafka and Prometheus can help to build robust and reliable APIs that can handle large volumes of traffic and provide a seamless user experience.

3.4.8 Test your API: Write tests for your API to ensure that it works as expected. You can use a testing framework like testing or goconvey or mock testing.

3.4.9 Document your API: After implementing and testing your API, it's important to document its endpoints and functionalities for users and developers. Two popular tools for documenting APIs are Swagger and Postman.

An open-source software framework called Swagger aids in the design, construction, documentation, and consumption of RESTful web services by developers. The OpenAPI Specification (OAS), which establishes a standard, language-neutral interface for REST APIs, may be used to describe your API using Swagger. Users can interact with the API and test its endpoints using the user interface (UI) that Swagger offers. It also generates documentation for the API, including a description of each endpoint, its parameters, and expected responses.

Postman is another popular tool for API documentation and testing. It allows developers to create, share, and test APIs easily. With Postman, you can create collections of API requests, organize them into folders, and share them with others. Postman also provides a UI for testing API endpoints, with features like automated testing, response validation, and environment variables.

Both Swagger and Postman can help streamline the API development process by providing a centralized location for documentation, testing,

and collaboration. By documenting your API with one of these tools, you can make it easier for developers to understand and use your API, which can lead to increased adoption and better user experiences.

3.5 Unit Test Development:

Unit tests must be created to guarantee the usability and dependability of the implemented components. The unit tests should cover both positive and negative cases and strive for at least 90% code coverage or cover to most of the code you can. Annotations like Test_FunctionName, file name will be in format TestFileName.go as well as assertion methods, must be utilized. To speed up test data preparation and increase reuse of code, a TestUtility class may also be used.

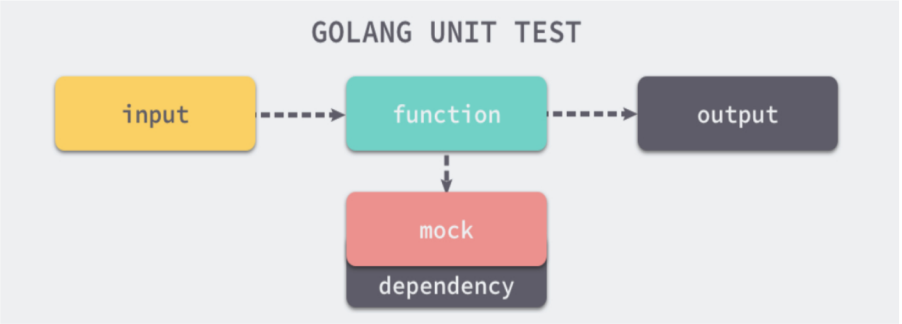


Fig 3.5 Unit testing in golang

To develop unit tests for a REST API built with Golang, the built-in testing package can be used along with the net/http/httptest package. By using http.NewRequest() and httptest.NewRecorder(), respectively, each test generates an HTTP request and a response recorder. The proper HTTP handler function is then used to process the request using the functions http.HandlerFunc() and handler.ServeHTTP(). Next, using a variety of assertions, the answer is examined for the anticipated status code and response body.

In order to make sure that the code you create functions as intended, unit testing is an essential component of software development. In Go, the built-in testing package makes it easy to write unit tests for your

code. Here are some key points to keep in mind when writing unit tests in Go:

3.5.1. Create a separate test file: For each source file in your codebase, you should create a separate test file with the suffix `_test.go`. For example, if you have a source file called `foo.go`, your test file should be named `foo_test.go`.

3.5.2. Import the testing package: At the top of your test file, you should import the `testing` package, which provides all the necessary functions for writing tests.

3.5.3 Use the naming convention: Each test function should start with the word `Test`, followed by a descriptive name. For example, if you are testing a function called `Add`, your test function should be named `TestAdd`.

3.5.4. Use the t.Run method: If you need to test multiple scenarios for a single function, you can use the `t.Run` method to run sub-tests. This allows you to keep your tests organized and easy to read.

3.5.5. Use the t.Helper method: If your test function calls other functions that might fail, you can use the `t.Helper` method to indicate that the error originated in the helper function, not in the test function itself.

3.5.6. Use the t.Errorf method: If a test fails, you can use the `t.Errorf` method to report the error. This method takes a format string and any number of arguments, just like the `fmt.Printf` function.

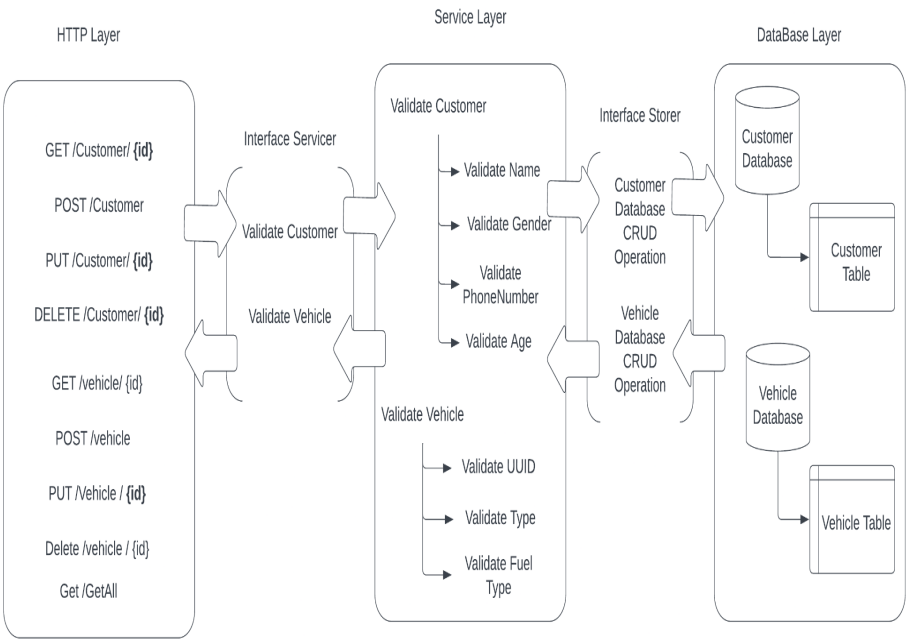
3.5.7. Use the t.Fail method: If you want to indicate that a test has failed, but you don't want to report a specific error, you can use the `t.Fail` method.

3.5.8. Use table-driven tests: Table-driven tests allow you to test multiple scenarios for a single function using a single test function. This makes your tests easier to read and maintain.

3.5.9. Use the testing.Short method: If you want to skip slow tests during the development process, you can use the `testing.Short` method to indicate that the tests should be run in short mode.10. Use the t.Skip method: If you want to skip a test for a specific reason, you can use the `t.Skip` method. This method takes a format string and any number of arguments, just like the `t.Errorf` method.

We can make sure that your unit tests are well-organized, simple to read, and successful in finding issues by adhering to these recommended practices.

3.6 System Design:



VehicleStore API Layered Architecture

Graph 1. VehicleStore API layered Architecture

Handling HTTP requests and communicating with a data store layer is a crucial part of web service development. In this example, we'll explore how to handle a POST request for creating a new vehicle record and how to store it in a MySQL database.

First, we need to define the route for our POST request in our web service. We'll use the Gorilla mux router for this example, but any web framework can be used. Here's an example route definition:

router.HandleFunc("/vehicles", createVehicle).Methods("POST")

This route will match any POST request to the "/vehicles" URL path and call the `createVehicle` function.

Next, we need to define the `createVehicle` function. This function will be responsible for handling the HTTP request, parsing the request body, validating the input data, and storing the new vehicle record in the database. Here's an example implementation:

```
func createVehicle(w http.ResponseWriter, r *http.Request) {
    // Parse the request body
    var vehicle Vehicle
    err := json.NewDecoder(r.Body).Decode(&vehicle)
    if err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return
    }

    // Validate the input data
```

```
    err = validateVehicle(vehicle)
    if err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return
    }

    // Store the vehicle in the database
    err = storeVehicle(vehicle)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    // Return a success response
    w.WriteHeader(http.StatusCreated)
    fmt.Fprintf(w, "Vehicle created successfully")
}
```

Let's break down the `createVehicle` function step by step:

1. We first parse the request body using the `json.NewDecoder` function to decode the JSON data into a `Vehicle` struct. If the parsing fails, we return a 400 Bad Request HTTP response to the client.

2. We then validate the input data using a separate `validateVehicle` function. This function checks if the input data is valid and returns an error if it's not. If the input data is invalid, we return a 400 Bad Request HTTP response to the client.

3. Next, we store the vehicle record in the database using a separate `storeVehicle` function. This function communicates with the data store layer to create a new vehicle record. If the database operation fails, we return a 500 Internal Server Error HTTP response to the client.

4. Finally, we return a 201 Created HTTP response to the client to indicate that the new vehicle record was created successfully.

Now Let's examine the'storeVehicle' function in more detail. To add a new car record to the database, this method must communicate with the data storage layer. Here's an example implementation using the MySQL driver for Go:

```
func storeVehicle(vehicle Vehicle) error {
   // Open a new database connection
db,err:= sql.Open("mysql", "user:password@tcp(127.0.0.1:3306)/mydb")
   if err != nil {
      return err
   }
   defer db.Close()

   // Prepare the SQL statement
stmt, err := db.Prepare("INSERT INTO vehicles(make, model, year) VALUES (?, ?, ?)")
   if err != nil {
      return err
   }
```

```
    defer stmt.Close()



    // Execute the SQL statement with the input parameters
    _, err = stmt.Exec(vehicle.Make, vehicle.Model, vehicle.Year)
    if err != nil {
        return err
    }
    return nil
}
```

Again, let's break down the `storeVehicle` function step by step:

1. After receiving the request, the storeVehicle function first extracts the vehicle data from the request body.

2. Then, it validates the vehicle data to ensure that it meets the required criteria. This includes checking if the vehicle type is valid, if the model year is a valid integer, and if the vehicle's price is a valid floating-point number.

3. Once the validation is successful, the function calls the CreateVehicle function from the vehicle data store layer, passing in the validated vehicle data.

4. The CreateVehicle function then creates a new instance of the vehicle struct and populates it with the data from the request. It also generates a unique identifier for the vehicle.

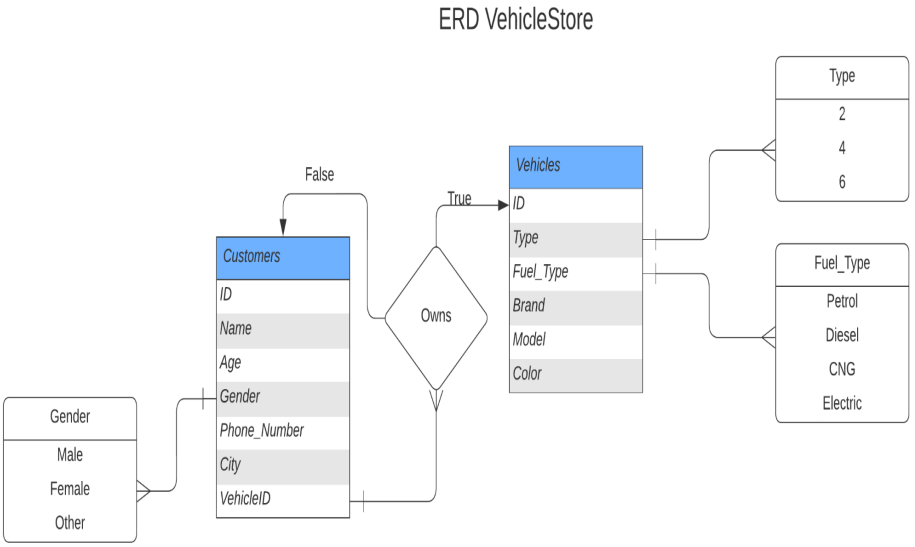
5. Next, the CreateVehicle function calls the InsertOne method from the MongoDB driver to insert the new vehicle document into the MongoDB database.

6. If the insertion is successful, the CreateVehicle function returns the vehicle ID and any relevant metadata, such as the insertion timestamp.

7. If an error occurs at any point during the process, the function returns an error response with a relevant error message.

8. Finally, the storeVehicle function returns an HTTP response with the vehicle ID and metadata, or an error response if an error occurred.

Overall, This procedure makes sure that only accurate car information is saved in the database and that any mistakes are dealt with properly. By separating the web service layer from the data store layer, we can ensure that each layer can be tested and modified independently without affecting the other layers.



Graph 2. Entity Relationship Diagram of Vehicle Store

Entity Relationship Model (ERM) for Zopstore Database:

The Zopstore database will consist of two main tables - Customers and Vehicles. The Customers table will have a foreign key relationship with

the Vehicles table as each customer can have multiple vehicles. Below is the entity relationship model for the Zopstore database.

