Building a RESTful API with 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

Piyushika Sachdeva (191217)

UNDER THE SUPERVISION OF

Dr. Ruchi Verma

to

Department of Computer 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. Ruchi Verma, **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.

[illegible] 15/May/23

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

Piyush

Submitted by:
Piyushika Sachdeva(191217)
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 **"Building RESTful API with Go using Three Layered Architecture"** 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 Piyushika Sachdeva (191217) over a period from February 2023 to May 2023 under the supervision of **Mithali R Shetty (Senior Lead Engineer Zopsmart Technology).** The matter embodied in the report has not been submitted for the award of any other degree or diploma.

Piyush

Piyushika Sachdeva
(191217)

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

MShetty

Mithali R Shetty
Senior Lead Engineer
Zopsmart Technology

Ruchi
13/May/23

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

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT

PLAGIARISM VERIFICATION REPORT

Date: 15 May 2023

Type of Document (Tick): PhD Thesis | M.Tech Dissertation/ Report | [✓ B.Tech Project Report] | Paper

Name: PIYUSHIKA SACHDEVA Department: CSE Enrolment No 191217

Contact No. 9910878208 E-mail. sachpiyushi@gmail.com

Name of the Supervisor: Dr. RUCHI VERMA

Title of the Thesis/Dissertation/Project Report/Paper (In Capital letters): BUILDING A RESTFUL API WITH GO USING THREE LAYERED ARCHITECTURE.

UNDERTAKING

I undertake that I am aware of the plagiarism related norms/ regulations, if I found guilty of any plagiarism and copyright violations in the above thesis/report even after award of degree, the University reserves the rights to withdraw/revoke my degree/report. Kindly allow me to avail Plagiarism verification report for the document mentioned above.

Complete Thesis/Report Pages Detail:

- Total No. of Pages = 61
- Total No. of Preliminary pages = 10
- Total No. of pages accommodate bibliography/references = 1

[illegible]

(Signature of Student)

FOR DEPARTMENT USE

We have checked the thesis/report as per norms and found **Similarity Index** at ....8.......... (%). Therefore, we are forwarding the complete thesis/report for final plagiarism check. The plagiarism verification report may be handed over to the candidate.

[illegible] 15/may/23

(Signature of Guide/Supervisor)

[illegible]

Signature of HOD

FOR LRC USE

The above document was scanned for plagiarism check. The outcome of the same is reported below:

Copy Received on	Excluded	Similarity Index (%)	Generated Plagiarism Report Details (Title, Abstract & Chapters)	
			Word Counts	
Report Generated on				
		Submission ID	Total Pages Scanned	
			File Size	

Checked by
Name & Signature

Librarian

....

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

ACKNOWLEDGEMENT

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

I am highly indebted to our training instructors Ms. Mithali R. Shetty, Mr. Vageesha BR and Mr. N for their guidance and constant supervision as well as for providing necessary information regarding the project. I also would like to thank Mr. V. Vaishnav and Ms. Spoorthi for mentoring me throughout my internship and helping me learn new concepts and technologies. It is indeed with a great sense of pleasure and immense sense of gratitude that I acknowledge the help of these individuals.

I am extremely grateful to my supervisor, Dr.Ruchi Verma, Assistant Professor (SG) , Department of CSE Jaypee University of Information Technology, Waknaghat, for her assistance. Her never-ending patience, intellectual direction, constant encouragement, constant and energetic supervision, constructive criticism, and good suggestions made it possible to finish this job.

I would also like to express my gratitude to everyone who has directly or indirectly assisted me in making this project a success. Finally, I must express my gratitude for my parents' unwavering support and patience.

Piyushika Sachdeva(191217)

Computer Science & Engineering and Information Technology Department.

Jaypee University of Information Technology, Waknaghat, Sola

TABLE OF CONTENT

LIST OF ABBREVIATIONS

URL	Uniform Resource Locator
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
SQL	Structured Query Language
GOFR	Go Framework
MYSQL	My Structured Query Language
IDE	Integrated Development Environment
BASH	Bourne Again Shell
SUDO	Superuser do or substitute user do

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 the request body and then parses anything that is required from that request. It then calls the service layer where all the business logic of the 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 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 affects the other.

CHAPTER 1 : INTRODUCTION

1.1.1 Company

Zopsmart Technologies is a dynamic and innovative technology company that provides tools and techniques to help offline businesses go online. Zopsmart's goal is to create value and deliver mature,end-to-end products for digital-savvy customers. Through human-centered design practices and proven analytics, Zopsmart helps businesses re-imagine their consumer interactions and innovate within predictable budgets.Founded in 2016, the company has rapidly established itself as a leading provider of software development, mobile app development, and web development services, catering to diverse industries such as e-commerce, healthcare, education, and more. 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 provides software solutions to some of the top-most firms and has its own framework to work on.

1.1.2 Introduction

Software development and web application development are important aspects of today's business world. In today's digital age, companies around the world rely on software and web applications to increase productivity, improve customer engagement and simplify operations.

Software Development comprises the process of designing, creating and implementing software solutions to meet business needs. This includes everything from creating new apps from the ground up to maintaining and improving existing systems. The purpose of software development is to create efficient, reliable and effective software solutions that improve business processes and provide value to stakeholders.

Web development is a subset of software development that entails designing software applications that operate on the Internet through the use of a web browser. Web applications can range from basic webpages to large commercial processes, and their design and development necessitate the use of specialized tools and techniques.

The web application covered in this project is a basic yet powerful three-layered implementation of Create, Read, Update, and Delete (CRUD) activities. The three independent levels of the three-layered architecture are the http/ handler layer, the service/ business layer, and the store/ datastore layer. Every layer has a unique set of jobs and responsibilities. The handler layer handles user input and output, the service layer handles business logic and rules, the store layer stores and retrieves data.

To ensure the app's quality and dependability, every layer of the architecture is thoroughly tested using suitable unit testing techniques.

The unit tests were created to test the functionality of the programmes and guarantee that each component of the application works as it should.

Furthermore, the application contains a middleware component that authenticates HTTP requests before passing them to the server. This prevents unauthorised access and guarantees that only authorised users have access to the resources. The middleware acts as a bridge between the client and the server, processing incoming requests and ensuring that they fulfil the relevant security requirements.

The web application described in this project, provides a strong illustration of how a three-layered architecture may be used to accomplish CRUD activities, combining middleware and unit testing for increased functionality and security.

1.2 Objectives

Employing industry best practises, to produce tested, organised, tidy, and maintainable web applications.

1.3 Motivation

Using the newest technology and development processes, the main goal of this project is to plan and create a web application that is quick, scalable, and safe.

1.4 Tools Used (Libraries or Frameworks)

A simple, dependable, and effective programming language called GO was created by Google developers to create code for apps. GO offers a number of the packages used in this project, including net/http that provides us with http client and server implementations. ‘json’ package implements json encoding/decoding. database/sql is another package that provides with SQL implementation.

The GOFR framework, created by ZopSmart with the intention of bringing efficiency, uniformity, and usability to all GO projects, is also used in this project.

WHAT IS ‘GOLANG’?

- The whole backend architecture of this project is developed in the Go programming language, including HTTP request execution, server response delivery, and programme logic creation.
- Go is an open-source programming language developed by Google[1] in 2009 with a focus on speed, simplicity, and efficiency.
- In contrast to other languages, Go is statically typed, which means that

variables must first be declared with a certain data type before they can be used. This approach improves programme reliability by assisting in compile-time error identification.

- Go also contains built-in concurrency support, allowing programmes to do many tasks at the same time. The language employs channels for communication between Goroutines, as well as lightweight threads known as goroutines.
- Go also has a garbage collector, which releases memory when it is no longer needed, making memory management easier for programmers and decreasing memory leaks.
- Also, Go prioritises code readability and maintains its syntax simple, making it easy to understand and update.
- Go features a robust standard library that provides programmers with a diverse set of useful functions and packages to work with.

1.5 Technical Requirements

GoLand IDE:

JetBrains created GoLand, IDE for the Go programming language which has capabilities like completion of code, debugging the code, version control, refactoring, testing.

Keyboard shortcuts and tools for code analysis which this IDE provides help developers in working more efficiently.

GitHub:

- GitHub is a version control and collaboration platform that enables developers to store and manage their code repositories in the cloud.
- It serves as a single area for developers to collaborate on projects, track code changes, and contribute to open-source software.

- GitHub is compatible with a wide range of programming languages and can be linked with a number of popular development tools and services.

Postman:

- Postman helps developers [7] to easily create, test, and document APIs.
- Postman may be used by developers to make HTTP requests to RESTful APIs, evaluate responses, and detect issues in real time.
- It also provides automated testing tools, documentation, and API sharing tools to developers and their teams.

Docker:

- Docker is a popular platform for developing, distributing, and executing containerized applications.[6]
- It provides a lightweight and efficient method of combining apps and their dependencies into a single unit that can be readily moved and operated across several contexts.
- Docker helps developers to build, test, and deploy applications more quickly and reliably, while simultaneously boosting scalability and portability.

1.5.1 Hardware Configuration

Table 1 : Hardware Configuration

Processor Intel® Core™ i5, 8-core CPU
RAM 16 GB
Hard Disk 512 GB
Monitor 14’’
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

Documentation for GO

The GO programming language comes with extensive documentation that developers may use while creating programmes. The GO documentation includes a detailed explanation to the language's syntax and usage, as well as numerous examples, best practices, and advice for developing efficient and secure code. The official GO documentation contains a language tour as well as reference and package papers that define the GO standard library.

Github and Git

The official documentation Introduction to Git and GitHub gives an introduction of version control systems and how they function. It focuses specifically on the Git version control system and how it combines with GitHub, a major web-based platform for code hosting and collaboration.

The manual describes how to utilise essential Git concepts including repositories, branches, commits, and merging in the context of software development. It also explains how to get started with Git and GitHub, including how to create accounts, create repositories, and collaborate with other developers.

Documentation for MySQL

MySQL is a prominent open-source[4] relational database management system for effectively storing, retrieving, and managing data and is built to manage massive volumes of data and several concurrent users, making it a popular choice for online applications that need a dependable database system.

Its SQL-based language supports a wide range of functions, including the ability to save and retrieve data, handle transactions, and run complicated queries.

GoMock

Gomock is a renowned mocking framework[2] that was created explicitly for the Go programming language. It enables developers to generate and design fake objects in order to test the behavior of their code without having an interaction with the actual implementation process.

Developers may use Gomock to construct effective unit tests that are segregated from external dependencies such as databases[3] or web services. This framework is well-known for integrating seamlessly with Go's built-in testing package, which makes it simple to use and set up.

CHAPTER 3 : SYSTEM DESIGN DIAGRAM

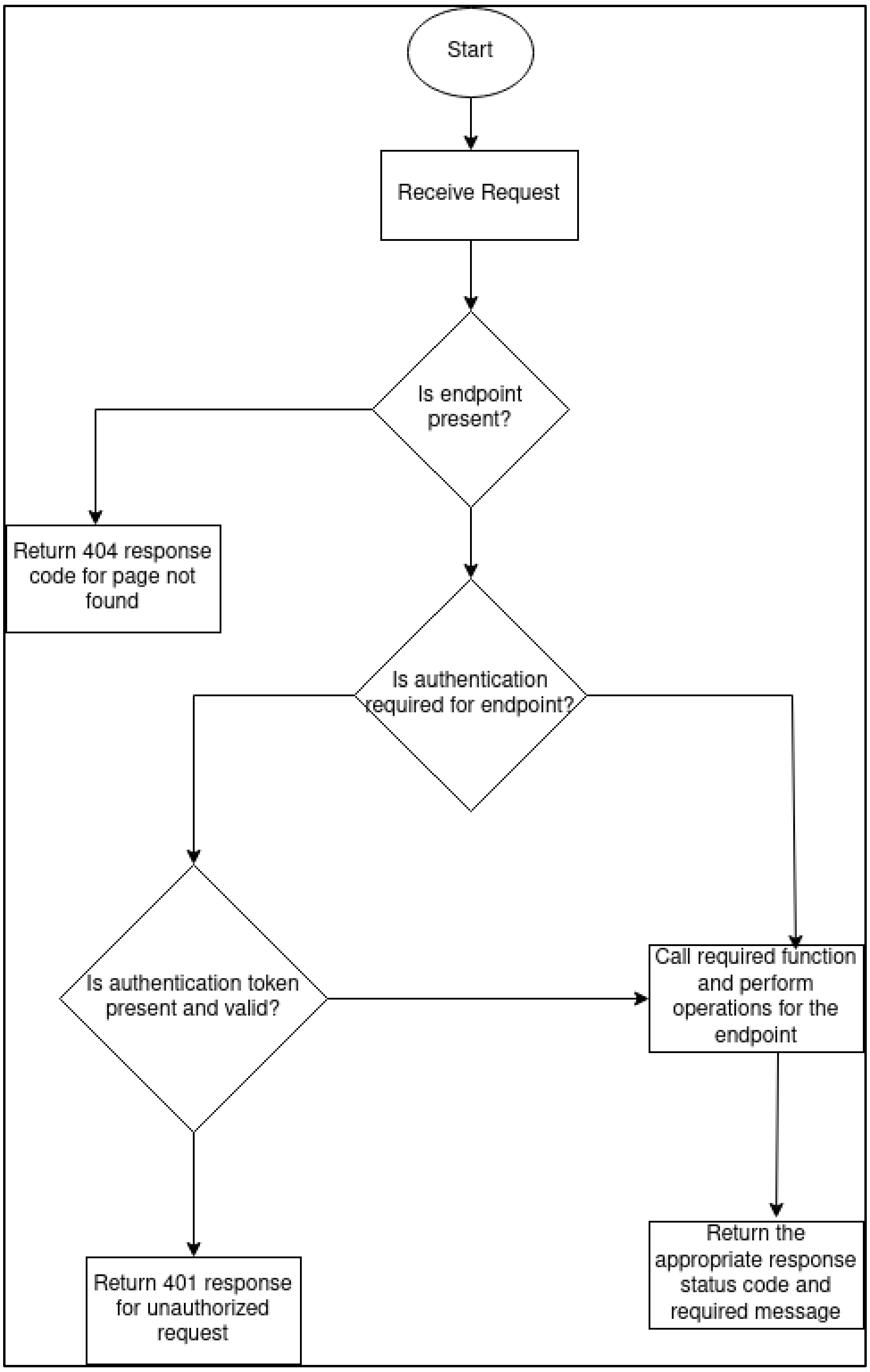

Start
Receive Request
Is endpoint present?
Return 404 response code for page not found
Is authentication required for endpoint?
Is authentication token present and valid?
Call required function and perform operations for the endpoint
Return 401 response for unauthorized request
Return the appropriate response status code and required message

