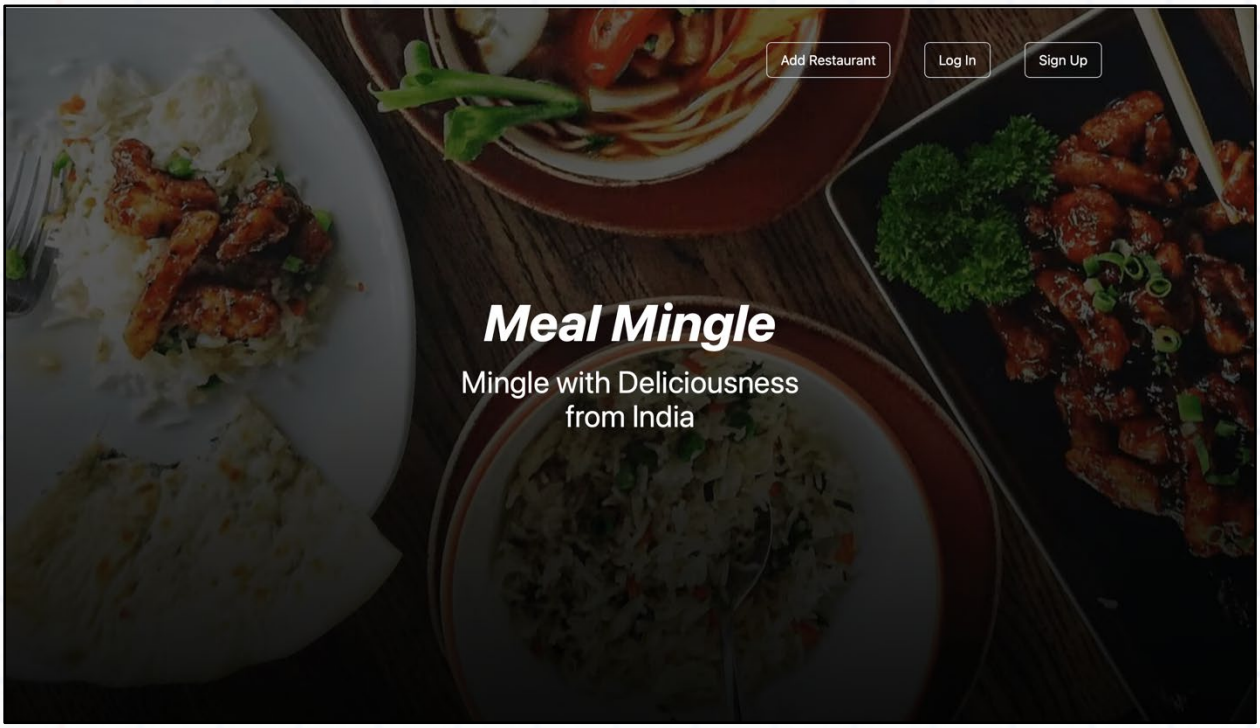
Build an E-marketplace web application



Contents

Introduction to Golang

1. (Go)Golang:

- Developed by Google: Go, often referred to as Golang, was created by Google engineers Robert Griesemer, Rob Pike, and Ken Thompson in 2007.
- Open Source: Released as an open-source language in 2009.
- Statically Typed: Ensures type safety and early detection of type errors.
- Compiled Language: Compiles to machine code, resulting in fast execution.
- Garbage Collected: Automatic memory management to prevent memory leaks and reduce manual intervention.
- Concurrency Support: Built-in concurrency model using goroutines and channels, making it easier to write concurrent programs.
- Minimalist Syntax: Clean and easy-to-read syntax designed to be simple and efficient.
- Standard Library: Rich standard library offering a wide range of utilities and functionalities.
- Cross-Platform: Can be compiled to run on various operating systems like Windows, macOS, and Linux.
- Tooling: Strong ecosystem with tools for testing, formatting, and dependency management.