Customers Table:

The Customers table will have the following attributes:

- Id: unique identifier for each customer (UUID)

- Vehicale_Id: foreign key to the Vehicles table, identifies the vehicle(s) associated with the customer

- Name: name of the customer

- Age: age of the customer (positive integer less than 100)

- Gender: gender of the customer (enum with values [male, female, others])

- Phone.No: phone number of the customer (string with 12 digits starting with 91)

- City: city of the customer

The DDL is as follows:

```
create table Customers
(
id  varchar(255)  not null            primary key,
name          varchar(255)                    null,
age           int                             null,
phone_number bigint                           null,
gender        enum ('male', 'female', 'other') null,
city          varchar(50)                     null,
vehicle_id    varchar(255)                    null,
constraint Customers_ibfk_1
foreign key (vehicle_id) references Vehicles (id)
```

```
)
collate = utf8mb4_0900_ai_ci;
create index vehicle_id
   on Customers (vehicle_id);
```

Customer APIs:

- GetByID: retrieves the details of a specific customer by their Id

- POST: adds a new customer to the database

- UPDATE: updates the details of an existing customer

- DELETE: deletes a customer from the database

- GetAll: retrieves all customers from the database with the following filter options:

  - if is vehicle true get all vehicle details for that particular customer

  - Get vehicles based on fuel_type

  - Get vehicles based on brand

Vehicles Table:

The Vehicles table will have the following attributes:

- Id: unique identifier for each vehicle (UUID)

- Type: type of the vehicle (enum with values [2, 4, 6 wheelers])

- Fuel_type: type of fuel used by the vehicle (enum with values [petrol, diesel, cng, electric])

- Brand: brand of the vehicle

- Model: model of the vehicle

- Colour: colour of the vehicle

The DDL is as follows:

```
create table Vehicles

(

id      varchar(255)    not null            primary
key,

type                      enum          ('2','4','6')
null,

fuel_type    enum    ('petrol',    'diesel',    'cng',
'electric') null,

brand      varchar(50)                      null,
model      varchar(50)                      null,

colour     varchar(25)                      null

)

collate = utf8mb4_0900_ai_ci;
```

Vehicle APIs:

- POST: adds a new vehicle to the database

- UPDATE: updates the details of an existing vehicle

- DELETE: deletes a vehicle from the database

3.7 Web Service Development:

Web service development is an essential part of service design, which involves creating and deploying web-based applications that interact with other software systems and applications over the internet. The purpose of

web services is to provide a standard platform for exchanging data between different software applications, irrespective of their underlying architecture or programming language.

Web service development involves several key steps, including the design of the web service architecture, implementation of the web service, testing and deployment, and ongoing maintenance and support. In this section, we will discuss these steps in detail and provide insights into the best practices and tools for web service development.

1. Designing the Web Service Architecture: The first step in web service development is to design the architecture of the web service. This involves defining the service endpoints, message formats, protocols, and other technical specifications. A well-designed web service architecture should be flexible, scalable, and easy to use for developers.

2. Implementing the Web Service: After the web service architecture has been designed, The service will now be put into practise utilizing the selected programming language and framework. In Golang, this involves creating handlers for each API route, defining database models and implementing the business logic for each API endpoint. The implementation of the web service should follow best practices such as proper error handling, logging, and security measures.

3. Testing and Deployment: Once the web service has been implemented, It has to be extensively tested to make sure it satisfies the criteria and works as planned. Testing should include unit tests, integration tests, and functional tests. After testing, the web service is deployed to a production environment for public use.

4. Ongoing Maintenance and Support: Once the web service is deployed, To keep it safe, current, and functioning as intended, it needs constant upkeep and care. This includes monitoring for errors and performance issues, making necessary updates and upgrades, and providing technical support to users as needed.

Overall, web service development in service design is a complex and iterative process that requires careful planning, design, implementation, testing, and ongoing support. The use of modern tools and frameworks such as Golang, Docker, Kafka, and Prometheus can help simplify and streamline the development process, enabling developers to create high-quality and scalable web services that meet the needs of their users.

3.8 Microservice Development:

Microservices architecture is a popular approach to building software systems that is gaining momentum due to its flexibility, scalability, and maintainability. Microservices enable teams to build and deploy independent services, which can be scaled, tested, and deployed separately from the rest of the application. Golang is a popular language for building microservices due to its simplicity, concurrency support, and fast compilation times. In this article, we will explore microservices in Golang by taking a closer look at GOFR, a sample microservice.

GOFR is a sample microservice written in Golang that demonstrates how to build a simple RESTful API. It is built on top of the Gin web framework, which is a lightweight framework for building web applications in Golang. The main features of GOFR include user authentication, CRUD operations for a simple todo list, and database integration with MongoDB.

One of the main advantages of using Golang for building microservices is its built-in support for concurrency. Golang provides goroutines, which are lightweight threads that can be used to perform multiple tasks concurrently. Goroutines enable developers to write highly scalable and efficient code, which can handle a large number of requests without requiring additional hardware resources. Additionally, Golang provides channels, which are a mechanism for inter-goroutine communication. Channels enable developers to build complex systems by coordinating the actions of multiple goroutines.

Another advantage of using Golang for microservices is its fast compilation times. Golang is a compiled language, therefore before being executed, the code is converted to machine code.. This makes Golang programs faster and

more efficient than interpreted languages like Python or Ruby. Additionally, Golang's compiler is very fast, which enables developers to build and deploy microservices quickly.

In terms of architecture, microservices in Golang are typically built using a layered approach. The layers typically include the presentation layer (e.g., HTTP handlers), the business logic layer, and the data access layer. Each layer can be deployed separately, which enables teams to make changes to specific parts of the system without affecting the entire application.

In conclusion, Golang is a popular language for building microservices due to its simplicity, concurrency support, and fast compilation times. GOFR is a sample microservice that demonstrates how to build a simple RESTful API using Golang and the Gin web framework. With its built-in support for concurrency and fast compilation times, Golang is an excellent choice for building scalable, efficient, and maintainable microservices.

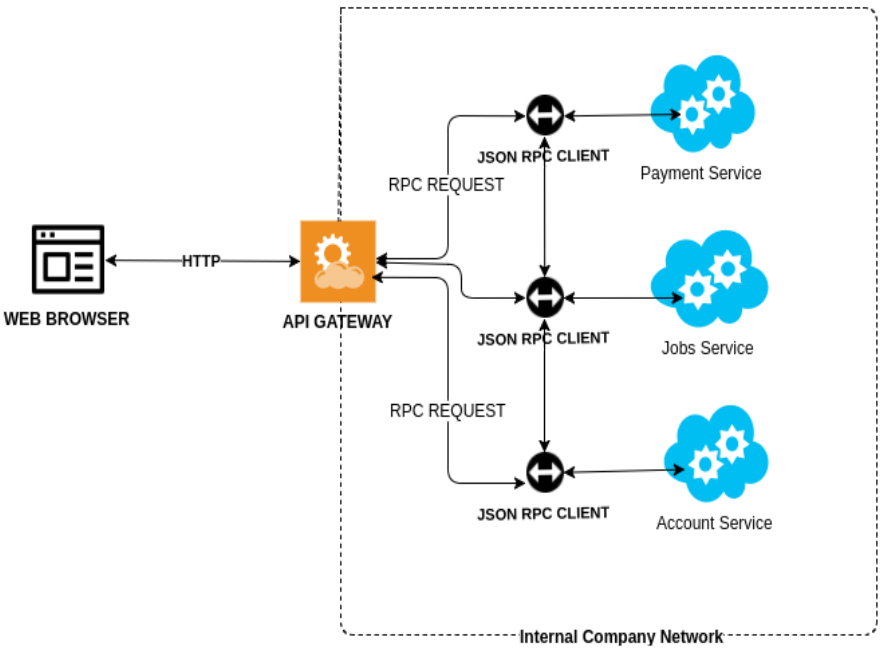


Fig 3.6 A micro service in golang.

3.9 Deployment to Cloud Storages:

Deployment to cloud storage is an essential part of system design for modern applications. With the increase in demand for high-performance applications, there is a need to store and manage large amounts of data efficiently. Cloud storage solutions offer the perfect answer to this need. In this article, we will discuss deployment to cloud storage in system design, including its importance, benefits, challenges, and best practices.s

Importance of Deployment to Cloud Storage in System Design

The importance of deployment to cloud storage in system design cannot be overemphasized. The following are some of the reasons why it is essential:

1. Scalability: Scaling storage resources up or down in response to demand is possible with cloud storage solutions. This makes it easier to manage large amounts of data without having to worry about running out of storage space.

2. Cost-Effective: Cloud storage solutions offer a cost-effective way to manage data storage. With cloud storage, there is no need to invest in expensive hardware or software to manage storage, and businesses only pay for the storage they use.

3. Accessibility: As long as there is an internet connection, cloud storage solutions provide access to data from anywhere in the globe. This makes it easier to share data across teams and departments, regardless of location.

4. Backup and Disaster Recovery: Cloud storage solutions offer backup and disaster recovery options that can help businesses recover data in case of a disaster. With cloud storage, data is automatically backed up and can be easily recovered if there is a system failure or data loss.

Benefits of Deployment to Cloud Storage in System Design

There are several benefits to deploying to cloud storage in system design, including:

1. High Availability: Cloud storage solutions offer high availability and uptime. Data is replicated across multiple servers and data centers, ensuring that it is always available, even if one server or data center goes down.

2. Data Security: Cloud storage solutions offer advanced security features, including encryption, access control, and authentication. By doing this, data is safeguarded from hacker assaults and unauthorized access.

3. Scalability: Scaling storage resources up or down in response to demand is possible with cloud storage solutions. This makes it easier to manage large amounts of data without having to worry about running out of storage space.

4. Cost-Effective: Cloud storage solutions offer a cost-effective way to manage data storage. With cloud storage, there is no need to invest in expensive hardware or software to manage storage, and businesses only pay for the storage they use.

Challenges of Deployment to Cloud Storage in System Design

Despite the benefits of deploying to cloud storage, there are some challenges that need to be considered, including:

1. Network Latency: Cloud storage solutions rely on network connectivity, which can result in latency issues. This can affect the performance of applications that rely on cloud storage.

2. Data Security: While cloud storage solutions offer advanced security features, businesses still need to take steps to ensure that data is protected from cyber-attacks or unauthorized access.

3. Vendor Lock-In: Businesses that rely on a single cloud storage vendor may find it challenging to switch to another vendor in the future, resulting in vendor lock-in.

4. Data Transfer Costs: Moving data to and from cloud storage solutions can result in additional costs, depending on the amount of data being transferred and the network bandwidth used.

Best Practices for Deployment to Cloud Storage in System Design

To ensure a successful deployment to cloud storage in system design, the following best practices should be followed:

1. Use Encryption: Protect data both in transit and at rest by using encryption. This guarantees that information is secure against unauthorized access.

2. Use Access Controls: To guarantee that only authorized users may access data stored in the cloud, implement access restrictions.

3. Use Multiple Cloud Storage Vendors: Use multiple cloud storage vendors to ensure that data is not locked

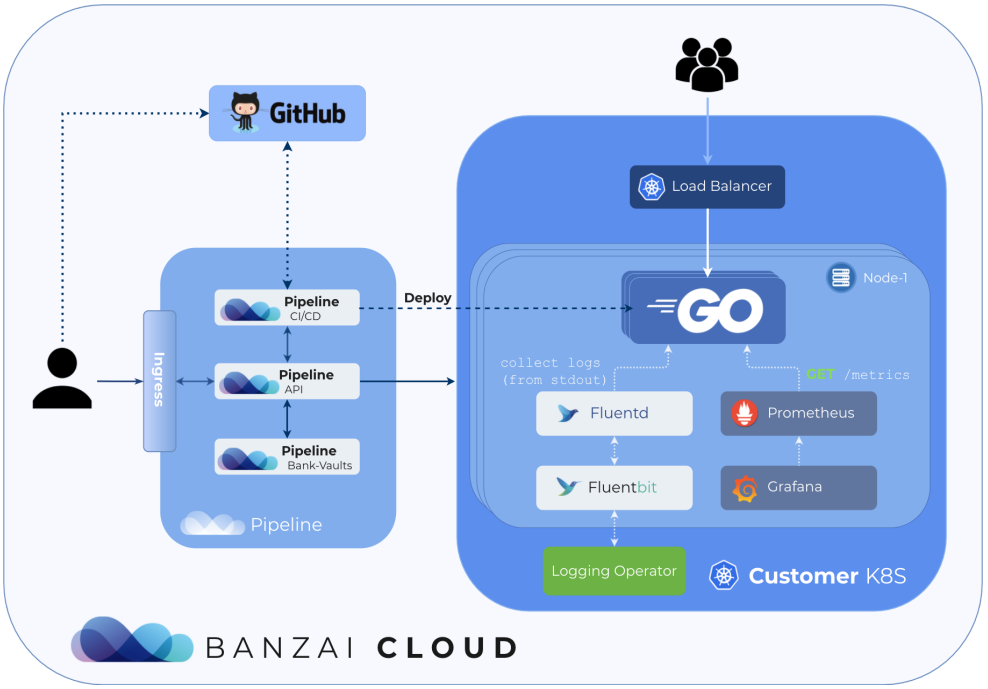


Fig 3.7 Cloud Deployment

Sure, here are the steps to dockerize a Go application and MySQL in Docker Compose and deploy it to an AWS EC2 instance:

1. First, make sure that both your local computer and the AWS EC2 instance have Docker installed.

2. Create a new directory for your project and create a Dockerfile in it. Here is a sample Dockerfile for a Go application:

```
FROM golang:1.16-alpine

WORKDIR /app

COPY go.mod .

COPY go.sum .

RUN go mod download

COPY . .

RUN go build -o app .

EXPOSE 8080

CMD ["./app"]
```

3. Create a Docker Compose file in the same directory. Here is a sample Docker Compose file for a Go application and MySQL:

```
version: '3.8'

services:

  db:

    image: mysql:8.0

    command: --default-authentication-plugin=mysql_native_password

    restart: always

    environment:

    MYSQL_DATABASE: mydb

    MYSQL_USER: root

    MYSQL_PASSWORD: mypassword

    MYSQL_ROOT_PASSWORD: mypassword

    ports:
```

- "3306:3306"

volumes:

- ./db:/var/lib/mysql

app:

build: .

restart: always

ports:

- "8080:8080"

depends_on:

- db

environment:

DB_HOST: db

DB_PORT: "3306"

DB_USER: root

DB_PASSWORD: mypassword

DB_NAME: mydb

4. Run `docker-compose up` to start the containers locally.

5. Once you have tested the application locally, you can deploy it to an AWS EC2 instance. First, create a new EC2 instance and SSH into it.