Fig 3.1: System design diagram

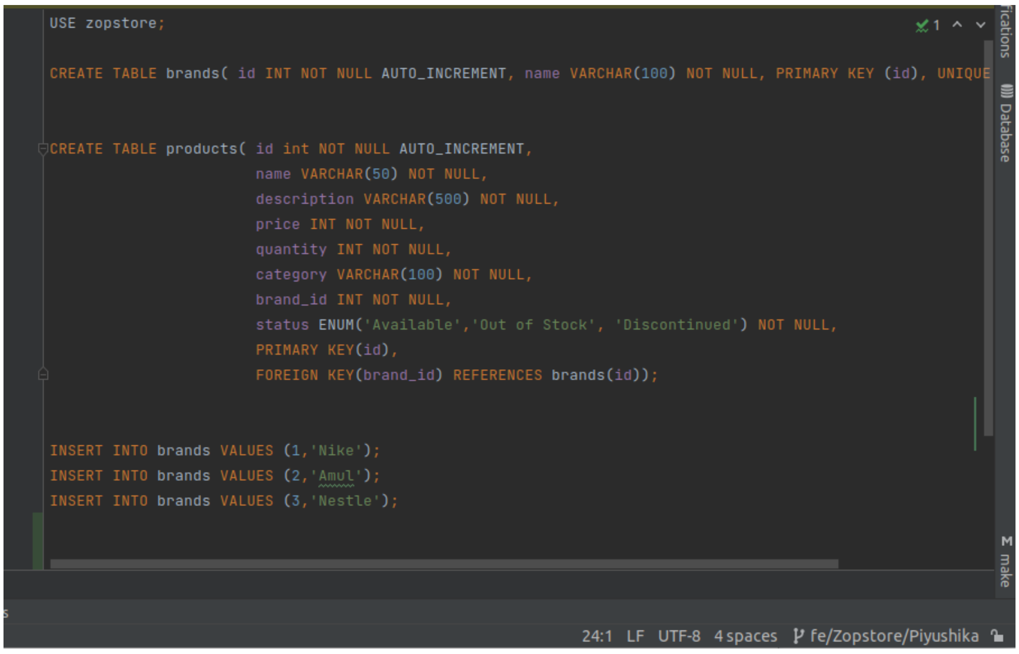
CHAPTER 4 : IMPLEMENTATION

4.1 Identification of features

Features of the web application:

- Creation of an entry of a brand and of a product with brand already existing with the store
- Updation of the existing brand records identified by brand ID
- Fetching details of brands based on ID
- Fetching details of all products
- Fetching details of products based on product name
- Updation of existing products identified by ID

4.2 SQL Schema



```sql
USE zopstore;

CREATE TABLE brands( id INT NOT NULL AUTO_INCREMENT, name VARCHAR(100) NOT NULL, PRIMARY KEY (id), UNIQUE


CREATE TABLE products( id int NOT NULL AUTO_INCREMENT,
                       name VARCHAR(50) NOT NULL,
                       description VARCHAR(500) NOT NULL,
                       price INT NOT NULL,
                       quantity INT NOT NULL,
                       category VARCHAR(100) NOT NULL,
                       brand_id INT NOT NULL,
                       status ENUM('Available','Out of Stock', 'Discontinued') NOT NULL,
                       PRIMARY KEY(id),
                       FOREIGN KEY(brand_id) REFERENCES brands(id));


INSERT INTO brands VALUES (1,'Nike');
INSERT INTO brands VALUES (2,'Amul');
INSERT INTO brands VALUES (3,'Nestle');
```

24:1 LF UTF-8 4 spaces fe/Zopstore/Piyushika

4.3 Study Material

Linux is a Unix-based[5] OS with both a command-line and graphical user interfaces. BASH, a shell for Unix systems that provides robust command-line features, is the standard shell in Linux.

In Linux, SUDO is a command that allows users to run commands as a different user, often the superuser. In the system environment, Linux runs lots of processes, and environment variables impact the programmes that use them. Within Linux, environment variables can be set through hidden files such as /.bashrc in the home folder (). The PATH variable indicates where to search for commands.

Linux has several built-in commands, including:

1. ls: lists the files in a folder
2. cd: changes the current directory
3. touch: creates an empty file
4. pwd: prints the path to the current working directory
5. mkdir: creates a new directory or directories. To create nested folders: mkdir -p folder/new
6. rmdir/rm: removes an empty directory (rmdir) or a directory and its contents (rm -rf folder)
7. mv: moves or renames files
8. cat: displays the contents of a file
9. chmod: sets file permissions flags to define who can read, write, or execute the file
10. vi: a text editor for creating and editing files.

Go Workspace

The Go language has a hierarchical structure with two root directories, including:

1. **src:** This directory contains Go source files.
2. **bin:** This directory contains executable commands.

In addition, the **GOPATH** environment variable is used to specify the location of your workspace, while the **GOROOT** variable is used to define where your Go SDK is located.

GO Packages

Each and every go program is made of packages. All the program in go environment 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:

Functions in go can take 0 or more arguments. A function may also take a variable number of arguments. Such a function is called a variadic function. Eg:

```
func multiply(a,b int) int{

return a*b

}
```

In the above example, the data type of the arguments can be clubbed if they are the same, and the return type of the function is specified after mentioning the name and arguments.

A function may return any number of arguments and is called from other functions using func_name(arguments) format.

Goimport:

In Go, every package that is imported must be formatted in a certain way. The built-in libraries are at the top of the bunch of imports. Then, third party imports are written like sql-driver etc. Financially, the local packages are mentioned at the bottom of the bunch. All of theses packages are imported in alphabetical order.

File Watchers:

File watchers is one of the tools offered which helps in making the code clean and concise.

go-imports: imports all the packages as and when needed as well as formats then as needed by go.

go-fmt: formats the code i.e. its indentation, line spacing, etc as per a standard which makes the code cleaner.

Variables

Variables in go can be declared in a lot of ways. The keyword 'var' is used to declare a single or a list of variables.

Eg: var age int

var (

age int

check bool

name, id string

)

Variables if not assigned any value, take the default value for that datatype. Zero values for:

int= 0

string= “”

bool= false

Except for global variables, shorthand notation can be used in Go, to declare and define the value of a variable at once. := operator is used for the same.

Eg: age:=22

The above statement assigns variable age the type of the value on the right hand side i.e ‘int’ and the value 22.

Global variables are mostly discouraged in GoLang, and it is a better practice to define variables in functions.

FOR

There is only For loop in go and no while or do while loop.

Eg: for i:0;i<10;i++{}

The for loop can behave like a while loop as follows:

Eg: for ;j<10;j++{}

IF

Parentheses are optional but curly brackets are required in the if statements of the Go language. If you are using a short statement, you can start the if statement with it before running the actual statement, but it needs to be followed by a semicolon. The else blocks can access any of the variables stated in the short statement of an if statement.

SWITCH

In contrast to other languages, the switch statement in Go behaves differently. Go only executes the selected case, not the ones that come after it. Furthermore, Go does not require the usage of break statements after each case. In contrast to other programming languages, Go's switch statement's cases and values can both be anything, not just integers.

In Go, a switch statement without a condition is similar to an if statement, while a switch statement with a condition can be used to form an if-else statement. In a Go switch statement, it is not permitted to have two cases with the same condition. Otherwise, a compile time error or type mismatch error would occur.

DEFER

The Go programming language has a special feature called "defer" that delays the execution of a function until the enclosing function has finished and returned. The arguments supplied to a deferred function are evaluated immediately after the defer statement is executed, but the function call itself is not executed until the enclosing function has finished running and is about to return.

This makes defer statements handy for tasks like shutting files, releasing resources, and other tasks that need to be completed after a function is done running.

The most recently deferred function call will be executed first because Go employs a stack to keep track of deferred function calls. "Last in, first out" (LIFO) sequence is used in this situation.

The "defer" keyword is used followed by the function that has to be deferred in a Go program to define a defer statement. This makes it simple to guarantee that specific actions are always carried out, despite a mistake or a panic.

POINTERS

In Go, pointers are variables that store a value's memory address. They enable oblique access to memory-stored values.

Use the * symbol and the type of the variable it will point to to declare a pointer. As an illustration, the statement var p *int declares a pointer with the name p that points to an integer value.

Use the & operator after the value to obtain the value's memory address. For instance, x:= 10; p:= &x assigns the pointer p the memory location of x.

Use the * operator and the pointer name to dereference a pointer. As an illustration, *p receives the value that p points to. Using *p = 20, a new value can also be assigned to the memory location pointed to by a pointer.

A pointer's zero value is nil. A pointer that is nil indicates that it points to no memory location.

Go creates a replica of the value you supply as an argument to a function. However, you can indirectly change the initial value when you give a pointer to a function.

Pointers can be used to build intricate data structures such as linked lists, trees, and graphs.

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

)
```

Using structures, you can combine fields of various types into a single composite type. For instance, a struct called "V" can be constructed with two integer-type entries, X and Y. The print function can be used to generate a new V value with the parameters X=1 and Y=2, which can then be printed to the console as an example of how to use this struct. On a variable of type V, structural fields can be accessed using dot notation.

A struct pointer can be declared using the & operator and used to access struct fields with a pointer. One can declare and initialize a variable of type *V called p to the address of a freshly formed V value, for instance. By utilizing the -> operator on the pointer variable, one can access structural fields.

Additionally, field names in the format X: 11—the field name is followed by a colon and the new value—can be used to assign values to struct fields. Unspecified fields will automatically receive a value of zero. When only a few of a struct's many fields need to be initialized with non-zero values, this can be helpful.

ARRAYS

Arrays are a data structure used to store a fixed number of elements of the same type.

Once an array is declared, its length cannot be modified because it is a component of the array's type.

In Go, the type and length of an array must be declared. For instance, you can write the following to declare an array of integers with no length specified:

```
int var arr []
```

You can use the following syntax to set the array's initial values:

```
a:= []int 1, 2
```

In this instance, the array has two starting elements: 1 and 2.

It is vital to remember that Go arrays are zero-indexed, which means that the index of the first element is 0. Using the index enclosed in square brackets, you can access an element in an array as follows:

```
arr[0] = 10
```

The first element of the array will now have the value 10 assigned to it.

SLICES

Slices are frequently utilized as dynamically scaled arrays in Go. A slice is a pointer to an array, and while not fixed, a slice's length is determined by the type. We can declare a slice with an inclusive range of indices, such as "s[low:high]". Slices just describe a portion of an array, they do not store any data.

When we modify a slice's elements, the corresponding elements of its underlying array are also modified. This has an effect on other slices that have similar elements. For instance, if we set 'a:= names[1:2]', 'a' would be a slice pointing to a particular area of the 'names' array.

A slice literal, which is an array literal without the length, can be used to generate a slice. A structure with a pointer to the underlying array, the slice's length, and its capacity is used to internally represent a slice. The number of elements in a slice is referred to as its length, while the slice's capacity is the most elements it can hold.

The built-in "make" function can be used to build a slice; it generates a zeroed array and returns a slice that refers to the array. By providing an integer input to the "make" command, we may specify the length of the slice. By providing a second integer input to the "make" command, we can also specify the slice's maximum capacity.

The "append" function allows us to add elements to a slice. The capacity of the slice is automatically doubled if it is less than the total amount of elements to be appended. When the capacity doubles, a new slice and more memory are created, but the underlying array is left unchanged.

RANGE

Iterating across a slice or map is done using a Range loop, a sort of for loop. Each time an iteration occurs, an index is returned along with a duplicate of the value at that position. You can assign "_" to the relevant variable to skip either the index or the value. For instance, skipping the value and index is indicated by the expressions "for j, _:= range power" and "for _, value:= range power." If we only require one of the variables, we can completely disregard the other.

A number can be multiplied or divided by a power of two using the bitwise shift operators and >>, respectively. For instance, "1 5" evaluates to 32 and signifies "1 times 2 raised to the power of 5". Similar to "32 >> 5", which evaluates to 1, "32 >> 5" signifies "32 divided by 2 raised to the power of 5"."

MAPS

Go has a built-in data structure called a map that associates keys with values. A map's keys must be distinct and can only be found once. A map with a zero value is considered to be nil, meaning it has no keys. We employ the built-in make function with the suitable type to produce a map. For instance, we can use the code below to make a map that converts texts to Vertex structs:

```
var m map[string]Vertex
m= make(map[string]Vertex)
```

We can also initialize a map with some key-value pairs using a map literal. For example:

```
M := map[string]int{
“Year”:2023
“Age”: 22
}
```

The syntax m[key] = value can be used to add or modify values in a map. The formula value = m[key] can be used to obtain a value. Use the built-in delete function with the map and the desired key, as in delete(m, key), to remove a key from a map. We can use the syntax value of ok = m[key] to determine whether a key is present in a map. Ok is a boolean variable that is true if the key is present in the map and false otherwise. Value is the zero value for the element type of the map if the key is absent from the map.

It is important to remember that a key cannot be a slice in any particular map. We can utilize maps to construct different data structures like hash tables, sets, and graphs because a map value can be a slice.

Function Closure

Functions in Go may access variables outside of their own scope thanks to the useful feature known as function closures. In other words, closures enable functions to "remember" the values of variables that were included in their scope at the time of their definition, even if those variables are not included in the scope at the time of the function call.

In Go, defining a function inside of another function results in a closure. After the outer function has returned and the inner function has access to the variables of the outside function. This is made feasible by the fact that Go produces a fresh closure every time the outer function is called, complete with a unique set of variables that are unrelated to those in any previous closures.

Closures can be used, for example, to build generator functions. For instance, you might create a function that, when called, returns a different function that produces a series of integers. The closure can "remember" the state of the generator, allowing it to pick up where it left off and continue producing numbers.

Closures can also be utilized to build filtering or transformers. One could, for instance, create a closure that accepts a slice of integers and returns a new slice that only contains even values. The filter condition can be "remembered" by the closure, making it possible to apply it on many slices without having to redefine the filter function each time.

METHODS

A method in Go is comparable to a function with a unique receiver parameter. A receiver whose type is defined in the same package as the method can be used to define a method. Between the func keyword and the method name is where the receiver can be found in its own argument list.

A type, even one that is not a struct, can have methods written on it. A method, including built-in types like int, can only be declared with a receiver whose type is defined in the same package as the method.

Use a pointer receiver to prevent copying the value on each method call and to let the method edit the value the receiver points to, respectively. When the recipient is a big struct, it might be more effective to do this. There should not be a combination of value and pointer receivers in any methods on a certain type.

A pointer receiver argument can refer to methods with both value and pointer receivers while a value receiver argument can only refer to methods with a value receiver when defining a receiver argument. When employing slices or maps, for example, or when we do not want changes to be reflected in the original value, we utilize value receivers. On the other side, we can use a pointer receiver to access methods either way or to ensure that changes are reflected. When the struct is big, pointer receivers are also helpful for preventing duplicate copies.

INTERFACES

A set of methods with specific signatures that an implementing value must have are defined by an interface type. The value of an interface type can include any value that implements those methods, allowing diverse value types to implement the same set of methods. An interface type can be defined using the syntax "type name interface," where "name" denotes the name of the interface. Without any explicit expression of purpose or use of the "implements" keyword, interfaces are implicitly implemented in Go.

Any value that implements the interface's methods may be stored in an

interface value once it has been constructed. Calling methods on the interface will not result in a null pointer exception if the concrete value held by the interface is nil. However, because there is no type inside the interface tuple to specify which concrete method to call, calling a method on a nil interface value will result in a run-time error.
A method and a pointer receiver can be used to change the value that the receiver points to. A larger struct may also benefit from not replicating the value each time a method is called. All methods on a given type in Go should have either a value receiver or a pointer receiver; neither should be present.

Interface, which has no methods and accepts any kind of value, is empty. In Go, every type implements at least zero methods, hence code that deals with values of unknown types frequently uses empty interfaces.

In conclusion, a Go interface type defines a collection of methods with specific signatures that can be implemented by various types. Any value that carries out those methods can be contained in an interface value. Calling a method on a nil interface value will result in a run-time error since an interface with a nil concrete value is itself non-nil. The value that the receiver points to can be changed using a pointer receiver, and any type of value can be stored in the empty interface.

TYPE ASSERTION & TYPE SWITCH

Accessing the concrete value that underlies an interface value is possible through a type assertion. By asserting that the interface value "i" has a concrete value of type "T" and assigning the underlying value of "T" to the variable "t," we can say that "t:=i.(T)". The statement will result in a panic if the interface value "i" does not contain a "T" value.

The underlying value and a boolean value that indicates if the assertion was successful are the two values that a type assertion in Go can deliver. For example, "t, ok:= i.(T)" will give "t" the underlying value and "ok" will be true if "i" contains a "T" value. "ok" will be false and "t" will be zero if "i" does not have a "T" value.

Type SWITCH

Multiple type assertions may be made sequentially using a type switch. In a type switch, cases are specified rather than values, and these types are compared to the type of the value carried by the given interface value. This is comparable to a switch statement in general. A type switch has the following syntax:

```
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
}
```

In a type switch, the declaration syntax is the same as in a type assertion i.(T), but the term type is used in place of a specific type T.

REST- REpresentational State Transfer

A set of standards for creating web services that follow the Representational State Transfer (REST) principles is known as the principles of RESTful design. Some principles of RESTful design are as follows:

- Client-server architecture: A RESTful web service ought to be developed with a client-server architecture, which requires the client and server components to be separated. The resulting division of responsibilities increases the system's flexibility and scalability.

- Statelessness: Every request should provide all of the information needed to complete it in a RESTful architecture. This means that no client-specific data or session state should be retained on the server. Instead, each request must include all relevant information from the client.
- Cacheability: Caching should be considered while developing RESTful web services. This implies that server responses should state if and for what duration the response can be kept. The server's load reduces while performance is improved.
- Layered architecture: When developing RESTful web services, an architecture of layers should be employed. This implies that there should be numerous layers between the client and the server. Each layer being in control of a specific set of responsibilities is flexible and expandable.

HTTP RESPONSE STATUS CODES

- An HTTP response status code indicates the status of the request made by a client to a server and also gives additional information about the response. A three-digit number identifies each among the five status code classes. The first digit indicates the response's class, while the next two digits provide further information about that specific response.

- The first type of response code is 1xx, which indicates that the website's server continues to process the request at the moment so the client should continue to wait for the end result of the request. These codes are purely for informational reasons and do not indicate whether a request was successful or unsuccessful.
- The second category of response codes are the 2xx response code, which indicate that the request for data was successful and that the website's server was able to satisfy the client's request. The most common 2xx code value of 200 indicates the success of the request and a response

from the server.

- The third group of response codes is 3xx codes, which indicate that the client needs to take more action to accomplish the request. The codes above are used for redirected links, which is when the server sends the client to a different site to obtain the requested item.
- The fourth category of response codes consists of the 4xx response codes, that represent client-side problems. 404 code is the most commonin these which indicates that a resource was not found.
- The final group of response codes are the 5xx codes, that indicate that the server encountered an issue. 500 is the most common among these signifying internal server error.

HTTP package

Common HTTP Protocols:

- Create: POST method for adding new data
- Read: GET method for retrieving data
- Update: PUT method for updating data
- Delete: DELETE method for deleting data

ServeMux(Multiplexer)

- ServeMux is an HTTP request multiplexer that executes requests by matching their URLs to the correct handlers.
- Use http.NewServeMux to create a new ServeMux and the Handle and HandleFunc methods to add a URL handler.
- A string and a http.Handler are accepted by the handle method. An interface with the ServeHTTP method is provided by the Handle method's second parameter.
- The handler implementation is sent to HandleFunc as a function along with the path for which it should be called.
- Server is an HTTP server that allows you to manage various routes and paths by passing an instance of a ServeMux.

- You can completely omit the ServeMux if you have a route or path that you want to manage by passing an instance of a http.Handler instead.
- The net/http package contains the Handle and HandleFunc methods as well as the DefaultServeMux.
- If you have used http.Handle and/or http.HandleFunc to define the handler implementations for the corresponding routes, the handler parameter for the http.ListenAndServe method, which starts the HTTP server, can be nil.

HTTPtest Package

- httpRequest: By simulating HTTP requests and responses, the HTTP Test package offers a straightforward way to test HTTP handlers. For testing purposes, a new incoming server request can be generated and delivered to an HTTP handler using the httptest.NewRequest function. A http.Request type instance is returned by this function.

- httpResponseWriter: In order to test the HTTP handler's response, we must record every piece of information that it will include in the response and then retrieve the information that was included later. ResponseRecorder, a type offered by the HTTP Test package, implements the http.ResponseWriter interface and saves the HTTP response for further review in tests. Using the httptest.NewRecorder function, we can construct a ResponseRecorder instance.

- w.Result(): The ResponseRecorder's Result() function can be used to get the HTTP response object after the HTTP handler has finished writing the response. The response produced by the handler is represented by a pointer to an instance of http.Response that is returned by the Result() function. The StatusCode, Header, Body, and optional Trailer fields of this returned object will all be filled in. The HTTP handler's functionality can then be checked by looking at these fields.

Layered Architecture:

Our application's layers are intended to be independent of one another, and they communicate with one another via clearly defined interfaces. Our codebase may be made modular, readable, and maintainable using this strategy. The HTTP layer, the Service layer, and the Store layer are the three separate layers of the application.

The HTTP layer is in charge of checking headers, managing request body data, and validating query and path parameters. The business logic is put into practice by the Service layer, which also interacts with the Store layer to carry out any required data storage actions. Database-level queries must be implemented by the Store layer.

Through clearly defined interfaces that outline input parameters and output types, each layer communicates with the one underneath it and the one above it. This makes it simple to test each layer by simulating the server, database, or interface as necessary.

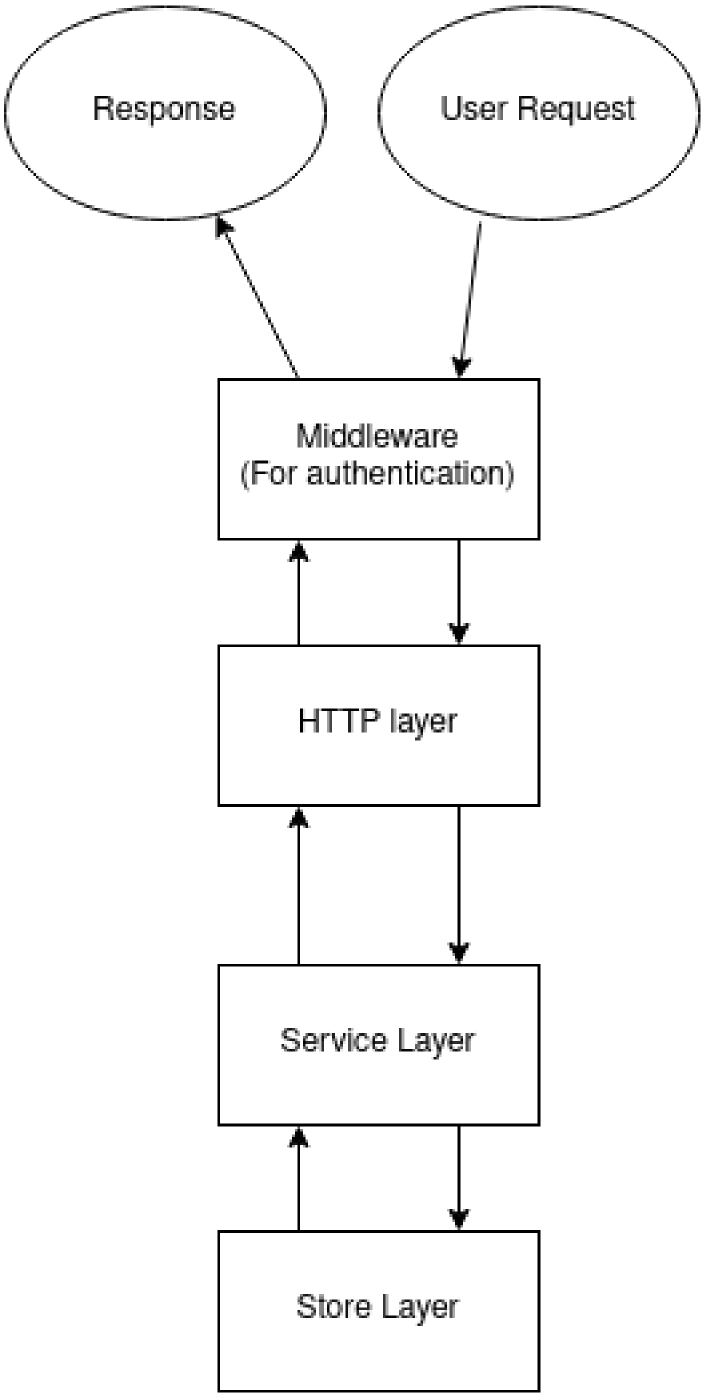


Fig 4.1: Three layered architecture flow

Dependency Injection:

Dependency injection is a technique used in software engineering that promotes the separation of concerns and the principle of modularity. It enables us to produce more tested, maintainable, and modular code.

Whenever an object or struct is initialized via dependency injection, its dependencies are supplied. This implies that we consciously decide when to employ previously created instances of our dependencies and when to build new ones. By doing this, we can simply change the dependency's implementation without having to change the code that depends on it.

This method also aids in simplifying and decoupling our code. Traditionally, dependencies are created and managed by objects in object-oriented programming. As a result, objects become tightly coupled, which makes it more difficult to test and manage them. We transfer this responsibility from the objects to a different component by using dependency injection. In this manner, objects are loosely connected to their dependents, which facilitates testing and maintenance.

UNIT TESTING

Unit testing is a crucial step in the development of software since it helps to assure code quality and catch problems as they arise. Go's built-in testing package, which offers a straightforward and effective way to build and run tests, makes unit testing simple.

Simply create a file with the "_test.go" suffix and import the "testing" package to construct a test in Go. Once that is done, create a function with the signature "func TestXxx(t *testing.T)" where "Xxx" stands for the name of the function being tested. You can create test cases within this function that use the "t" argument to report test results.

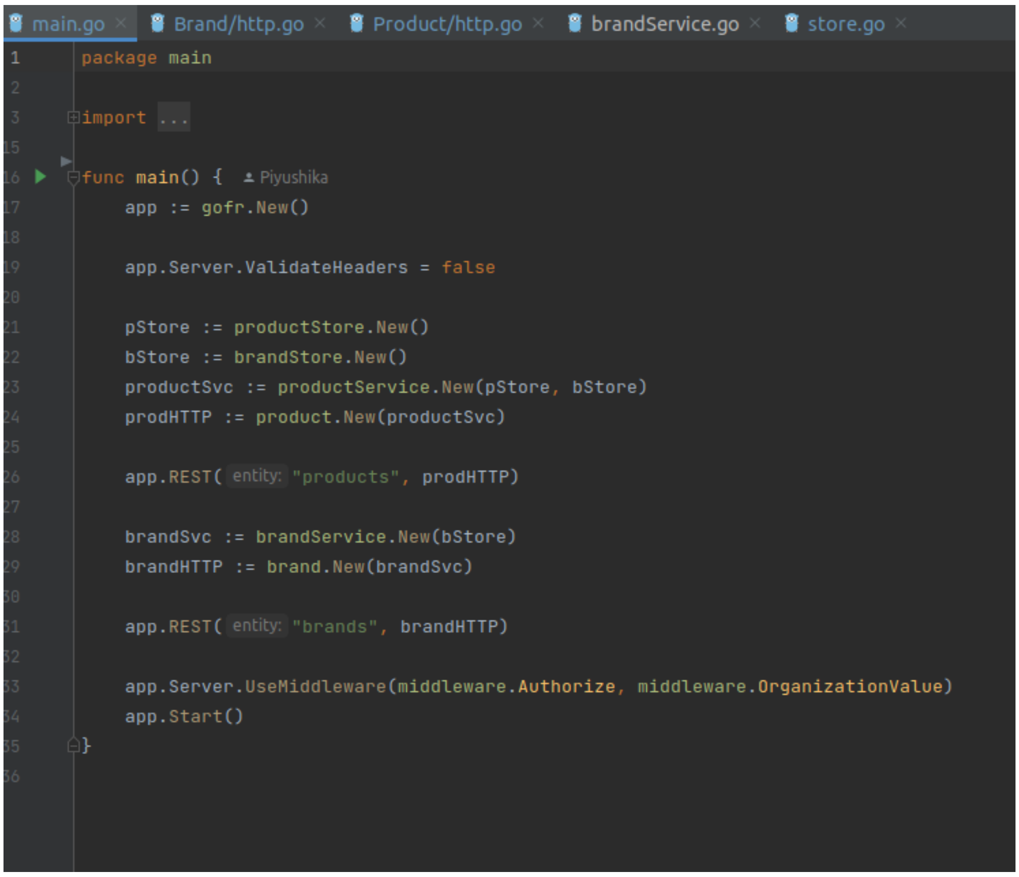
The testing package in Go also includes helpful functions to indicate problems and failures, such as "t.Errorf" and "t.Fatalf." Additionally, performance testing and optimization are possible when creating benchmarks using the "func BenchmarkXxx(b *testing.B)" signature.

Table-driven testing, where test cases are described in a table format to make it easier to create and manage test cases, is a key idea in Go unit testing. This method enhances code coverage and identifies edge cases that conventional testing could have overlooked.

4.4 Layer working and Swagger Documentation

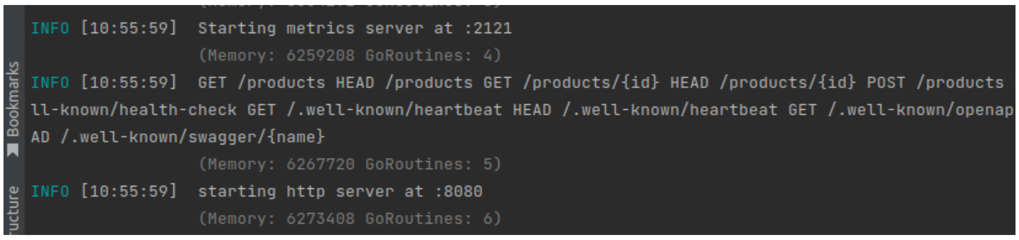
Code

main.go : source file all routers