2. Why Use Go (Golang):

- Performance: High performance due to its compiled nature, close to C/C++ speeds.
- Concurrency: Excellent concurrency support with lightweight goroutines, making it suitable for modern, multicore processors.
- Scalability: Designed to scale efficiently for both small and large applications.
- Simplicity: Simple and clean syntax reduces development time and enhances readability.
- Strong Standard Library: Comprehensive standard library reduces the need for external dependencies.
- Fast Compilation: Fast compilation times enhance the development workflow.
- Developer Productivity: Ease of use and a rich set of tools improve developer productivity.
- Robustness: Statically typed and compiled language helps catch errors early in the development cycle.
- Maintainability: Code simplicity and readability make it easier to maintain and understand.
- Community and Ecosystem: Growing community and ecosystem with numerous libraries and frameworks.

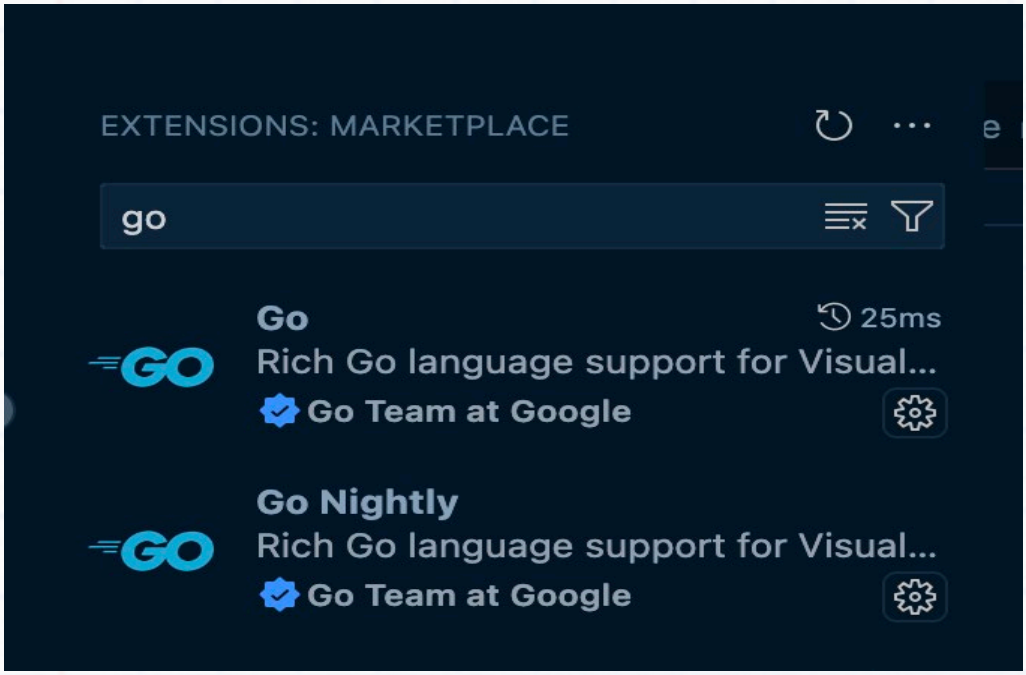
- Microservices: Ideal for developing microservices due to its efficiency and simplicity in handling concurrent tasks.
- Cloud Native: Well-suited for cloud-native development with native support for modern architectures and containerization.

Fundamental of Golang

1. **Setup** Guide Visual Studio Code.
    - Go to the Visual Studio Code website (Click Here to download).
    - Click on the "Download" button for your operating system (Windows, macOS, or Linux).
    - Follow the installation instructions provided for your operating system.
    - Launch Visual Studio Code:
    - Once installed, launch VS Code from your applications or programs menu.

2. **Installing** Visual Studio Code (go) Extensions.

- Open Visual Studio Code.
- Click on the Extensions Icon.
- Type `go` in the spacebar.



3. **Installing** go

Make use to install go version 1.22.2, this version we will use in throught the project.

- Window Setup Guide Reference → Link
- Linux Setup Guide Reference → Link
- MacOS Setup Guide Reference → Link
- Verify the go version, by running the command below.

```
go version
```

4. **Writing** first code in go. [External Reference Link]

- Clone the golang-fundamental repository from the GitHub. (Link)

- For the Reference this is the final source code.

Final Source Code Link >> golang-fundamentals

- Open golang-fundamental in Visual Studio Code.
- Go to the main.go file.

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

- Go the visual studio code integrated terminal and run the below command.

```
go run main.go
```

PS: - In Go, for a file to be runnable as an executable program, it must be part of the main package and must contain a main function. This main function serves as the entry point of the program.