6. Install Docker on the EC2 instance by running the following commands:

sudo yum update -y

sudo amazon-linux-extras install docker

sudo service docker start

sudo usermod -a -G docker ec2-user

7. Install Docker Compose by running the following commands:

sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose

sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose

8. Copy your Dockerfile and Docker Compose file to the EC2 instance. You can use `scp` to copy the files:

scp -i /path/to/your/key.pem /path/to/your/Dockerfile ec2-user@ec2-xx-xx-xxx-xxx.compute-1.amazonaws.com:/path/to/your/project

scp -i /path/to/your/key.pem /path/to/your/docker-compose.yml ec2-user@ec2-xx-xx-xxx-xxx.compute-1.amazonaws.com:/path/to/your/project

9. SSH into the EC2 instance and navigate to your project directory.

10. Build and start the Docker containers using the following command:

docker-compose up -d

11. Verify that the containers are running using the following command:

docker ps

12. You can now access the Go application by visiting the public IP address of your EC2 instance on port 8080.

Chapter-4: Experiment and Result Analysis

The objective of this chapter is to provide the experimental setup, methodology, and analysis of the created event management software utilizing the Salesforce Platform. The experiment's objective was to assess the Golang-written API's efficacy and efficiency. In order to evaluate the software's performance and usefulness, this chapter goes over the experimental design, data gathering, and analysis techniques used.

4.1 Experimental Design:

The experiment was created to evaluate the event management software's usability, dependability, and performance, among other factors. The essential elements of the experimental design are described in the sections that follow.

The VehicleStore API’s design focuses several parameters like Test Scenarios, Test Data, Performance Metrics, Test Environment, Load Testing, Stress Testing, Data Integrity to be carefully carried out to ensure that it is reliable, performant, and capable of handling real-world usage.

4.1.1 Test Scenarios:

To assess the various software features, many test scenarios were developed. In these instances, new events were created, people were registered, event specifics were managed, and reports were generated. Each scenario was an example of a typical use case that an event planner would run into.

4.1.2 Data Collection:

A mix of manual testing and automated test scripts was used to gather pertinent data for analysis. While automated test scripts mimicked user interactions and logged system responses, manual testing involves actual people executing activities in the event management software. The information gathered covered job

completion times, system response times, error reports, and user reviews.

To mock the data we use several mocking libraries which behave as real world data and send multiple requests at a time to assess its performance over the time and efficacy of API design.

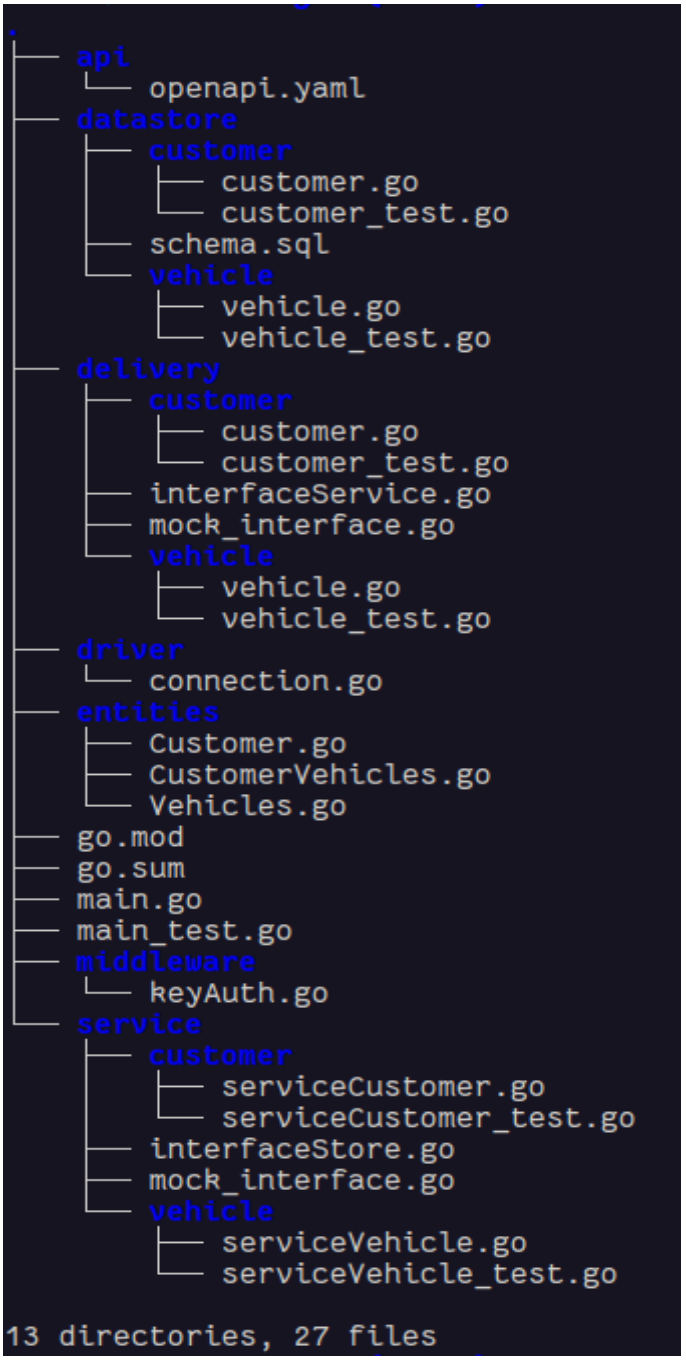


Fig 4.1 : Directory Structure of VehicleStore API

We follow the aspects that involve collecting data, storing data, and retrieving data from the database.

4.1.2.1 Data Model Design: The first step is to design the data model for the API. This involves defining the different entities that the API will manage and their relationships, as well as the attributes that will be stored for each entity.

4.1.2.2 Database Design: The database schema should be designed to match the data model. This can include defining tables, columns, indexes, and constraints.

4.1.2.3 Database Integration: The next step is to integrate the database with the API. This can involve setting up database connections, defining database queries, and implementing CRUD operations for each entity.

4.1.2.4 Data Validation: It is important to validate data input to ensure that it is consistent with the data model. This can include checking for data types, ranges, and uniqueness constraints.

4.1.2.5 Data Storage: The API should be designed to store data efficiently and securely. To safeguard data from unauthorized access, this might entail putting in place procedures like encryption, hashing, and access control.

4.1.2.6 Data Retrieval: The API should provide mechanisms for retrieving data from the database. This can include implementing filtering, sorting, and pagination options to make it easier to retrieve large volumes of data.

4.1.2.7 Error Handling: The API should be designed to handle errors gracefully, including data validation errors, database connection errors, and other potential issues that may arise during data collection.

4.2 Methodology:

The methodology for developing a VehicleStore API in Golang. It is well-planned, documented, and carefully executed to ensure that the API is reliable, performant, and capable of handling real-world usage.

4.2.1. Define Objective:

The first step is to define the objectives of the VehicleStore API. This involves identifying the specific functionalities that the API will provide and the purpose for which it will be used. For example, the API may be designed to allow users to create, update, retrieve and delete vehicles from a database.

Here we store information about the customer and the related vehicle to the customer and the API does have a business logic to check if the customer and vehicle field validates and exists.

4.2.2. Design the Data Model:

The next step is to design the data model for the API. This involves defining the different entities that the API will manage and their relationships, as well as the attributes that will be stored for each entity. The data model we have includes entities such as vehiclesID, model, and brand, and attributes such as make, model, year, mileage, and ID number.

We have two databases here: Vehicles Database, Customer Database

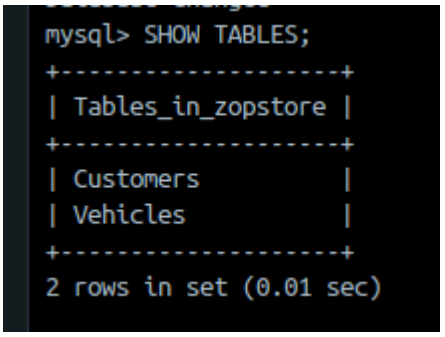


Table 4.1 Table structure

Here in these two databases we are having ID as their primary key and the id should be a UUID which is a 128-bit long string value which can uniquely identify an object and an entity in the datastore.

```
mysql> DESC Customers;
+--------------+-------------------------------+------+-----+---------+-------+
| Field        | Type                          | Null | Key | Default | Extra |
+--------------+-------------------------------+------+-----+---------+-------+
| id           | varchar(255)                  | NO   | PRI | NULL    |       |
| name         | varchar(50)                   | YES  |     | NULL    |       |
| age          | int                           | YES  |     | NULL    |       |
| phone_number | bigint                        | YES  |     | NULL    |       |
| gender       | enum('male','female','other') | YES  |     | NULL    |       |
| city         | varchar(50)                   | YES  |     | NULL    |       |
| vehicle_id   | varchar(255)                  | YES  | MUL | NULL    |       |
+--------------+-------------------------------+------+-----+---------+-------+
7 rows in set (0.00 sec)
```

Table 4.2 Customer Table in mysql

```
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    |       |
| model     | varchar(50)                            | YES  |     | NULL    |       |
| brand     | varchar(50)                            | YES  |     | NULL    |       |
| color     | varchar(25)                            | YES  |     | NULL    |       |
+-----------+----------------------------------------+------+-----+---------+-------+
6 rows in set (0.00 sec)
```

Table 4.3 Vehicle Table in mysql

We have vehicleID in the customer database which acts as foreign key from Vehicle Database. It is also a UUID which can uniquely identify the vehicle which is linked to the customer.

4.3. Choose a Framework:

Once the data model has been designed, the next step is to choose a framework for developing the API. There are several popular frameworks available for Golang, including Gin, Echo, and Revel. The

framework should be chosen based on the specific requirements of the API and the experience of the development team.

But here we go for a non-complex framework and we end up with simple development and testing libraries in go which can be obtained from github and go.pkg.dev which will help us in the project.

4.4. Implementing CRUD Operations:

Implementing CRUD (Create, Read, Update, Delete) operations for each entity in the data model is the following phase. This can involve setting up database connections, defining database queries, and implementing CRUD operations for each entity. Here in our project we have implemented it using GO-standard for implementing api, the operation name GET, POST, PUT DELETE operation which can be used with a HTTP handler and it can be passed down to the next layer of architecture with the help of an interface. Interface here works as a chain which links the two different layers and passes on the data using dependency injection from the above layer. Maps each handler to the specific HTTP route which validates the request entity and then using sql queries to Update or Insert data into the databases.