```go
package main

import ...

func main() {
	app := gofr.New()

	app.Server.ValidateHeaders = false

	pStore := productStore.New()
	bStore := brandStore.New()
	productSvc := productService.New(pStore, bStore)
	prodHTTP := product.New(productSvc)

	app.REST( entity: "products", prodHTTP)

	brandSvc := brandService.New(bStore)
	brandHTTP := brand.New(brandSvc)

	app.REST( entity: "brands", brandHTTP)

	app.Server.UseMiddleware(middleware.Authorize, middleware.OrganizationValue)
	app.Start()
}
```

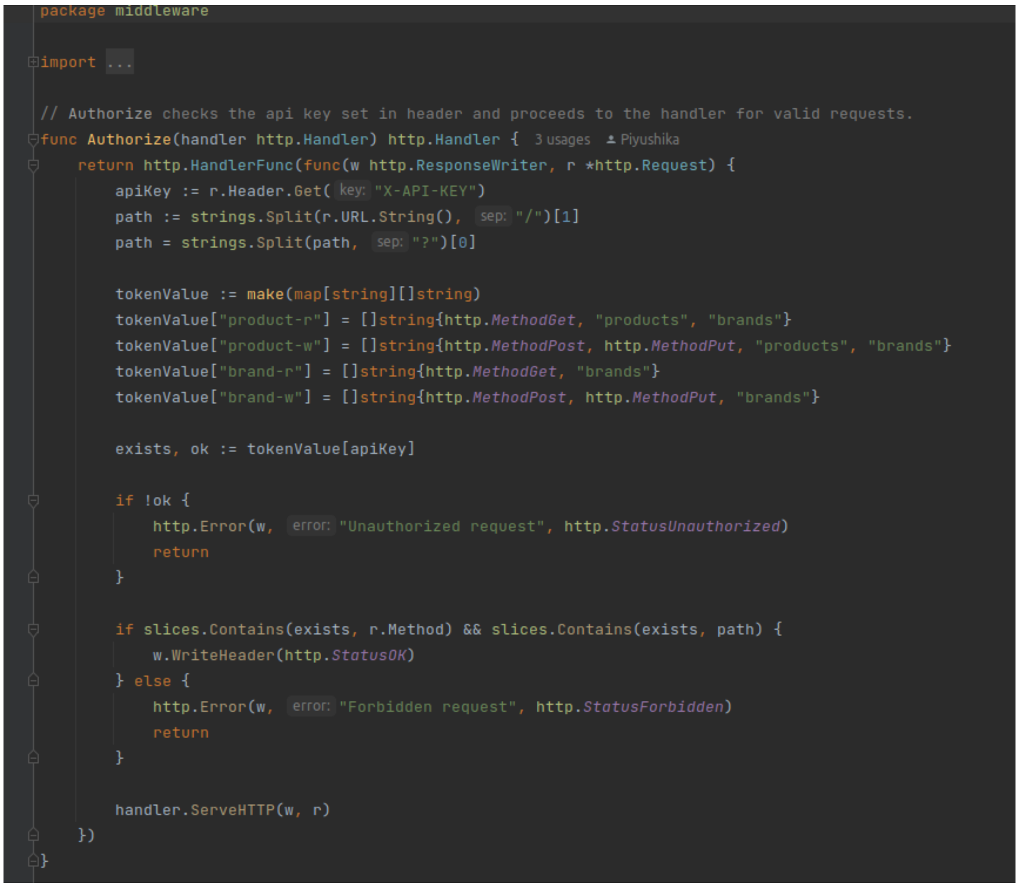
Upon running the main file using : go run main.go,



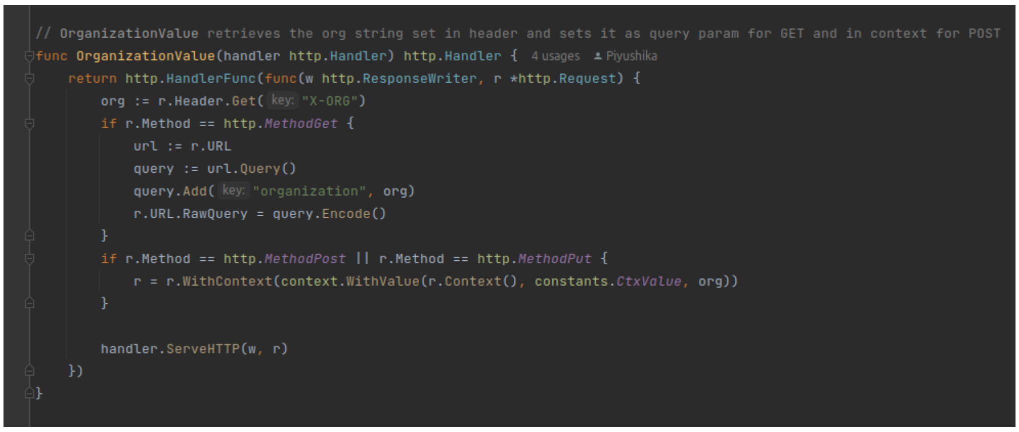
```
INFO [10:55:59]  Starting metrics server at :2121
                 (Memory: 6259208 GoRoutines: 4)
INFO [10:55:59]  GET /products HEAD /products GET /products/{id} HEAD /products/{id} POST /products
ll-known/health-check GET /.well-known/heartbeat HEAD /.well-known/heartbeat GET /.well-known/openap
AD /.well-known/swagger/{name}
                 (Memory: 6267720 GoRoutines: 5)
INFO [10:55:59]  starting http server at :8080
                 (Memory: 6273408 GoRoutines: 6)
```

- Http server established on port 8080.
- Metric server that is part of the GOFR framework, starts on port 2121.
- Upon hitting endpoint on port 8080 on postman, the control transfers to main and then our middleware for authentication.

MIDDLEWARE



```go
package middleware

import ...

// Authorize checks the api key set in header and proceeds to the handler for valid requests.
func Authorize(handler http.Handler) http.Handler {  3 usages  Piyushika
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        apiKey := r.Header.Get( key: "X-API-KEY")
        path := strings.Split(r.URL.String(),  sep: "/")[1]
        path = strings.Split(path,  sep: "?")[0]

        tokenValue := make(map[string][]string)
        tokenValue["product-r"] = []string{http.MethodGet, "products", "brands"}
        tokenValue["product-w"] = []string{http.MethodPost, http.MethodPut, "products", "brands"}
        tokenValue["brand-r"] = []string{http.MethodGet, "brands"}
        tokenValue["brand-w"] = []string{http.MethodPost, http.MethodPut, "brands"}

        exists, ok := tokenValue[apiKey]

        if !ok {
            http.Error(w,  error: "Unauthorized request", http.StatusUnauthorized)
            return
        }

        if slices.Contains(exists, r.Method) && slices.Contains(exists, path) {
            w.WriteHeader(http.StatusOK)
        } else {
            http.Error(w,  error: "Forbidden request", http.StatusForbidden)
            return
        }

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



```
// OrganizationValue retrieves the org string set in header and sets it as query param for GET and in context for POST
func OrganizationValue(handler http.Handler) http.Handler {  4 usages  Piyushika
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        org := r.Header.Get( key: "X-ORG")
        if r.Method == http.MethodGet {
            url := r.URL
            query := url.Query()
            query.Add( key: "organization", org)
            r.URL.RawQuery = query.Encode()
        }
        if r.Method == http.MethodPost || r.Method == http.MethodPut {
            r = r.WithContext(context.WithValue(r.Context(), constants.CtxValue, org))
        }

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

If required authentication is valid, the control then goes over to the handler layer.

For Product entity:

HANDLER LAYER

http pkg : in the handler layer, we unmarshal the json body (in create and update), check for invalid parameters - id and body and then send it to the service layer.

Index: Index for products performs the GetAll and GetByName functionality as the response can have multiple records.

Read: This function performs the GetByID functionality, which retrieves record for a particular product ID.

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

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

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.

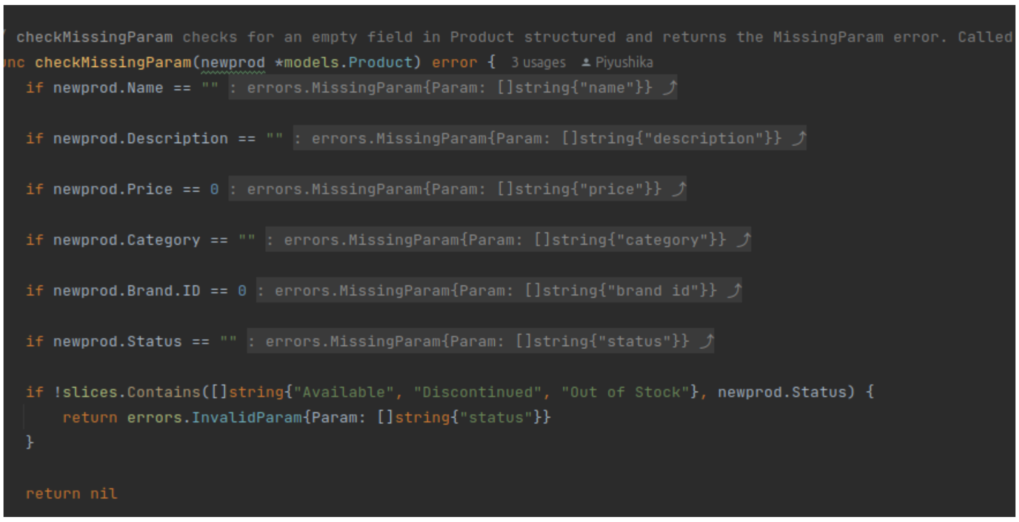
GetProduct: this function calls the service layer with product id received from handler and bool value corresponding to inclusion of brand details in product record. Any error returned by the service layer call is in turn returned to the handler, else a response with product details is sent.

GetProductByName: This function returns the response and error value received from the store layer function GetByName.

GetAllProducts: A slice of all products is sent as response from the service layer if no error is received from the store layer.

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

UpdateProduct: We check if the data to be updated is all valid, i.e. no missing parameters. If no error occurs, store layer function UpdateProduct is called by passing the context, id and product details. If rows affected value returned is 0 then response for no record found is returned and GetByID for brand’s store layer is called, to include brand name in response body.