Package Declaration:

The file must declare package main at the top.
This indicates that the file is part of the main package, which is necessary for creating an executable.

Main Function:

The file must contain a main function defined as func main().
This function is the starting point of the program, and the Go runtime will call it when the program starts.

5. **Modules** in go. [Reference Link]

- **Purpose**: Go modules were introduced to address the challenges of dependency management in Go projects. They ensure reproducible builds by specifying exact versions of dependencies and managing updates in a controlled manner.
- **Initialization**: To start using modules, developers create a go.mod file at the root of their project using go mod init [module-path]. This file includes the module's path and specifies the Go version compatibility.
- **Dependency Management**: Modules handle dependency resolution and versioning automatically. Dependencies are listed in the go.mod file along with the specific versions required, ensuring that builds are consistent across different environments.
- **Versioning**: Go modules follow Semantic Versioning (SemVer), allowing developers to specify compatible versions of dependencies. The go.mod file specifies these constraints, and tools like go get manage updates within those constraints.
- **Commands:**
  - **go mod init [module-path]**: Initializes a new module with the specified path.
  - **go get [module-path]**: Adds a new dependency to the project and updates the go.mod file.

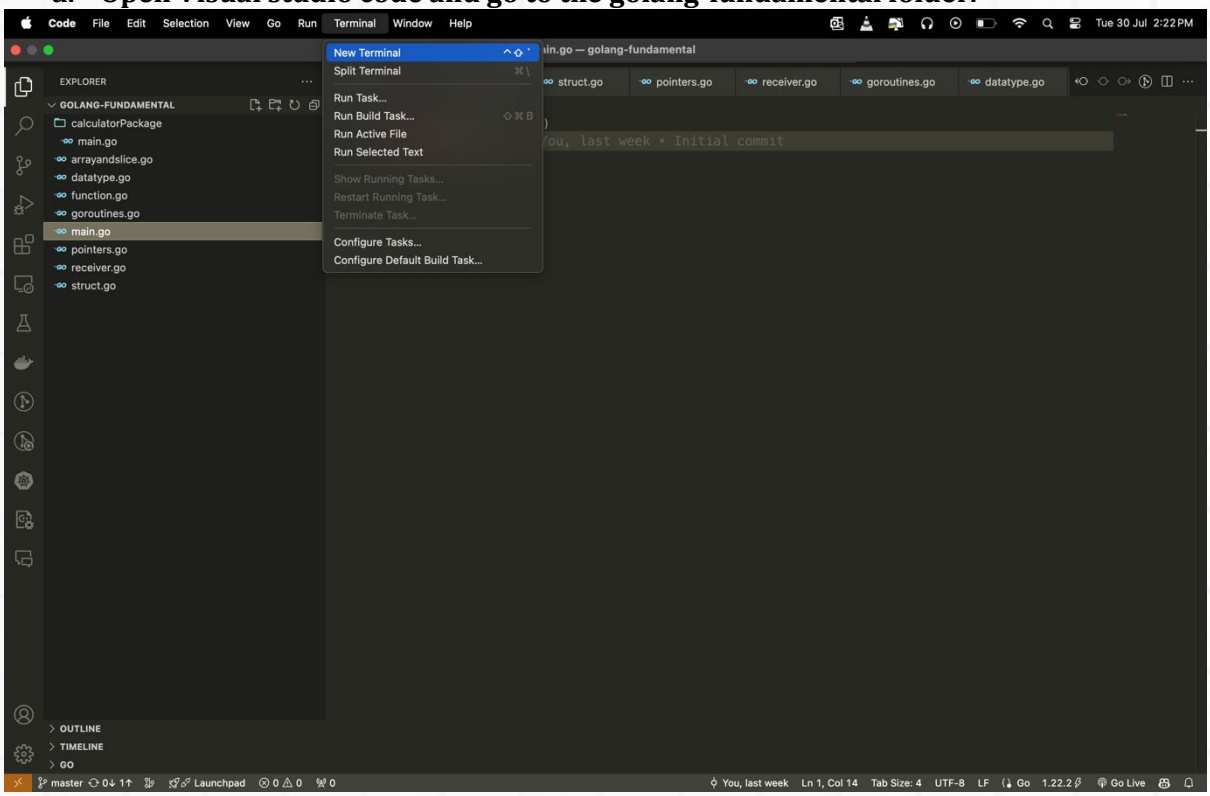
- **go mod tidy**: Removes unused dependencies from go.mod, ensuring only necessary modules are listed.

