INTERNSHIP REPORT

Submitted in the fulfilment of the requirement for the degree of Bachelor of Technology

in

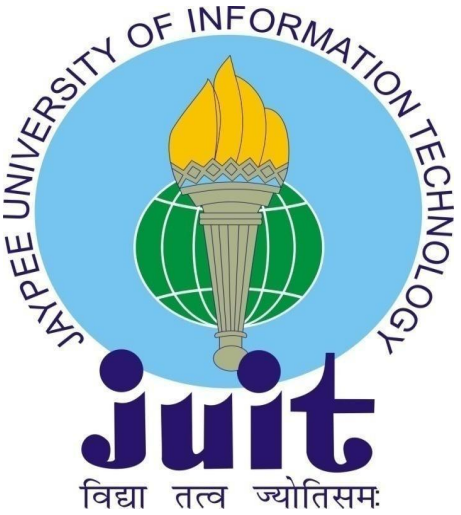
Computer Science and Engineering

By

Aakanksha Jaiswal (171304)

UNDER THE SUPERVISION OF

Mr. Akshat Kumar Singhal



Department of Computer Science & Engineering and Information Technology

Jaypee University of Information Technology, Waknaghat, 173234, Himachal Pradesh, INDIA

DECLARATION BY CANDIDATE

I hereby declare that the work presented in this report entitled **"Developing Scalable Backend Systems in GoLang"** in the 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 at ZopSmart Pvt. Ltd. carried out over a period from February 2021 to May 2021 .

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

(Student's Signature)
Aakanksha Jaiswal
171304

This is to certify that the above statement made by the candidate is true to the best of my knowledge.
Mr. Akshat Singhal
(Senior Backend Developer, Manager)
ZopSmart

ACKNOWLEDGEMENT

I take this opportunity to express my sincere thanks and deep gratitude to all those people who extended their wholehearted cooperation and have helped me in completing this internship successfully.

First of all, I would like to thank **Ms. Mithali R. Shetty**, **Mr. Akshat Singhal** and **Mr. Vikash Kumar,** my mentors, who mentored me, guided me and challenged me. Last but not the least, I would like to thank our founders for considering me a part of their organization and providing such a great Platform to learn and enhance my skills.

A very special thanks goes to all the faculties of Jaypee University of Information Technology under whose guidance I have been able to excel in my career and become a part of the **ZopSmart** family.

Aakanksha Jaiswal
(CSE, 171304)
Jaypee University of Information Technology

TABLE OF CONTENT

COMPANY'S PROFILE

Founded by Mr. Mukesh Sigh, ZopSmart expertises in the retail domain which is built on its team's immense knowledge in the fields of FMCG, retail, supply-chain and logistics- all the experience has been baked into business processes that are embedded in their products. Their solution has been built over 7 years and is handling millions of transactions every day. The suite of products are one of the most advanced, stable and scalable solutions available in the world of retail technology.

The team consists of passionate retail professionals who hail from world's most prestigious educational institutions- Mukesh Singh , Founder (B.Tech. IIT Kanpur, PhD Massachusetts Institute of Technology); Raj Pander, CEO and Co-Founder (B.Tech. IIT Kanpur, MBA Wharton School of Business) ; and Vikash Kumar, CTO (B.Tech IIT Bombay).

ZopSmart tends to collaborate with Kroger and McAfee and is helping them shift their backend from conventional non-performant systems to scalable, maintainable, fast and reliable solutions. It uses google's newly developed Go Language as the primary tool for development. Other than these, ZopSmart owns numerous products of its own such as SmartStore (E-commerce platform), Pagenics, Hiring Portal, Quizzing Tool, etc. One of the major users of SmartStore is Tammimi from Saudi Arabia, which is one of the largest online stores in the UAE.

Chapter 01: INTRODUCTION

1.1 Web Application and its Components

All web-based database applications have three primary components: A web browser (or client), a web application server, and a database server.

Web-based database applications rely on a database server, which provides the data for the application. The clients handle the presentation logic, which controls the way in which users interact with the application. A Web Application Server responds to the clients request by interacting with the database. It contains all the business logic and acts as a mediator between the two.

Hence we have three layers in a web application:

- **View Layer**
  It provides an interface to the application, regardless of who it is for- the users with a browser or for another application using Web services. View layer is the bridge for getting the data in and out of the application. It does not have business logic, it is more focused on the interface.

- **Business Layer**
  It accepts user requests from the browser, processes them, and determines the routes through which the data will be accessed. The workflows by which the data and requests travel through the backend lay encoded in a business layer.

- **Data Access Layer**
  It is built to keep the code we use to pull data from our data store like database, flat files, or web services separate from business logic and presentation code. Since it focuses only on interacting with the data, it can be replaced easily as per needs. This helps in scaling applications well.