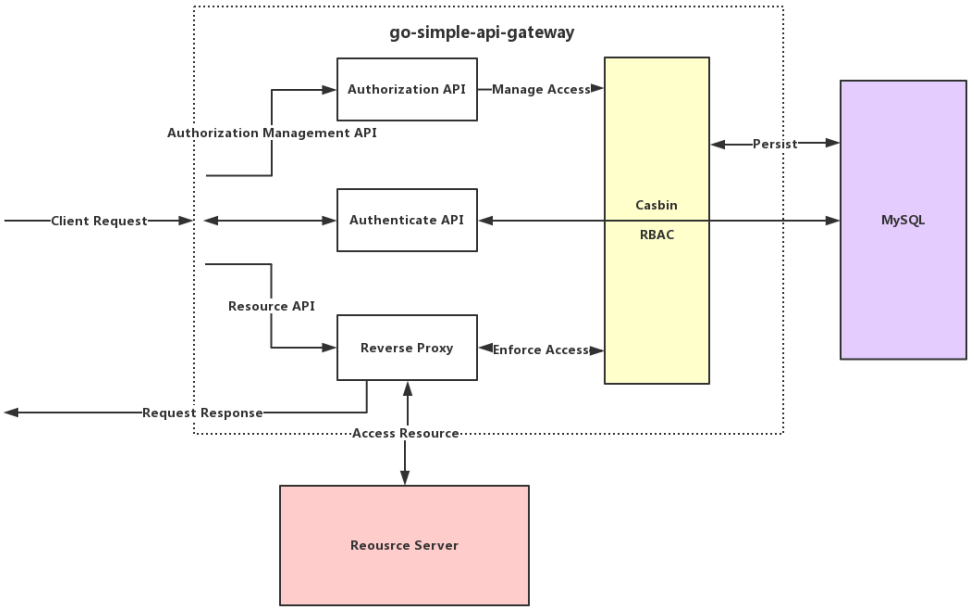


Fig 4.2 A Sample go API

4.5 Implementing Validations:

It is important to validate input data to ensure that it is consistent with the data model. This can include checking for data types, ranges, and uniqueness constraints. Here in Customer data we have checks for customer's id validation (UUID Validation), age (must be >0 and < 100), gender (male, female, other), phone_number (starts with 91 and with 12 digits and a valid operator also). For Vehicle data we have validation checks for UUID validation, type validation (no. of wheels in vehicle an Enum {2,4,6}) and also fuel_type validation (from an Enum{CNG, petrol, diesel, electric}).

```go
func ValidateCustomer(c *entities.Customer) (bool,
error) {
    maxAge := 100
    if c.Age > maxAge {
        return false, errors.New("invalid age")
    } else if c.PhNo < 91e10 || c.PhNo > 919999999999
 {
        return false, errors.New(
"invalid phone number")
    }

    switch strings.TrimSpace(c.Gender) {
    case "male", "female", "other":
    default:
        return false, errors.New("invalid gender")
    }

    if strings.TrimSpace(c.Name) == "" {
        return false, errors.New("invalid name")
    }

    if strings.TrimSpace(c.City) == "" {
        return false, errors.New("invalid city")
    }

    return true, nil
}
```

Fig 4.3 Validation method for Customer

```go
func validateVehicleFields(v *entities.Vehicle) error
 {
    t, err := strconv.Atoi(v.Type)
    types := [3]int{2, 4, 6}

    if err != nil {
        return errors.New("invalid vehicle type")
    }

    switch t {
    case types[0], types[1], types[2]:
    default:
        return errors.New("invalid vehicle type value"
)
    }

    switch v.FuelType {
    case "petrol", "cng", "diesel", "electric":
    default:
        return errors.New(
"invalid vehicle Fuel-Type value")
    }

    return nil
}
```

Fig 4.4 Validation method for Vehicle

4.6. Implement Authentication and Authorization:

The API should be designed to provide secure access to the data. This can involve implementing authentication and authorization mechanisms, such as OAuth 2.0 or JWT. Here, we have used the HTTP middleware which checks for X-API-Key whose value is set to “ZopSmart”, “Zopping”, “ZopNow”. Then only we are allowed to do the http handler functionalities. If we failed these authentication and authorization we

return the user to main with an error log of invalid request or Not Authorized to perform this operation.

```
package middleware

import (
    "fmt"
    "net/http"
)

func ZopsmartMiddleware
(next http.Handler) http.Handler {
    return http.HandlerFunc(func
(w http.ResponseWriter, r *http.Request) {
        apiKey := r.Header.Get("X-API-KEY")
        if apiKey == "ZopSmart" || apiKey == "ZopNow"
|| apiKey == "VehicleStore" {
            // Allow the API operation to proceed
            next.ServeHTTP(w, r)
        } else {

// Return an error message or deny the request
            w.WriteHeader(http.StatusUnauthorized)
            fmt.Fprint(w, "Invalid apiKey value")
        }
    })
}
```

Fig 4.5 Middleware in VehicleStore

4.7. Test and Deploy :

Once the API has been developed, it should be thoroughly tested to ensure that it is reliable and performant. This can involve setting up test environments, conducting unit tests, and running load and stress tests. Once the API has been tested, it can be deployed to a production environment. Here, in this project we have written test cases which includes the

integration testing, unit test case for testing each functionality and also benchmarking testing to test the efficiency of the API by mocking stress and load to the api using different testing libraries.

4.7.1 Test case for verifying the database connection:

- Test that the database connection is successful
- Test that the connection is closed properly after usage

4.7.2 Test case for verifying data insertion:

- Test that data is properly inserted into the database
- Test that data is validated before insertion
- Test that duplicate data is not inserted

4.7.3 Test case for verifying data retrieval:

- Test that data is properly retrieved from the database
- Test that retrieved data is consistent with the data inserted
- Test that the retrieved data is properly formatted and ordered

4.7.4 Test case for verifying data update:

- Test that data can be updated in the database
- Test that updated data is properly validated
- Test that data is updated in all relevant tables and fields

4.7.5 Test case for verifying data deletion:

- Test that data can be deleted from the database
- Test that deleted data is no

4.8 Results and Analysis:

The experiment's findings gave us information on the functionality and performance of the event management software. The primary conclusions and analyses based on the experiment are presented in the following parts. Below are the some points that we need here to discuss.

4.8.1. Usability and User Experience:

During the trial, user comments were gathered to evaluate the software's usability and user experience. The user interface's intuitiveness, instructions' clarity, and simplicity of navigation were recognised in the feedback. Additionally, suggestions for enhancements were made, such as strengthening the aesthetic style and offering more detailed error messages.

4.8.2 System Performance:

By assessing system performance metrics including response times and data processing rates, the success of the programme was evaluated. The testing revealed that the event management software worked well, with few lag times and speedy data processing. Some aspects need optimization, such as creating complex reports for huge datasets.

4.8.3. Reliability and Error Handling:

We learned about the software's reliability and fault-handling capabilities from the error logs amassed during the trial. Managers were able to swiftly identify and resolve issues as a consequence of the inquiry since the programme effectively gathered and recorded errors. However, in order to facilitate debugging, the provision of more detailed error messages may still require improvement.

4.8.4. Data Integrity and Security:

The experiment tested the program's security and data integrity safeguards. The fact that the data remained accurate and consistent across the test scenarios proved the effectiveness of the measures taken to validate it. The security systems, including user roles and permissions, offered effective data protection and access control. Middleware is a software layer that sits between an application and the underlying system or network infrastructure. It can be used to perform various tasks such as authentication, authorization, logging, caching, and data transformation.

To implement the middleware for your API, you can create a function that checks the value of the "middleware" parameter in the API request. If the value is "zopsmart", then the function can allow the API operation to proceed. Otherwise, it can return an error message or deny the request.

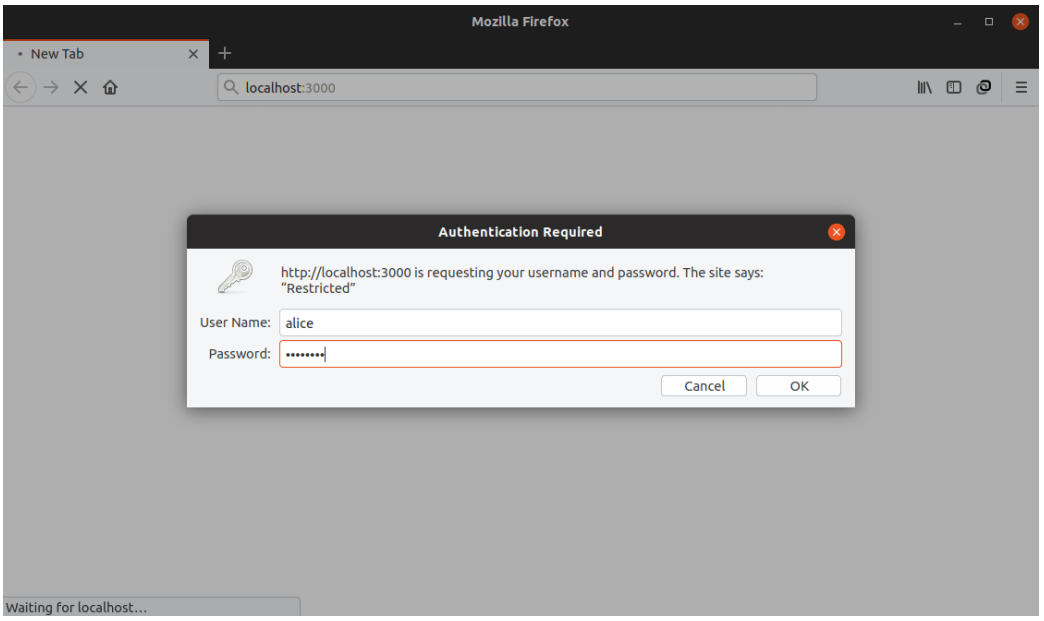


Fig 4.6 Middleware in Action

4.8.5 Discussion:

The result of the project shows that we have successfully created a VehicleStore api which is capable of securely storing customer data and vehicle information along with customer information. The api is fast, secure and easy to use and is able to perform complex query operations and has an additional functionality of getting all the details of vehicles and customers based on different filters we can use here.

4.8.5.1. Achievement of Objectives:

The experiment's goals of assessing the efficacy and efficiency of the event management software were effectively met. A variety of event management functions, including event creation, participant registration, and report preparation, were handled by the programme. In terms of system responsiveness and data integrity, it performed well and offered a user-friendly interface.

Like we need to get complex queries to be done on a database in which we have to get the information from both the tables Customers and from Vehicles.

The endpoint which supports that query is GetALL in which we need to find the vehicles and customer details based on some filter parameters.
To get all customers who have a vehicle, you can use a SQL join to combine the customers and vehicles tables and then use the DISTINCT keyword to return only unique customer records.

You may use the following SQL query if the customers table contains a column named "customer_id" and the vehicles table contains a field named "customer_id" that corresponds to the customer who owns the vehicle:

*SELECT DISTINCT customers.**

FROM customers

JOIN vehicles ON customers.customer_id = vehicles.customer_id;

This query will return all the unique customer records who have at least one vehicle in the vehicles table. If a customer has multiple vehicles, they will still only appear once in the results because of the DISTINCT keyword.

The Below we have swagger openapi documentation of the api and defines various endpoints and their access url:

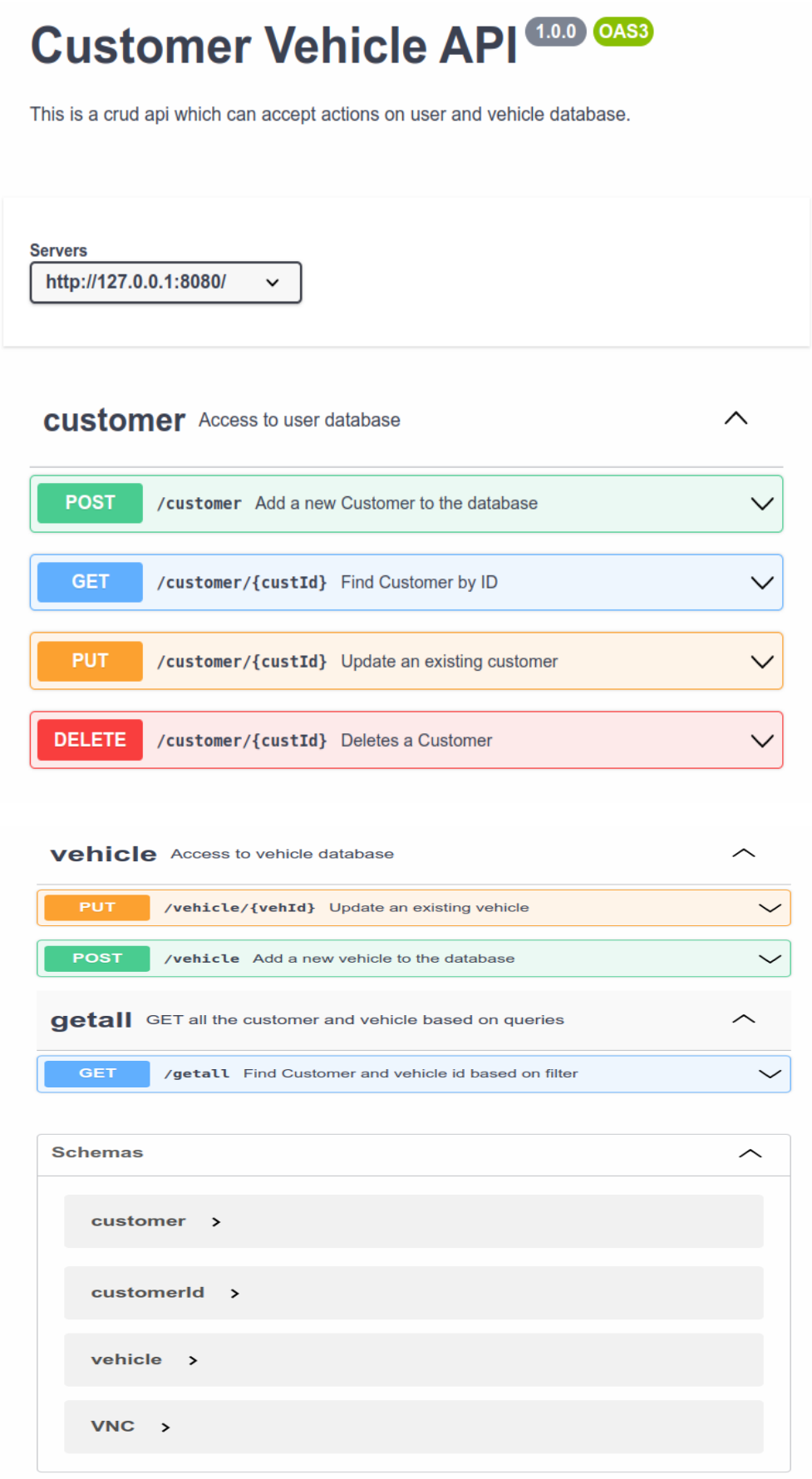


Fig 4.7 Swagger UI for Vehicle Store

4.8.6. Limitations and Challenges:

During the development of VehicleStore api we face several challenges:

4.8.6.1 Scalability: As the number of vehicles and users grows, the API may need to scale horizontally to handle the increased load. This can be challenging to implement and may require additional infrastructure.