```
 checkMissingParam checks for an empty field in Product structured and returns the MissingParam error. Called
nc checkMissingParam(newprod *models.Product) error {  3 usages  Piyushika
  if newprod.Name == "" : errors.MissingParam{Param: []string{"name"}}

  if newprod.Description == "" : errors.MissingParam{Param: []string{"description"}}

  if newprod.Price == 0 : errors.MissingParam{Param: []string{"price"}}

  if newprod.Category == "" : errors.MissingParam{Param: []string{"category"}}

  if newprod.Brand.ID == 0 : errors.MissingParam{Param: []string{"brand id"}}

  if newprod.Status == "" : errors.MissingParam{Param: []string{"status"}}

  if !slices.Contains([]string{"Available", "Discontinued", "Out of Stock"}, newprod.Status) {
      return errors.InvalidParam{Param: []string{"status"}}
  }

  return nil
```

DATA STORE LAYER

Store pkg: 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 record in the db by executing sql query and storing data in db

Update: In this function an existing record is updated in the db with the data received from layers above.

Get By Id: retrieves product details for a given ID from the db

Get By Name: retrieves product details from the db for a given product name

Get All : retrieves all product details from the db

For Brand Entity:

HANDLER LAYER

http.go : in the handler layer, we unmarshal the json body (in create and update), check for invalid parameters - id and body and then send it to the service layer.

Read: for a valid id given as path parameter, this function calls the service layer GetBrand function. The response returned from the service layer is then returned if no error occurs.

Create: After unmarshalling the request body, for a valid request, service layer's CreateBrand function is called and if no error is returned then the created brand details are returned as response.

Update: For a given id, after unmarshalling the request body, for a valid request, service layer's UpdateBrand is called. For 0 rows affected, error for no record found is returned. Else the updated brand details are sent as response.

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.

GetBrand: Calls store layer's GetByID function by passing the id received from the handler layer. For nil error, the brand details response is returned to the handler layer.

Create Brand: After checking for missing name parameter, if no error occurs, store layer's Create function is called.

Update Brand: After checking for missing name parameter, if no error occurs, store layer's Update function is called.

DATA STORE LAYER

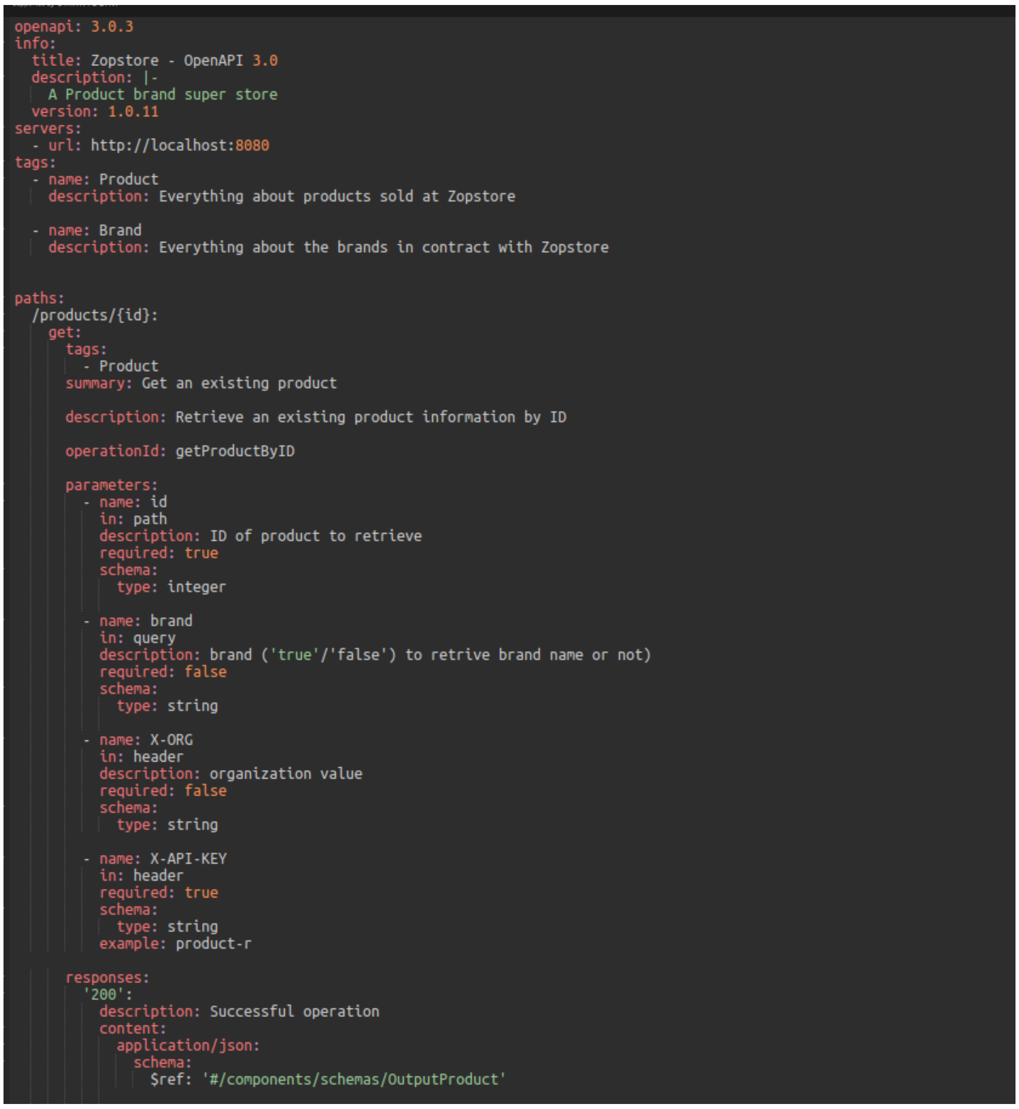
store.go: In this layer we write the query to store data in our database and check there are no db based errors.

Create: This layer interacts with the database to insert brand record details sent from layers above.

GetByID: This function retrieves id,name of brand from db for a particular brand ID received from layers above. For any scan errors, 'entity not found' error is returned to the service layer.

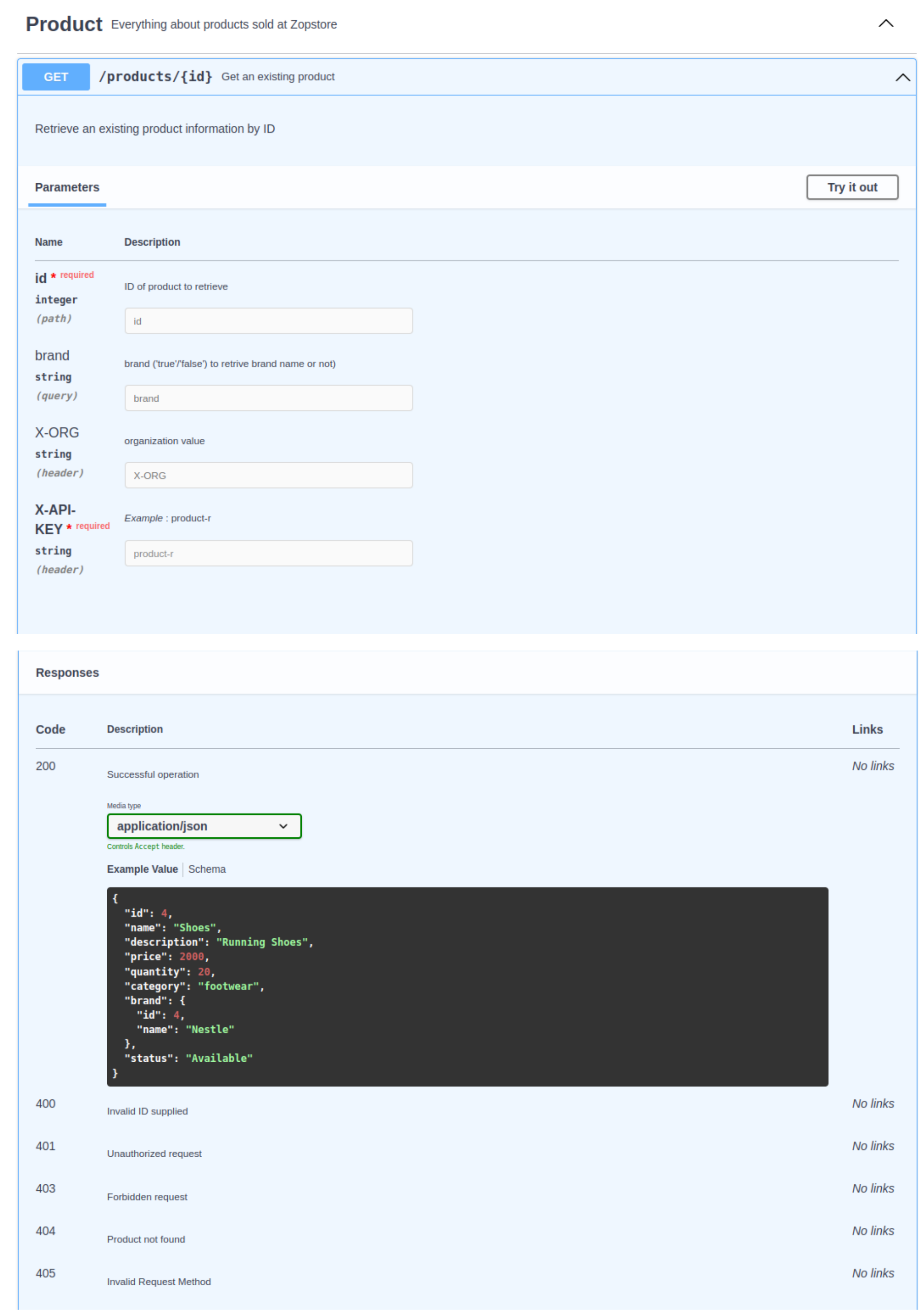
Update: Updates brand details for a particular brand ID

4.5 SWAGGER DOCUMENTATION



```yaml
openapi: 3.0.3
info:
  title: Zopstore - OpenAPI 3.0
  description: |-
    A Product brand super store
  version: 1.0.11
servers:
  - url: http://localhost:8080
tags:
  - name: Product
    description: Everything about products sold at Zopstore

  - name: Brand
    description: Everything about the brands in contract with Zopstore


paths:
  /products/{id}:
    get:
      tags:
        - Product
      summary: Get an existing product

      description: Retrieve an existing product information by ID

      operationId: getProductByID

      parameters:
        - name: id
          in: path
          description: ID of product to retrieve
          required: true
          schema:
            type: integer

        - name: brand
          in: query
          description: brand ('true'/'false') to retrive brand name or not)
          required: false
          schema:
            type: string

        - name: X-ORG
          in: header
          description: organization value
          required: false
          schema:
            type: string

        - name: X-API-KEY
          in: header
          required: true
          schema:
            type: string
          example: product-r

      responses:
        '200':
          description: Successful operation
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/OutputProduct'
```

PRODUCT:



Product Everything about products sold at Zopstore

GET /products/{id} Get an existing product

Retrieve an existing product information by ID

Parameters

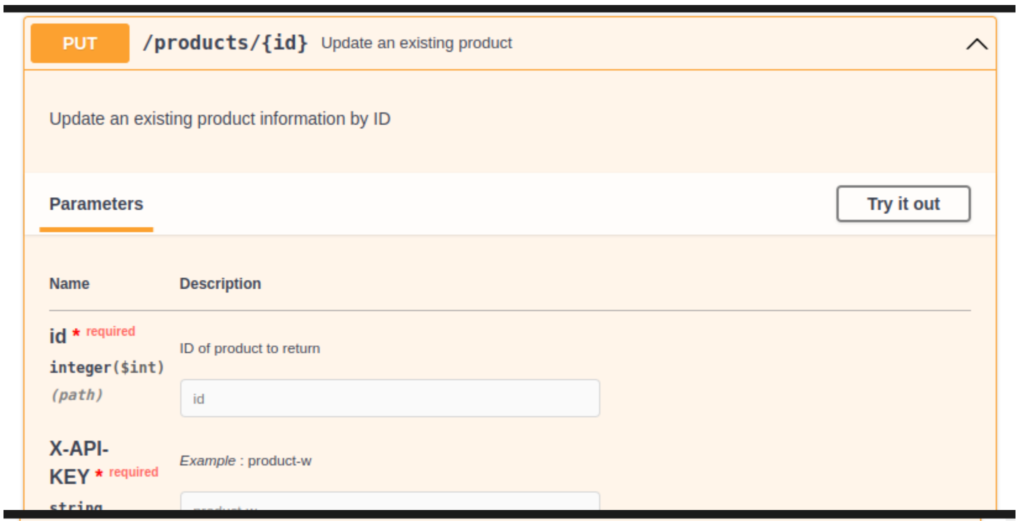
Try it out

Name	Description
id * required integer (path)	ID of product to retrieve
brand string (query)	brand ('true'/'false') to retrive brand name or not)
X-ORG string (header)	organization value
X-API-KEY * required string (header)	*Example* : product-r

Responses

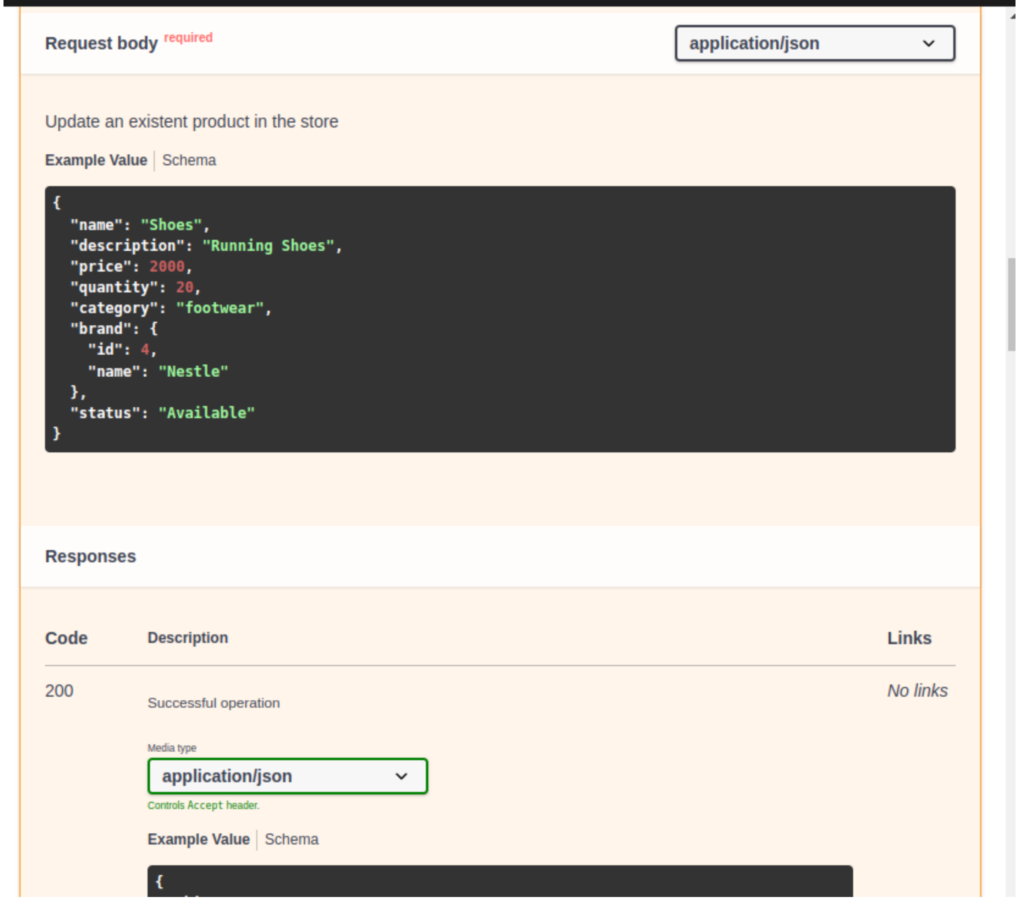
Code	Description	Links
200	Successful operation Media type: application/json Controls Accept header. Example Value | Schema	*No links*
400	Invalid ID supplied	*No links*
401	Unauthorized request	*No links*
403	Forbidden request	*No links*
404	Product not found	*No links*
405	Invalid Request Method	*No links*