1.2 Software Architecture

In this project, we propose a two-stage CNN architecture, where the first stage detects human faces while the second one uses a lightweight image classifier to classify the faces detected in the first stage as ‘With Mask’ or ‘Without Mask’ and draws bounding boxes around them along with the confidence score of the predicted category.

1.2.1 Monolith Architecture

Monolith means composed all in one piece. The Monolithic application describes a single-tiered software application in which different components combined into a single program from a single platform.

Benefits:

- Simple to develop.
- Simple to test.
- Simple to deploy.
- Simple to scale horizontally by running multiple copies behind a load balancer.

Drawbacks:

- *Maintenance* — If Application is too large and complex to understand entirely, it is challenging to make changes fast and correctly.
- The size of the application can slow down the start-up time.
- The entire application must be deployed on each update.
- Monolithic applications can also be challenging to scale when different modules have conflicting resource requirements.
- *Reliability* — Bug in any module (e.g. memory leak) can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug impact the availability of the entire application
- Regardless of how easy the initial stages may seem, Monolithic applications have difficulty adopting new and advanced technologies. Since changes in languages or frameworks affect an entire application, it requires effort to thoroughly work with the app details, hence it is costly considering both time and effort.

Monolithic applications fit best for use cases where the requirements are pretty simple, the app is expected to handle a limited amount of traffic.

1.2.2 Layered Architecture

This pattern can be used to structure programs that can be decomposed into groups of subtasks, each of which is at a particular level of abstraction. Each layer provides services to the next higher layer.

The most common layers are:

1. *Delivery Layer :* The delivery layer will receive the request and parse anything that is required from the request. It calls the use case layer, ensures that the response is the required format and writes it to the response writer.
2. *Use Case Layer :* The use case layer does the business logic that is required for the application. This layer will communicate with the datastore layer. It takes whatever it needs from the delivery layer and then calls the datastore layer. Before and after calling the datastore layer, it applies the business logic that is required.
3. *Datastore Layer :* The datastore stores the data. It can be any data storage. The use case layer is the only layer that communicates with the datastore. This way each layer can be tested independently without depending on the other.

1.2.3 Microservice Architecture

Microservices are an approach to application development in which a large application is built as a suite of modular services (i.e. loosely coupled modules/components). Each module supports a specific business goal and uses a simple, well-defined interface to communicate with other sets of services.

Instead of sharing a single database as in Monolithic application, each microservice has its own database. Having a database per service is essential if we want to benefit from microservices, because it ensures loose coupling. Each of the services has its own database. Moreover, a service can use a type of database that is best suited to its needs.

1.2.4 Model-View-Controller (MVC) Architecture

The MVC architecture is a software architectural pattern in which the application logic is divided into three components on the basis of functionality.

These components are called:

- *Models* - represent how data is stored in the database
- *Views* - the components that are visible to the user, such as an

output or a GUI

- *Controllers* - the components that act as an interface between models and views

At first glance, the three tiers may seem similar to the model-view-controller (MVC) concept; however, topologically they are different. A fundamental rule in a three tier architecture is the client tier never communicates directly with the data tier; in a three-tier model all communication must pass through the middle tier. Conceptually the three-tier architecture is linear. However, the [model-view-controller] MVC architecture is triangular: the view sends updates to the controller, the controller updates the model, and the view gets updated directly from the model.

The MVC architecture is used not only for desktop applications but also for mobile and web applications.

1.3 Software Design Patterns

Software design is responsible for the code level design such as, what each module is doing, the classes scope, and the functions purposes, etc. When used strategically, they can make a programmer significantly more efficient by allowing them to avoid reinventing the wheel, instead using methods refined by others already. Most commonly used software designed patterns are: Singleton, Factory Method, Strategy, Observer, Builder, Adapter and State.

1.4 Software Design Principles

SOLID is an acronym formed by the names of 5 design principles centered around better code design, maintainability, and extendability. The principles were first introduced by Robert Martin (more familiar in the developer circles as Uncle Bob) in his 2000 paper Design Principles and Design Patterns. The principles were later named by Michael Feathers who switched their order so they can form the acronym.

The SOLID software principles will guide us to:

- write code that's easy to maintain;
- make it easier to extend the system with new functionality without breaking the existing ones;

- write code that's easy to read and understand.

- **Single Responsibility Principle**
  Single Responsibility Principle is the S in SOLID. Single responsibility means that our class (any entity for that matter, including a method in a class, or a function in structured programming) should only do one thing. If our class is responsible for getting users' data from the database, it shouldn't care about displaying the data as well. Those are different responsibilities and should be handled separately.

- **Open/Closed Principle**
  The Open/Closed Principle states that a module should be open for extension, but closed for modification. That means one should be able to extend a module with new features not by changing its source code, but by adding new code instead. The goal is to keep working, tested code intact, so over time, it becomes bug resistant.

- **Liskov Substitution Principle**
  One should be able to substitute a parent class with any of its child classes, without breaking the system, that is implementations of the same interface should never give a different result.