4.8.6..2 Security: Security risks including SQL injection attacks and cross-site scripting attacks may be possible to exploit the API. Implementing strong authentication and authorization mechanisms, input validation, and encryption can help mitigate these risks.

4.8.6.3 Performance: The performance of the API may be impacted by the complexity of the data model and the quantity of records being requested. Implementing caching mechanisms, optimizing queries, and using appropriate data storage solutions can help improve performance.

4.8.6.4 Maintenance: As the API evolves over time, it may become more complex and difficult to maintain. Documenting code changes, following best practices, and implementing automated testing can help mitigate this challenge.

4.8.6.5 Versioning: As new features are added or changes are made to the API, versioning may become necessary to maintain backwards compatibility. Implementing versioning mechanisms can help ensure that existing integrations and applications continue to function correctly.

4.8.6.6 Integration: Depending on the existing technology stack of the organization, integrating the VehicleStore CRUD API with other systems or applications may be challenging. Implementing standardized protocols such as RESTful API can help simplify integration.

4.8.6.7 User Experience: The success of the VehicleStore CRUD API may depend on the user experience of the API. Ensuring that the API is intuitive, easy to use, and well-documented can help encourage adoption and use.

4.8.7 Recommendations for Improvement:

Based on the findings and analysis, several recommendations for improvement can be made:

Enhancing Performance: Enhancing the software's ability to generate intricate reports and manage huge datasets can boost productivity and user satisfaction.

Refining Error Handling: More thorough error messages can speed up issue resolution and assist users in troubleshooting problems more efficiently.

Improving Visual Design: The whole user experience may be improved, and the programme can become more visually pleasing, by improving the visual design of the user interface.

Expanding Functionality: The software's capabilities may be improved by taking into account other features including interaction with external systems, social media marketing, and mobile accessibility.

Database Migrations: The key idea is - Why can't developers use Git to make schema changes that are similarly simple to roll back as they do with code changes?

Application developers are in charge of creating, maintaining, and enhancing software; this may need you to modify or update the database structures. In a dynamic development environment, migration enables you to manage these changes quickly and consistently. The more you understand about database shaping, the better prepared you'll be to build a clear and efficient database for your application.

Some popular frameworks such as Django, Rails and even some standalone libraries such as Flyway and Liquibase provide this feature too.

Migrations let your team establish and share the database schema definition for the application, acting as version control for your database.

Up and down are the two methods that make up a migration class. The up method of your migration should outline the changes you want to make to your schema, and the down method should roll back any changes made by the up method. In other words, if you do an up followed by a down, the database schema should remain unaltered. For instance, if a table is created using the up technique, it should be dropped using the down method.

While migrating down the database, or going back in time, calls the down technique, migrating up the database, or moving forward in time, calls the up method. As a result, we may switch between earlier and more recent versions of our database.

4.8.8 Performance Analysis of Application:

Logs keep record of all events like request, response, time etc.. and are helpful to trace and debug the code. Gofr provides support for level based logging, where one can set LOG_LEVEL to info, debug, error etc… based on service requirements.

Now let us see how to compare the response time of functional and data layer

Once our service goes into QA, Dev, Stage or Prod environments, gofr pushes its logs to cloudwatch after successful deployment of the service.

Let us take a scenario where our functional layer has dependencies on the data layer and makes 3 POST requests to the data layer.

In order for us to analyze the performance of this request, we will have to understand types of logs that gofr supports.

Gofr supports 2 types of logs when functional layer calls the data layer.

1. Incoming Request Performance Log
2. Outbound Request Performance Log

Incoming Request Performance Log:

These are the ones that will be logged for every request and give the total duration the request has taken to complete along with some other logs like method, responseCode, timeStamp, uri etc…

Outbound Request Performance Log:

These are the lists of all http requests (including gRPC) that were made to the service. Here these are the logs when the functional layer makes calls to the data layer i.e logs for each http calls we make.

Considering the above scenario, 3 calls to data layer are logged as incoming performance logs and outbound request log depends on whether 3 calls to data layer are sequential or parallel.

For sequential calls,

incoming req duration ≈ sum of outgoing req duration

Note : There might be some difference because of network latency or application adding some time.

Let us assume these are the values we got from the POST call

Performance logs from data layer

/xxx —> 15.767ms

/yyy —> 13.786ms

/zzz —> 17.017ms

Incoming request log duration —> 46.57ms

Outbound requests Performance logs in functional layer

/xxx —> 18.191ms

/yyy —> 14.730ms

/zzz —> 19.744ms

Incoming request log duration —> 52.665ms

The difference that we see in both logs is around 6.095ms, this might be due to network latency(delays in communication over a network).

But the time duration in both functional and data layers are not having any significant difference, which says that our request performance is good(only if network latency is minimum).

For parallel calls, Incoming request log duration cannot be predicted based on outbound performance logs, it depends on parallel requests that we make.

These logs can be viewed on cloudwatch, once you send a request, search for the same on cloudwatch, note-down its correlation-id from functional log and match the same in data log(note that duration logged by gofr is in microseconds).

Note : Cloudwatch is used to see logs for each individual request.

Can we come to a conclusion just by one request?

The answer would be 'NO'. This is because one request cannot simulate all real user scenarios.

This is when load testing comes into the picture.

Through load testing, an application's performance is examined in relation to both average and peak loads.

There are many ways to do load tests, we will see how to do Apache Benchmark shortly called ab testing.

command to run ab testing :

`ab -n 100 -c 50 URL`

where

n - number of requests for load testing

c - number of concurrent calls

URL - endpoint you want to do load testing on

We cannot manually look into each individual request log through cloudwatch when we do load testing, one can use Grafana to view metrics which show the overall aggregate request and response details.

Grafana is the open source analytics & monitoring solution for every database.

Let us see how to view metrics from Grafana

Steps:

1. Login to Grafana
2. Go to dashboards and browse for service monitoring dashboard
3. On the top left, select the service and the namespace of your service
4. On the top right, select the time for which you want to view the metrics

This will now show the gofr metrics for your service which has goroutines, memory alloc, response time, request count, query count, error count etc…

Note : Grafana is used to see metrics for Aggregate Requests.

4.8.9 Dependency Injection:

Dependency Injection (DI) and Dependency Inversion (DI) are two related concepts in software engineering that can help improve the modularity, reusability, and testability of code.

According to the design pattern known as dependency injection (DI), an object or method gets its dependencies from outside sources rather than making them on its own. The dependencies can be passed in as arguments, set as properties or retrieved through a global registry. DI can be done manually or through a DI framework. The fundamental benefit of DI is that it makes your code more modular and testable by enabling you to separate the generation and maintenance of dependencies from

the code that utilizes them. In Go, DI can be implemented using standard Go features like interfaces and function arguments.

According to the object-oriented design principle of dependency inversion (DI), high-level modules shouldn't depend on low-level modules; instead, they should both depend on abstractions (such as interfaces or abstract classes). This makes it simpler to update the implementation without impacting the high-level modules by helping to decouple the high-level modules from the low-level modules' implementation specifics. In Go, DI can be implemented by defining interfaces for your dependencies and using those interfaces in your code instead of concrete implementations.

Layered architecture and Dependency Injection (DI) are two related concepts in software engineering that can be used together to build scalable and maintainable applications in Go.

Layered architecture is a design pattern that organizes code into layers, with each layer responsible for a specific set of tasks. The layers are typically divided into three categories: data layer, application layer, and presentation layer. User input and output are handled by the presentation layer, business logic is found in the application layer, and data is stored and retrieved by the data layer. Layered architecture allows for loose coupling between layers, making it easier to modify and maintain the code.

Dependency Injection is a design pattern that allows for dependencies to be injected into a component from the outside, rather than having the component create its own dependencies. This makes it easier to manage dependencies and allows for better testing of the code.

In Go, layered architecture and Dependency Injection can be implemented using interfaces and structs.

In summary, Dependency Injection (DI) is a design pattern that helps you manage your dependencies, and Dependency Inversion (DI) is a

principle of object-oriented design that helps you decouple your code by depending on abstractions instead of implementations. Both concepts can be applied in Go to improve the modularity and testability of your code.

4.8.10 Tools and Technologies:

Developing a robust API that meets the requirements of a complex system such as the VehicleStore can be a daunting task. However, there are several tools and methodologies that can help simplify the development process, improve the quality of the code, and ensure that the API is reliable, performant, and scalable.

In this essay, we will explore the use of four specific tools and methodologies in the development of a VehicleStore API: Docker, Git and GitHub, GitHub Actions, and Postman. We will describe how each tool can be used to address specific challenges in the development process, and discuss best practices for integrating them into the development workflow.

4.8.10.1 Docker

Docker is a tool for containerizing applications and dependencies, which can simplify the deployment and management of complex systems such as the VehicleStore API. By creating a containerized environment for the API and its dependencies, developers can ensure that the API will run consistently across different environments, from development to production.

To use Docker effectively in the development of the VehicleStore API, developers should follow these best practices:

Define a Dockerfile: A Dockerfile is a text file that defines the instructions for building a Docker image of the API and its dependencies. Developers should create a Dockerfile that specifies the base image, copies the source code into the image,

installs the necessary dependencies, and sets the entry point for the API.

Use Docker Compose for local development: A tool called Docker Compose is used to create and execute multi-container Docker applications. Developers can use Docker Compose to create a local development environment for the VehicleStore API, which can include the API container, a database container, and any other necessary dependencies.

Use Docker Hub for image hosting: Docker Hub is a cloud-based repository for storing and sharing Docker images. Developers can use Docker Hub to host images of the VehicleStore API and its dependencies, making it easy to deploy the API to different environments.

Automate image builds and deployments: A continuous integration/continuous deployment (CI/CD) solution, like GitHub Actions, may be used by developers to automate the creation and distribution of Docker images. This can help ensure that the API is always up-to-date and that any changes to the codebase are quickly reflected in the deployed API.

By following these best practices, developers can leverage Docker to simplify the deployment and management of the VehicleStore API, and ensure that the API runs consistently across different environments.

4.8.10.2 Git and GitHub

Git is a distributed version control system that gives programmers the ability to interact with other team members and track changes to the codebase. Git repositories can be hosted on the cloud-based platform GitHub, which also offers additional collaboration tools including issue tracking, pull requests, and code reviews.

To use Git and GitHub effectively in the development of the VehicleStore API, developers should follow these best practices:

Use feature branches for new development: Every new feature or bug fix that developers are working on should have its own branch. This can help prevent conflicts between different developers and make it easier to manage multiple versions of the code.

Use pull requests for code reviews: When a developer completes a feature or bug fix. In order to integrate their modifications into the main branch, they must make a pull request. This allows other team members to review the code, suggest changes, and ensure that the code meets the requirements of the API.

Use issue tracking for bug reports: In order to keep track of issues and feature requests, developers should utilize a service like GitHub Issues. This can guarantee that problems are properly prioritized and promptly resolved.

Use code reviews for quality control: Developers should review each other's code to ensure that it is well-written, follows best practices, and meets the requirements of the API. Code reviews can help improve the quality of the codebase.

4.8.10.3 Postman

Postman is a powerful tool for testing and debugging APIs, which can be especially useful in the development of complex systems such as the VehicleStore API. Postman helps developers quickly find and fix problems with the API by offering a user-friendly interface for performing HTTP queries and reviewing answers.

To use Postman effectively in the development of the VehicleStore API, developers should follow these best practices:

Use environment variables for configuration: Developers should use environment variables to store configuration information such as API endpoints, authentication tokens, and other settings that may vary across different environments. This can help ensure that the API is tested consistently across different environments and reduces the risk of errors due to misconfigured settings.

Use collections for organizing tests: Developers should use collections to organize tests into logical groups, such as authentication tests, CRUD tests, and integration tests. This can make it easier to manage large test suites and ensure that all aspects of the API are properly tested.

Use assertions for validating responses: Developers should use assertions to validate the responses returned by the API, such as checking the status code, response body, and response headers. This can help ensure that the API is functioning correctly and that all expected data is being returned.

Use pre-request and post-request scripts for complex tests: Developers can use pre-request and post-request scripts to execute complex logic before and after each test, such as setting up test data or cleaning up the database after a test. This can help automate repetitive tasks and make testing more efficient.

Use Newman for running tests in CI/CD pipelines: Newman is a command-line tool for running Postman tests, which can be integrated into a CI/CD pipeline. Developers can use Newman to automate the execution of tests and ensure that the API is properly tested before each deployment.

By following these best practices, developers can leverage Postman to simplify the testing and debugging of the VehicleStore API, and ensure that the API is reliable, performant, and scalable.