```
{
  "id": 4,
  "name": "Shoes",
  "description": "Running Shoes",
  "price": 2000,
  "quantity": 20,
  "category": "footwear",
  "brand": {
    "id": 4,
    "name": "Nestle"
  },
  "status": "Available"
}
```


PUT /products/{id} Update an existing product

Update an existing product information by ID

Name	Description
id * required integer($int) (path)	ID of product to return id
X-API-KEY * required string	*Example* : product-w


Request body required

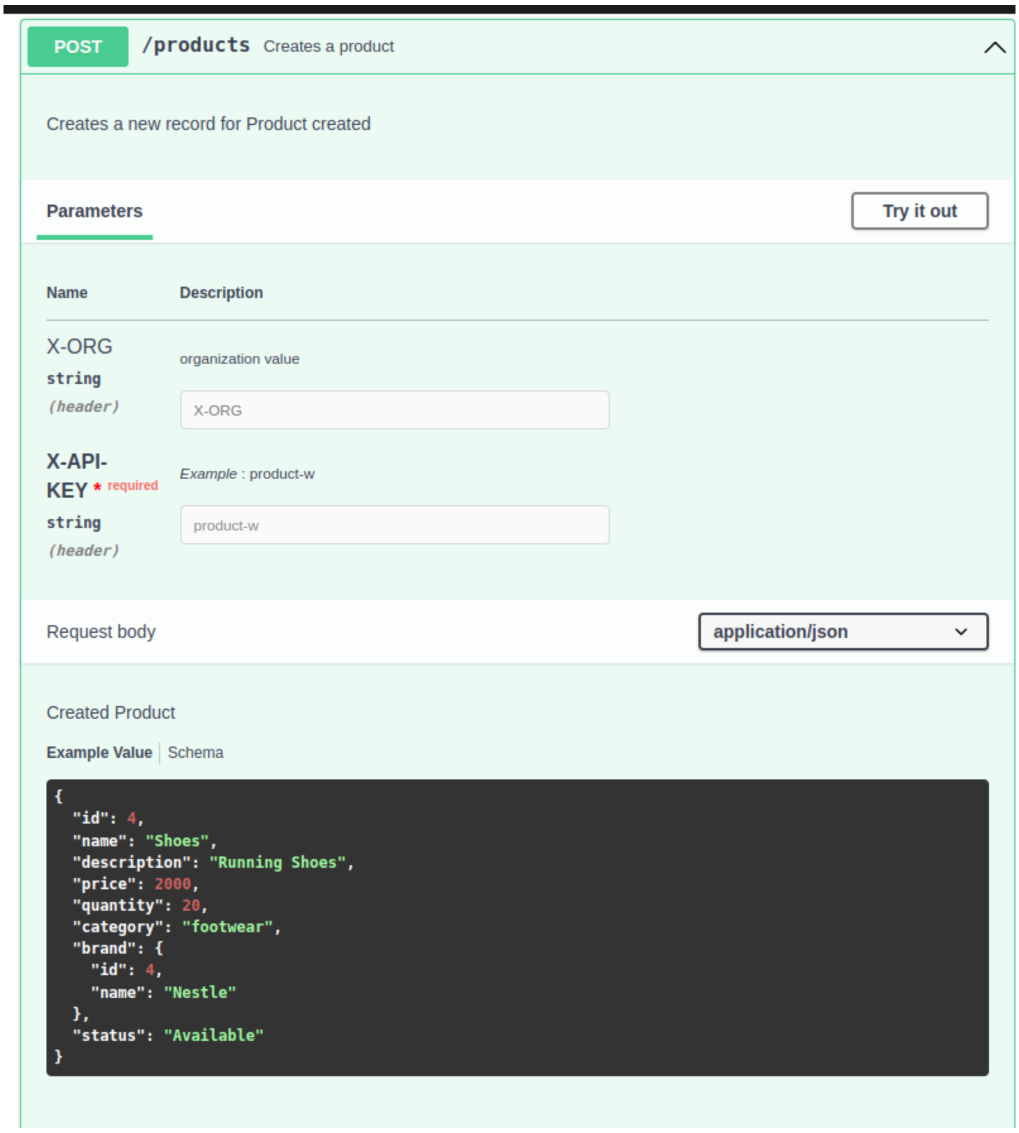
Update an existent product in the store

```
{
  "name": "Shoes",
  "description": "Running Shoes",
  "price": 2000,
  "quantity": 20,
  "category": "footwear",
  "brand": {
    "id": 4,
    "name": "Nestle"
  },
  "status": "Available"
}
```

Responses

Code	Description	Links
200	Successful operation Media type: application/json Controls Accept header. Example Value | Schema	*No links*

```
{
```



/products Creates a product

Creates a new record for Product created

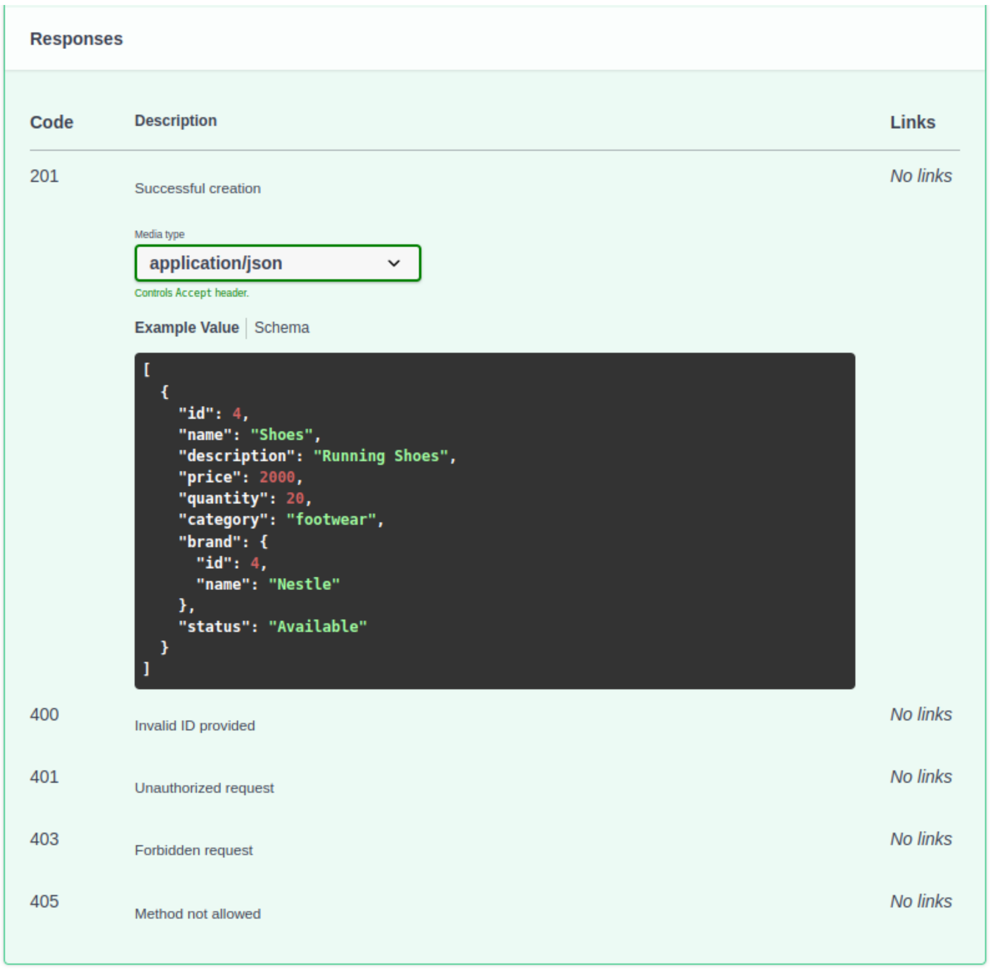
Parameters

Name	Description
X-ORG string *(header)*	organization value X-ORG
X-API-KEY * required string *(header)*	*Example* : product-w product-w

Request body

Created Product

```
{
  "id": 4,
  "name": "Shoes",
  "description": "Running Shoes",
  "price": 2000,
  "quantity": 20,
  "category": "footwear",
  "brand": {
    "id": 4,
    "name": "Nestle"
  },
  "status": "Available"
}
```



Responses

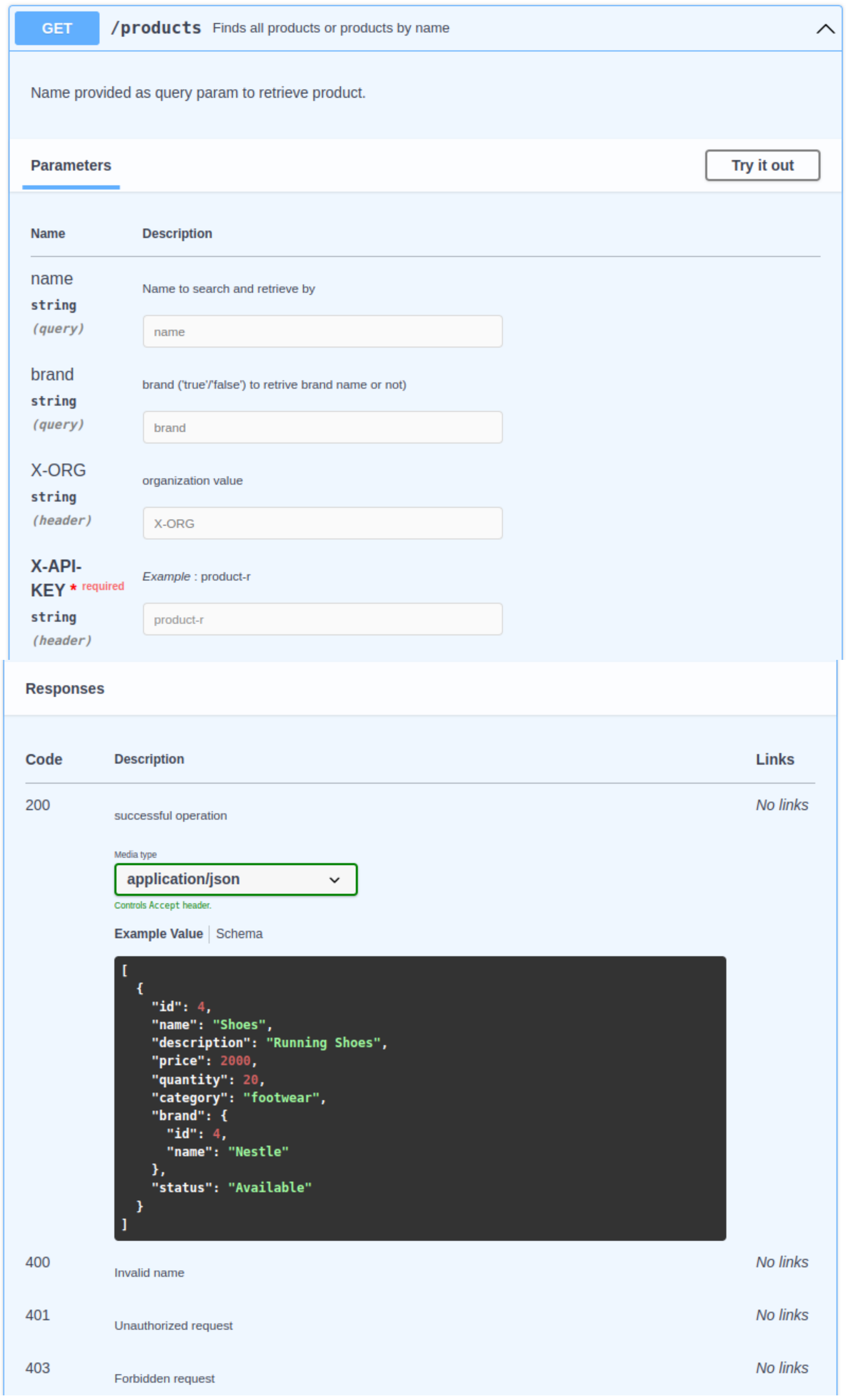
Code	Description	Links
201	Successful creation	*No links*
400	Invalid ID provided	*No links*
401	Unauthorized request	*No links*
403	Forbidden request	*No links*
405	Method not allowed	*No links*

Media type: application/json

Controls Accept header.

Example Value | Schema