- **Interface Segregation Principle**
  Also known as the principle of lean interface, the Interface Segregation Principle states that one should never force the client to depend on methods it doesn't use.

- **Dependency Inversion Principle**
  The dependency inversion principle states that high-level modules should not depend on low-level modules - both should depend on abstractions. By combining the dependency injection technique with the concept of binding an interface to a concrete implementation, we can make sure that we never depend on concrete classes. This will allow us to easily change the implementation of specific parts of the system without breaking it. A good example of this is switching our database driver from SQL to NoSQL. If we depend on the abstract interfaces for accessing the database, we'd be able to easily change the specific implementations and make sure the system works properly.

Chapter 02: USING GoLang FOR BACKEND

"Building High-Performance Apps with Golang"

Developed by Google in 2009, Golang is an open-source language that has been steadily gaining traction. While there is a right tool/language for every job, Golang or Go makes a perfect choice in many scenarios.

Major Benefits of Golang:

Concurrency

Modern applications are built for real-time collaboration and increasingly rely on microservices. A language that has built-in features to support concurrent web requests is highly desirable. Go manages concurrency efficiently while keeping the execution straightforward. It enables connections with millions of users from a single instance and interaction with any number of services without blocking web requests.

Simplicity

Golang has a clean syntax. To keep it simple, Go forgoes many features such as classes, inheritance, and annotations. While this could lead to a few extra lines of code, Golang ensures clarity. The language becomes an ideal choice if you have a very large codebase with different teams working on different segments of the code. Modifications and code maintenance becomes a breeze compared to languages like Java.

Performance

Golang races past high-level languages like Java in performance. Because it can be compiled directly and does not need a virtual machine to convert the code into machine-readable format, Go code executes fast similar to JavaScript on a web page. Moreover, sub-millisecond garbage collection pauses in Golang helps to avoid loading delays boosting application speed.

Chapter 03: IMPLEMENTATION OF THE TRAINING PROJECT

3.1 Problem Statement

The Problem statement given was:

Create a service to manage a multi brand car dealership. The dealer works with only the following brands:

- Tesla
- Porsche
- Ferrari
- Mercedes
- BMW

This dealership handles 3 types of engines/motors (will be referred as engine henceforth):

- Petrol
- Diesel
- Electric

Engine details will have

- Displacement - for petrol and diesel
- Number of cylinders - for petrol and diesel
- Range(in km) - for electric only

Following are the requirements for the application:

The Dealer can add models for the above brands. Following information are required for adding:

- Name
- Year
- Fuel type
- Brand
- Engine details

View all models available. There should be an option to view engine details. Output:

- Name
- Year
- Fuel
- Brand Name
- Engine details

View model by brand. There should be an option to view engine details. Output:

- Name
- Year
- Fuel
- Engine details

Update the model details. Following details can be updated:

- Year
- Fuel type
- Engine details

Delete a model. When a model is deleted, the engine details should also be deleted.

Technical details

- Brand and fuel type will be constant and cannot be changed.
- Brand table should be pre-populated and no new brands can be added
- New fuel types cannot be added.

Authorisation Header
apikey - 5n6j76bv4v3h3ji4b5v6

GET /model?include=engine

Query Params: include=engine (optional)

Response: 200

```
[
    {
      "name" : "",
      "year" : 1999,
      "fuelType" : "diesel",
      "brand" : "",
      "engine" : {
           "displacement" : 1000,
           "numberOfCylinders" : 4
      }
```

```
    },
    {
      "name" : "",
      "year" : 1999,
      "fuelType" : "electric",
      "brand" : "",
      "engine" : {
           "range" : 800
      }
    }
]
```

GET /model?brand={name}

Response: 200

```
[
    {
      "name" : "Carrera GT",
      "year" : 1999,
      "fuelType" : "petrol",
      "brand" : "",
      "engine" : {
           "displacement" : 1000,
           "numberOfCylinders" : 4
      }
    },
    {
      "name" : "Taycan Turbo S",
      "year" : 2020,
      "fuelType" : "electric",
      "brand" : "",
      "engine" : {
           "range" : 400
      }
    }
]
```

GET /model/{id}

Response:

```
{
   "name" : "",
   "year" : 1999,
   "fuelType" : "electric",
   "brand" : "",
   "engine" : {
        "range" : 800
   }
}
```

POST /model

Request Body:

```
{
   "name" : "",
   "year" : 1999,
   "fuelType" : "petrol",
   "brand" : "",
   "engine" : {
        "displacement" : 1000,
        "numberOfCylinders" : 4
   }
}
```

Response: 201

DELETE /model/{id}

Response: 204

PUT /model/{id}

This should support partial update of data i.e. only brand etc.