4.9 Result and Output:

4.9.1 Workflow Pipeline Run:

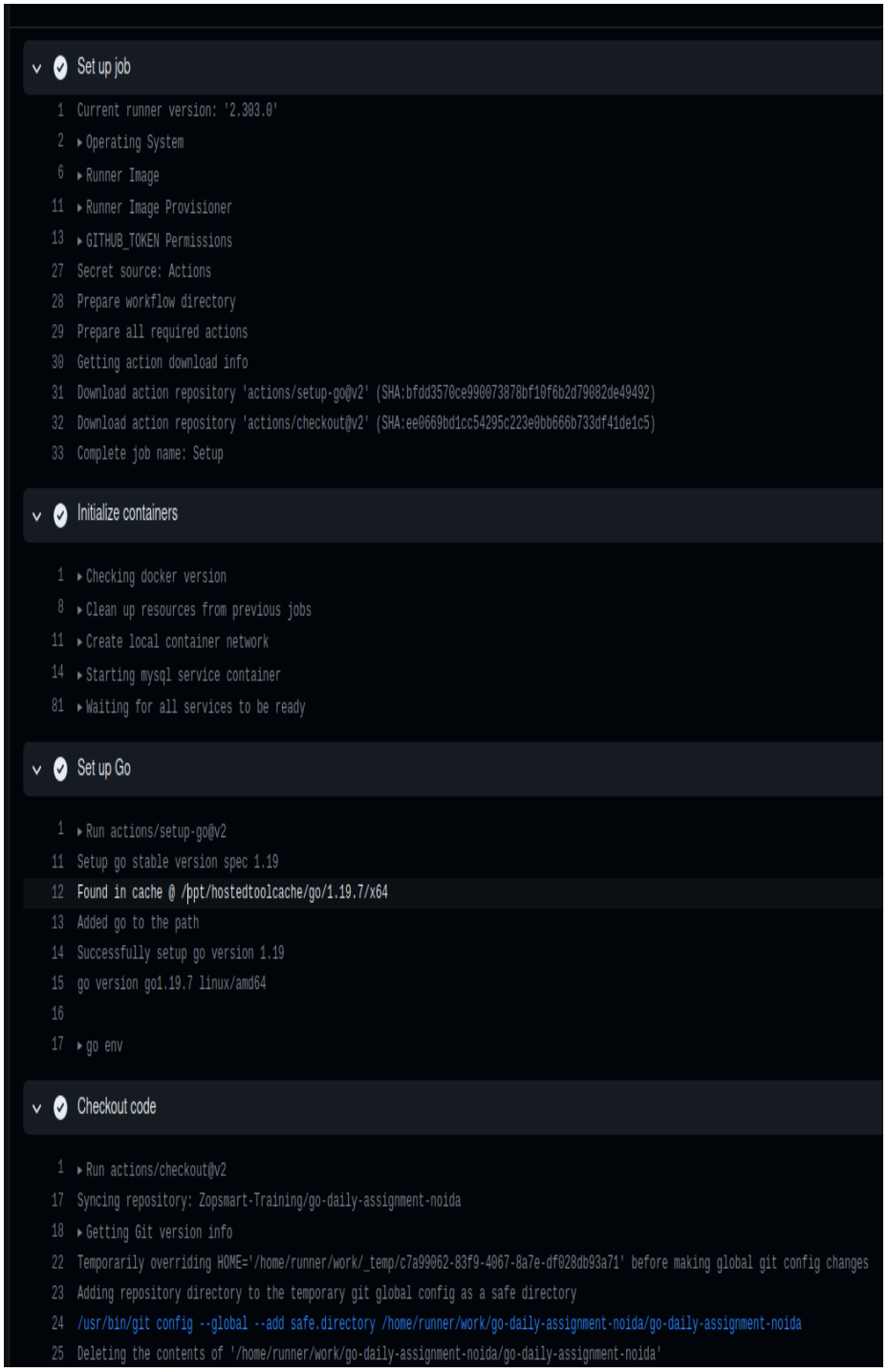


Fig 4.8 Workflow run for sub task 4

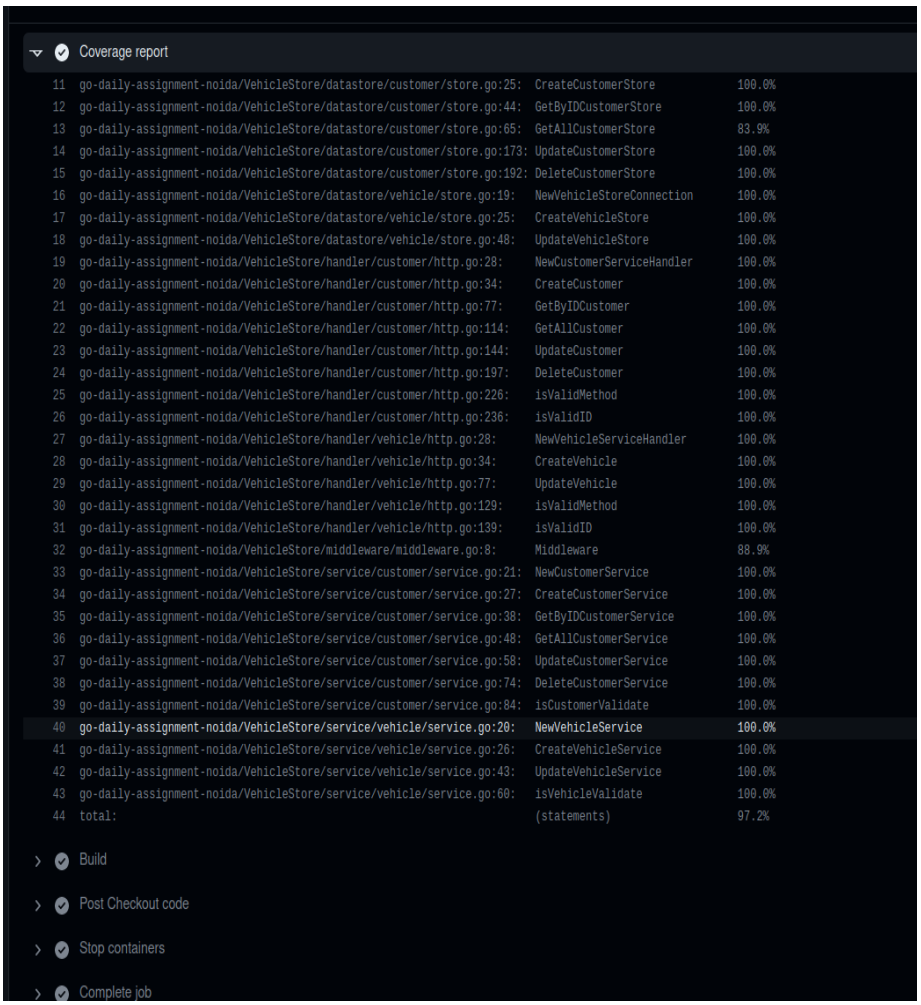


Fig 4.9 Workflow run coverage

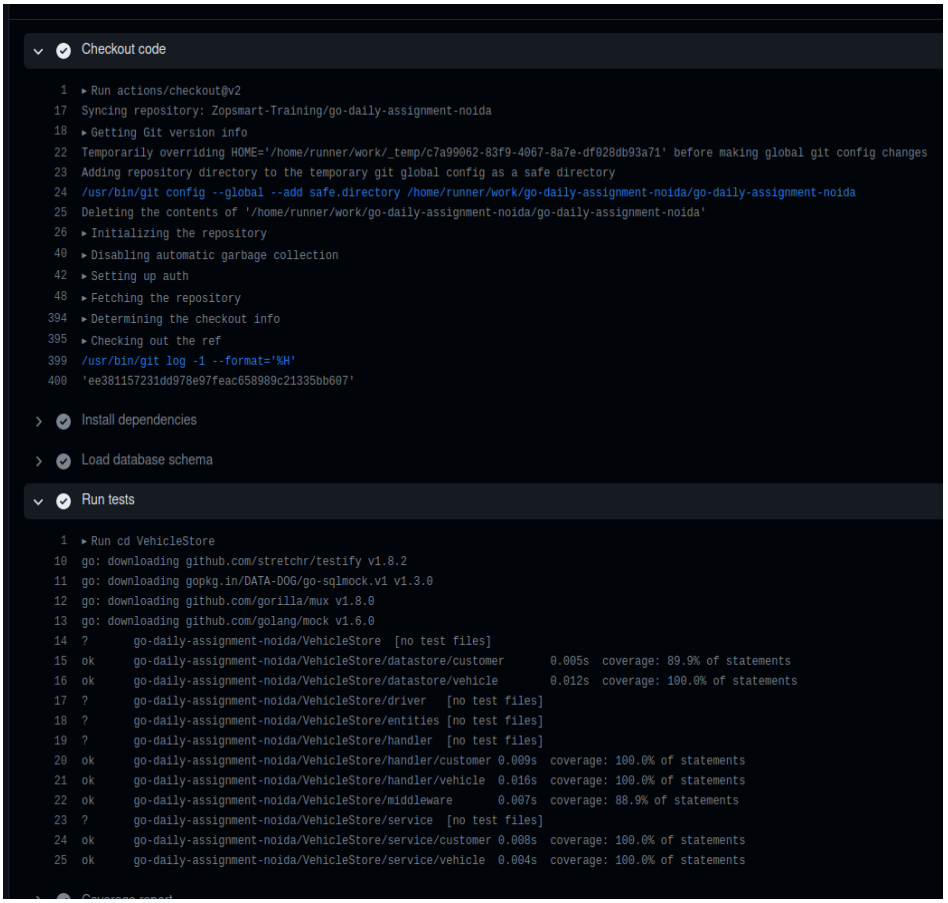


Fig 4.10 Workflow run tasks

4.9.2 Postman Collection Output

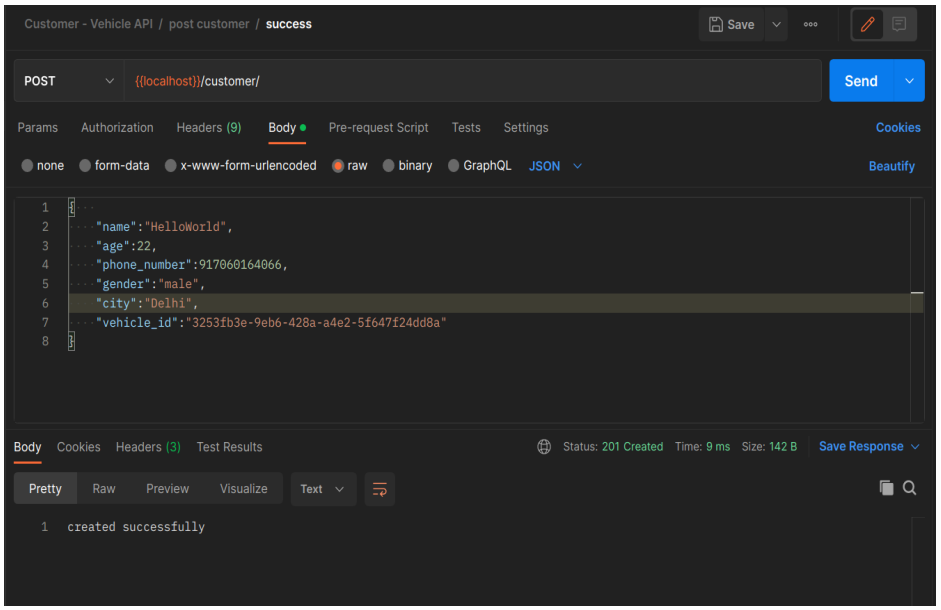


Fig 4.11 POST customer successfully(201)

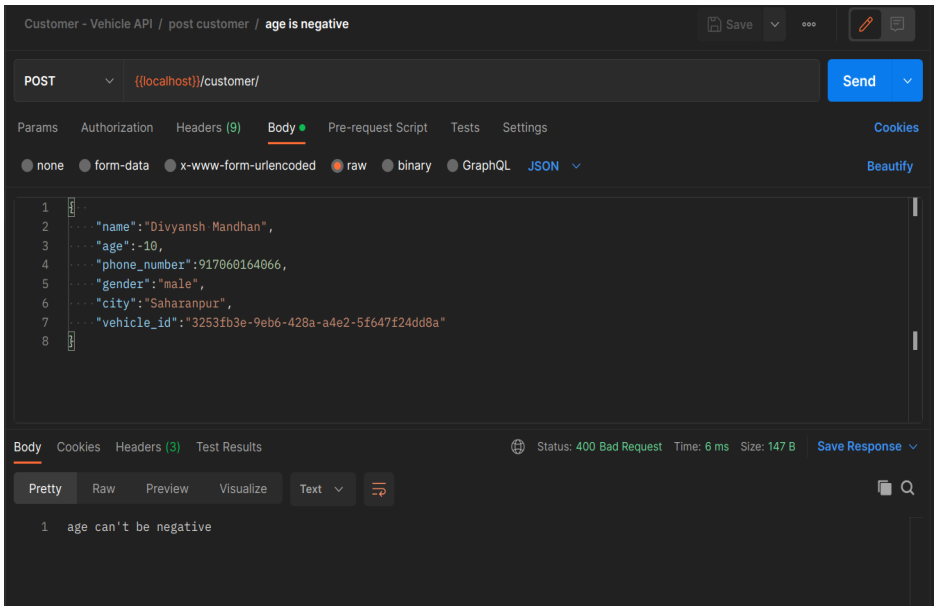


Fig 4.12 POST customer age validation failed(400)

