Web Application in Go using Three Layered Architecture

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

in
Computer Science and Engineering

By
Kashish Gupta (181476)

UNDER THE SUPERVISION OF
Dr. Rajni Mohana

Department of Comp

uter Science & Engineering and Information Technology

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

DECLARATION

I hereby declare that this project has been done by me under the supervision of Dr.Rajni Mohan**,** **Affiliation,** Jaypee University of Information Technology. I also declare that neither this project nor any part of this project has been submitted elsewhere for award of any degree or diploma.

Supervised by:
Dr.Rajni
Assistant Professor (SG)
Department of Computer Science & Engineering and Information Technology
Jaypee University of Information Technology

Submitted by:
Kashish Gupta(181476)
Computer Science & Engineering Department
Jaypee University of Information Technology

CERTIFICATE

This is to certify that the work which is being presented in the project report titled **“Three Layer Architecture in GO”** in partial fulfillment of the requirements for the award of the degree of B.Tech in Computer Science And Engineering and submitted to the Department of Computer Science And Engineering, Jaypee University of Information Technology, Waknaghat is an authentic record of work carried out by Kashish Gupta (181476) during the period from 7 Feb & 2022 to 29 May under the supervision of **Dr.Rajni,** Department of Computer Science and Engineering, Jaypee University of Information Technology, Waknaghat.

Kashish Gupta
(181262)

The above statement made is correct to the best of my knowledge.

[illegible]

Dr.Rajni
Assistant Professor (SG)
Computer Science & Engineering and Information Technology
JUIT, Waknaghat

[illegible]

Mithali R Shetty
Senior Lead Engineer
Zopsmart Technology

Dated: 27-05-21

ACKNOWLEDGEMENT

Foremost, I would like to express my heartiest gratefulness to almighty God for his divine blessing that made it possible for mr to complete the project successfully.

I am extremly grateful to my supervisor, **Dr.Rajni Mohana**, Assistant Professor (SG) , Department of CSE Jaypee University of Information Technology, Waknaghat, for her assistance. To complete this assignment, my supervisor has extensive knowledge and a deep interest in the subject of Web development. His never-ending patience, intellectual direction, constant encouragement, constant and energetic supervision, constructive criticism, good suggestions, and reading many poor versions and fixing them at all stagesmade it possible to finish this job.

I'd like to thank **Dr.Rajni Mohana**, Assistant Professor (SG) , Department of CSE, for her invaluable assistance in completing my project.

I would also like to express my gratitude to everyone who has directly or indirectly assisted me in making this project a success. In this unique scenario, I'd want to appreciate the different staff members, both teaching and non-teaching, who have developed their helpful assistance and facilitated my project. Finally, I must express my gratitude for my parents' unwavering support and patience.

Kashish Gupta

TABLE OF CONTENTS

REFERENCES

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Creating a web application is quite simple but the challenge comes when the code has to be tested, structured, cleaned and maintained and thus here we follow the Three Layered Architecture using Go language.

The three layers are handler, service and datastore which are all independent of each other. The handler layer receives request body and then parse anything that is required from that request. It then calls the service layer where all the logic of program is defined, ensures that the response is the required format and writes it to the response writer. This layer further communicates with the datastore layer. It takes whatever it needs from the handler layer and then calls the datastore layer. The datastore layer is where all the the data is stored. It can be any data storage. The use case layer is the only layer that communicates with the datastore. That is how we test each layer independently making sure that no layer affect other.

CHAPTER 1 : INTRODUCTION

1.1 Company

ZopSmart is a softaware solution technology company that provides you with all the tools to build your e-commerce business, . ZopSmart has a suite of products that will help you build the perfect business you're aiming to open and run. It has different products such as Smart Store Eazy, Smart Payment Gateway, etc. Zopsmart is building next generation technology for the retail sector and their customers range from a small furniture shop to multinational retail chains and solutions include an e-commerce platform,Digital Marketing , m-Commerce, automated logistics systems, management platform, order management platform, and iOT devices. It also provide software solutions to some of the top-most firms and has it's ownframework to work on.

1.2 Introduction

It is a basic web application that implements CRUD operations based on the three layered architecture. Programs at each layer have their own unit test. There is also an implementation of middleware that authenticates the http request before sending it to the server.

1.3 Objectives

To create testable, structured, clean and maintainable web applications by using industrial best practices.

1.4 Motivation

To apply industrial best practices and create a fast, scalable and secure web application.

1.5 Libraries/Frameworks Used

GO - *A*n open source programming language developed by Google engineers aim to build simple, reliable, and efficient code for applications.
GO provides various packages that are used in this project such as:

1. **net/http**: This package provides http client/server implementations.
2. **json**: Encoding and decoding of JSON is implemented by this package.
3. **errors**: Manipulation of errors is implemented by this function.
4. **database/sql**: This package provides a SQL-like databases.

And for unit testing: gomock and sqlMock is used

MOVING FORWARD WITH GO

All the backend framework such as implementing http request, sending response to server, writing program logic etc is written in Go.

1.5 Technical Requirements

- **GOLand** is an IDE to write clean code .
- **Postman** API platform for building and using APIs.
- **Mysql** server provides a database management system with querying and connectivity capabilities

1.5.1 Hardware Configuration

Table 1 : Hardware Configuration

Processor	Apple M1 chip, 8-core CPU
RAM	8 GB
Hard Disk	256 GB SSD
Monitor	13”
Mouse	
Keyboard	

1.5.2 Software Configuration

Table 2 : Software Configuration

Operating System	Ubuntu
Language	GO
Runtime environment	GO runtime
Package Manager	GO

CHAPTER 2 : LITERATURE SURVEY

1) **GO Documentation**
   Designed at google by R Griesemer, R Pike, and K Thompson, go is a statically typed, open source, compiled programming language.

2) **MySQL Documentation**

   MySQL i.e My Structured Query Language is an open-source relational database management system that helps us to store data, fetch details, delete an entry etc.

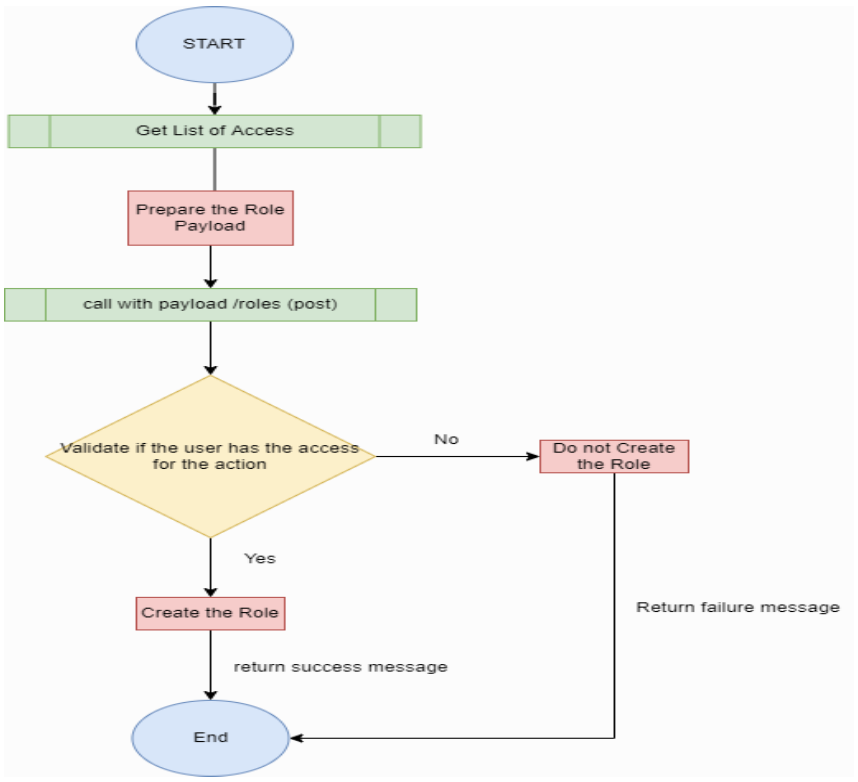
3) **GoMock**

   gomock is a mocking framework for the GO programming language and is used to integrate well with Go's built-in testing package.

4) **Git and Github**

   Official documentary that familiarizes you with the concepts of a version control system i.e Git and how it works with GitHub.

CHAPTER 3 : System Design Diagram



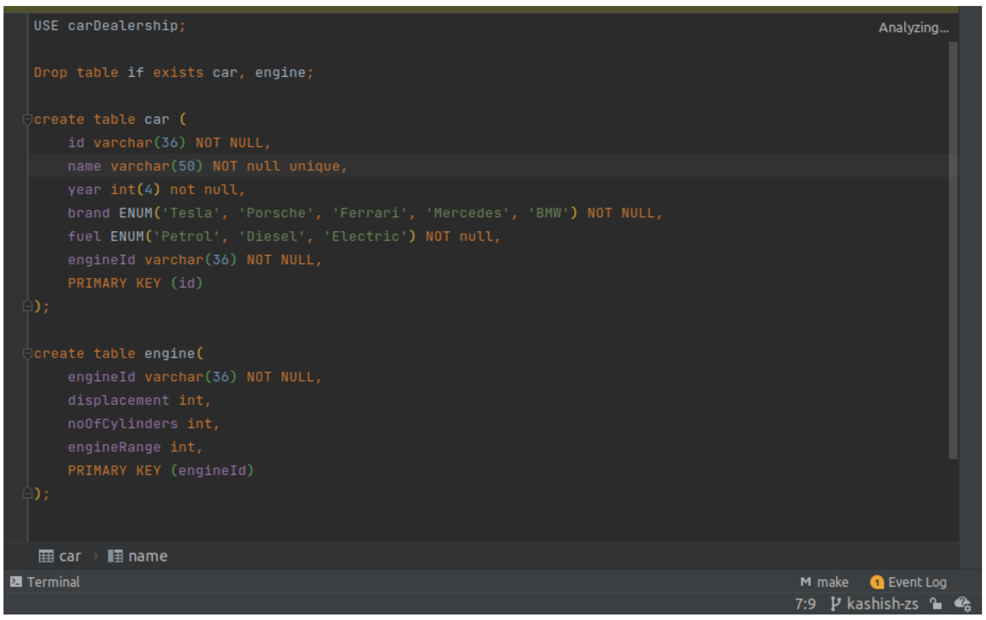
CHAPTER 4 : IMPLEMENTATION

4.1 Identification of features

The web application features:

- Creation of an entry with car and engine details
- Updation of the existing system
- Deletion of an existing entry
- Fetching details based on car/ engine ID
- Fetching details based on car brand name
- Fetching details based on car brand name and availability of engine details

4.2 SQL Schema



```sql
USE carDealership;

Drop table if exists car, engine;

create table car (
    id varchar(36) NOT NULL,
    name varchar(50) NOT null unique,
    year int(4) not null,
    brand ENUM('Tesla', 'Porsche', 'Ferrari', 'Mercedes', 'BMW') NOT NULL,
    fuel ENUM('Petrol', 'Diesel', 'Electric') NOT null,
    engineId varchar(36) NOT NULL,
    PRIMARY KEY (id)
);

create table engine(
    engineId varchar(36) NOT NULL,
    displacement int,
    noOfCylinders int,
    engineRange int,
    PRIMARY KEY (engineId)
);
```

4.3 Study Material

LINUX

Unix based OS with both command line interface and graphical user interface.
BASH(Bourne Again Shell) is a Unix shell and is the default shell in Linux.

Package manager is used for installing, upgrading and cleaning packages. Default package manager of Ubuntu is apt-get: *Advanced Packaging Tool* (APT), SUDO: **superuser do or substitute user do**

There are several processes running in the system environment, there are several variables set in the environment too called Environment variables and they affect the processes using them. If the value of these variables changed, processes using them will be affected.
We can set environment variables in ~/.bashrc or ~/.bash_profile. These are hidden files (hence start with a dot). Both are present in the home directory(~).

PATH: specifies the locations to be searched to find a command.

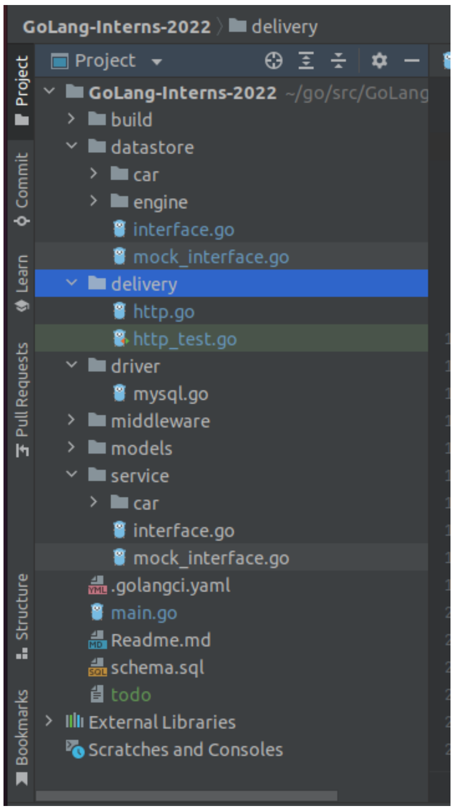
LINUX Commands:

1. ls: lists the files in a folder.
2. cd: change directory
3. touch: used to create an empty file.
4. pwd: print path to the folder we are currently working
5. mkdir: creates a folder or multiple folders
   To create nested folder -p mkdir folder/new
6. rmdir/rm: to delete an empty folder/ rm -rf folder(if another files/folder inside)
7. mv: move/ rename files cat: concatenate -> lists the contents of file

8. chmod: It sets the file permissions flags(define who can read, write to or execute the file) on a file or folder.
9. vi: visual editor

Go Workspace

It's a hierarchy with two directories as it's root:
1. src: containing goo source files,
2. bin: containing executable commands



```
GoLang-Interns-2022 〉 delivery
Project ▾
⌄ GoLang-Interns-2022 ~/go/src/GoLang
  › build
  ⌄ datastore
    › car
    › engine
      interface.go
      mock_interface.go
  ⌄ delivery
      http.go
      http_test.go
  ⌄ driver
      mysql.go
  › middleware
  › models
  ⌄ service
    › car
      interface.go
      mock_interface.go
    .golangci.yaml
    main.go
    Readme.md
    schema.sql
    todo
› External Libraries
  Scratches and Consoles
```

GOPATH environment variable: It specifies the location of your workspace.
GOROOT is **a variable that defines where your Go SDK is located**.