Request Body:

```
{
    "year" : 1999,
    "fuelType" : "",
    "brand" : "",
    "engine" : {
         "displacement" : 1000,
         "numberOfCylinders" : 4
    }
  }
```

Response: 200

NOTE:

- Invalid request should send 400
- Invalid method should send 405
- Invalid path should send 404
- For any unknown error, return 500 along with error message

3.2 Test Driven Development

Test-driven development (TDD) is a software development process relying on software requirements being converted to test cases before software is fully developed, and tracking all software development by repeatedly testing the software against all test cases. This is opposed to software being developed first and test cases created later.

3.2.1 Layered Architecture

The three independent layers are delivery, use-case and datastore.

- **Delivery Layer:** The delivery layer will receive the request and parse anything that is required from the request. It calls the use case layer, ensures that the response is the required format and writes it to the response writer.
- **Use-Case Layer:** The use case layer does the business logic that is required for the application. This layer will communicate with the datastore layer. It takes whatever it needs from the delivery layer and then calls the datastore layer. Before and after calling the datastore layer, it applies the business logic that is required.
- **Datastore Layer:** The datastore stores the data. It can be any data storage. The use case layer is the only layer that communicates with the datastore. This way each layer can be tested independently without depending on the other.

Since each layer is independent of each other, if the application grows to have gRPC support, only the delivery layer will change. Datastore and use case layer will remain the same. Even when there is a change in datastore, the entire application need not change. Only the datastore layer will change. This way, it is easy to isolate any bugs, maintain the code and grow the application.

Each layer will communicate with each other through interfaces.

- Schema for Engine and Model entities:

```
type Engine struct {
        ID                int `json:"-"`
        Displacement      int `json:"displacement"`
        NumberOfCylinders int `json:"number_of_cylinders"`
        Range             int `json:"range"`
}
```

```
type Model struct {
        ID            int     `json:"id"`
        Name          string  `json:"name"`
        Year          int     `json:"year"`
        FuelType      string  `json:"fuel_type"`
        Brand         string  `json:"brand"`
        EngineDetails *Engine `json:"engine_details,omitempty"`
}
```

- Connecting to mysql:

```
package driver

import (
        "database/sql"
        "fmt"

        _ "github.com/go-sql-driver/mysql"
)

type SQLConfigs struct {
        Host     string
        Username string
        Password string
        Port     int
        Database string
}

func ConnectToDB(c SQLConfigs) (*sql.DB, error) {
        connectionString := fmt.Sprintf("%s:%s@tcp(%s:%d)/%s", c.Username, c.Password, c.Host, c.Port, c.Database)

        db, err := sql.Open("mysql", connectionString)
        if err != nil {
                return nil, err
        }

        return db, nil
}
```

- Defining constants:

```
package constants

type FuelType string

const (
        Electric FuelType = "electric"
        Petrol   FuelType = "petrol"
        Diesel   FuelType = "diesel"
)

type Brand string

const (
        Tesla    Brand = "tesla"
        Porsche  Brand = "porsche"
        Ferrari  Brand = "ferrari"
        Mercedes Brand = "mercedes"
        BMW      Brand = "BMW"
)
```

- Defining custom errors:

```
package errors

import (
        "fmt"
)

type EntityNotFound struct {
        Entity string
}

func (e EntityNotFound) Error() string {
        return fmt.Sprintf("error: %s is invalid; Not Found", e.Entity)
}

type EmptyEntity struct {
        Entity string
}

func (e EmptyEntity) Error() string {
        return fmt.Sprintf("error: %s is empty; Bad Request", e.Entity)
}

type Error string

func (e Error) Error() string {
        return string(e)
}

const (
        ConstantField   Error = "cannot update fixed fields"
        InvalidFuelType Error = "invalid fuel type"
        InvalidYear     Error = "invalid year"
        InvalidEngine   Error = "invalid engine details"
)
```

3.2.2 Tests for Storage Layer

For Storage Layer, we use SQL mocks that mocks the behaviour of mysql-driver for us.