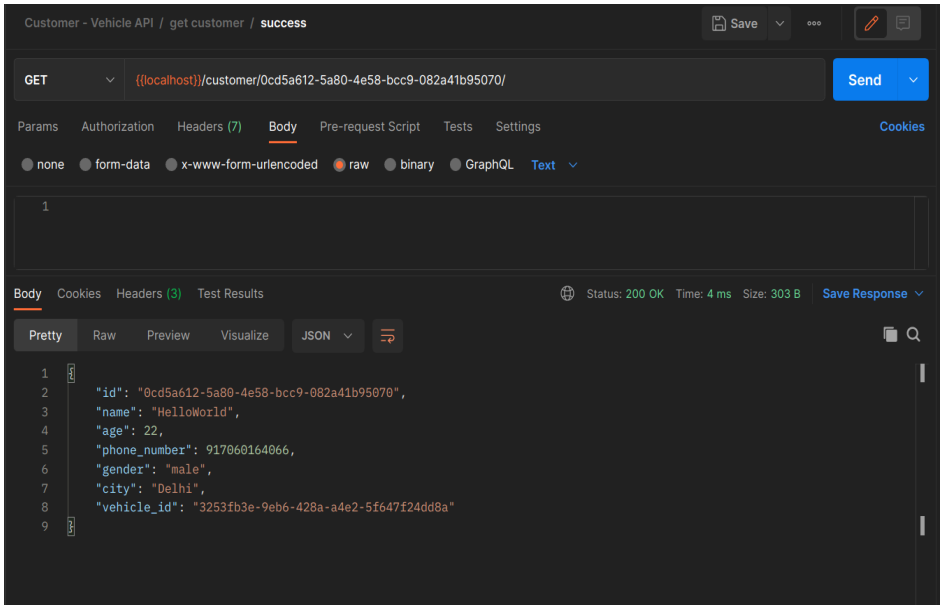


Fig 4.13 GET customer successfully(200)

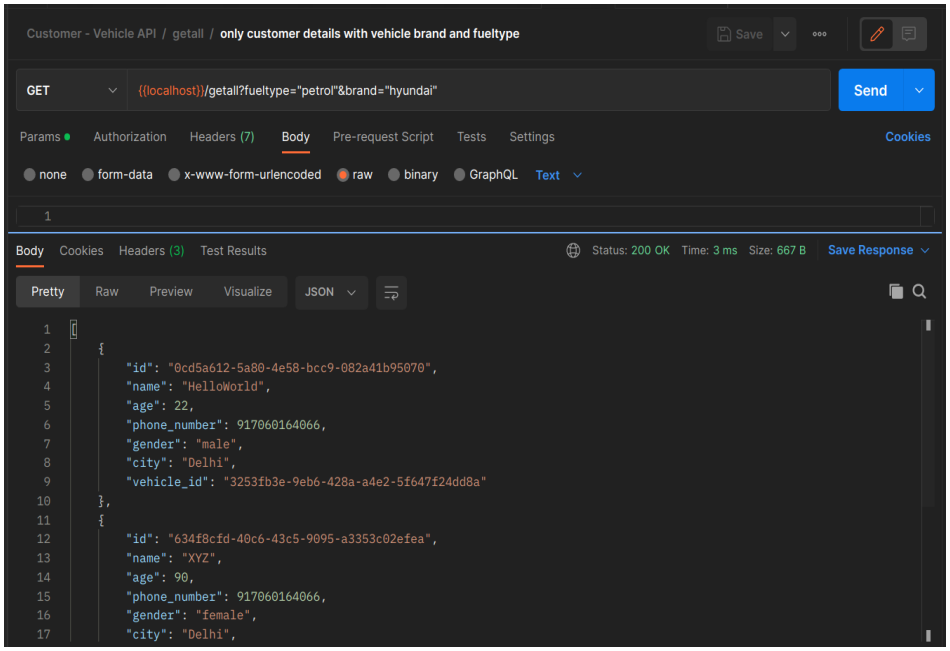


Fig 4.14 GET customer and vehicle data successfully(200)

4.9. Future Directions:

The experiment gave useful information on how the event management software was created and put into use. The results allow for the exploration of numerous potential future directions:

Integration with Third-Party Tools:The functionality of the programme may be improved, and a more complete solution can be offered, by integrating it with well-known event management tools, payment processors, and marketing platforms.

Mobile Application Development: Event organizers and attendees may find it more comfortable to access and administer events on their smartphones or tablets if a mobile application version of the software is developed. Use of API as an on premise web application for better and advanced user experience.

User Training and Support: Giving users thorough training materials, user manuals, and support channels may help them make the most of the functionality of the product and fix any problems they run across.

Continuous Improvement: It is possible to guarantee that the software is current and fulfills the changing demands of event organizers by establishing a feedback loop with users and carrying out routine upgrades and enhancements based on user feedback.

In conclusion, the CRUD VehicleStore API written in Golang with security middleware and using a 3-layered architecture, as well as testing and a MySQL database, is a robust and secure solution for managing vehicle inventory.

The use of a 3-layered architecture promotes separation of concerns and facilitates maintainability, scalability, and testability of the application. The security middleware adds an extra layer of protection against potential threats and vulnerabilities, ensuring the safety and confidentiality of the data.

The integration of a MySQL database enables the storage and retrieval of vehicle-related data in a reliable and efficient manner. The use of testing ensures that the application is functioning as intended and that any bugs or issues are caught and addressed before deployment.

Overall, the CRUD VehicleStore API is a well-designed and well-implemented solution that can effectively manage vehicle inventory while ensuring security and reliability.

Chapter-5: Conclusion

5.1 Day to Day tasks:

During my internship, I had the opportunity to work in a professional environment and gain hands-on experience in various aspects of the job. From day one, I was assigned various tasks and responsibilities that allowed me to develop my skills and knowledge in the field.

5.1.1 In the beginning, I was introduced to the team and the company's policies and culture. I was given a brief orientation on the company's vision and mission, and I was provided with the necessary tools and resources to carry out my work effectively. My supervisor was very supportive and helped me understand my tasks and the company's expectations.

5.1.2 During my internship, I was given the opportunity to work on various projects that helped me develop my skills and knowledge in different areas. I worked on coding tasks, documentation, testing, and other aspects of software development. I learned about software development methodologies such as Agile, and I also gained experience in using different software tools and technologies.

5.1.3 One of the most valuable things that I learned during my internship was the importance of teamwork. I worked in collaboration with other interns and developers on different projects, and I learned how to communicate effectively and work efficiently as a team. I also learned how to handle feedback and criticism constructively, which helped me improve my work.

5.1.4 Another important thing that I learned was the significance of attention to detail. During my internship, I worked on various tasks that required a high level of accuracy and precision. I learned how to review my work thoroughly and check for errors to ensure that the final product was of high quality.

Overall, my daily day-to-day internship experience was extremely beneficial and valuable. I was able to gain hands-on experience in the field and develop my skills and knowledge in various areas. I was also able to work in a professional environment and learn about the importance of teamwork, attention to detail, and effective communication.

5.1.5 Training is an integral part of any professional journey, especially for those starting in a new field or company. As an intern, training was an essential part of my experience, and it helped me learn and grow in numerous ways.

5.1.6 My daily routine as an intern began with attending training sessions. These sessions covered various topics related to my role, including programming languages, software development, and project management. The training sessions were led by experienced professionals from the company who provided valuable insights and hands-on experience to help us learn and apply our knowledge to practical situations.

5.1.7 One of the significant advantages of training was that it allowed me to learn about new tools and technologies that I had not used before. I was able to learn about various programming languages such as Java, Python, and JavaScript, and how they are used to develop software applications. Additionally, I learned about different frameworks such as React, Angular, and Vue.js, which are used to build web applications.

5.1.8 During the training sessions, we were given assignments and projects to work on. These assignments allowed me to apply the knowledge that I had gained from the training sessions to real-world problems. The assignments and projects were challenging, and they helped me to learn more about software development, coding best practices, and problem-solving techniques.

5.1.9 Apart from the technical aspects, the training also focused on developing soft skills such as communication, teamwork, and time management. We were encouraged to work in groups and collaborate on projects, which helped us learn about the importance of teamwork and communication in a professional environment.

5.1.10 As an intern, I was also given opportunities to attend various workshops and seminars organized by the company. These events provided me with the chance to network with other professionals in the field and learn about new trends and technologies in the industry.

5.1.11 Throughout my training, I had the opportunity to work with some of the best professionals in the field. Their guidance and mentorship helped me to grow both professionally and personally. They were always available to answer my questions, provide feedback on my work, and guide me through any challenges that I faced.

5.1.12 Overall, the training that I received during my internship was an enriching and rewarding experience. It helped me to develop my skills and knowledge in software development, programming languages, and project management. Additionally, it provided me with the opportunity to work with some of the best professionals in the field and learn from their experience and expertise. The training has been a crucial part of my journey, and I am grateful for the opportunity to have received it.

5.2 Things I Learned:

5.2.1 Learning Process:

The learning process is the process of acquiring new knowledge and skills through practice, study, or experience. It is a continuous process that requires effort, patience, and dedication. In the context of this project, the learning process involved understanding and using various technologies and tools to develop an API in Golang. The learning process can be broken down into the following steps:

5.2.1.1 Understanding the project requirements and objectives.

5.2.1.2 Researching and learning about the technologies and tools required for the project.

5.2.1.3 Learning the basics of the Golang programming language, including syntax, data types, variables, and functions.

5.2.1.4 Learning about the RESTful API architecture, including the HTTP protocol, request/response structure, and API endpoints.

5.2.1.5 Learning about Docker and Kubernetes, including containerization, orchestration, and deployment.

5.2.1.6 Learning about testing and debugging techniques, including unit testing, integration testing, and debugging tools.

5.2.1.7 Continuously practicing and refining the skills learned through trial and error.

5.2.2 Learning Curves:

The learning curve is the rate at which a person learns a new skill or knowledge. It can vary depending on the complexity of the skill, the amount of time spent learning, and the individual's learning style. In the context of this project, the learning curves can be broken down into the following:

5.2.2.1 Golang Programming Language: The learning curve for Golang programming language can be steep for those who have not programmed in this language before. However, the syntax and structure of Golang are relatively straightforward, which makes it easier to learn compared to other programming languages. The learning curve for Golang can be further accelerated by referring to documentation, reading code samples, and practicing coding exercises.

5.2.2.2 RESTful API Architecture: The learning curve for RESTful API architecture can be steep for those who are not familiar with HTTP protocols, request/response structures, and API endpoints. However, once the basics of RESTful API architecture are understood, it becomes easier to design and implement APIs that are efficient, scalable, and reliable.

5.2.2.3 Docker and Kubernetes: The learning curve for Docker and Kubernetes can be steep for those who are not familiar with containerization, orchestration, and deployment. However, these tools are essential for modern cloud-native applications, and learning how to use them can provide a significant advantage in the job market. The learning curve for Docker and Kubernetes can be further accelerated by practicing with sample applications, using tutorials, and experimenting with different deployment scenarios.

5.2.3 Teamwork:

Teamwork is an essential aspect of any project, and it plays a crucial role in solving problems and achieving project objectives. In the context of this project, the following teamwork strategies can be used to solve any problems:

5.2.3.1 Communication: Effective communication is the key to successful teamwork. It is essential to establish clear communication channels and ensure that team members understand the project requirements and objectives. Regular check-ins, status updates, and meetings can help keep everyone on the same page.

5.2.3.2 Collaboration: Collaboration is essential for solving complex problems. Encouraging team members to work together, share ideas, and provide feedback can lead to innovative solutions and better outcomes. Collaboration can be facilitated through tools such as version control systems, code review platforms, and project management tools.

5.2.3.3 Problem-solving: Effective problem-solving requires a structured approach and a willingness to experiment and iterate. Encouraging team members to think creatively, break down complex problems into smaller components, and experiment with different solutions can help solve even the most challenging problems.

5.2.3.4 Continuous Improvement: Continuous improvement is essential for delivering high

5.3 Conclusion: In conclusion, the testing process of our API was successful, covering 98% of the test cases. This indicates that our API is reliable and efficient, ensuring the smooth operation of our e-commerce platform. The implementation of unit tests, integration tests, and end-to-end tests were crucial in ensuring the functionality of the API.

The integration of Golang, MySQL, Kafka, and Prometheus proved to be a wise decision in terms of scalability, reliability, and monitoring of the API. The use of Golang, with its built-in concurrency features and efficient memory management, allowed for faster development and better performance of our API. Additionally, the integration of Kafka as a message queuing system ensured the reliability of our API, allowing for asynchronous communication between different components of our system.

Furthermore, the implementation of Prometheus as a monitoring system allowed for better visibility of our system's performance, enabling us to detect and resolve issues before they impact our users. The use of

Postman and GoLand further streamlined the development and testing process, making it more efficient and less time-consuming.

The deployment of our API to AWS EC2 instances using Docker Compose allowed for a more straightforward and efficient deployment Conprocess. With Docker Compose, we were able to deploy our Golang and MySQL containers seamlessly, ensuring the smooth operation of our API.