GO Packages

Each and every go program is made of packages. All the program in go enviroment start running in the main package
math/rand:- In package rand, environment is deterministic i.e. when run rand.In return same number, and if we want different results each time we use, rand.Seed
With import use ()-for clarity[Factored statement] and " " with packages
When exporting names use Capital letter with its package- ex: Pi(math.Pi)
We can use fmt: formatted i/o package to format all this.

Functions:func()

In GO a function can take 0 or more agrs. The type of a function comes after variable name

func add(a int, b int) int{ return a+b}

To call a function- *func main(){ add(9,2)}*

ex- *add(int a, int b): add(a, b int)*

A function can return any number of results

ex- func swapString(str1, str 2string) (string, string)

{ return str2,str1}

func main(){ str1,str2:=swap("A","B") } [:= assignment operator]

Import:

In go we import all the paxkges and libraries in alphabetic order.
First all the inbuilt packages are written followed by third party packages.

File Watchers:

It is a tool that is used to Imports all packages, formatting: Correct all the indentation, makes code standard and clean, etc.

Variables

We use var to declare a variable or a list of variables. The variables can be at package or function level

declaration: var i int; default int=0, bool=false

We can include initializer one per var.

```
Ex: var i, j int=4,2
    var k, j= false, "no!"
```

Inside a function, we use := for short assignment statement can be used in place of var decleration with implicit type but outside a function, we can't use :=; we have to use var, func,etc.

```
Ex:func main(){
var m, n int=3,2
k,n:= true,"no!"
}
```

White creating variables we should always avoid global variables

Variable declared without an explicit initial value are 0, numeric; false for boolean and "" for string

Type Conversion

Type Converts say integer to float or vice-versa i.e convers obe var type to other.

eg: var j int=32

f:= float64(j)

Type Interface

If we have a declaration on rhs, the new var takes the same type.
Ex- var i int
    j:= i
But when not specified, say v:=42; then the type depends on precision.

Constants

We define constants using const.
Constasts can be char(character), string, boolean, or numeric values.
They cannot be declared using :=
Numeric const have high precision values

FOR

In go we have only for; no while or do while loop

1. for loop:

for i:= 0; i<20; i++

2. for loop behaving like a while loop:

for ; j<20; —> j>=1
for i<10→i>=1

3. for loop behaving like do while loop

for v=0;v<10; here v is local to for

IF

In if statements of fo we do not need to use () but {} is required.
If with short statements meaning if can start with a short statement before excution but it should end with a semicolon.
Variables that are declared insid an if short statement are also available inside any of the else blocks.

SWITCH

Switch of go works a little differently. Go runs only selected cases, not all cases that follows
In go, break statements are not required after every case.
Switch cases need not be constant.
In switch values can be anything, not specifically an integer.
In go, switch evaluates from top to bottom
Switch without a condion is same as true- if,if-else
We cannot have two cases with same condition ie. no duplicate case or it gives us type mismatch error or compile time error.

DEFER

A defer statement defers the execution of a function until the surrounding function returns. The deferred call's arguments are evaluated immediately, but thefunction call is not executed until the surrounding function returns.
Defer calls are pushed in a stack and thus it prints in LIFO, last in first out order.
Use defer keyword to define defer statements.

POINTERS

Pointers holds the memory address of any value that is provided to it.
*K, a pointer to K value, *pointer value
To dereferencing/ indirecting, *k=28 ()
Pointer’s zero value is nil.
declaration: var k* float, for example
& operator generates a pointer to its operand: k=&i→ k=*p

STRUCTS

Structs are collection of fields
ex: type V struct{

```
        X int

        Y int }

        Func main(){

        print(V{1,2})

        } - basically prints 1,2
```

Struct fields are accessed using v= V{11,12}.
To access an struct field we use a struct pointer.
var (

```
  v1 = Vtx{11,1 2}  // has type Vertex

  v2 = Vtx{X: 11}  // Y:0 is implicit

  p  = &Vtx{11, 12} // has type *Vertex

)
```

ARRAYS

Arrays are where we can store a collection of elements. Array length is part of its type i.e. cannot be resized

to declare and array :
Arr[] int
Arr=[] int {1,2}

SLICES

In go we use slices as dynamically sized array
declaration S [low:high]; incl. low
Slices are like references to array
It doesn't store any data, we just describe a section of an array
Changing the elements of a slice modifies the corresponding elements of its underlying array and is reflected in other slices that share common elements.
Eg: a:= names[1:2]; names-array
A slice literal is an array literal without the length.
A slice structure, internally contains a pointer, length and a capacity field.
len(s): length of slice or we can say number of elements in slice.
cap(s): capacity of slice or we can say number of elements in underlying array
Zero value of slice is nil. and thus length and capacity is nil i.e len=cap=0
Slice can be made with build-in func make that is how we create dynamic sized array.
Make creates zeroed array and returns slice that refers to the array.
Ex: a:=make([]int , len)
    a:=make([]int, len, cap)
Slice can contain any type including other slices. We can also append elements in a slice using append(slice, element1,element2…).
If the capacity of the slice is less than the no. of elements to be appended, it automatically doubles the capacity. : cap(s)+1)*2: and creates a new slice, new memory is allocated and changes are not reflected on underlying array.

RANGE

A Range is a form of for loop that iterates over a slice/map.
For each iteration it returns an index and copy of value at that index.
We can also skip the index or value by assigning _.
Syntax:
  for j, _ := range power
  for _, value := range power
If we only want index we can omit the second variable
for j:=range power

<< : times 2
>>: divide 2
For example, 1 << 5 is "1 times 2, 5 times" or 32. And 32 >> 5 is "32 divided by 2, 5 times" or 1.

MAPS

A map maps keys to values
Zero map has value nil. A nil map has no keys
If the top-level type is just a type name, you can omit it from the elements of the literal.

Ex: var m map[string]Vertex

```
func main() {
   map = make(map[string]Vertex)
   map["B Labs"] = Vertex{
         40.68423, -74.39867,
   } ; Or we can omit vertex
```

Mutating maps
To insert/update: m[key]=elem
To retrieve : elem=m[key]
To delete: delete(m,key) : to delete a key
To test if key is present: elem, ok=m[key] ; ok=true if key present else false
If we can’t find key in the map, then elem is the zero value for the map's element type.
A key in any given map cannot be slice.

PANIC: run-time error

Variadic Fuctions

fmt.Println: It is an empty interface
Variadic PARAMETER: …: can pass 0 or more values and can be of any type
In an argument list, variadic is last variable

Function Values

Functions are values too which can be passed around just like other values.
Function values may be used as function arguments and return values.
When it comes to Go function, it maybe *closure*.
A closure is a function value that references variables from outside its body.
The function may access and assign to the referenced variables; in this sense the function is "bound" to the variables.

METHODS

A method is like a function with a special receiver argument.
We can define a method with a receiver whose type is defined in the same package as the method.
The receiver appears in its own argument list between the func keyword and the method name.

We can define methods on type.
We can define methods on non-struct types also.
You can only declare a method with a receiver whose type is defined in the same package as method including built in types like int.
There are two reasons to use a pointer receiver:
The first is so that the method can modify the value that its receiver points to.
The second is to avoid copying the value on each method call. This can be more efficient if the receiver is a large struct.
All methods on a given type should have either value or pointer receivers, but not a mixture of both.

Receiver Arguments:

- Value receiver argument can only reference methods with value receiver whereas pointer receiver argument references methods with both value and pointer receiver: METHOD SETS
- We use value receivers when we don’t want changes to be reflected in the original value, while using slices, maps, etc
- We can use a pointer receiver when we want the changes to be reflected or when we want to access methods either way or when the struct is quite large to avoid duplicate copies.

INTERFACES

Interface type is defined as method signature of a particlare underlying base.

A value of interface type can hold any value that implements those methods i.e same methods with different type is implemented by interface]

Syntax: type name interface{}

Interfaces are implemented implicitly. There is no explicit declaration of intent, no "implements" keyword.

Zero value of interface is nil.

Abstract type underlying which is our concrete type (struct, float, etc): can be thought of as a tuple of a value and a concrete type: (value, type)

In case of pointer receiver: (&{Hello}, *main.T)

If the concrete value inside the interface itself is nil, the method will be called with a nil receiver & doesn’t trigger a null pointer exception.
Interface value that holds a nil concrete value is itself non-nil.
A nil interface value holds neither value nor concrete type.
Callin a method on a nil interface is a run-time error because there is no type inside the interface tuple to indicate which *concrete* method to call.
The interface type that specifies zero methods is known as the *empty interface*: interface{}

An empty interface may hold values of any type. Every type implements at least zero methods. Empty interfaces are used by code that handles values of unknown type.
var i interface{}
i=42 (type->int)

TYPE ASSERTION

A *type assertion* provides access to an interface value's underlying concrete value.

t:=i.(T): This statement asserts that the interface value i hold the concrete type T and assigns the underlying T value to the variable t. If I do not hold a T, the statement will trigger a panic.

A type assertion can return two values: the underlying value and a boolean value that reports whether the assertion succeeded.
t, ok := i.(T)-> If i holds a T, then t will be the underlying value and ok will be true. If not , ok will be false and t will be the zero value of type T, and no panic occurs.

Type SWITCH

A *type switch* is a construct that permits several type assertions in series.
A types witch is like a regular switch statement,but the cases in a type switch specify types (not values), and those values are compared against the type of the value held by the given interface value.

```
switch s := i.(type) {
case T:
   // here s has type A
case S:
   // here s has type B
default:
   // no match; here s has the same type as i
}
```

The declaration in a type switch has the same syntax as a type assertion i.(T), but the specific type T is replaced with the keyword type.

STRINGERS

It is an ubiquitous interfaces defined by the format(fmt) package.

```
type Stringer interface {

   String() string

}
```

a type that can describe itself as a string.

Readers

The io package specifies the io.Reader interface, which represents the read end of a stream of data.
The io.Reader interface has a Read method:

```
func (T) Read(b []byte) (n int, err error)
```

Read populates the given byte slice with data and returns the number of bytes populated and an error value. It returns an io.EOF error when the stream ends.

Type Image:

The actual struct that implements the Image interface is the RGBA type.

REST- REpresntation State Transfer

A REST API (also known as a RESTful API) is an application programming interface (API or web API) that allows users to interact with RESTful web services while adhering to the REST architectural style's restrictions. REST, which stands for representational state transfer, was created by computer scientist Roy Fielding.
Principles of RESTful Design
Decoupling of client and server - Client and server programmes must be totally independent of one another in a REST API architecture. Only the URI of the requested resource should be known by the client software; it cannot connect to the server application in any other way. A server application should not change the client software except to provide it with the necessary data over HTTP.

Due to the statelessness of REST APIs, each request must include all of the data required to process it. REST APIs, in other words, do not require any server-side connectivity. Any data relating to a client request cannot be saved by server programmes.
Cacheability - Whenever practical, resources should be cacheable on both the client and server sides. Server responses must also indicate if the requested resource can be cached. The goal is to boost server-side scalability while improving client-side performance.

REST API calls and responses pass through multiple layers in a layered system architecture. In most circumstances, client and server programmes will communicate indirectly. In the communication loop, there could be several different middlemen. REST APIs must be built in such a way that neither the client nor the server can access them.

Response Status Codes

1. 200:OK, Success
2. 201: Success+Created
3. 202: Accepted, request received but not completed
4. 204: No content
5. 400: Bad Request, incorrect syntax
6. 404: Not found
7. 405: Method Not Allowed
8. 500: Internal Server Error

HTTP package

The http package provides a client and a server. The server is made of handlers. The handler takes a request and based on that it returns a response.

1. HTTP protocols

Create : Post-> new data
Read : Get-> retrieve data
Update: Put-> update data
Delete: Delete-> delete data

2. ServeMux(Multiplexer)

- ServeMux is an HTTP request multiplexer.
- Responsible for matching URLs in request to an appropriate handler and executing it.

http.NewServerMux.[url handler using Handle and HandleFun methods]

Handle Method:

- It accepts a String and an http.Handler
- Func (mux *ServeMux) Handle(pattern string, handler Handler)
- http.Handler is an interface (second parameter in the Handle method) with the ServeHTTP method

```
type Handler interface {
   ServeHTTP(ResponseWriter, *Request)
}
Eg: func (h home) ServeHTTP(rw http.ResponseWriter, r *http.Request) {
        rw.Write([]byte("Welcome to the Just Enough Go! blog series!"))
}
mux := http.NewServeMux()
mux.Handle("/", home{})
```

- HandleFunc accepts the handler implementation in the form of a function (along with the path for which it is to be invoked).

func (mux *ServeMux) HandleFunc(pattern string, handler func(ResponseWriter, *Request))

Server (HTTP server)

server := http.Server{Addr: ":8080", Handler: mux} -> Addr is the address on which the server listens e.g. http://localhost:8080 and Handler is actually an http.Handler instance.

- If you just had a route or path which you wanted to handle, you can pass an instance of an http.Handler (e.g. home{} in this case) and skip the ServeMux altogether.
- You can/should pass an instance of a ServeMux so that you can handle multiple routes/paths (e.g. /home, /items etc.) Internally, it works by dispatching or routing to the appropriate handler based on the path (URL) in http.Request.

func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Reques

- DefaultServeMux: We don't need to use an explicit ServeMux. The Handle and HandleFuncmethods available in a ServeMux are also exposed as global functions in the net/http package.

- To start the HTTP server, you can use the http.ListenAndServe function, just as you would with a Server instance.

func ListenAndServe(addr string, handler Handler) error

- The handler parameter can be nil if you have used http.Handle and/or http.HandleFunc to specify the handler implementations for the respective routes.

Functions as handlers

type HandlerFunc func(ResponseWriter, *Request)
HandlerFunc allows you to use ordinary functions as HTTP handlers. For example:

```
func welcome(rw http.ResponseWriter, req *http.Request) {
        rw.Write([]byte("Welcome to Just Enough Go"))
}
```
Or http.ListenAndServe(":8080", http.HandlerFunc(welcome))

Note: HandlerFunc(f) is a Handler that calls the function f

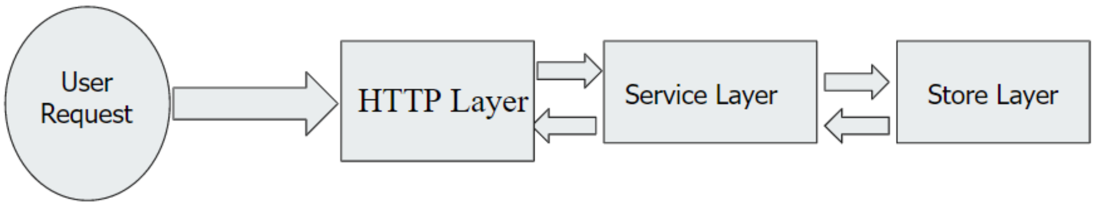
HTTPtest Package

1. httpRequest: httptest.NewRequest- returns a new incoming server request, suitable for passing to an http.Handler for testing.
2. httpResponseWriter: w=httptest.NewRecorder type which returns httptest.ResponseRecorder- ResponseRecorder is an implementation of http.ResponseWriter that records its mutations for later inspection in tests; an be used to be passed into our server

handler, record all the data that the handler will write to the response and return the data written afterwards.