Test for Engine Store:

```go
func TestGetEngineByID(t *testing.T) {
	e := entities.Engine{ID: 110, Range: 4000}
	row := sqlmock.NewRows([]string{"modelID", "displacement", "numberOfCylinders", "model_range"}).
		AddRow(e.ID, e.Displacement, e.NumberOfCylinders, e.Range)

	const query = "SELECT modelID, displacement, numberOfCylinders, model_range FROM engines WHERE modelID = ?"

	id := 110
	db, mock := GetMockDB()
	c := New(db)

	mock.ExpectQuery(query).WithArgs(id).WillReturnRows(row)

	_, err := c.GetEngineByID(id)
	if err != nil {
		t.Error(err)
	}
}
```

```go
func TestInsertEngine(t *testing.T) {
	var testCases = []struct {
		description string
		engine      entities.Engine
	}{
		{
			description: "Successful Insertion",
			engine: entities.Engine{
				ID:               107,
				Displacement:     1000,
				NumberOfCylinders: 40,
			},
		},
	}

	for _, tc := range testCases {
		db, mock := GetMockDB()
		c := New(db)

		mock.ExpectBegin()

		res := sqlmock.NewResult(int64(tc.engine.ID), 1)
		q2 := "INSERT INTO engines"
		mock.ExpectExec(q2).
			WithArgs(tc.engine.ID, tc.engine.Displacement, tc.engine.NumberOfCylinders, tc.engine.Range).
			WillReturnResult(res)

		mock.ExpectCommit()

		err := c.InsertEngine(tc.engine)
		if err != nil {
			t.Errorf("error was not expected while updating stats: %s", err)
		}
	}
}
```

```go
func TestDeleteEngineByID(t *testing.T) {
	var testCases = []struct {
		description string
		modelID     int
	}{
		{description: "Successful Deletion", modelID: 110},
	}

	for _, tc := range testCases {
		db, mock := GetMockDB()
		c := New(db)

		mock.ExpectBegin()

		res := sqlmock.NewResult(int64(tc.modelID), 1)

		query := "DELETE FROM engines WHERE modelID = ?"
		mock.ExpectExec(regexp.QuoteMeta(query)).
			WithArgs(tc.modelID).
			WillReturnResult(res)

		mock.ExpectCommit()

		err := c.DeleteEngineByID(tc.modelID)
		if err != nil {
			t.Errorf("error was not expected while updating stats: %s", err)
		}

		err = mock.ExpectationsWereMet()
		if err != nil {
			t.Errorf("there were unfulfilled expectations: %s", err)
		}
	}
}
```

Test for Model Store:

```
func GetMockDB() (*sql.DB, sqlmock.Sqlmock) {
        db, mock, err := sqlmock.New()
        if err != nil {
                fmt.Println("Failed to open mock sql database: ", err)
        }

        return db, mock
}

func TestGetAllModels(t *testing.T) {
        e1 := entities.Model{ID: 110, Name: "Elsa GT", Year: 2015, FuelType: "electric", Brand: "Tesla"}
        e2 := entities.Model{ID: 112, Name: "Nova NZ", Year: 2020, FuelType: "diesel", Brand: "Porsche"}

        rows := sqlmock.NewRows([]string{"modelID", "name", "year", "fuelType", "brand"}).
                AddRow(e1.ID, e1.Name, e1.Year, e1.FuelType, e1.Brand).
                AddRow(e2.ID, e2.Name, e2.Year, e2.FuelType, e2.Brand)

        query := "SELECT modelID, name, year, fuelType, brand  FROM models"
        db, mock := GetMockDB()
        c := New(db)

        mock.ExpectQuery(query).WillReturnRows(rows)

        _, err := c.GetAllModels()
        if err != nil {
                t.Error(err)
        }

        if err := mock.ExpectationsWereMet(); err != nil {
                t.Errorf("there were unfulfilled expectations: %s", err)
        }
}
```

```go
func TestDeleteModelByID(t *testing.T) {
	var testCases = []struct {
		description string
		modelID     int
	}{
		{description: "Successful Deletion", modelID: 110},
	}

	for _, tc := range testCases {
		db, mock := GetMockDB()
		c := New(db)

		mock.ExpectBegin()

		res := sqlmock.NewResult(int64(tc.modelID), 1)

		mock.ExpectPrepare("DELETE FROM models WHERE modelID = ?").
			ExpectExec().
			WithArgs(tc.modelID).
			WillReturnResult(res)

		mock.ExpectCommit()

		err := c.DeleteModelByID(tc.modelID)
		if err != nil {
			t.Errorf("error was not expected while updating stats: %s", err)
		}

		err = mock.ExpectationsWereMet()
		if err != nil {
			t.Errorf("Expectations not fulfill: %s", err)
		}
	}
}
```

3.2.3 Tests for Service Layer

- **Test GetAll**

```
{description: "GET all Models", output: []entities.Model{
        {
                ID:            110,
                Name:          "Elsa GT",
                Year:          2015,
                FuelType:      "electric",
                Brand:         "Tesla",
                EngineDetails: &entities.Engine{ID: 110, Range: 400},
        },
        {
                ID:            112,
                Name:          "Nova NZ",
                Year:          2020,
                FuelType:      "petrol",
                Brand:         "Porsche",
                EngineDetails: &entities.Engine{ID: 112, Displacement: 1000, NumberOfCylinders: 40},
        },
}},
```

- **Test Get model by ID**

```
        {
                description: "Database Error",
                id:          115,
                output:      entities.Model{},
                err:         goerrors.New("database error"),
        },
        {
                description: "GET Model with valid ID",
                id:          112,
                output: entities.Model{
                        ID:            112,
                        Name:          "Verona",
                        Year:          2020,
                        FuelType:      "electric",
                        Brand:         "Tesla",
                        EngineDetails: &entities.Engine{ID: 110, Range: 400},
                },
        },
        {
                description: "Invalid ID",
                id:          1,
                err:         errors.EntityNotFound{Entity: "id"},
        },
        {
                description: "Empty ID",
                err:         errors.EmptyEntity{Entity: "id"},
        },
```