```
[
  {
    "id": 4,
    "name": "Shoes",
    "description": "Running Shoes",
    "price": 2000,
    "quantity": 20,
    "category": "footwear",
    "brand": {
      "id": 4,
      "name": "Nestle"
    },
    "status": "Available"
  }
]
```


GET **/products** Finds all products or products by name

Name provided as query param to retrieve product.

Parameters

Name	Description
name **string** *(query)*	Name to search and retrieve by name
brand **string** *(query)*	brand ('true'/'false') to retrive brand name or not) brand
X-ORG **string** *(header)*	organization value X-ORG
X-API-KEY * required **string** *(header)*	*Example* : product-r product-r

Responses

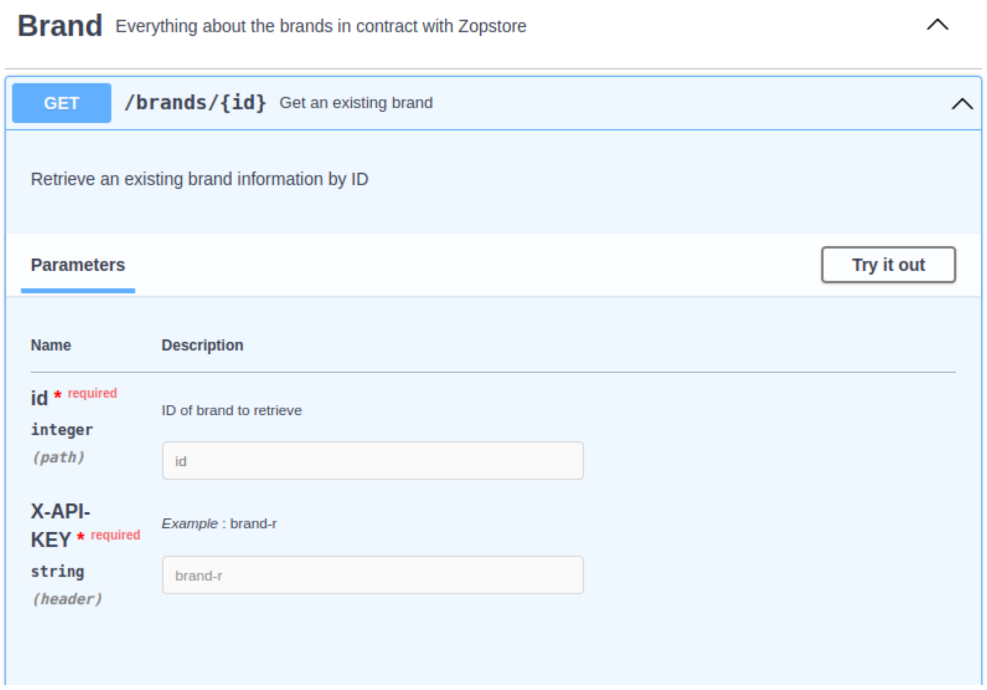
Code	Description	Links
200	successful operation	*No links*
400	Invalid name	*No links*
401	Unauthorized request	*No links*
403	Forbidden request	*No links*

200 — Media type: application/json (Controls Accept header.)

Example Value | Schema

```
[
  {
    "id": 4,
    "name": "Shoes",
    "description": "Running Shoes",
    "price": 2000,
    "quantity": 20,
    "category": "footwear",
    "brand": {
      "id": 4,
      "name": "Nestle"
    },
    "status": "Available"
  }
]
```

BRAND:



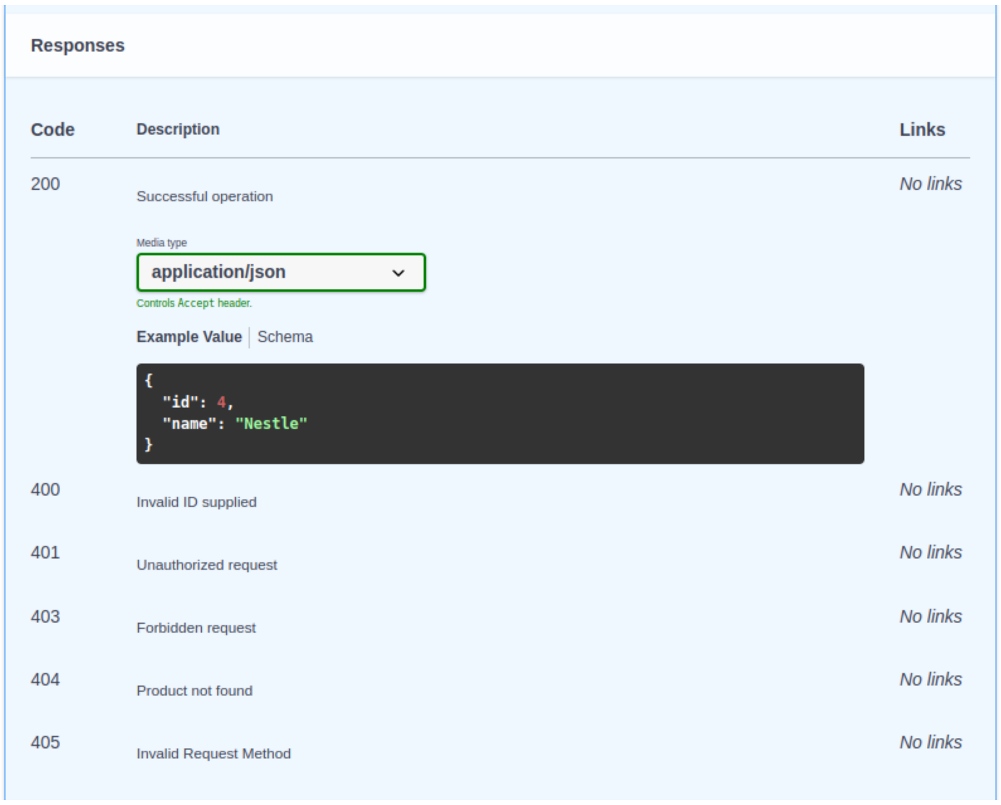
Brand Everything about the brands in contract with Zopstore

GET `/brands/{id}` Get an existing brand

Retrieve an existing brand information by ID

Parameters Try it out

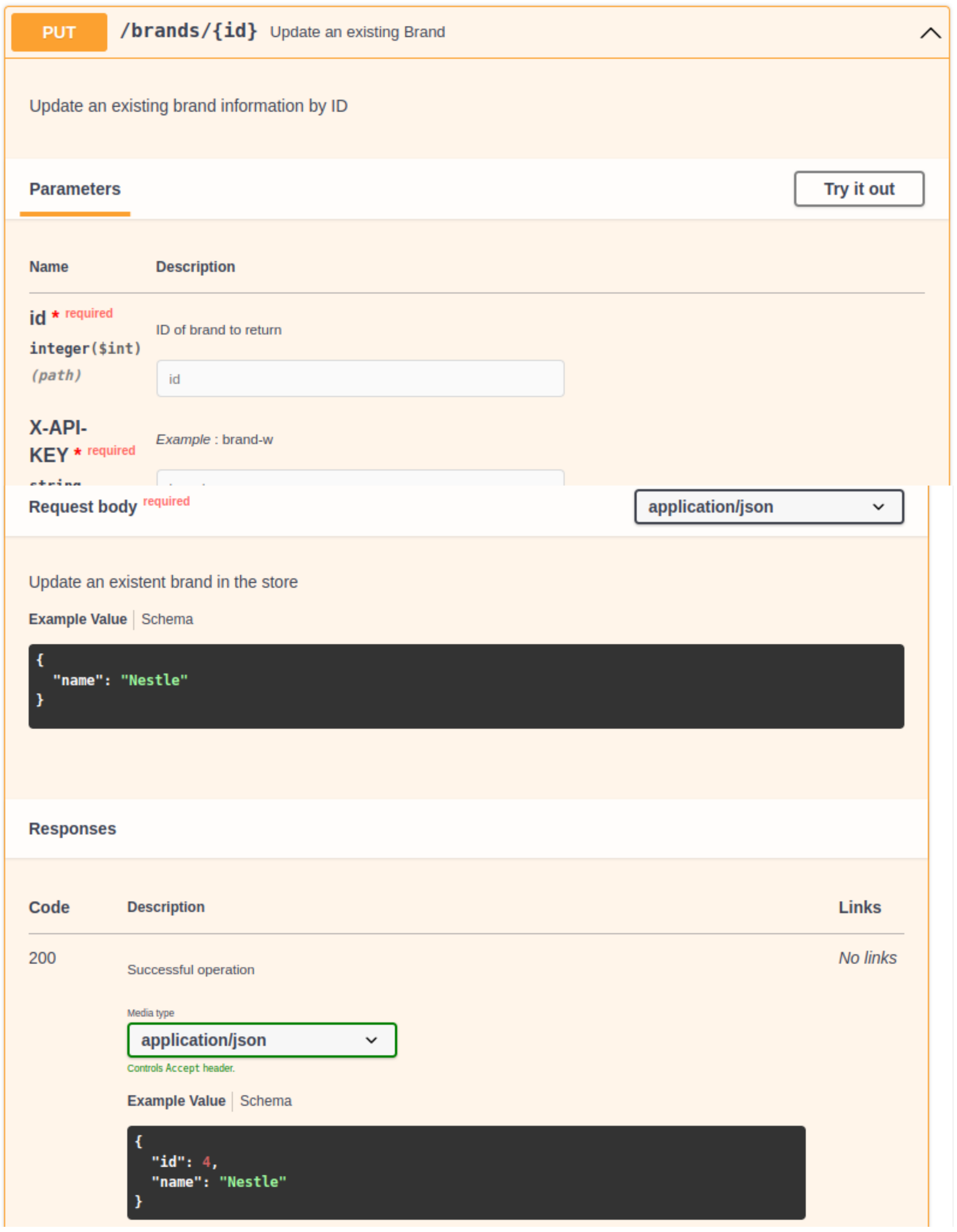
Name	Description
id * required integer *(path)*	ID of brand to retrieve id
X-API-KEY * required string *(header)*	*Example* : brand-r brand-r



Responses

Code	Description	Links
200	Successful operation Media type: application/json Controls Accept header. **Example Value** | Schema	*No links*
400	Invalid ID supplied	*No links*
401	Unauthorized request	*No links*
403	Forbidden request	*No links*
404	Product not found	*No links*
405	Invalid Request Method	*No links*

```
{
  "id": 4,
  "name": "Nestle"
}
```



PUT /brands/{id} Update an existing Brand

Update an existing brand information by ID

Parameters

Try it out

Name	Description
id * required integer($int) (path)	ID of brand to return id
X-API-KEY * required	Example : brand-w

Request body required

application/json

Update an existent brand in the store

Example Value | Schema

```
{
  "name": "Nestle"
}
```

Responses

Code	Description	Links
200	Successful operation	No links

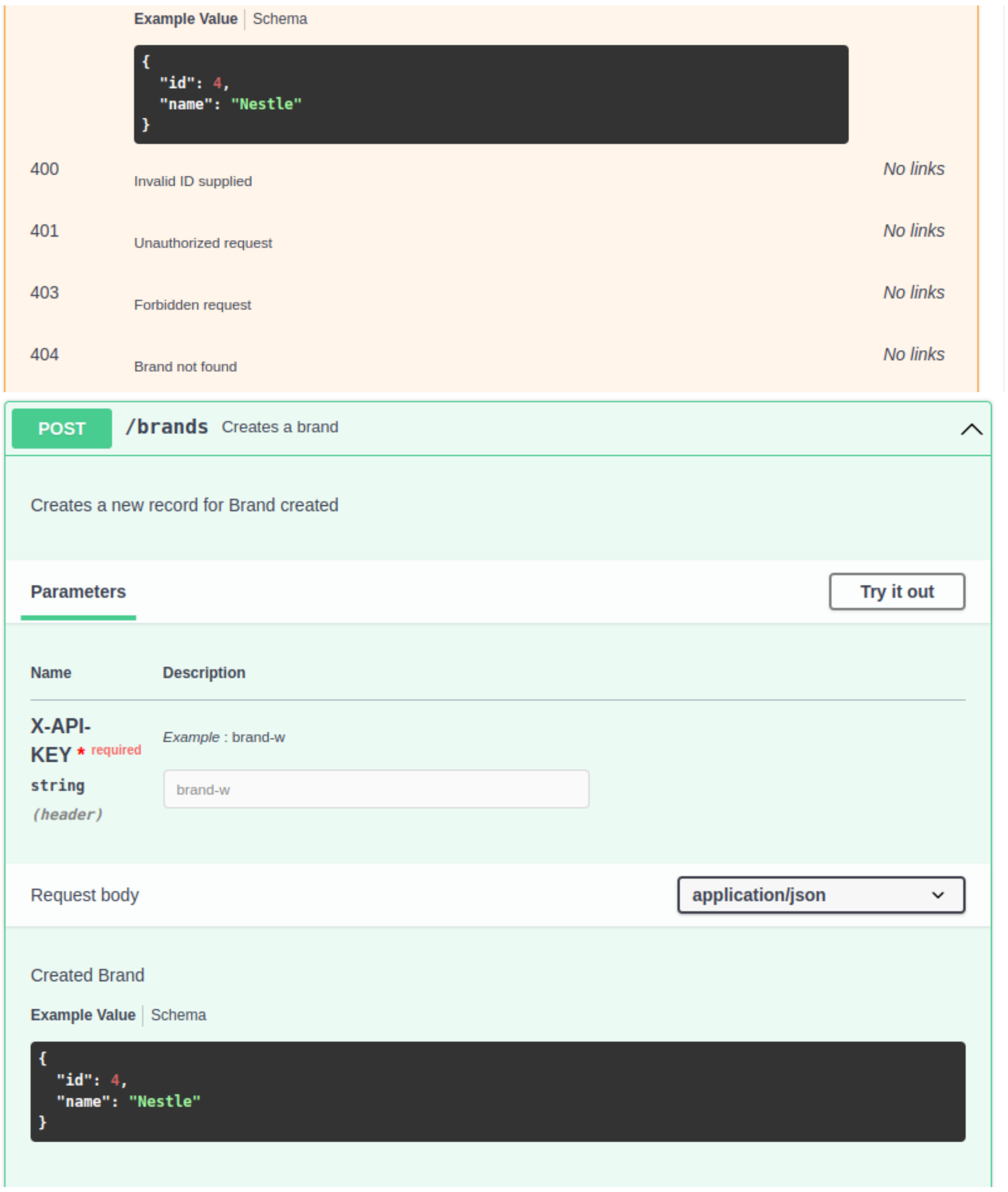
Media type

application/json

Controls Accept header.

Example Value | Schema