3. w.Result(): func (rw *ResponseRecorder) Result() *http.Response- Result returns the response generated by the handler.The returned Response will have at least its StatusCode, Header, Body, and optionally Trailer populated.

Layered Architecture: Layers are independent of each other and communicate with each other through interface.



Basically this helps us make our application modular, readable and maintainable.

This has 3 layers - HTTP layer, Service layer, Store layer.

1. HTTP layer : validates query/path parameters, request body, header checks.
2. Service layer : Implements business logic and communicates with datastore layer
3. Store layer : Implements database level queries.

   1. Each layer communicates with its previous/next layer using an interface(methods with input parameters and output types are defined).
   2. Testing of each layer is done by mocking its interface/DB/Server based on necessity.

Dependency Injection:

It is a style of writing code such that at the time the object is initialization the dependencies of a particular object/struct are provided.
We can explicitly choose when to create new instances of our dependencies and when to reuse the same instance.
Our structs no longer have the responsibility for building their dependencies thus making our structs less tightly coupled to their dependencies.

Factory method

It is a design pattern which solves the problem of creating product objects without specifying their concrete classes. It defines a method, which is used to create objects instead of direct calling new operator.

1. **Simple factory**
2. **Interface factories**.

MICRO SERVICES

Microservices is an architectural and organizational approach to software development that consists of small, independent services that communicate over well-defined APIs. Small, self-contained teams own and operate these services.

Microservices designs make it easier to expand and develop applications, allowing for more creativity and faster time-to-market for new features.

Benefits of micro services

- **Flexible Scaling**

  Each microservice can be scaled independently to meet demand for the app feature it serves. This enables teams to correctly size infrastructure, accurately estimate the cost of a feature, and maintain service availability during peak demand periods.

- **Easy Deployment**

  Microservices enable continuous integration and delivery, making it easy to try out new ideas and roll back if they don't work out. More experimentation, quicker code modifications, and faster time-to-market for new features are all possible thanks to the low cost of failure.

- **Reusable Code**

  Teams can use functions for many purposes by dividing software into discrete, well-defined modules. A service created for one function can be used as a foundation for another feature. This allows an application to self-bootstrap since developers may add new features without having to write code from scratch.

Database Migration

Developers are responsible for building, maintaining, and improving applications- this could need you to change or update the database structures. **Migration gives you the ability to handle these changes easily and consistently in an active development environment.** The more you know about shaping your database, the better equipped you'll be to create an effective and concise database for your application.

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

Migrations act as a version control system for your database, allowing your team to define and share the database schema definition for the application.

There are two methods in a migration class: up and down. The up method of your migration should specify the schema modification you want to do, and the down method should reverse the alterations made by the up method. In other words, if you conduct an up followed by a down, the database schema should remain identical. If you make a table in the up method, for example, you should dump it in the down method.

When migrating up the database – forward in time – the up method is used, whereas when migrating down the database – back in time – the down approach is used. Thus, can go back and forth to older and newer versions of our database.

PROMETHEUS

Prometheus is an open-source monitoring and alerting tool created to monitor a highly dynamic container enviroment in real-time. It can also be used for a traditional (non-container) bare server in which applications are directly deployed.

ARCHITECTURE

PROMETHEUS SERVER: It does the actual monitoring work.

- Time Series Database: stores metrics data (*storage*)
- Data Retrieval Worker: Pulls data from metrics, applications, servers, etc. (*retrieval*)
- Accepts PromQL queries: consumed by external systems via the HTTP api.

PROMETHEUS MONITORS: It monitors single application, apache server, linux/windows server etc called *targets*.

TARGET: The target units such CPU status, memory/disk storage space, exceptions counts, request count/duration, etc are monitored.

MATRICS: Unit that is monitored for a specific target. It is defined as human-readable, text based and have two entities:

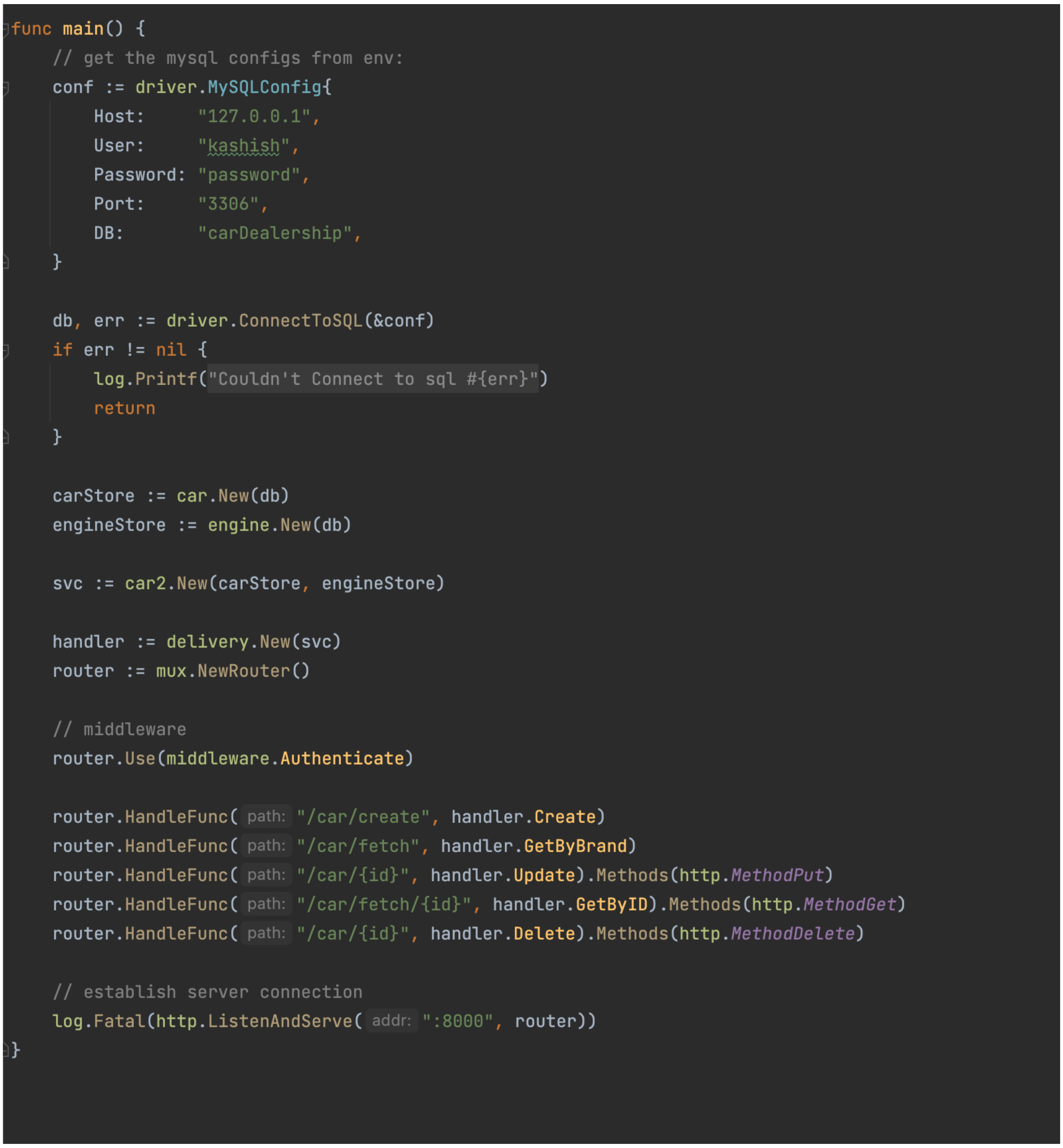
- Help: Description of what metrics is.
- Type: 4-types-
    1. Counter
    2. Summary
    3. Gauge
    4. Histogram

EXPORTER: Some servers already expose prometheus endpoints therefore don’t need extra service to gather metrics but many services need another component called *exporter.*

It’s a script or service that fetches metrics from target and converts it to correct format that prometheus understands and exposes it to its own/matric endpoint where prometheus can scrape them.

CODE

main.go : source file all routers



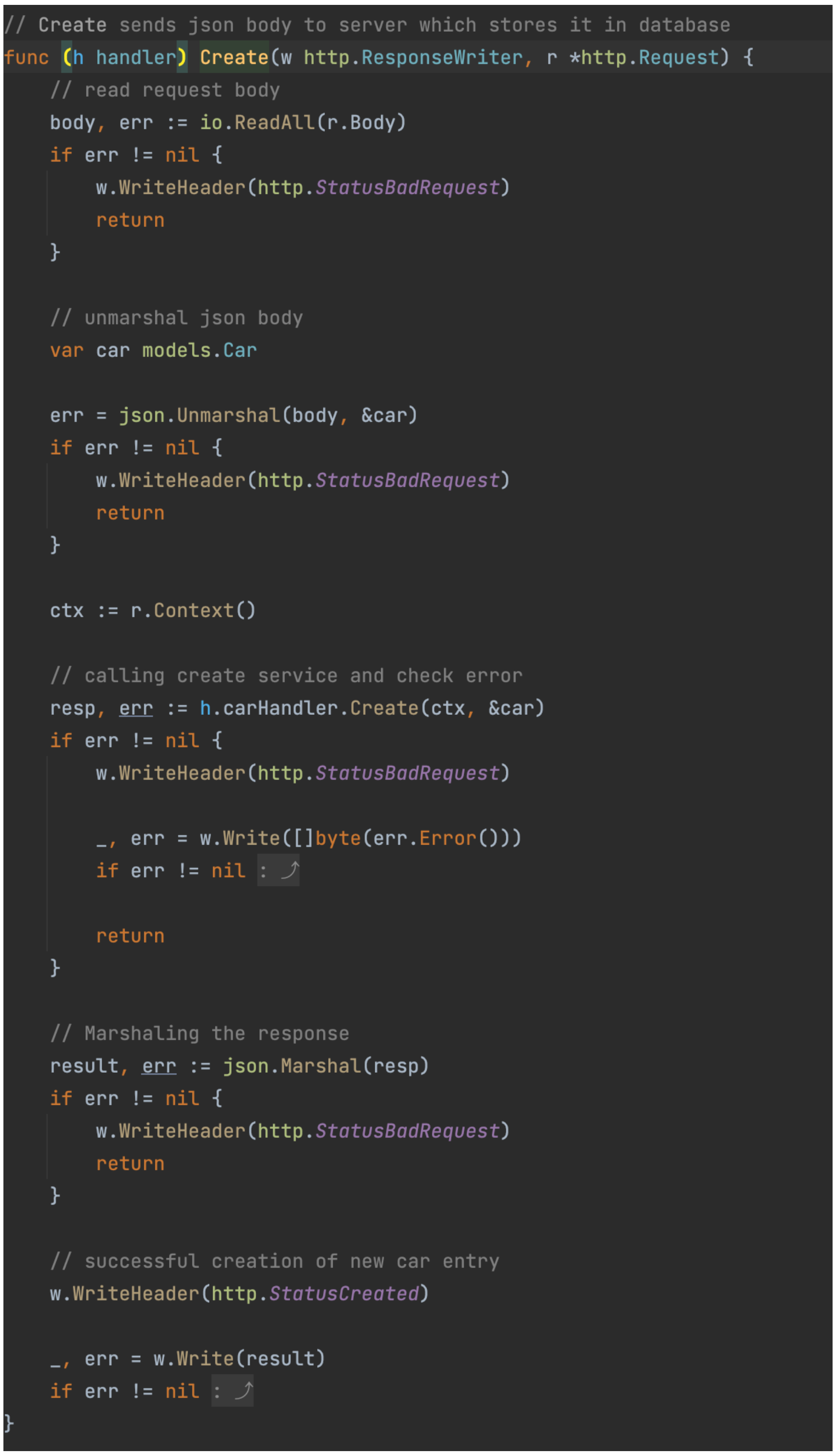
```go
func main() {
    // get the mysql configs from env:
    conf := driver.MySQLConfig{
        Host:     "127.0.0.1",
        User:     "kashish",
        Password: "password",
        Port:     "3306",
        DB:       "carDealership",
    }

    db, err := driver.ConnectToSQL(&conf)
    if err != nil {
        log.Printf("Couldn't Connect to sql #{err}")
        return
    }

    carStore := car.New(db)
    engineStore := engine.New(db)

    svc := car2.New(carStore, engineStore)

    handler := delivery.New(svc)
    router := mux.NewRouter()

    // middleware
    router.Use(middleware.Authenticate)

    router.HandleFunc( path: "/car/create", handler.Create)
    router.HandleFunc( path: "/car/fetch", handler.GetByBrand)
    router.HandleFunc( path: "/car/{id}", handler.Update).Methods(http.MethodPut)
    router.HandleFunc( path: "/car/fetch/{id}", handler.GetByID).Methods(http.MethodGet)
    router.HandleFunc( path: "/car/{id}", handler.Delete).Methods(http.MethodDelete)

    // establish server connection
    log.Fatal(http.ListenAndServe( addr: ":8000", router))
}
```

Our server gets started here at port 8000 and once we hit the end-point in postman, this calls the handler layer.