- **Test Post model**

```
{
        description: "POST Model with Empty Name",
        input: entities.Model{
                ID:            110,
                Name:          "",
                Year:          2015,
                FuelType:      "electric",
                Brand:         "Porsche",
                EngineDetails: &entities.Engine{ID: 110, Displacement: 1000, NumberOfCylinders: 4},
        },
        err: errors.EmptyEntity{Entity: "model name"},
},
{
        description: "POST Model with Invalid FuelType",
        input: entities.Model{
                ID:            110,
                Name:          "Verona",
                Year:          2015,
                FuelType:      "CNG",
                Brand:         "Porsche",
                EngineDetails: &entities.Engine{ID: 110, Displacement: 1000, NumberOfCylinders: 4},
        },
        err: errors.InvalidFuelType,
},
```

- **Test Delete**

```
{description: "Valid Deletion", id: 110},
{description: "Valid Deletion", id: 112},
{
        description: "Deleting non-existing ID",
        id:          90,
        err:         errors.EntityNotFound{Entity: "id"},
},
```

3.2.4 Tests for Handler Layer

- **Test Get All**

```
{
        description: "GET All models with Engine Details",
        url:         "/model/?include=engine",
        resBody: []entities.Model{
                {
                        ID:            110,
                        Name:          "Elsa GT",
                        Year:          2015,
                        FuelType:      "electric",
                        Brand:         "Tesla",
                        EngineDetails: &entities.Engine{Displacement: 1000, NumberOfCylinders: 4, Range: 400},
                },
                {
                        ID:            112,
                        Name:          "Nova NZ",
                        Year:          2020,
                        FuelType:      "petrol",
                        Brand:         "Porsche",
                        EngineDetails: &entities.Engine{Displacement: 1000, NumberOfCylinders: 40},
                },
        }, statusCode: http.StatusOK,
},
{
        description: "GET All models with Invalid URL",
        url:         "/model/?include=",
        statusCode:  http.StatusBadRequest,
},
```

- **Test Get by ID**

```
  {
          description: "GET Request for valid API Endpoint",
          modelID:     "110",
          resp: entities.Model{
                  ID:            110,
                  Name:          "Elsa GT",
                  Year:          2015,
                  FuelType:      "electric",
                  Brand:         "Tesla",
                  EngineDetails: &entities.Engine{Displacement: 1000, NumberOfCylinders: 4, Range: 400}},
          statusCode: http.StatusOK,
  },
  {
          description: "GET Request for Incorrect Model ID",
          modelID:     "010",
          statusCode:  http.StatusNotFound,
  },
```

- **Test Post**

```
{
        description: "Posting a Model with Valid Details",
        req: entities.Model{
                ID:           113,
                Name:         "Carrera GT",
                Year:         1999,
                FuelType:     "petrol",
                Brand:        "Ferrari",
                EngineDetails: &entities.Engine{Displacement: 1000, NumberOfCylinders: 4},
        },
        statusCode: http.StatusCreated,
},
{
        description: "Posting a Model with Invalid Details",
        req: entities.Model{
                ID:           121,
                Name:         "Carrera GT",
                Year:         1999,
                FuelType:     "petrol",
                Brand:        "Ferrari",
                EngineDetails: &entities.Engine{NumberOfCylinders: 4, Range: 400},
        },
        statusCode: http.StatusBadRequest,
},
```

- **Test Update**

```
{
        description: "PUT Request on Valid API Endpoint",
        modelID:     "110",
        input: entities.Model{
                Year:     1999,
                FuelType: "Petrol",
                EngineDetails: &entities.Engine{
                        Displacement:     1000,
                        NumberOfCylinders: 4,
                        Range:            400,
                },
        },
        statusCode: http.StatusOK,
},
{
        description: "PUT Request with empty Model ID",
        input: entities.Model{
                Year:         1999,
                FuelType:     "Petrol",
                EngineDetails: &entities.Engine{Displacement: 1000, NumberOfCylinders: 4}},
        statusCode: http.StatusBadRequest,
},
```

- **Test Delete**

```
{description: "Deleting an existing model", modelID: "113", statusCode: http.StatusNoContent},
{description: "Deleting a non-existing model", modelID: "114", statusCode: http.StatusNotFound},
{description: "Deleting without Model ID", statusCode: http.StatusBadRequest},
{description: "Deleting without Float Type Model ID", modelID: "10.5", statusCode: http.StatusBadRequest},
{description: "Database Error", modelID: "205", statusCode: http.StatusInternalServerError},
```