- **File**: The go.mod file serves as the central manifest for the module, listing dependencies and their versions. It also records replacements and excluded versions.

Q:- How to create golang-fundamental project as a go module ?

1. Process of creating go module.

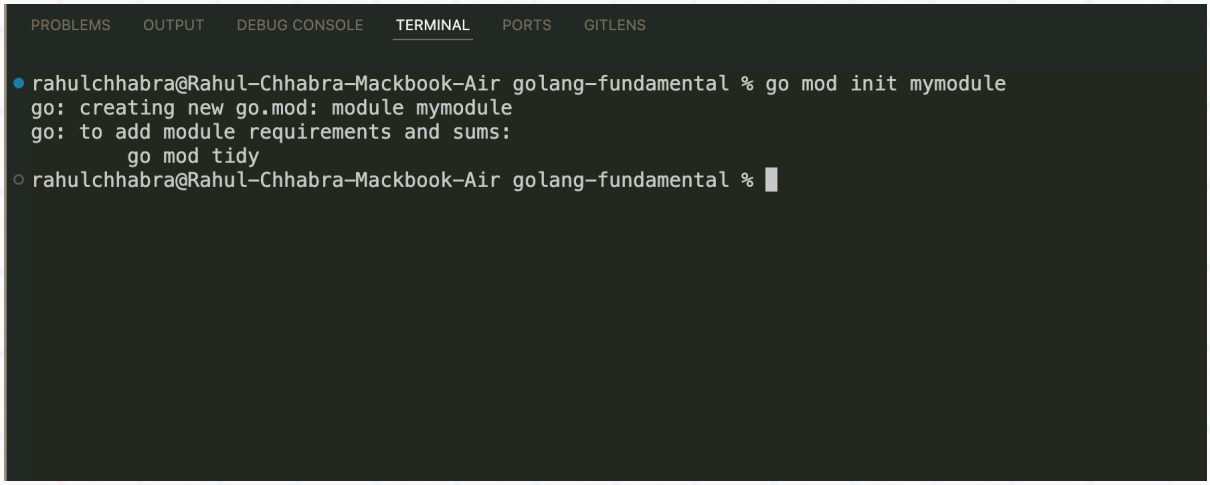
a. Open Visual studio code and go to the golang-fundamental folder.



b. Open Visual studio code integrated terminal.