HANDLER LAYER

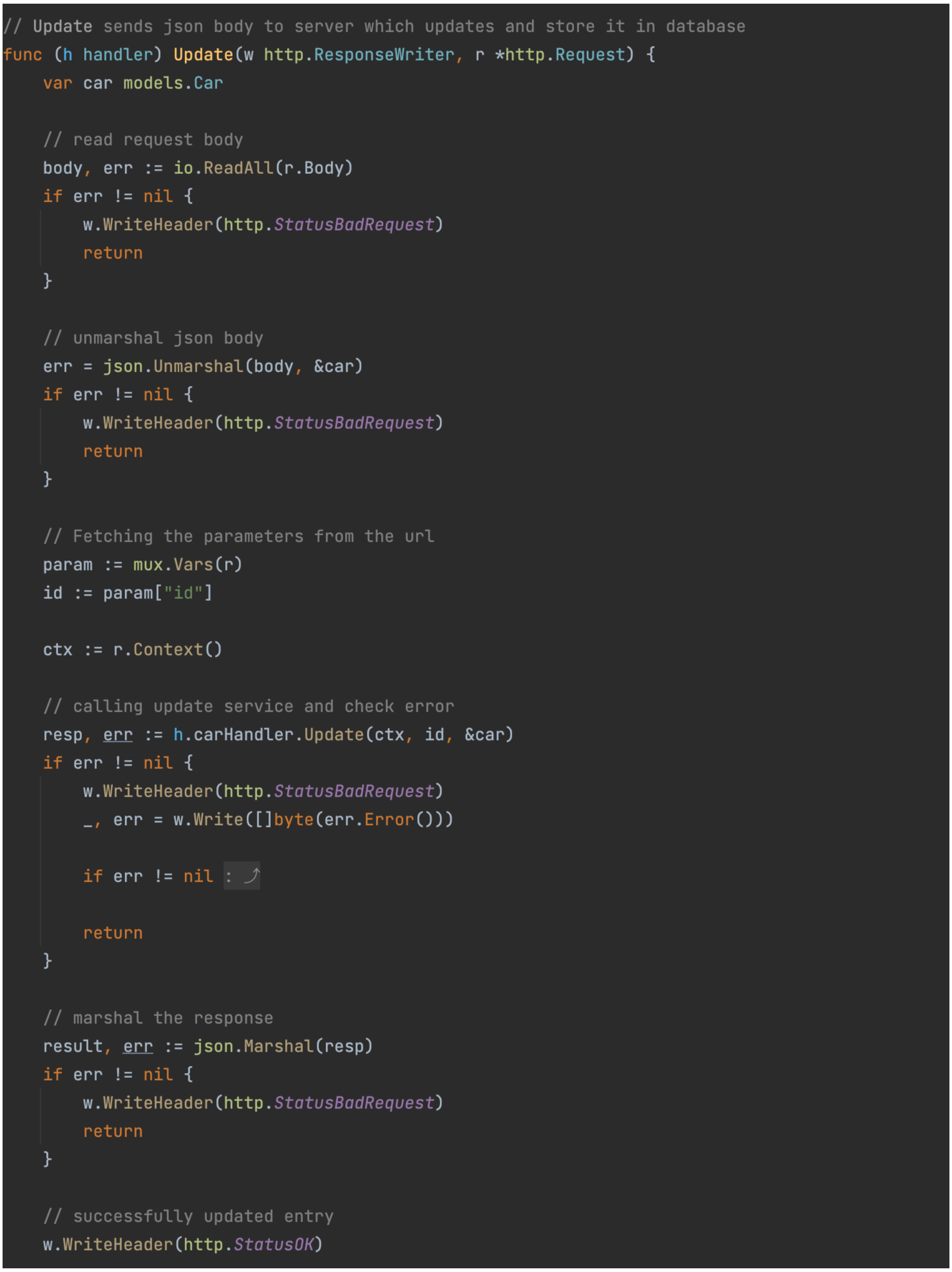
http.go : here we unmarshal the json body and send it to the service layer.

Create: Here we create a new entry by passing the requested json body followed by unmarshiling it and then passing it to the service layer to check if all the data passed is valid and according to the parameters defined.



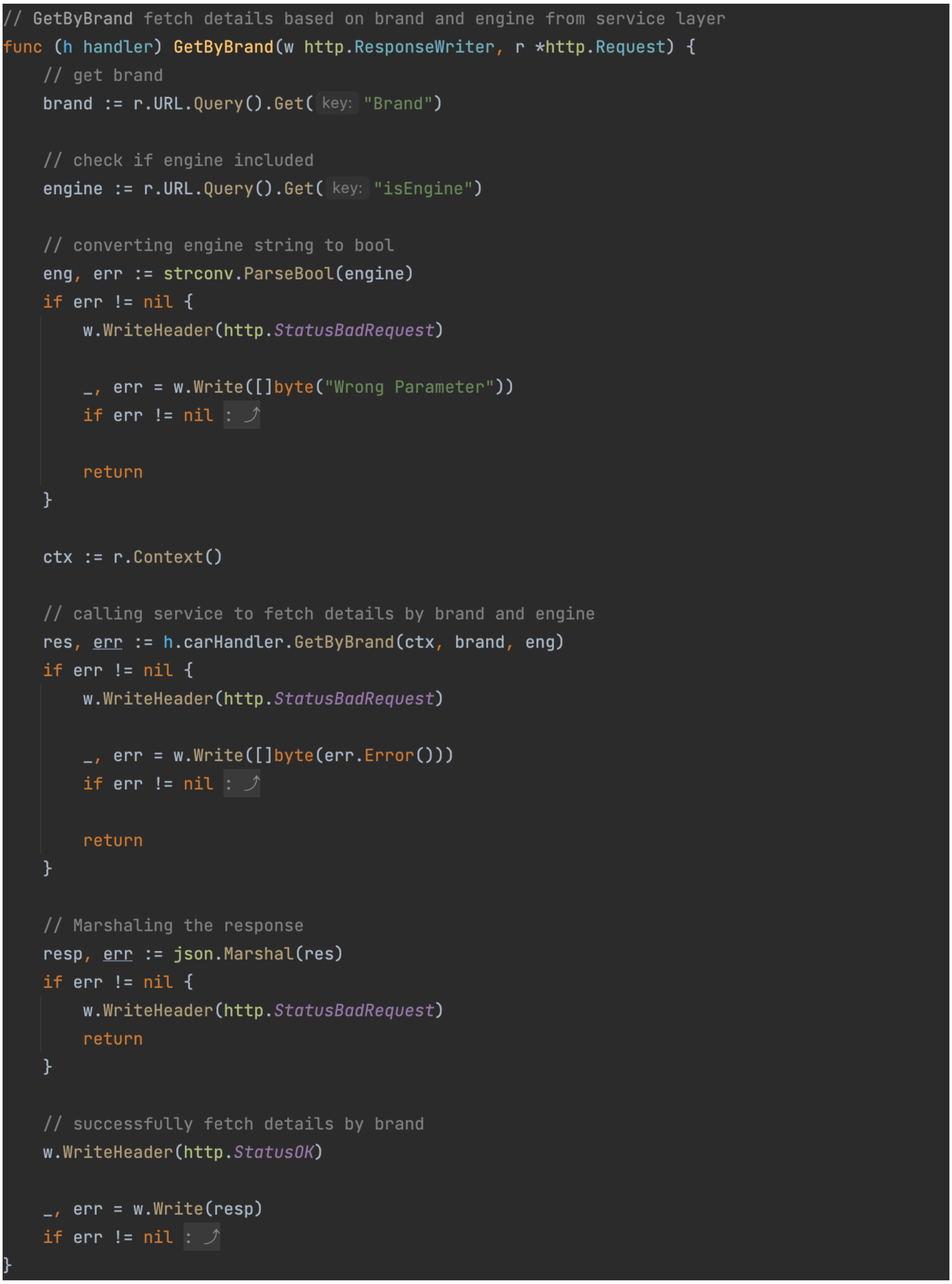
```go
// Create sends json body to server which stores it in database
func (h handler) Create(w http.ResponseWriter, r *http.Request) {
    // read request body
    body, err := io.ReadAll(r.Body)
    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        return
    }

    // unmarshal json body
    var car models.Car

    err = json.Unmarshal(body, &car)
    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        return
    }

    ctx := r.Context()

    // calling create service and check error
    resp, err := h.carHandler.Create(ctx, &car)
    if err != nil {
        w.WriteHeader(http.StatusBadRequest)

        _, err = w.Write([]byte(err.Error()))
        if err != nil : ⤴

        return
    }

    // Marshaling the response
    result, err := json.Marshal(resp)
    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        return
    }

    // successful creation of new car entry
    w.WriteHeader(http.StatusCreated)

    _, err = w.Write(result)
    if err != nil : ⤴
}
```

Update: Here we update an existing entry by passing the requested json body followed by unmarshiling it and then passing it to the service layer to check if all the data passed is valid and according to the parameters defined.



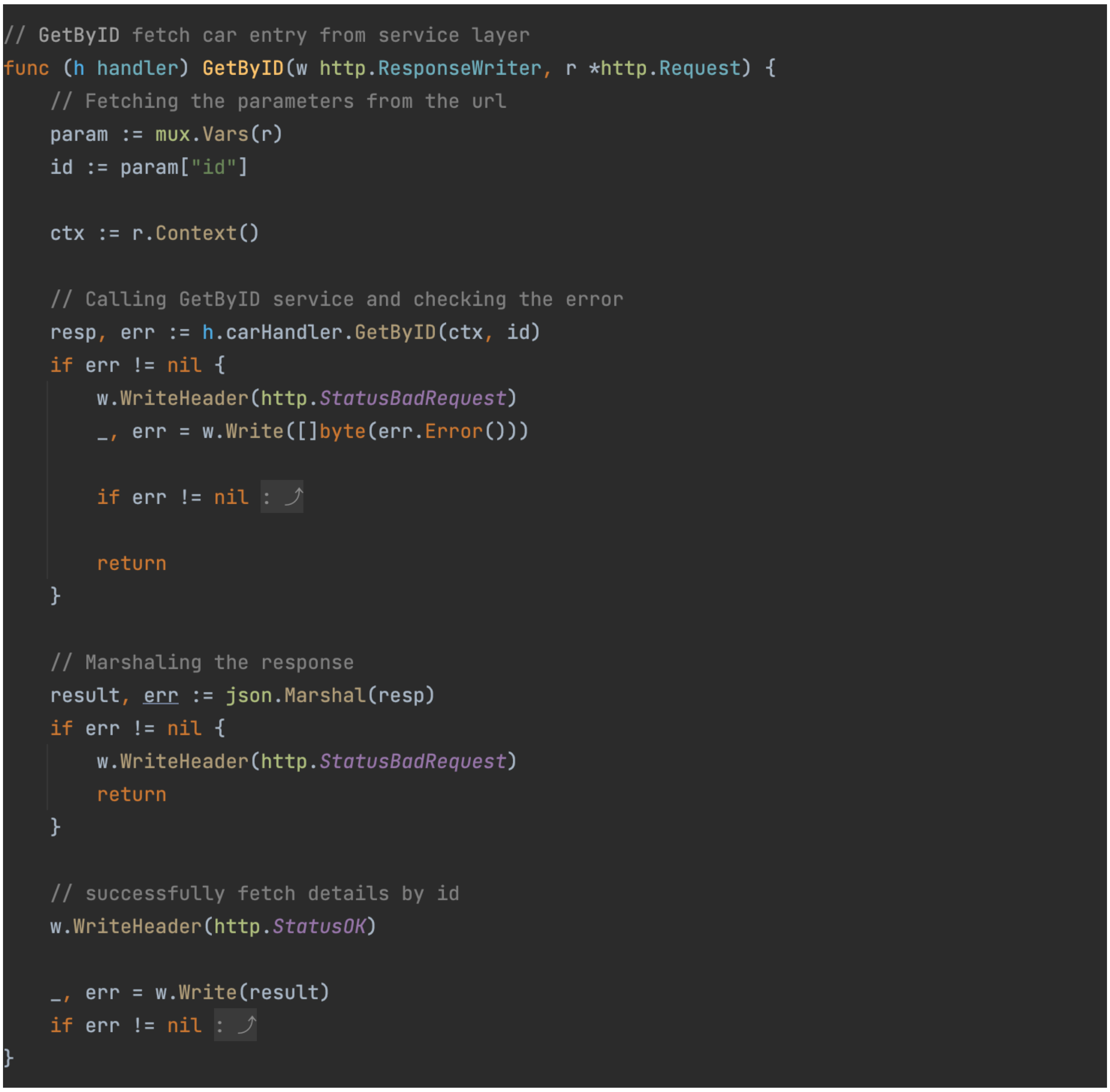
```go
// Update sends json body to server which updates and store it in database
func (h handler) Update(w http.ResponseWriter, r *http.Request) {
    var car models.Car

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

    // unmarshal json body
    err = json.Unmarshal(body, &car)
    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        return
    }

    // Fetching the parameters from the url
    param := mux.Vars(r)
    id := param["id"]

    ctx := r.Context()

    // calling update service and check error
    resp, err := h.carHandler.Update(ctx, id, &car)
    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        _, err = w.Write([]byte(err.Error()))

        if err != nil : ⤴

        return
    }

    // marshal the response
    result, err := json.Marshal(resp)
    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        return
    }

    // successfully updated entry
    w.WriteHeader(http.StatusOK)
```

Get By Brand: Here we fetch details of car by car brand and then passing it to the service layer to check if all the data passed is valid and according to the parameters defined.