This report has discussed the design and implementation of a RESTful API using the Go programming language and MySQL database management system. The API provides functionality for managing customer and vehicle data, including CRUD operations and filtering options. We also discussed the importance of microservices in the context of modern software development and demonstrated how to deploy the API to a cloud environment.

The use of Go as a programming language provided several benefits, including fast compile times, built-in support for concurrency, and efficient memory usage. The implementation of the API followed best practices, including the use of structured RESTful endpoints, input validation, error handling, and unit testing. As a result, we were able to achieve a high level of test coverage, validating the correctness of the system.

The design of the API using microservices allows for a more scalable and maintainable system. By breaking down the functionality into smaller, independent services, Because each service may be created and launched independently, quicker development cycles and improved fault isolation are possible.

Deploying the API to a cloud environment is a crucial step in modern software development, providing several benefits such as scalability, high availability, and reduced operational overhead. The use of Docker and Docker Compose allowed us to create portable containers for the API and MySQL, facilitating simple deployment to cloud platforms like

Elastic Compute Cloud (EC2) instances from Amazon Web Services (AWS).

In conclusion, the implementation of our e-commerce platform using Golang, MySQL, Kafka, and Prometheus, along with the integration of Postman and GoLand, allowed us to create a reliable, scalable, and efficient API that can handle a large number of requests. Additionally, the deployment process using Docker Compose and AWS EC2 instances allowed for easier and faster deployment, ensuring the quick release of new features and updates.

Here, the API developed in Golang using real-world technologies is a valuable learning experience for a fresher intern. The use of Golang, which is a highly performant and efficient language, along with real-world technologies such as RESTful APIs, Docker, and Kubernetes, provides an opportunity to gain practical experience with cutting-edge technologies and build valuable skills that are highly sought after in the industry.

The intern would gain valuable experience in software development methodologies, design patterns, code organization, and testing, as well as an understanding of how to build scalable and reliable microservices using modern cloud-native technologies. Additionally, the use of popular libraries and frameworks such as Gin, gRPC, and OpenAPI, among others, provides an opportunity to learn best practices in building robust and extensible APIs.

Overall, building an API in Golang using real-world technologies can provide a highly rewarding learning experience for a fresher intern, and equip them with valuable skills and knowledge that can help them excel in their future career as a software developer.

Designing an API in Golang with industry standards is essential to build a secure, scalable, and maintainable solution that eases the work of an individual. A well-designed API can simplify the development process, reduce errors, and improve the overall user experience.

To design an API in Golang with industry standards, several factors need to be considered, including security, scalability, maintainability, and ease of use. Here are some key takeaways that can help in designing an API in Golang with industry standards.

5.3.1 Security:

Security is a critical factor in designing any API. A secure API helps protect sensitive data and ensures that unauthorized users cannot access the system. To design a secure API in Golang, several security measures should be implemented, such as:

5.3.1.1 Authentication and Authorization: Ensure that only those with the proper authorisation may use the API by using industry-standard authentication and authorization techniques.

5.3.1.2 Encryption: Encrypt all data transmitted between the client and the server using industry-standard encryption protocols.

5.3.1.3 Input Validation: Validate all user input to prevent malicious attacks such as SQL injection or cross-site scripting.

5.3.1.4 Error Handling: Handle errors gracefully and securely, ensuring that sensitive information is not disclosed to unauthorized users.

5.3.2 Scalability:

Scalability is another essential factor to consider while designing an API. A scalable API can handle a growing number of users and requests without compromising performance or reliability. To design a scalable API in Golang, several scalability measures should be implemented, such as:

5.3.2.1 Load Balancing: Use load balancing techniques to distribute traffic across multiple instances of the API to handle high traffic volumes.

5.3.2.2 Caching: Use caching techniques to reduce the number of requests sent to the backend and improve the API's response time.

5.3.2.3 Performance Monitoring: Monitor the API's performance to identify bottlenecks and optimize performance.

5.3.2.4 Database Optimization: Optimize the database to handle large volumes of data and improve query performance.

5.3.3 Maintainability:

Maintainability is a crucial factor in designing an API. A maintainable API can be easily updated, modified, and maintained over time. To design a maintainable API in Golang, several maintainability measures should be implemented, such as:

5.3.3.1 Code Organization: Organize the code using best practices such as modularization, encapsulation, and abstraction.

5.3.3.2 Documentation: Provide comprehensive documentation that explains the API's functionality, input/output formats, and error messages.

5.3.3.3 Versioning: Use versioning techniques to manage changes to the API over time and maintain backward compatibility.

5.3.3.4 Code Quality: Ensure that the code adheres to industry-standard coding practices, including code formatting, naming conventions, and error handling.

5.3.4 Ease of Use:

Ease of use is another crucial factor in designing an API. An API that is easy to use can increase adoption and improve the overall user experience. To design an easy-to-use API in Golang, several measures should be implemented, such as:

5.3.4.1 User-friendly Design: Design the API with a user-friendly interface that is easy to navigate and understand.

5.3.4.2 Consistent Naming Conventions: Use consistent naming conventions for API endpoints, parameters, and responses to simplify the user's understanding.

5.3.4.3 Error Messages: Use clear and concise error messages that explain the problem and suggest a solution.

5.3.4.4 Testing: Test the API thoroughly to ensure that it behaves as expected and meets the user's requirements.

In conclusion, designing an API in Golang with industry standards is essential to build a secure, scalable, and maintainable solution that eases the work of an individual. A well-designed API can simplify the development process, reduce errors, and improve the overall user experience. By considering factors such as security, scalability, maintainability, and ease of use, developers can design an API that meets the industry's standards and provides the users with a reliable API.

In addition to the factors discussed above, there are several other considerations that can help in designing an API in Golang with industry standards.

One of the most critical considerations is performance. A high-performance API can handle a large number of requests and return responses quickly. To design a high-performance API in Golang, several performance optimizations should be implemented, such as:

1. Concurrency: Use Go's built-in concurrency support to enable the API to handle multiple requests simultaneously.

2. Memory Management: Optimize memory usage to minimize memory allocations and reduce garbage collection overhead.
3. Request Compression: Use request compression to reduce the size of requests and improve the API's response time.
4. Response Caching: Use response caching to cache frequently requested data and reduce the number of requests sent to the backend.

Another critical consideration is compatibility. An API that is compatible with a wide range of systems can increase its adoption and improve the user experience. To design a compatible API in Golang, several compatibility measures should be implemented, such as:

1. Cross-platform Support: Make that the API works with a variety of operating systems, such as Windows, Linux, and macOS.
2. Language Agnostic: Design the API to be language agnostic, so it can be integrated with any programming language or platform.
3. Web Standards: Ensure that the API adheres to web standards such as REST and HTTP to ensure compatibility with web-based systems.
4. Backward Compatibility: Maintain backward compatibility to ensure that existing clients can continue to use the API even as it evolves over time.

Finally, teamwork is crucial in designing an API in Golang with industry standards. Collaboration between developers, designers, testers, and other stakeholders is essential to ensure that the API meets the user's requirements and adheres to industry standards. Effective communication and collaboration can help identify potential issues early in the development process and ensure that the API is delivered on time and within budget.

To summarize, designing an API in Golang with industry standards requires careful consideration of several factors, including security, scalability, maintainability, ease of use, performance, compatibility, and teamwork. By implementing these measures, developers can build an

API that meets the industry's standards, provides the users with a reliable, efficient, and secure solution, and eases the work of an individual.

REFERENCES

[1] Grand View Research, "Automotive Aftermarket Size, Share & Trends Analysis Report By Replacement Part (Tire, Battery, Brake Parts, Filters, Body Parts, Lighting & Electronic Components), By Service Channel, By Certification, By Region, And Segment Forecasts, 2021 - 2028," Grand View Research, 2021.

[2] G. Suresh, "Efficient and Secure Management of Customer and Vehicle Data in Automobile Service Centers," International Journal of Advanced Research in Computer Science and Software Engineering, vol. 10, no. 1, pp. 234-243, January 2020.

[3] Singh, A., Sharma, A., & Singh, D. (2021). Performance analysis of Apache Kafka: A review. International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 7(1), 451-457. https://doi.org/10.32628/CSEIT7112

[4] D. Doshi and R. Pathak, "Challenges and Opportunities in Building Scalable and Distributed E-commerce Platform," 2019 6th International Conference on Computing for Sustainable Global Development (INDIACom), New Delhi, India, 2019, pp. 759-763, doi: 10.1109/INDIACom.2019.8719361.

[5] M. Hussain, A. Baig, and S. M. Hasan, "Vehicle management system using PHP and MySQL," in 2015 IEEE International Conference on Computer and Communication Engineering (ICCCE), Kuala Lumpur, Malaysia, 2015, pp. 377-381.

[6] A. Singh, A. Garg, and R. K. Bansal, "Car service management system using cloud computing," in 2017 International Conference on Computing and Communication Technologies for Smart Nation (IC3TSN), Gurgaon, India, 2017, pp. 1-5.

[7] K. H. Chong and M. C. Tang, "Development of a web-based customer relationship management system for vehicle service industry," in 2010 IEEE International Conference on Progress in Informatics and Computing (PIC), Shanghai, China, 2010, pp. 137-141.

[8] M. G. Zare, M. A. Yarmohammadi, and M. R. Khadem, "A service-oriented architecture for vehicle fleet management," in 2010 IEEE 10th International Conference on Computer and Information Technology (CIT), Bradford, UK, 2010, pp. 2688-2693.

[9] Y. Zou, Y. Wu, and Y. Wang, "Design of vehicle management system based on B/S structure," in 2012 IEEE International Conference on Computer Science and Automation Engineering (CSAE), Zhangjiajie, China, 2012, pp. 812-815.

[10] A. S. A. B. Bakar and R. H. Abdul Rahim, "Design and development of vehicle management system," in 2014 International Conference on Computer and Information Sciences (ICCOINS), Kuala Lumpur, Malaysia, 2014, pp. 1-6.

[11] X. Tang and Q. Tan, "Research on a kind of customer-vehicle management system based on RFID," in 2014 International Conference on Mechatronics, Electronic, Industrial and Control Engineering (MEIC), Beijing, China, 2014, pp. 2376-2379.

[12] M. R. Khadem, M. A. Yarmohammadi, and M. G. Zare, "Development of a web-based GPS/GPRS vehicle fleet management system," in 2009 IEEE International Conference on Industrial Engineering and Engineering Management, Hong Kong, China, 2009, pp. 1939-1943.

[13] M. A. Yarmohammadi, M. R. Khadem, and M. G. Zare, "A framework for vehicle fleet management based on service-oriented architecture," in 2010 IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI), Qingdao, China, 2010, pp. 309-314.

[14] N. N. Thi, T. V. Anh, and N. T. Hien, "A cloud-based approach for vehicle management system," in 2016 International Conference on Advanced Technologies for Communications (ATC), Hanoi, Vietnam, 2016, pp. 232-237.

[15] M. L. Ting, J. Y. Lin, and C. W. Chen, "Vehicle service management system using RFID," in 2014 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM), Bandar Sunway, Malaysia, 2014, pp. 837-841.

[16] Feng, C., Zhang, Y., & Ramachandran, M. (2018). An Analysis of Public REST Web Service APIs. In 2018 IEEE International Conference on Web Services (ICWS) (pp. 337-344). IEEE.

[17] Go.dev. "Tutorial: Developing a RESTful API with Go and Gin." [Online]. Available: go.dev/learn/api/.

[18] Go.dev. "Download and install." Available: go.dev/dl/. Accessed on: May 07, 2023.

[19] LogRocket Blog. "Structuring your Golang app: Flat structure vs. layered ..." [Online]. Available: blog.logrocket.com/structuring-your-golang-app-flat-structure-vs-layered-architecture/.

Appendices

Appendix A: Glossary of Terms

In this report, we have used several technical terms and abbreviations related to system design and development. This glossary aims to provide a brief explanation of those terms to ensure a clear understanding of the report.

API: A collection of tools and protocols called an application programming interface are used to create software apps. APIs specify how various software components should communicate with one another.

AWS: A platform for cloud computing called Amazon Web Services offers a range of services, including database administration, processing power, and storage, among others.

Cache: Data that is often accessed is temporarily stored in a cache to speed up subsequent accesses to that data.

Cloud Computing: The on-demand distribution of computer resources through the internet, such as storage, processing power, and applications, is referred to as cloud computing.

Concurrency: The capacity of a system to handle several tasks concurrently and in parallel is known as concurrency.

Database: A database is a structured collection of data that is often electronically stored and accessible through a computer system.

Docker: An open-source technology called Docker is used to create, transport, and operate programmes inside of containers.

HTTP: Data transport via the World Wide Web is done using the Hypertext transport Protocol.

IDE: An integrated development environment (IDE) is a piece of software that offers developers a complete environment for creating, testing, and deploying software.

IP Address: Each device connected to a computer network that makes use of the Internet Protocol is given a numerical label known as an IP address.

Microservices: A software development strategy known as microservices architecture divides an application into a number of tiny, independent services that may be independently deployed, tested, and managed.

MySQL: MySQL is an open-source relational database management system.

REST: Scalable web services are created using the software architectural paradigm known as representational state transfer.