c. Run the below-command: -

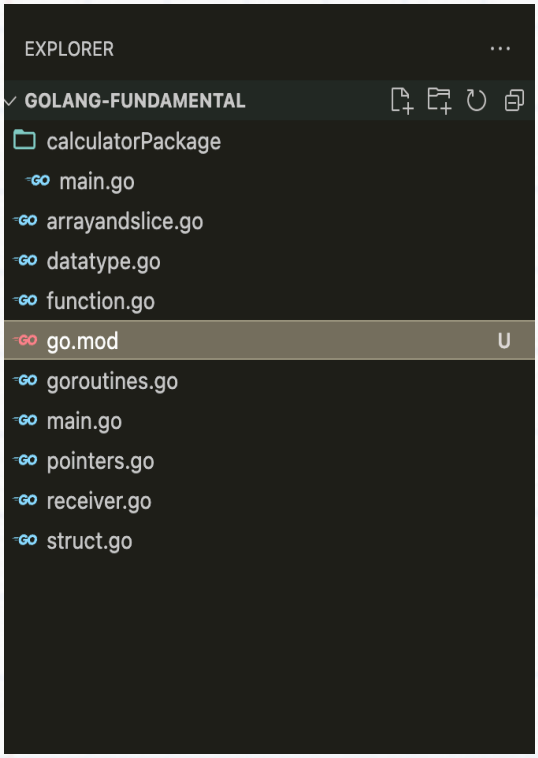
```
go mod init mymodule.
```

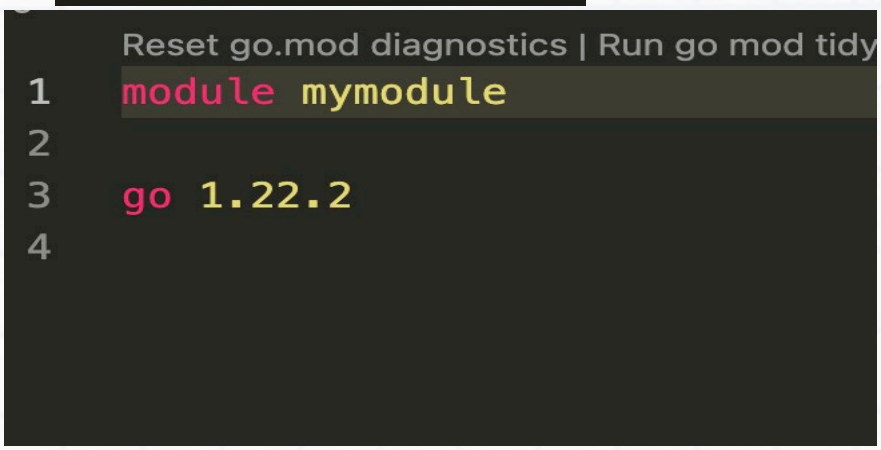


d. O/P:

```
go: creating new go.mod: module mymodule
go: to add module requirements and sums:
go mod tidy
```

2. If the module is successfully created then a new *go.mod* file will be added.





6. **Datatypes** in go. [External Reference Link]

1. Numeric Type:
    - **int**: Signed integer type. Its size can vary based on the platform (32 or 64 bits).
    - **uint**: Unsigned integer type. Its size also varies (32 or 64 bits).
    - byte: Alias for uint8, represents ASCII characters.

- **rune**: Alias for int32, represents a Unicode code point.
- int8, int16, int32, int64: Signed integers of specific sizes.
- **uint8**, uint16, uint32, uint64: Unsigned integers of specific sizes.
- **float32, float64**: Floating-point types for single and double precision, respectively.
- complex64, complex128: Complex number types with float32 and float64 real and imaginary parts.

2. Boolean Type:
   - **bool**: Represents true or false values.
3. String Type:
   - **string**: Represents a sequence of characters. Strings are immutable in Go.

Please note that the default values are initialized when variables are declared but not explicitly initialized. Default variable types in Go are not reference types, so they are not nullable. Therefore, variables have zero values, which vary depending on the variable type, as you may know. In Go, for all numeric types, the default value is **0**, for boolean it's **false**, and for strings, it's an empty string ("").

datatype.go file

```go
package main

import "fmt"
// global declaration of variables
var globalVar1 int = 10
var globalVar2 int = 20
var globalVar3 int = globalVar1 + globalVar2

// global declaration of variables with short hand
var (
    t1       int = 35
    t2       int = 35
    t3, t4, t6    = 23, 33, 43
)
```

```
// we can't use this way(num := 10) of declaring variables outside the function body in golang

func Datatypes() {

    // Ways to declare variables
    var a int = 10
    var b int = 20
    var c int = a + b

    fmt.Println("Sum of a and b is: ", c)
    // Short hand declaration
    d := 10
    e := 20
    f := d + e
    fmt.Println("Sum of d and e is: ", f)

    // Multiple variable declaration
    var g, h, i int = 10, 20, 30
    fmt.Println("Sum of g, h and i is: ", g+h+i)

    // Multiple variable declaration with short hand
    j, k, l := 10, 20, 30
    fmt.Println("Sum of j, k and l is: ", j+k+l)

    // printing global variables
    fmt.Println("Sum of globalVar1 and globalVar2 is: ", globalVar3);


    // string datatype
    var str string = "Hello, World!"
    fmt.Println(str)

    // short hand declaration of string
```

```
    str1 := "Hello, World!"
    fmt.Println(str1)


    // boolean datatype
    var bool1 bool = true
    fmt.Println(bool1)

    // short hand declaration of boolean
    bool2 := true
    fmt.Println(bool2)

    // float datatype
    var n float32 = 10.5
    fmt.Println(n)

    // short hand declaration of float
    o := 20.5
    fmt.Println(o)

    fmt.Println(t1 + t2 + t3 + t4 + t6)
}
```

How to run this Datatypes function in datatype.go file ??
Solution: -

1. In Go, for a file to be runnable as an executable program, it must be part of the main package and must contain a main function. This main function serves as the entry point of the program.