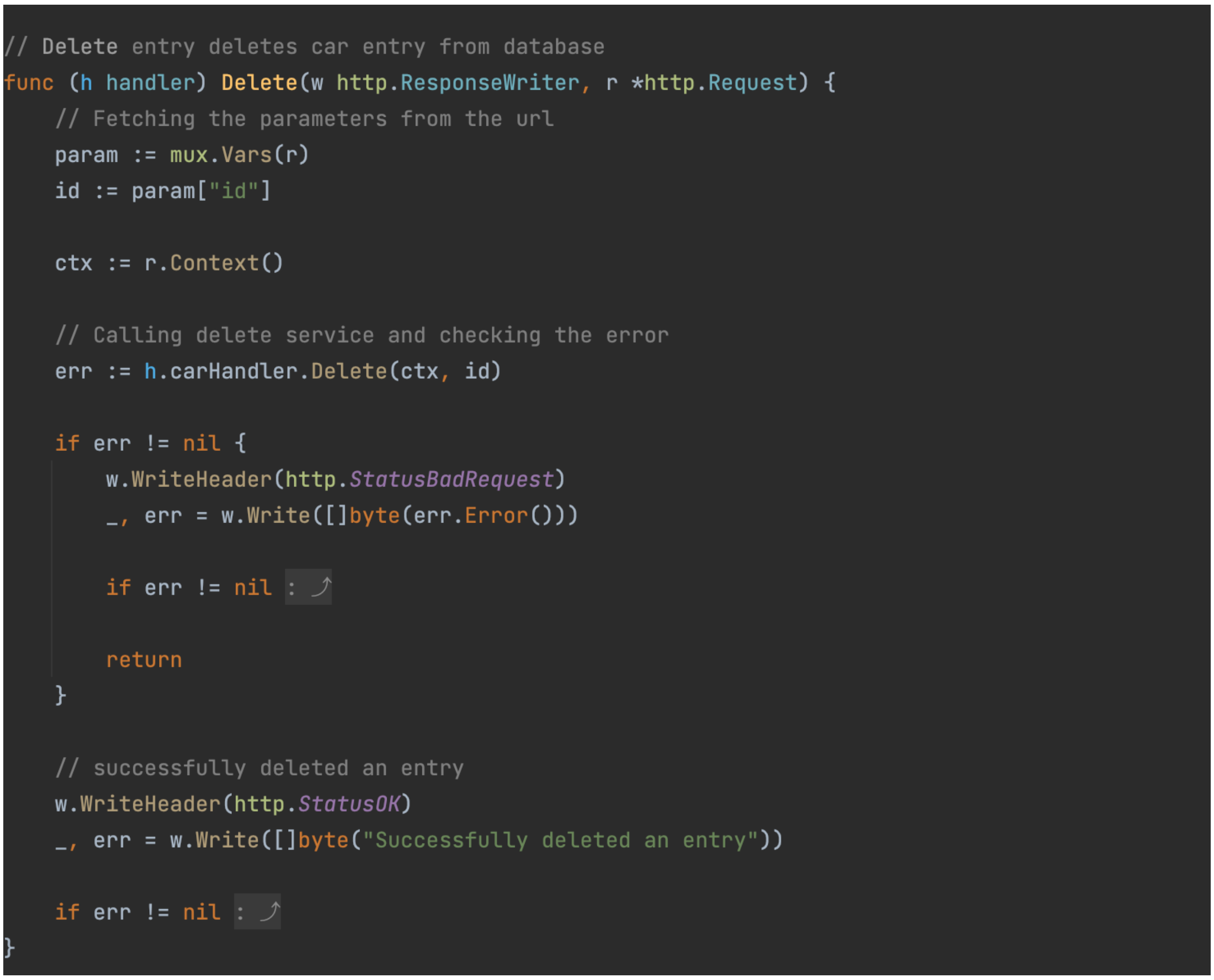
```
// GetByBrand fetch details based on brand and engine from service layer
func (h handler) GetByBrand(w http.ResponseWriter, r *http.Request) {
    // get brand
    brand := r.URL.Query().Get( key: "Brand")

    // check if engine included
    engine := r.URL.Query().Get( key: "isEngine")

    // converting engine string to bool
    eng, err := strconv.ParseBool(engine)
    if err != nil {
        w.WriteHeader(http.StatusBadRequest)

        _, err = w.Write([]byte("Wrong Parameter"))
        if err != nil : ⤴

        return
    }

    ctx := r.Context()

    // calling service to fetch details by brand and engine
    res, err := h.carHandler.GetByBrand(ctx, brand, eng)
    if err != nil {
        w.WriteHeader(http.StatusBadRequest)

        _, err = w.Write([]byte(err.Error()))
        if err != nil : ⤴

        return
    }

    // Marshaling the response
    resp, err := json.Marshal(res)
    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        return
    }

    // successfully fetch details by brand
    w.WriteHeader(http.StatusOK)

    _, err = w.Write(resp)
    if err != nil : ⤴
}
```

Get By Id: Here we fetch details of car by car Id and then passing it to the service layer to check if all the data passed is valid and according to the parameters defined.



```go
// GetByID fetch car entry from service layer
func (h handler) GetByID(w http.ResponseWriter, r *http.Request) {
    // Fetching the parameters from the url
    param := mux.Vars(r)
    id := param["id"]

    ctx := r.Context()

    // Calling GetByID service and checking the error
    resp, err := h.carHandler.GetByID(ctx, id)
    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        _, err = w.Write([]byte(err.Error()))

        if err != nil :

        return
    }

    // Marshaling the response
    result, err := json.Marshal(resp)
    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        return
    }

    // successfully fetch details by id
    w.WriteHeader(http.StatusOK)

    _, err = w.Write(result)
    if err != nil :
}
```

Delete : Here we delete an entry and then pass it to the service layer to check if all the data passed is valid and according to the parameters defined.

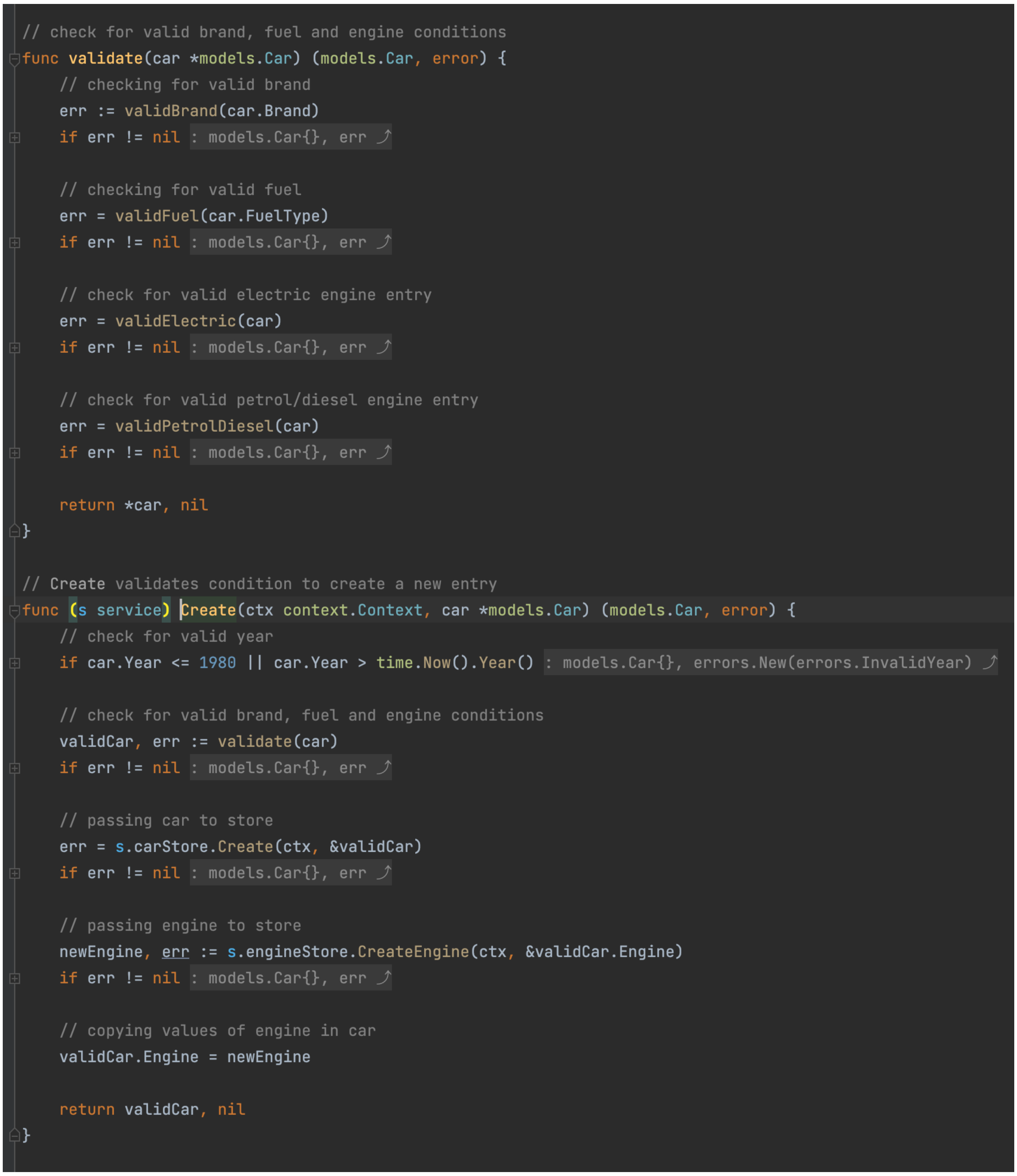


```go
// Delete entry deletes car entry from database
func (h handler) Delete(w http.ResponseWriter, r *http.Request) {
    // Fetching the parameters from the url
    param := mux.Vars(r)
    id := param["id"]

    ctx := r.Context()

    // Calling delete service and checking the error
    err := h.carHandler.Delete(ctx, id)

    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        _, err = w.Write([]byte(err.Error()))

        if err != nil : ⤴

        return
    }

    // successfully deleted an entry
    w.WriteHeader(http.StatusOK)
    _, err = w.Write([]byte("Successfully deleted an entry"))

    if err != nil : ⤴
}
```

SERVICE LAYER

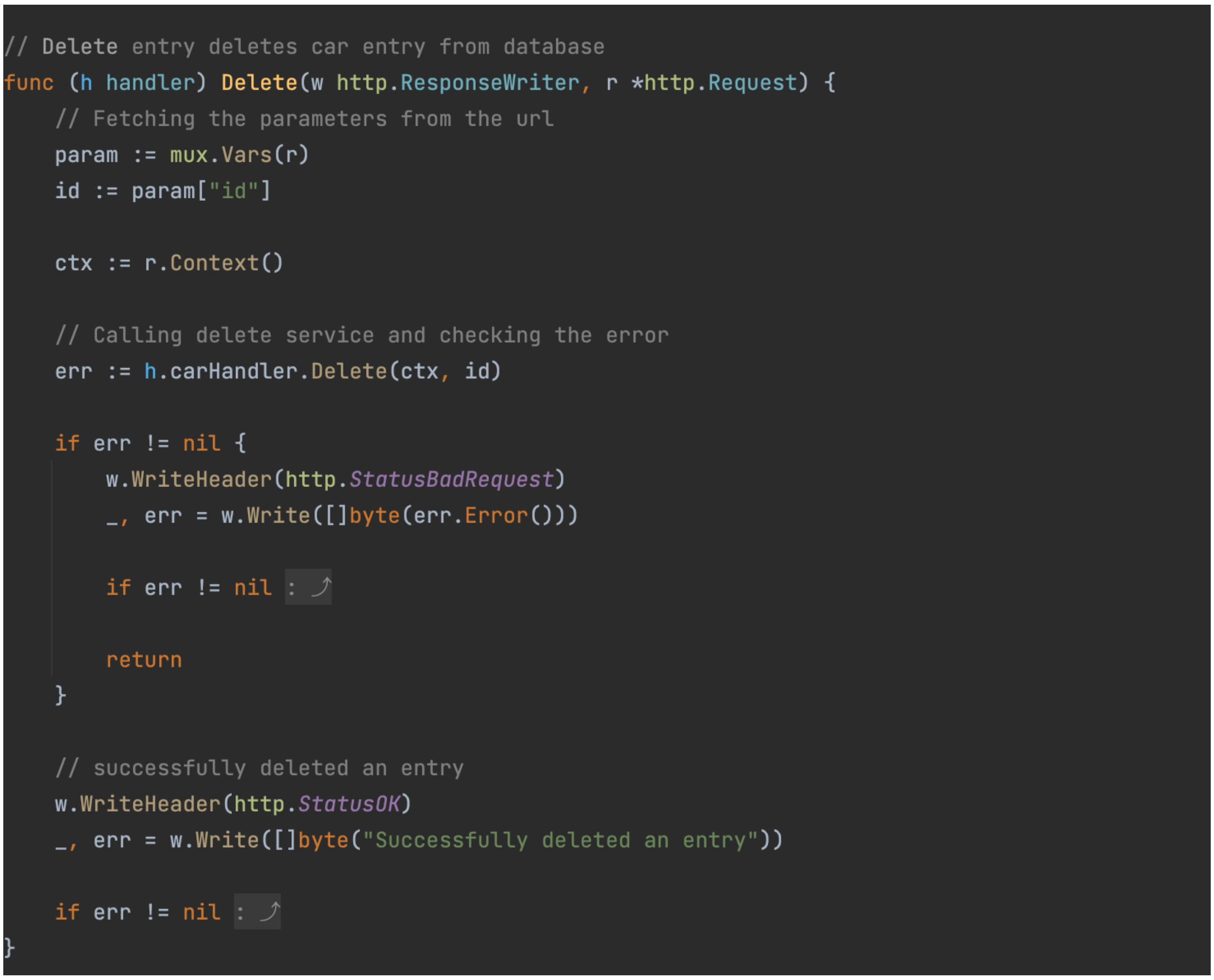
service.go : Before storing the data in the database, we want to make sure all the business logic is correct and thus we do the same in this layer. We make sure all the fields are validated according to the rules designed.

create: We check if the data added is all valid and accoring to rules defined. Once we discover that there is no error we pass it to datastore layer to store data in database.