3.3 Implementation for Storage Layer

```
type Model struct {
        db *sql.DB
}

func New(db *sql.DB) Model {
        return Model{db: db}
}


 func (m Model) InsertModel(model entities.Model) error {
         tx, err := m.db.Begin()
         if err != nil {
                 return err
         }

         defer func() {
                 switch err {
                 case nil:
                         _ = tx.Commit()
                 default:
                         _ = tx.Rollback()
                 }
         }()

         query := "INSERT INTO models (modelID, name, year, fuelType, brand) VALUES (?, ?, ?, ?, ?)"

         _, err = tx.Exec(query, model.ID, model.Name, model.Year, model.FuelType, model.Brand)
         if err != nil {
                 return err
         }

         return err
 }
```

```
func (m Model) DeleteModelByID(id int) error {
	tx, err := m.db.Begin()
	if err != nil {
		return err
	}

	defer func() {
		switch err {
		case nil:
			_ = tx.Commit()
		default:
			_ = tx.Rollback()
		}
	}()

	pre, err := tx.Prepare("DELETE FROM models WHERE modelID = ?")
	if err != nil {
		return err
	}

	_, err = pre.Exec(id)
	if err != nil {
		return err
	}

	defer func() {
		er := pre.Close()
		if er != nil {
			log.Println(er)
		}
	}()

	return err
}
```

3.3 Implementation for Service Layer

```
type Model struct {
        modelStore  stores.Model
        engineStore stores.Engine
}

func New(m stores.Model, e stores.Engine) Model {
        return Model{
                modelStore:  m,
                engineStore: e,
        }
}

func (m Model) ServiceGetAllWithEngine() (modelList []entities.Model, err error) {
        modelList, err = m.modelStore.GetAllModels()
        if err != nil {
                return
        }

        engineList, err := m.engineStore.GetAllEngines()
        if err != nil {
                return
        }

        for i := range modelList {
                modelList[i].EngineDetails = &engineList[i]
        }

        return
}
```

```go
func (m Model) ServiceUpdateModelByID(model entities.Model, id int) (err error) {
	_, err = m.ServiceGetModelByID(id)

	switch {
	case err != nil:
		return err
	case model.ID != 0 || model.Brand != "" || model.Name != "":
		return errors.ConstantField
	case reflect.DeepEqual(model, entities.Model{}):
		return errors.EmptyEntity{Entity: "update body"}
	case model.FuelType != "" && !validFuelType(model.FuelType):
		return errors.InvalidFuelType
	case model.Year != 0 && (model.Year <= 1950 || model.Year >= time.Now().Year()):
		return errors.InvalidYear
	case !validModelType(model):
		return errors.InvalidEngine
	}

	err = m.modelStore.UpdateModelByID(model, id)
	if err != nil {
		return
	}

	if model.EngineDetails == nil {
		return
	}

	return m.engineStore.UpdateEngineByID(*model.EngineDetails, id)
}
```

```
func (m Model) ServiceDeleteModelByID(id int) (err error) {
        _, err = m.ServiceGetModelByID(id)
        if err != nil {
                return
        }

        err = m.modelStore.DeleteModelByID(id)
        if err != nil {
                return err
        }

        return m.engineStore.DeleteEngineByID(id)
}
```

3.5 Implementation for Handler Layer

```
type Model struct {
        service services.Model
}

func New(s services.Model) Model {
        return Model{service: s}
}

func (m Model) GetAllWithEngine(w http.ResponseWriter, r *http.Request) {
        include := r.FormValue("include")
        if include != "engine" {
                http.Error(w, http.StatusText(http.StatusBadRequest), http.StatusBadRequest)
                return
        }

        models, err := m.service.ServiceGetAllWithEngine()
        if err != nil {
                w.WriteHeader(http.StatusInternalServerError)
                writeResponse(w, []byte(err.Error()))

                return
        }

        resp, err := json.Marshal(models)
        if err != nil {
                w.WriteHeader(http.StatusInternalServerError)
                return
        }

        writeResponse(w, resp)
}
```

```go
func (m Model) PostModel(w http.ResponseWriter, r *http.Request) {
	var model entities.Model

	body, err := ioutil.ReadAll(r.Body)
	if err != nil {
		w.WriteHeader(http.StatusBadRequest)
		return
	}

	err = json.Unmarshal(body, &model)
	if err != nil {
		w.WriteHeader(http.StatusBadRequest)
		writeResponse(w, []byte(err.Error()))

		return
	}

	err = m.service.ServicePostModel(model)

	if err == nil {
		w.WriteHeader(http.StatusCreated)
		return
	}

	writeErrorResponse(w, err)
}
```

```
func (m Model) DeleteModelByID(w http.ResponseWriter, r *http.Request) {
        var err error

        vars := mux.Vars(r)
        id := vars["id"]

        var i int

        if id != "" {
                i, err = strconv.Atoi(id)
                if err != nil {
                        w.WriteHeader(http.StatusBadRequest)
                        writeResponse(w, []byte("error in parsing the ID"))

                        return
                }
        }

        err = m.service.ServiceDeleteModelByID(i)
        if err == nil {
                w.WriteHeader(http.StatusNoContent)
                return
        }

        writeErrorResponse(w, err)
}
```

```go
func (m Model) UpdateModelByID(w http.ResponseWriter, r *http.Request) {
	var err error

	vars := mux.Vars(r)
	id := vars["id"]

	var i int
	if id != "" {
		i, err = strconv.Atoi(id)
		if err != nil {
			w.WriteHeader(http.StatusBadRequest)
			writeResponse(w, []byte("error in parsing the ID"))

			return
		}
	}

	var model entities.Model

	body, err := ioutil.ReadAll(r.Body)
	if err != nil {
		w.WriteHeader(http.StatusBadRequest)
		return
	}

	err = json.Unmarshal(body, &model)
	if err != nil {
		w.WriteHeader(http.StatusBadRequest)
		writeResponse(w, []byte(err.Error()))

		return
	}

	err = m.service.ServiceUpdateModelByID(model, i)
	if err == nil {
		return
	}

	writeErrorResponse(w, err)
}
```