```
{
  "id": 4,
  "name": "Nestle"
}
```


Example Value | Schema

```
{
  "id": 4,
  "name": "Nestle"
}
```

400	Invalid ID supplied	*No links*
401	Unauthorized request	*No links*
403	Forbidden request	*No links*
404	Brand not found	*No links*

POST **/brands** Creates a brand

Creates a new record for Brand created

Parameters

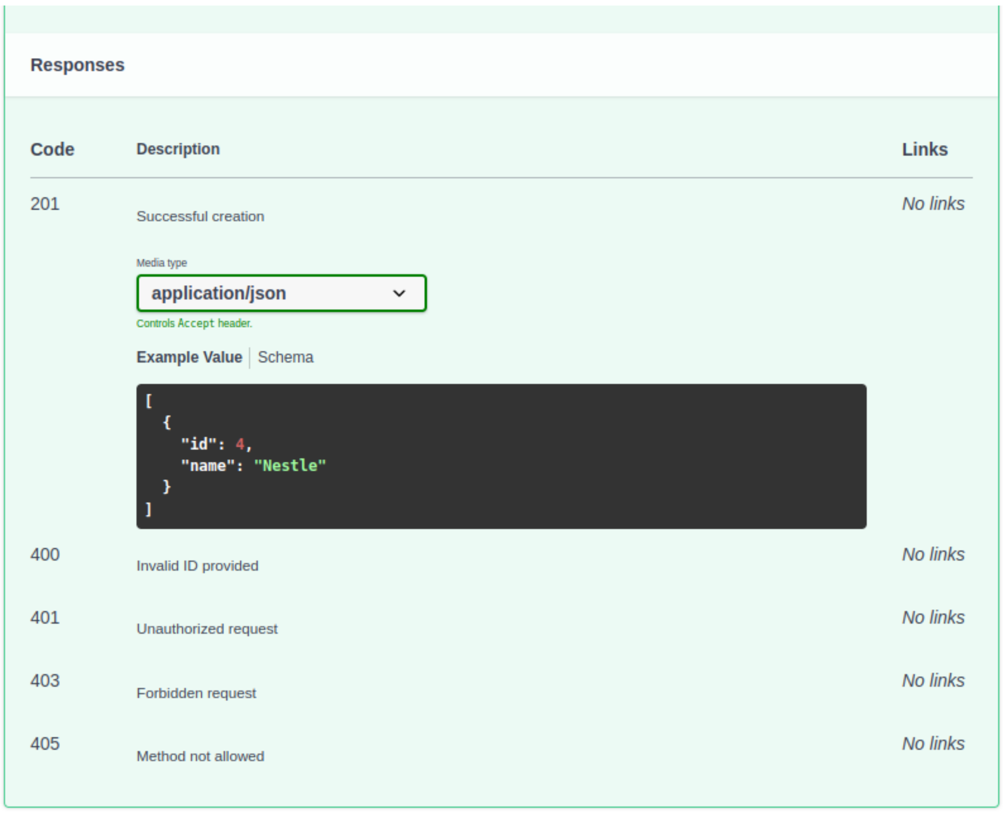
Name	Description
X-API-KEY * required `string` *(header)*	*Example* : brand-w brand-w

Request body: application/json

Created Brand

Example Value | Schema

```
{
  "id": 4,
  "name": "Nestle"
}
```



Responses

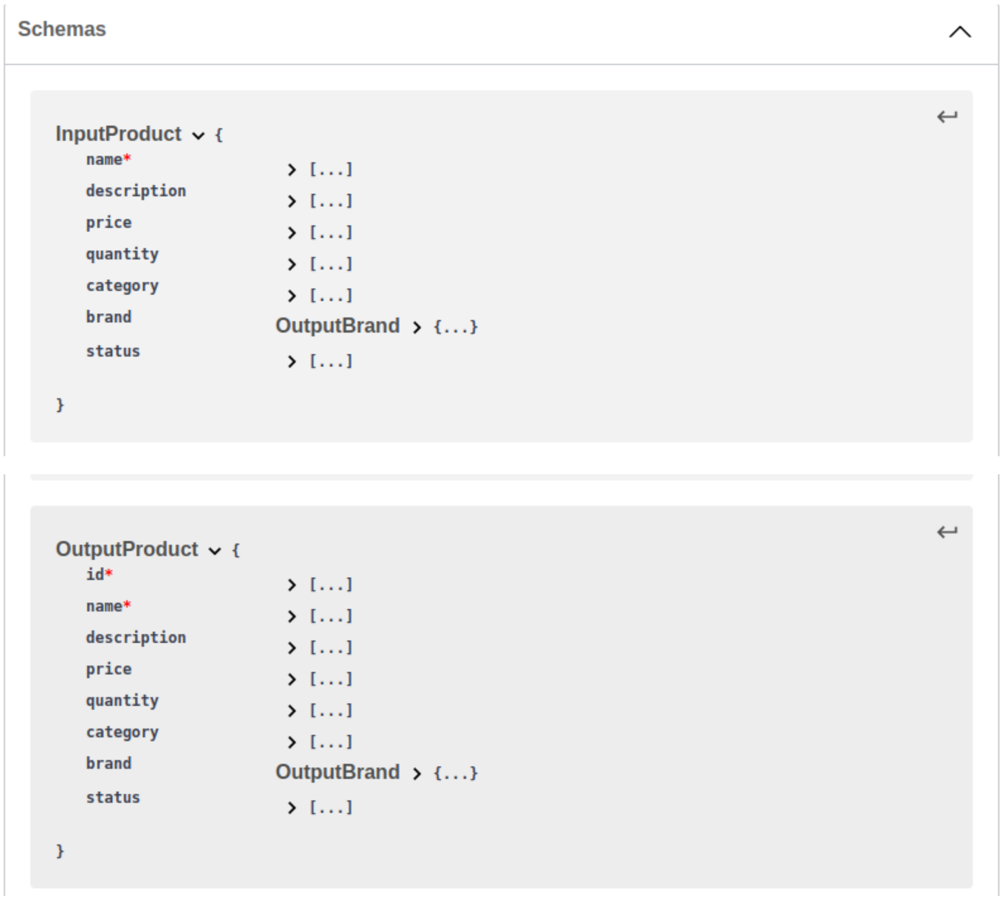
Code	Description	Links
201	Successful creation	*No links*
400	Invalid ID provided	*No links*
401	Unauthorized request	*No links*
403	Forbidden request	*No links*
405	Method not allowed	*No links*

Media type: application/json

Controls Accept header.

Example Value | Schema

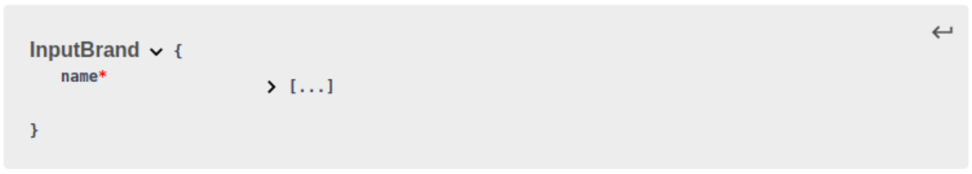
```
[
  {
    "id": 4,
    "name": "Nestle"
  }
]
```



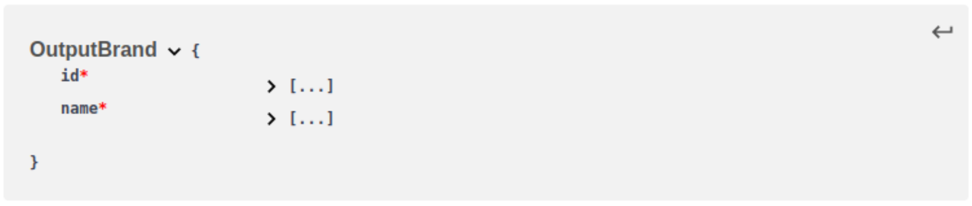
Schemas

```
InputProduct ⌄ {
   name*          › [...]
   description    › [...]
   price          › [...]
   quantity       › [...]
   category       › [...]
   brand          OutputBrand › {...}
   status         › [...]
}
```

```
OutputProduct ⌄ {
   id*            › [...]
   name*          › [...]
   description    › [...]
   price          › [...]
   quantity       › [...]
   category       › [...]
   brand          OutputBrand › {...}
   status         › [...]
}
```



```
InputBrand ⌄ {
    name*                         › [...]
}
```



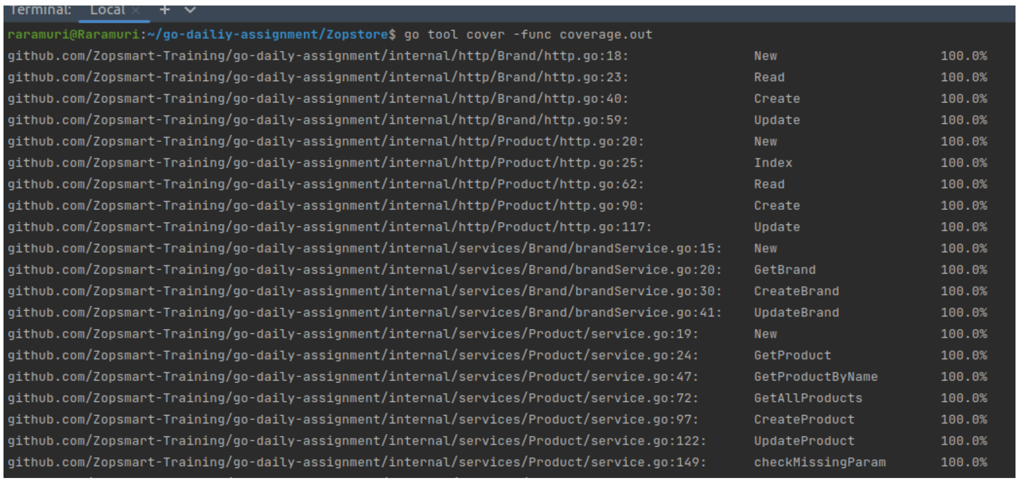
```
OutputBrand ⌄ {
    id*                           › [...]
    name*                         › [...]
}
```

CHAPTER 5: CONCLUSION

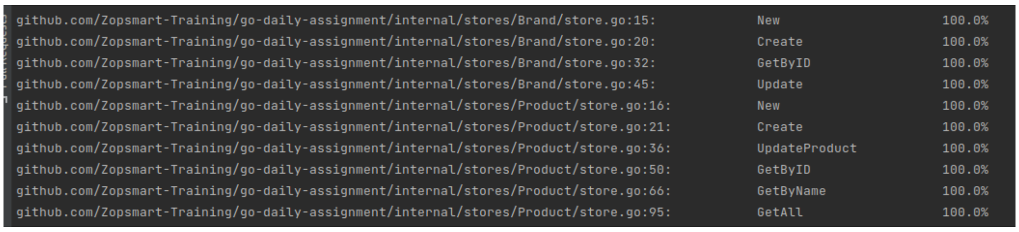
5.1 Performance Analysis

1. Unit Test Coverage

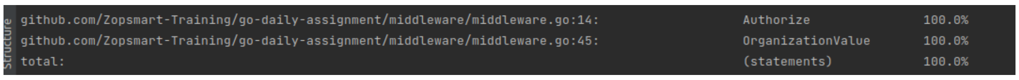
Performed unit test coverage and found all 32 tests ran successfully i.e PASS with a total coverage of 100%.



```
raramuri@Raramuri:~/go-dailiy-assignment/Zopstore$ go tool cover -func coverage.out
github.com/Zopsmart-Training/go-daily-assignment/internal/http/Brand/http.go:18:                 New                     100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/http/Brand/http.go:23:                 Read                    100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/http/Brand/http.go:40:                 Create                  100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/http/Brand/http.go:59:                 Update                  100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/http/Product/http.go:20:               New                     100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/http/Product/http.go:25:               Index                   100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/http/Product/http.go:62:               Read                    100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/http/Product/http.go:90:               Create                  100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/http/Product/http.go:117:              Update                  100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/services/Brand/brandService.go:15:     New                     100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/services/Brand/brandService.go:20:     GetBrand                100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/services/Brand/brandService.go:30:     CreateBrand             100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/services/Brand/brandService.go:41:     UpdateBrand             100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/services/Product/service.go:19:        New                     100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/services/Product/service.go:24:        GetProduct              100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/services/Product/service.go:47:        GetProductByName        100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/services/Product/service.go:72:        GetAllProducts          100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/services/Product/service.go:97:        CreateProduct           100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/services/Product/service.go:122:       UpdateProduct           100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/services/Product/service.go:149:       checkMissingParam       100.0%
```



```
github.com/Zopsmart-Training/go-daily-assignment/internal/stores/Brand/store.go:15:              New                     100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/stores/Brand/store.go:20:              Create                  100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/stores/Brand/store.go:32:              GetByID                 100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/stores/Brand/store.go:45:              Update                  100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/stores/Product/store.go:16:            New                     100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/stores/Product/store.go:21:            Create                  100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/stores/Product/store.go:36:            UpdateProduct           100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/stores/Product/store.go:50:            GetByID                 100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/stores/Product/store.go:66:            GetByName               100.0%
github.com/Zopsmart-Training/go-daily-assignment/internal/stores/Product/store.go:95:            GetAll                  100.0%
```



```
github.com/Zopsmart-Training/go-daily-assignment/middleware/middleware.go:14:          Authorize               100.0%
github.com/Zopsmart-Training/go-daily-assignment/middleware/middleware.go:45:          OrganizationValue       100.0%
total:                                                                                  (statements)            100.0%
```

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.

5.2 Results Achieved

The primary objective of the training was to grasp and implement the fundamentals of GoLang, MySQL, and Unit Testing while building a web application that performs essential CRUD operations. The application was designed using a three-layered architecture and could be tested using Postman.

5.3 Applications Contributions

GoLang has been utilized in various open source and real-world applications. Below are some notable applications:

1. Docker, which is a suite of tools used to deploy Linux containers and the Kubernetes container management system.
2. Swagger
3. Postman

5.4 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.

5.5 Future Work / Scope

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

REFERENCES

[1] *Documentation Go*. Available at: https://go.dev/doc/

[2] Golang (no date) *Golang/Mock: Gomock is a mocking framework for the go programming language.*, *GitHub*. Available at: https://github.com/golang/mock

[3] Data-Dog *Data-dog/GO-sqlmock: SQL mock driver for Golang to test database interactions*, *GitHub*. Available at: https://github.com/DATA-DOG/go-sqlmock

[4] *MySQL documentation MySQL*. Available at: https://dev.mysql.com/doc/

[5] *Linux.org*. Available at: https://www.linux.org/

[6] *Docker docs: How to build, share, and run applications* (2023) *Docker Documentation*. Available at: https://docs.docker.com/

[7] Postman (2023) *Overview*, *Postman Learning Center*. Available at: https://learning.postman.com/docs/introduction/overview/