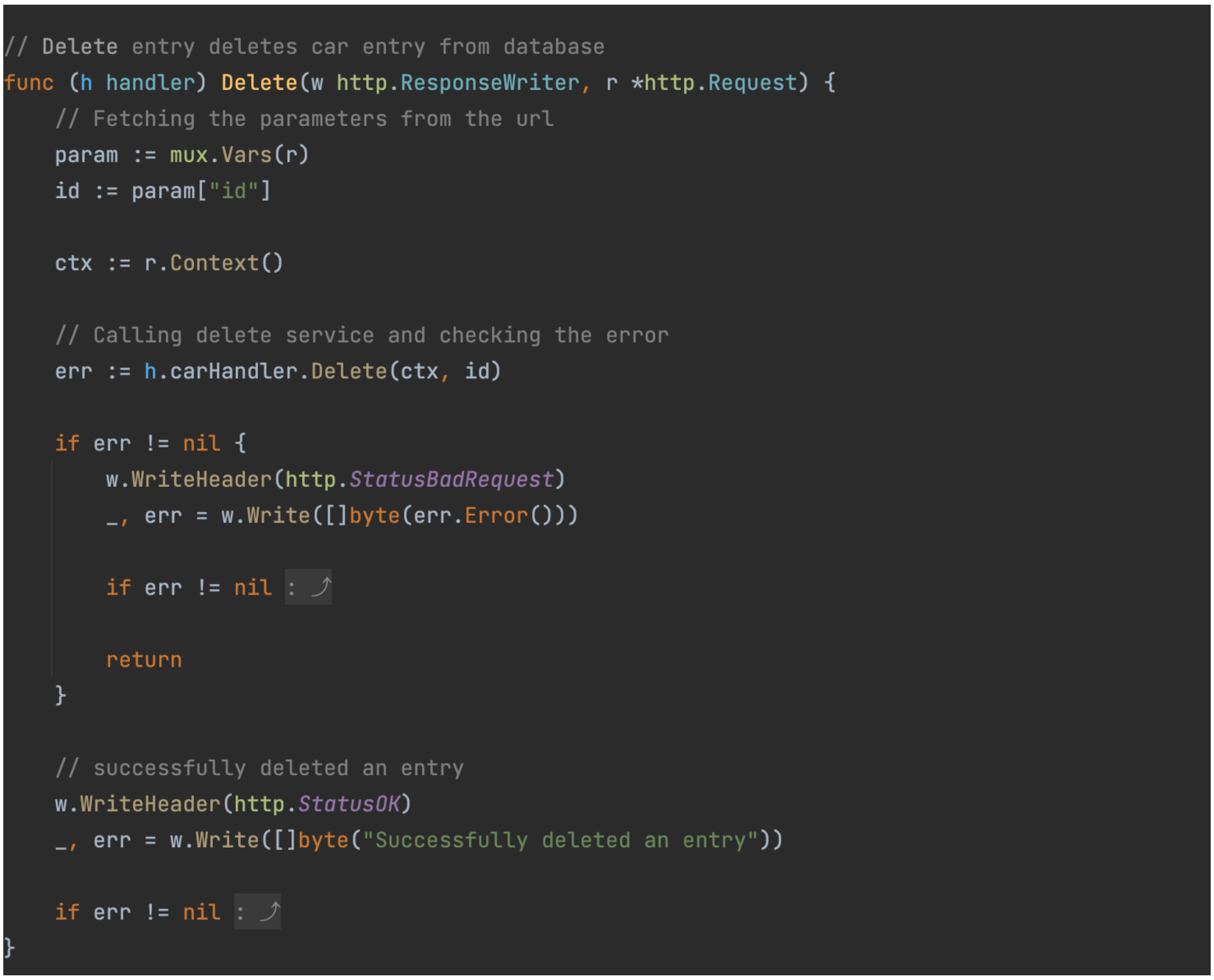
```
// check for valid brand, fuel and engine conditions
func validate(car *models.Car) (models.Car, error) {
    // checking for valid brand
    err := validBrand(car.Brand)
    if err != nil : models.Car{}, err ⤴

    // checking for valid fuel
    err = validFuel(car.FuelType)
    if err != nil : models.Car{}, err ⤴

    // check for valid electric engine entry
    err = validElectric(car)
    if err != nil : models.Car{}, err ⤴

    // check for valid petrol/diesel engine entry
    err = validPetrolDiesel(car)
    if err != nil : models.Car{}, err ⤴

    return *car, nil
}

// Create validates condition to create a new entry
func (s service) Create(ctx context.Context, car *models.Car) (models.Car, error) {
    // check for valid year
    if car.Year <= 1980 || car.Year > time.Now().Year() : models.Car{}, errors.New(errors.InvalidYear) ⤴

    // check for valid brand, fuel and engine conditions
    validCar, err := validate(car)
    if err != nil : models.Car{}, err ⤴

    // passing car to store
    err = s.carStore.Create(ctx, &validCar)
    if err != nil : models.Car{}, err ⤴

    // passing engine to store
    newEngine, err := s.engineStore.CreateEngine(ctx, &validCar.Engine)
    if err != nil : models.Car{}, err ⤴

    // copying values of engine in car
    validCar.Engine = newEngine

    return validCar, nil
}
```

update: We check if the data updatd is all valid and accoring to rules defined. Once we discover that there is no error we pass it to datastore layer to store data in database.



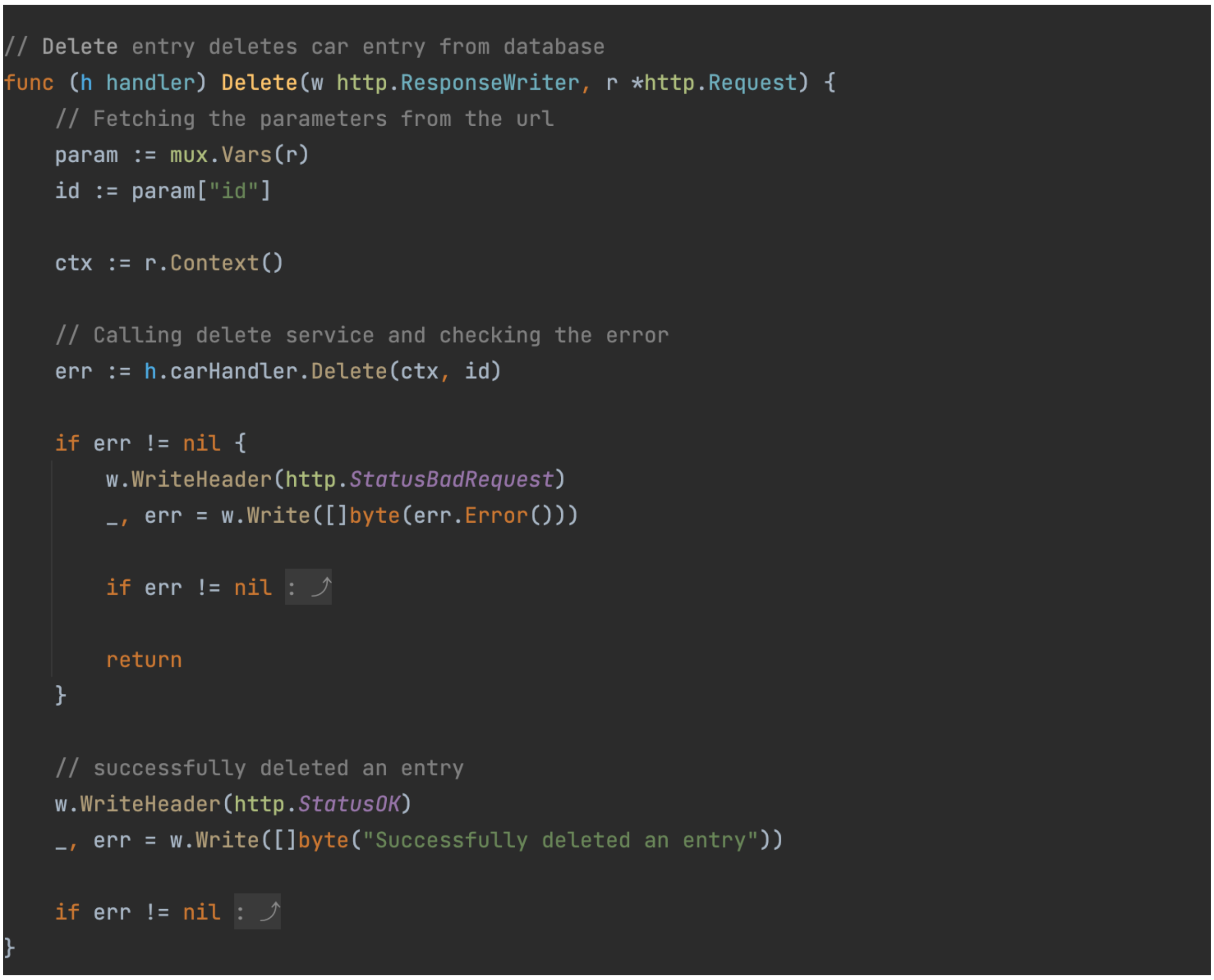
```go
// Delete entry deletes car entry from database
func (h handler) Delete(w http.ResponseWriter, r *http.Request) {
    // Fetching the parameters from the url
    param := mux.Vars(r)
    id := param["id"]

    ctx := r.Context()

    // Calling delete service and checking the error
    err := h.carHandler.Delete(ctx, id)

    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        _, err = w.Write([]byte(err.Error()))

        if err != nil : ⤴

        return
    }

    // successfully deleted an entry
    w.WriteHeader(http.StatusOK)
    _, err = w.Write([]byte("Successfully deleted an entry"))

    if err != nil : ⤴
}
```

Get by Id : We check if the data fetch by Id is all valid and accoring to rules defined. Once we discover that there is no error we pass it to datastore layer to fetch data in database.



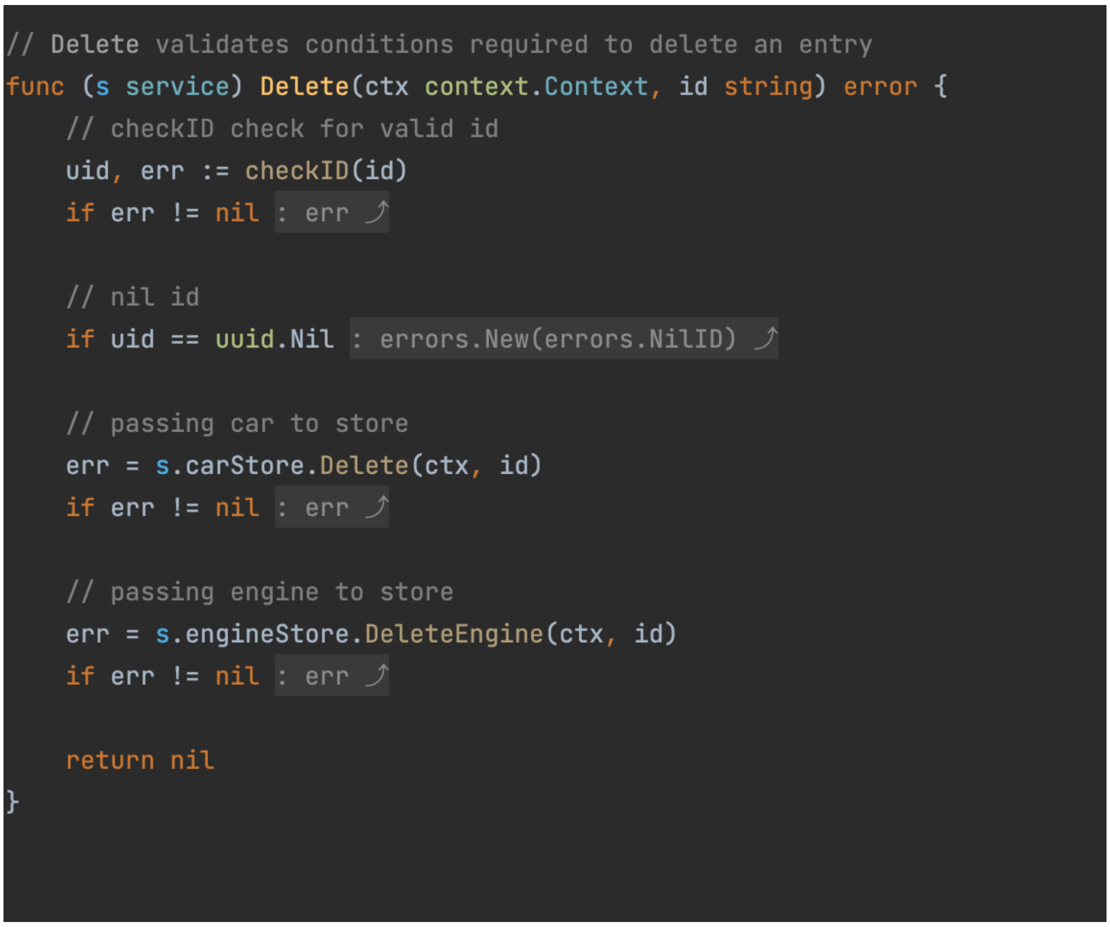
```go
// Delete entry deletes car entry from database
func (h handler) Delete(w http.ResponseWriter, r *http.Request) {
    // Fetching the parameters from the url
    param := mux.Vars(r)
    id := param["id"]

    ctx := r.Context()

    // Calling delete service and checking the error
    err := h.carHandler.Delete(ctx, id)

    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        _, err = w.Write([]byte(err.Error()))

        if err != nil :

        return
    }

    // successfully deleted an entry
    w.WriteHeader(http.StatusOK)
    _, err = w.Write([]byte("Successfully deleted an entry"))

    if err != nil :
}
```

Get by Brand: We check if the data fecthed by brand is all valid and accoring to rules defined. Once we discover that there is no error we pass it to datastore layer to fetch data in database.



```go
// Delete entry deletes car entry from database
func (h handler) Delete(w http.ResponseWriter, r *http.Request) {
    // Fetching the parameters from the url
    param := mux.Vars(r)
    id := param["id"]

    ctx := r.Context()

    // Calling delete service and checking the error
    err := h.carHandler.Delete(ctx, id)

    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        _, err = w.Write([]byte(err.Error()))

        if err != nil :

        return
    }

    // successfully deleted an entry
    w.WriteHeader(http.StatusOK)
    _, err = w.Write([]byte("Successfully deleted an entry"))

    if err != nil :
}
```

Delete: We check if the data deleted is all valid and accoring to rules defined. Once we discover that there is no error we pass it to datastore layer to delete data in database.