3.6 Communicating Using Interfaces

```
package stores

import (
        "github.com/zopsmart/golang-training/aakanksha-jaiswal-zs/car-dealership-project/entities"
)

type Engine interface {
        GetAllEngines() ([]entities.Engine, error)
        GetEnginesByIDList([]int) ([]entities.Engine, error)
        InsertEngine(entities.Engine) error
        GetEngineByID(int) (entities.Engine, error)
        DeleteEngineByID(int) error
        UpdateEngineByID(entities.Engine, int) error
}

type Model interface {
        GetAllModels() ([]entities.Model, error)
        GetModelsByBrand(string) ([]entities.Model, []int, error)
        InsertModel(entities.Model) error
        GetModelByID(int) (entities.Model, error)
        DeleteModelByID(int) error
        UpdateModelByID(entities.Model, int) error
}
```

```
package services

import "github.com/zopsmart/golang-training/aakanksha-jaiswal-zs/car-dealership-project/entities"

type Model interface {
         ServiceGetAllWithEngine() ([]entities.Model, error)
         ServiceGetAllWithoutEngine() ([]entities.Model, error)
         ServiceGetByBrandWithEngine(string) ([]entities.Model, error)
         ServiceGetByBrandWithoutEngine(string) ([]entities.Model, error)
         ServiceGetModelByID(int) (entities.Model, error)
         ServicePostModel(entities.Model) error
         ServiceDeleteModelByID(int) error
         ServiceUpdateModelByID(entities.Model, int) error
}
```

main.go

```
main() {
    const PORT = 3306

    configurationDB := driver.SQLConfigs{
            Host:     "localhost",
            Username: "root",
            Password: "123",
            Port:     PORT,
            Database: "car_dealership",
    }

    db, err := driver.ConnectToDB(configurationDB)
    if err != nil {
            log.Println("could not connect to sql, err:", err)
            return
    }

    defer func() {
            err := db.Close()
            if err != nil {
                    log.Println(err)
            }
    }()

    modelDataStore := model.New(db)
    engineDataStore := engine.New(db)
    sr := services.New(modelDataStore, engineDataStore)
    h := handlers.New(sr)

    r := mux.NewRouter()

    r.HandleFunc("/model", h.GetByBrandWithEngine).Queries("brand", "{name}", "include", "{engine}").Methods(http.MethodGet)
    r.HandleFunc("/model", h.GetByBrandWithoutEngine).Queries("brand", "{name}").Methods(http.MethodGet)
    r.HandleFunc("/model", h.GetAllWithEngine).Queries("include", "{engine}").Methods(http.MethodGet)
    r.HandleFunc("/model", h.GetAllWithoutEngine).Methods(http.MethodGet)
    r.HandleFunc("/model/{id:[0-9]*}", h.GetModelByID).Methods(http.MethodGet)
    r.HandleFunc("/model", h.PostModel).Methods(http.MethodPost)
    r.HandleFunc("/model/{id:[0-9]*}", h.DeleteModelByID).Methods(http.MethodDelete)
    r.HandleFunc("/model/{id:[0-9]*}", h.UpdateModelByID).Methods(http.MethodPut)
    r.Use(middlewares.AuthenticationMiddleware)

    srv := &http.Server{
            Handler: r,
            Addr:    "192.168.1.68:8080",
    }

    log.Fatal(srv.ListenAndServe())
```

Chapter 04: LEARNING AND CONCLUSION

4.1 Discussion on the Results Achieved

The following results were achieved upon the implementation of this project:

- All tests were passed, include the edge cases
- Code coverage was 100%

4.2 Afterworks

- From this Project we learnt principles of system designing, architectures, patterns and embrace the power of golang when it comes to developing scalable backend systems,
- After this project, I was assigned to the SmartStore Ecommerce platform wherein I migrated a php service to golang, using the same design principles and architecture.