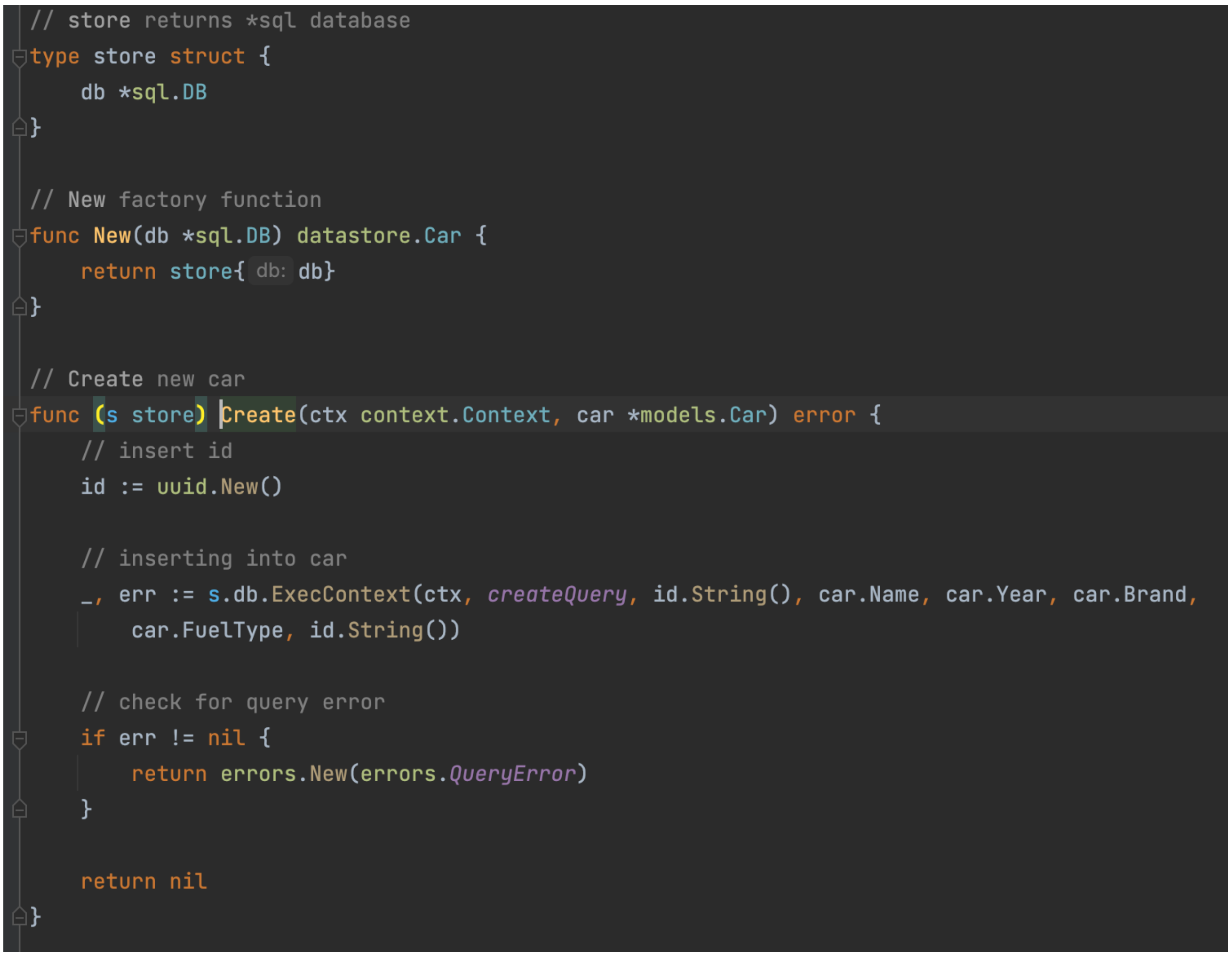


```go
// Delete validates conditions required to delete an entry
func (s service) Delete(ctx context.Context, id string) error {
    // checkID check for valid id
    uid, err := checkID(id)
    if err != nil : err ⤴

    // nil id
    if uid == uuid.Nil : errors.New(errors.NilID) ⤴

    // passing car to store
    err = s.carStore.Delete(ctx, id)
    if err != nil : err ⤴

    // passing engine to store
    err = s.engineStore.DeleteEngine(ctx, id)
    if err != nil : err ⤴

    return nil
}
```

DATA STORE LAYER

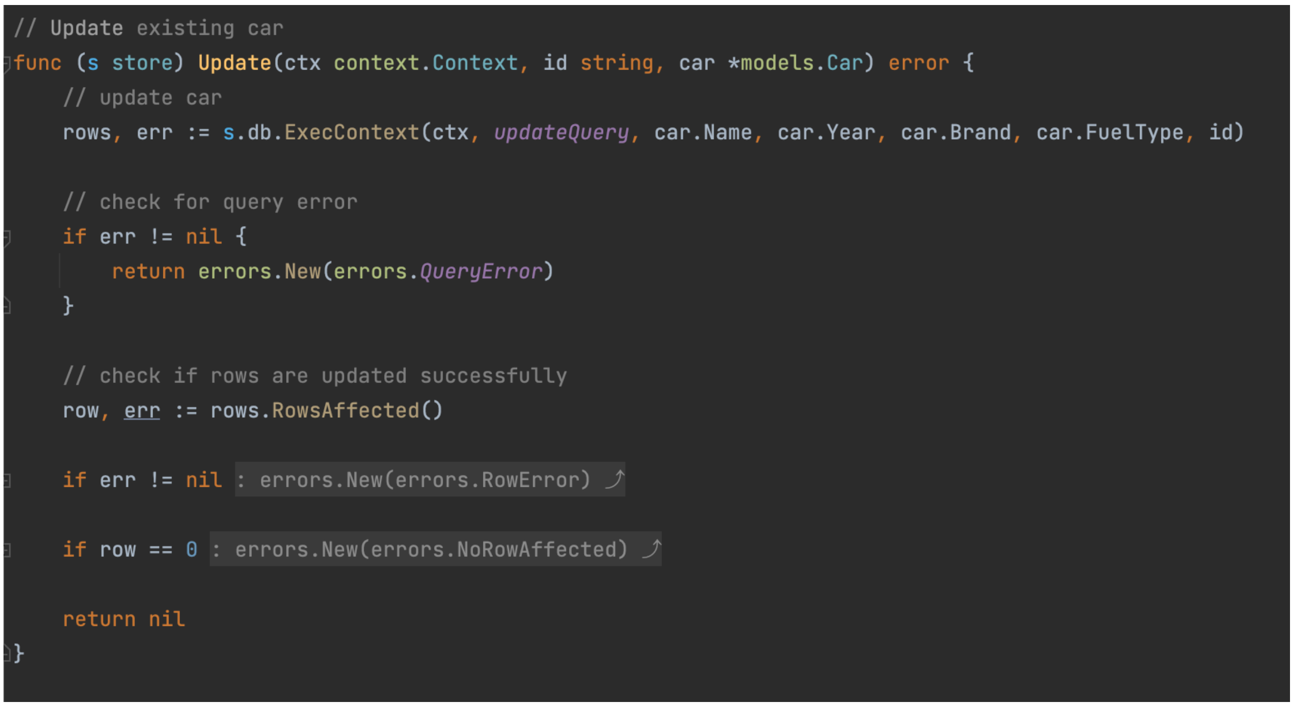
store.go: In this layer we write the query to store data in our database and check there are no db based errors.

Create: In this layer we create a new entry by performing sql query and then store the data in database.



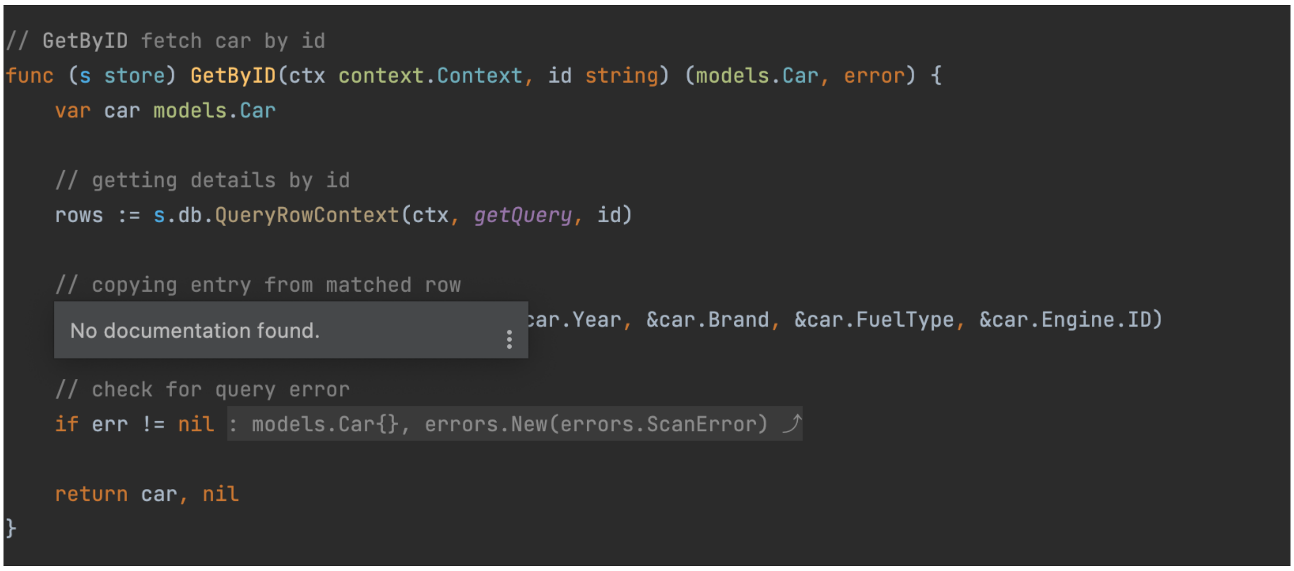
```go
// store returns *sql database
type store struct {
    db *sql.DB
}

// New factory function
func New(db *sql.DB) datastore.Car {
    return store{ db: db}
}

// Create new car
func (s store) Create(ctx context.Context, car *models.Car) error {
    // insert id
    id := uuid.New()

    // inserting into car
    _, err := s.db.ExecContext(ctx, createQuery, id.String(), car.Name, car.Year, car.Brand,
        car.FuelType, id.String())

    // check for query error
    if err != nil {
        return errors.New(errors.QueryError)
    }

    return nil
}
```

Update: In this layer we update an existing entry by performing sql query and then storing the data in database.



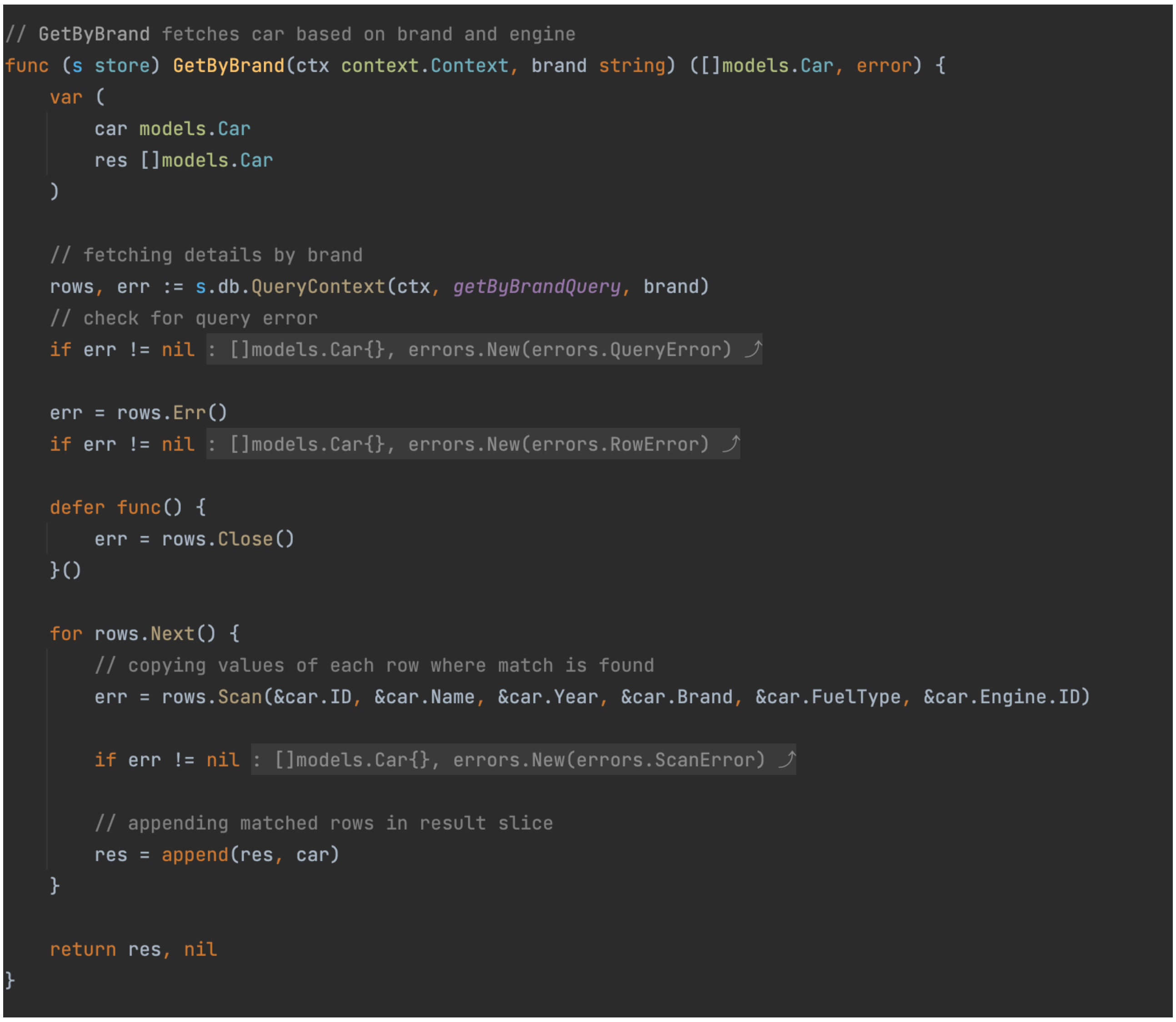
```go
// Update existing car
func (s store) Update(ctx context.Context, id string, car *models.Car) error {
    // update car
    rows, err := s.db.ExecContext(ctx, updateQuery, car.Name, car.Year, car.Brand, car.FuelType, id)

    // check for query error
    if err != nil {
        return errors.New(errors.QueryError)
    }

    // check if rows are updated successfully
    row, err := rows.RowsAffected()

    if err != nil : errors.New(errors.RowError) ⤴

    if row == 0 : errors.New(errors.NoRowAffected) ⤴

    return nil
}
```

Get By Id: In this layer we fetch an entry by performing sql query and then storig the data in database.



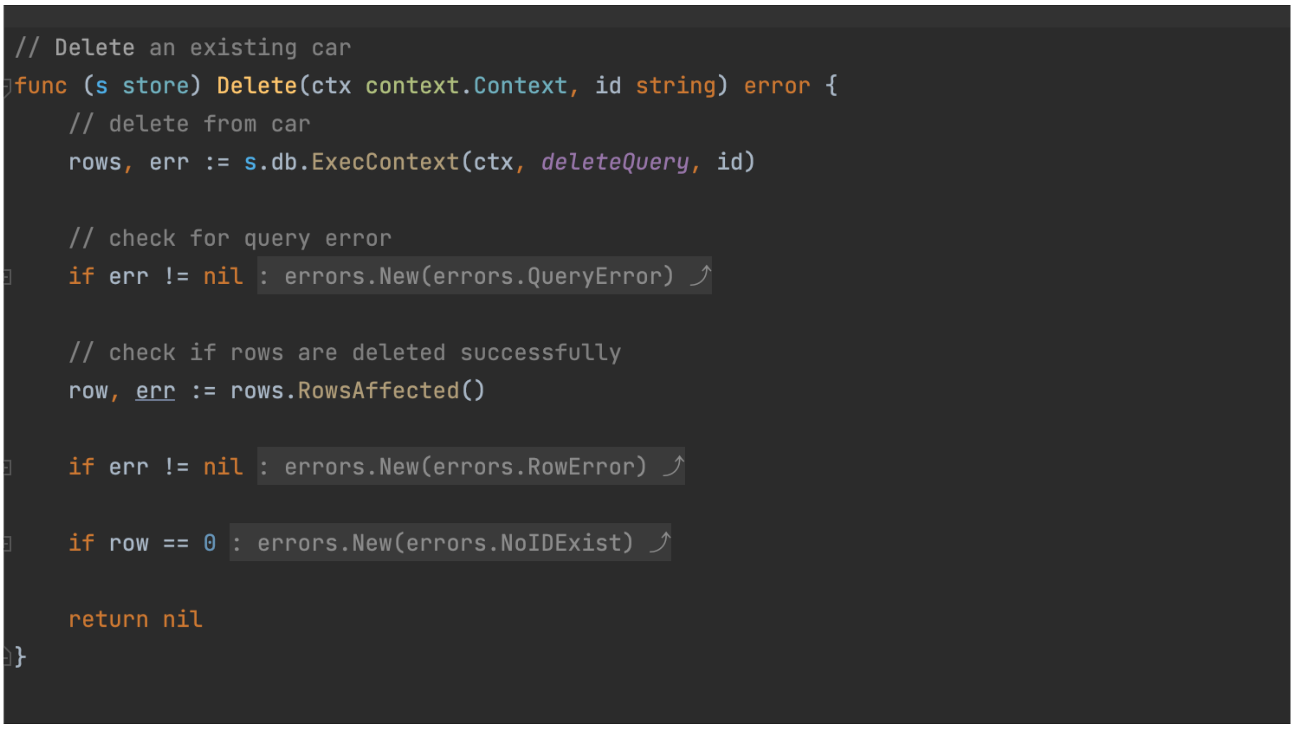
```
// GetByID fetch car by id
func (s store) GetByID(ctx context.Context, id string) (models.Car, error) {
    var car models.Car

    // getting details by id
    rows := s.db.QueryRowContext(ctx, getQuery, id)

    // copying entry from matched row
    No documentation found.       ⋮ car.Year, &car.Brand, &car.FuelType, &car.Engine.ID)

    // check for query error
    if err != nil : models.Car{}, errors.New(errors.ScanError) ⤴

    return car, nil
}
```

Get By Brand : In this layer we fetch details by brand and performing sql query and then store the data in database.



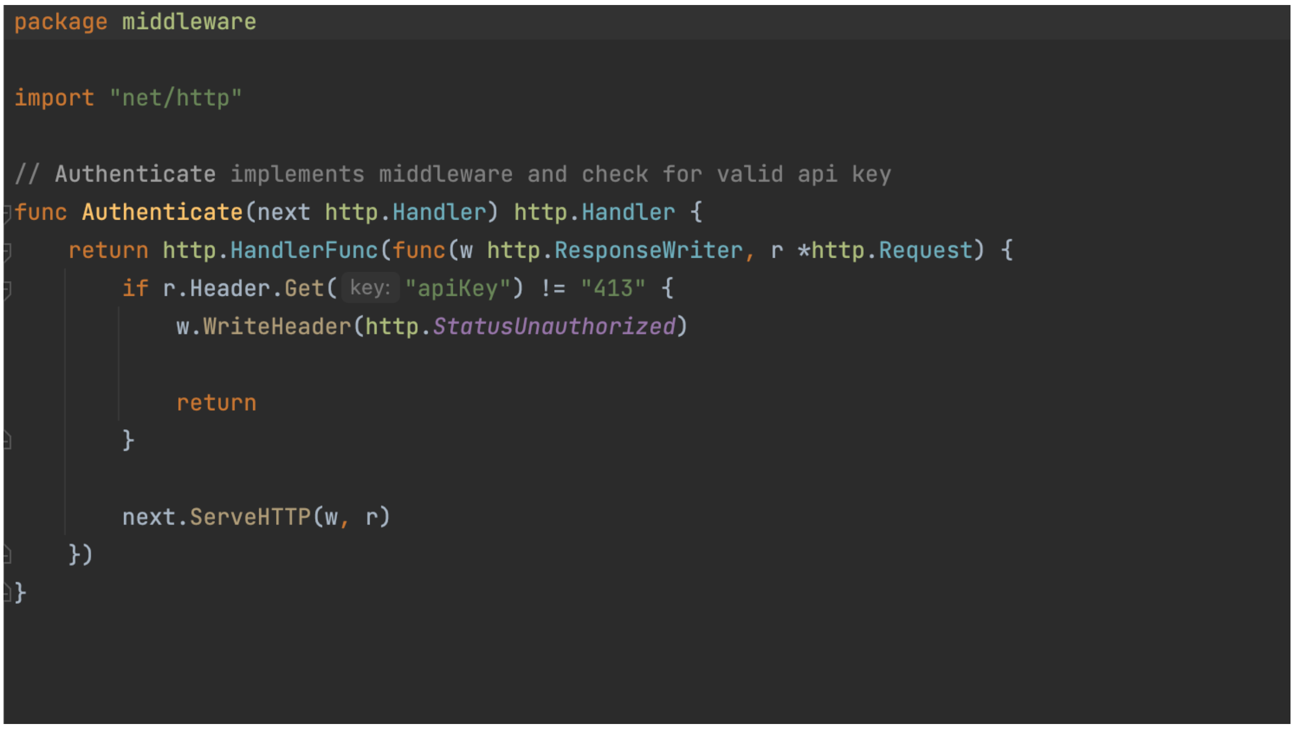
```go
// GetByBrand fetches car based on brand and engine
func (s store) GetByBrand(ctx context.Context, brand string) ([]models.Car, error) {
    var (
        car models.Car
        res []models.Car
    )

    // fetching details by brand
    rows, err := s.db.QueryContext(ctx, getByBrandQuery, brand)
    // check for query error
    if err != nil : []models.Car{}, errors.New(errors.QueryError) ⤴

    err = rows.Err()
    if err != nil : []models.Car{}, errors.New(errors.RowError) ⤴

    defer func() {
        err = rows.Close()
    }()

    for rows.Next() {
        // copying values of each row where match is found
        err = rows.Scan(&car.ID, &car.Name, &car.Year, &car.Brand, &car.FuelType, &car.Engine.ID)

        if err != nil : []models.Car{}, errors.New(errors.ScanError) ⤴

        // appending matched rows in result slice
        res = append(res, car)
    }

    return res, nil
}
```

Delete : In this layer we delete and existing entry by performing sql query and then deleting the data in database.



```go
// Delete an existing car
func (s store) Delete(ctx context.Context, id string) error {
    // delete from car
    rows, err := s.db.ExecContext(ctx, deleteQuery, id)

    // check for query error
    if err != nil : errors.New(errors.QueryError) ⤴

    // check if rows are deleted successfully
    row, err := rows.RowsAffected()

    if err != nil : errors.New(errors.RowError) ⤴

    if row == 0 : errors.New(errors.NoIDExist) ⤴

    return nil
}
```

MIDDLEWARE:

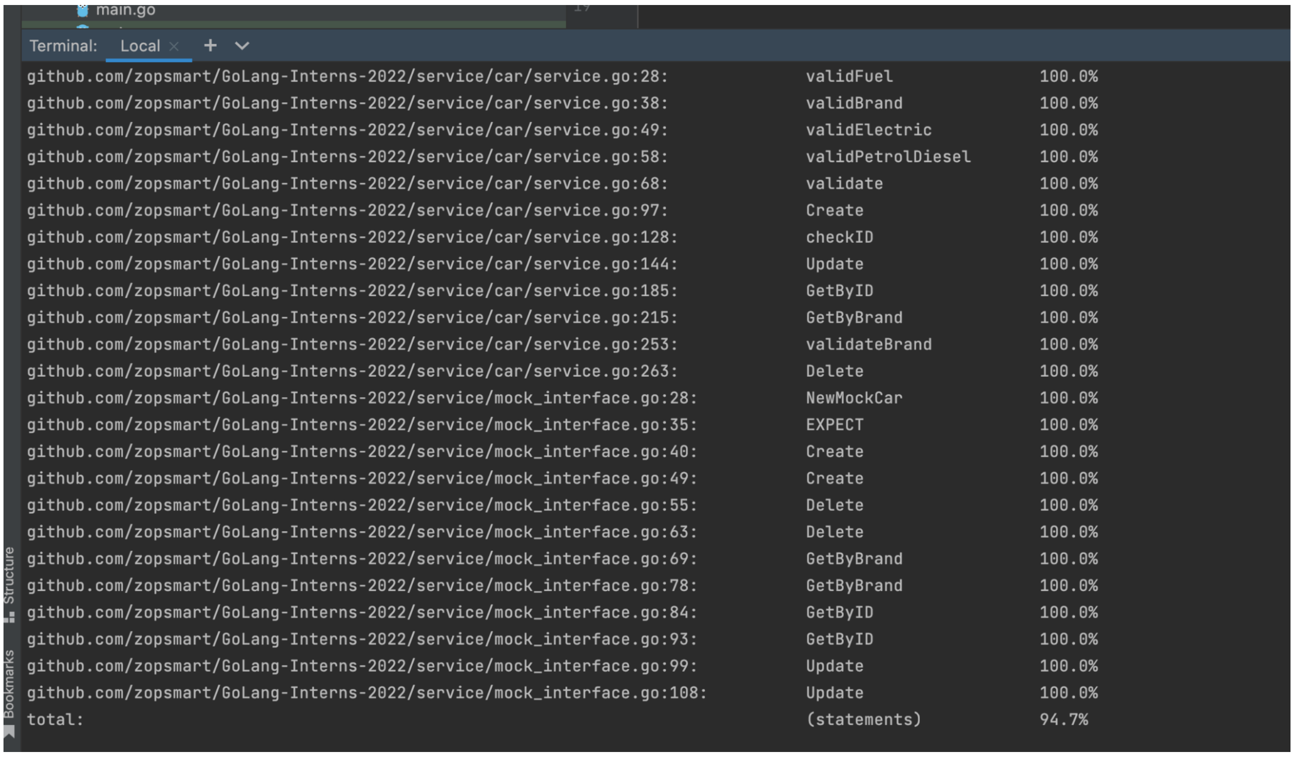


```go
package middleware

import "net/http"

// Authenticate implements middleware and check for valid api key
func Authenticate(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        if r.Header.Get( key: "apiKey") != "413" {
            w.WriteHeader(http.StatusUnauthorized)

            return
        }

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

CHAPTER 4 : Performance Analysis

1. Unit Test Coverage

Performed unit test coverage and found all 44 tests ran successfully i.e PASS with a total coverage of 94.7%.



```
github.com/zopsmart/GoLang-Interns-2022/service/car/service.go:28:            validFuel             100.0%
github.com/zopsmart/GoLang-Interns-2022/service/car/service.go:38:            validBrand            100.0%
github.com/zopsmart/GoLang-Interns-2022/service/car/service.go:49:            validElectric         100.0%
github.com/zopsmart/GoLang-Interns-2022/service/car/service.go:58:            validPetrolDiesel     100.0%
github.com/zopsmart/GoLang-Interns-2022/service/car/service.go:68:            validate              100.0%
github.com/zopsmart/GoLang-Interns-2022/service/car/service.go:97:            Create                100.0%
github.com/zopsmart/GoLang-Interns-2022/service/car/service.go:128:           checkID               100.0%
github.com/zopsmart/GoLang-Interns-2022/service/car/service.go:144:           Update                100.0%
github.com/zopsmart/GoLang-Interns-2022/service/car/service.go:185:           GetByID               100.0%
github.com/zopsmart/GoLang-Interns-2022/service/car/service.go:215:           GetByBrand            100.0%
github.com/zopsmart/GoLang-Interns-2022/service/car/service.go:253:           validateBrand         100.0%
github.com/zopsmart/GoLang-Interns-2022/service/car/service.go:263:           Delete                100.0%
github.com/zopsmart/GoLang-Interns-2022/service/mock_interface.go:28:         NewMockCar            100.0%
github.com/zopsmart/GoLang-Interns-2022/service/mock_interface.go:35:         EXPECT                100.0%
github.com/zopsmart/GoLang-Interns-2022/service/mock_interface.go:40:         Create                100.0%
github.com/zopsmart/GoLang-Interns-2022/service/mock_interface.go:49:         Create                100.0%
github.com/zopsmart/GoLang-Interns-2022/service/mock_interface.go:55:         Delete                100.0%
github.com/zopsmart/GoLang-Interns-2022/service/mock_interface.go:63:         Delete                100.0%
github.com/zopsmart/GoLang-Interns-2022/service/mock_interface.go:69:         GetByBrand            100.0%
github.com/zopsmart/GoLang-Interns-2022/service/mock_interface.go:78:         GetByBrand            100.0%
github.com/zopsmart/GoLang-Interns-2022/service/mock_interface.go:84:         GetByID               100.0%
github.com/zopsmart/GoLang-Interns-2022/service/mock_interface.go:93:         GetByID               100.0%
github.com/zopsmart/GoLang-Interns-2022/service/mock_interface.go:99:         Update                100.0%
github.com/zopsmart/GoLang-Interns-2022/service/mock_interface.go:108:        Update                100.0%
total:                                                                        (statements)          94.7%
```

2. **Linter Check**

   Performed a linter check using command ***golangci-lint run*** which makes sure that the program is properly formatted and follows standard code guidelines such as no gocognit complexity or funlen to be 0 etc. There were **no** linter errors found in this project.

CHAPTER 5 : CONCLUSION

5.1 Results Achieved

The main aim of the training was to be able to understand and implement the concepts of **GoLang, MySQL, Unit Testing,** being able to create a web application successfully performing basic CRUD operations and can be tested using postman using the **three layered architecture.**

5.2 Applications Contributions

GoLang have been part of a variety of real world/ open source applications, some of the which are listed below:

1. Docker, a set of tools for deploying Linux containers, Kubernetes container management system
2. Dropbox, who migrated some of their critical components from Python to Go
3. Ethereum, The go-ethereum implementation of the Ethereum Virtual Machine, blockchain for the Ether cryptocurrency
4. Gitlab, a web-based DevOps lifecycle tool that provides a Git-repository, wiki, issue-tracking, continuous integration, deployment pipeline features etc.

5.3 Limitations

The application implements only the backend part but front end can be done for the same to make the application more attractive and user friendly.

5.4 Future Work / Scope

1. Front-end for application
2. Make the program more extensive

REFERENCES

https://go.dev/doc/
https://github.com/golang/mock
https://github.com/DATA-DOG/go-sqlmock
https://github.com/gorilla/mux
https://dev.mysql.com/doc/
https://www.linux.org/
https://docs.docker.com/
https://kubernetes.io/docs/home/
https://ngdocs.harness.io/
https://prometheus.io/docs/introduction/overview/