2. Package Declaration:

3. The file must declare package ***main*** at the top.

4. This indicates that the file is part of the ***main*** package, which is necessary for creating an executable.

5. Main Function:

6. The file must contain a ***main*** function defined as ***func main()***.

7. This function is the starting point of the program, and the Go runtime will call it when the program starts.

8. . ***main.go*** file can be used as a runnable file, we can call the Datatypes function from from main function.

PS: Idea of having one main.go file :-

1. In Go, the main function serves as the entry point for the program. It's where the execution of the program begins. Therefore, it's common to have a single main.go file in a Go project that contains the main function.

2. Simplicity: Having a single main.go file makes it clear where the program starts. This is especially helpful for new developers or when you come back to the project after a while.

3. Organization: You can use the main.go file to set up and coordinate the various parts of your program. This might include setting up logging, parsing command-line arguments, configuring settings, starting servers, and so on.

4. Separation of Concerns: By keeping the main function in its own file, you can keep your other Go files focused on their own tasks. This makes your code easier to understand and maintain.

5. Build Tool Compatibility: The Go build tool expects a single main package with a main function as the entry point for building executable programs. Having a single main.go file aligns with this expectation.

main.go file (calling Datatypes function)

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
   // calling Datatype function from datatype.go file.
    Datatypes();
}
```

Run the command

```
go run
```

O/P: -

```
Hello, World!
Sum of a and b is:  30
Sum of d and e is:  30
Sum of g, h and i is:  60
Sum of j, k and l is:  60
Sum of globalVar1 and globalVar2 is:  30
Hello, World!
Hello, World!
true
true
10.5
20.5
169
```

Note: - For every file, we will use main.go file as a entry point first.
Or
main.go file will act as entry point file of our project.

7. **Functions** in go. [External Reference Link]

- Functions are fundamental building blocks in Go programs. They allow you to organize your code into reusable blocks that perform specific tasks. This promotes code modularity, readability, and maintainability.

- **Here's a breakdown of functions in Golang:**

    - **Creating a Function**: You use the **func** keyword to define a function in Go. The general syntax is as follows:

```
func funtion_name (parameters) return_type {
    // function body
}
```

    - **func**: This keyword indicates you're defining a function.

    - **function_name**: This is a name you choose for your function. It should be descriptive and reflect the function's purpose.

    - **parameter_list (optional):** This is a comma-separated list of parameters the function accepts. Each parameter has a name and a data type specifying the kind of value it can receive.

- **return_type (optional)**: This specifies the data type of the value the function returns. If a function doesn't return a value, you can use void.

- **Function Body**: The function body contains the statements that define the function's logic. These statements are enclosed in curly braces {}. The function body executes when the function is called.

- **Calling a Function**: To execute a function, you use its name followed by parentheses (). If the function takes parameters, you provide the actual values (arguments) within the parentheses when you call it.

- **Defer keyword:**

    - **Purpose:** The defer keyword is used to schedule a function call to be run after the function that contains the defer statement has finished executing.

    - **Execution Order:** Deferred functions are executed in last-in, first-out order. The most recently deferred function call is executed first.

function.go file

```
package main

import "fmt"

// function with return type
func greetMessage(name string) string {
    return "Hello, " + name + "!"
}
// function without return type
func printMessage(message string) {
    fmt.Println(message)
}
// funtions with defer keyword
func deferFunction1() {
    fmt.Println("Defer Function 1")
}
func deferFunction2() {
    fmt.Println("Defer Function 2")
}

func Caller() {
    result := greetMessage("World")
    fmt.Println("Result: ", result)
    // calling function without return type

    printMessage("Hello, World!")
    // Immediately Invoked Function Expression (IIFE)
    // This function will be called immediately after its declaration and definition
    // and it will not be called again in the program execution

func() {
```

```
    fmt.Println("IIFE")
}()


// defer keyword
// defer keyword is used to delay the execution of a function until
// the surrounding function returns
// defer functions are executed in Last In First Out (LIFO) order
defer deferFunction1()
defer deferFunction2()
}
```

main.go file

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
    // Datatypes();
    Caller();
}
```

Run the Command: -

```
go run .
```

O/P: -

```
Hello, World!
Result:  Hello, World!
Hello, World!
IIFE
Defer Function 2
Defer Function 1
```

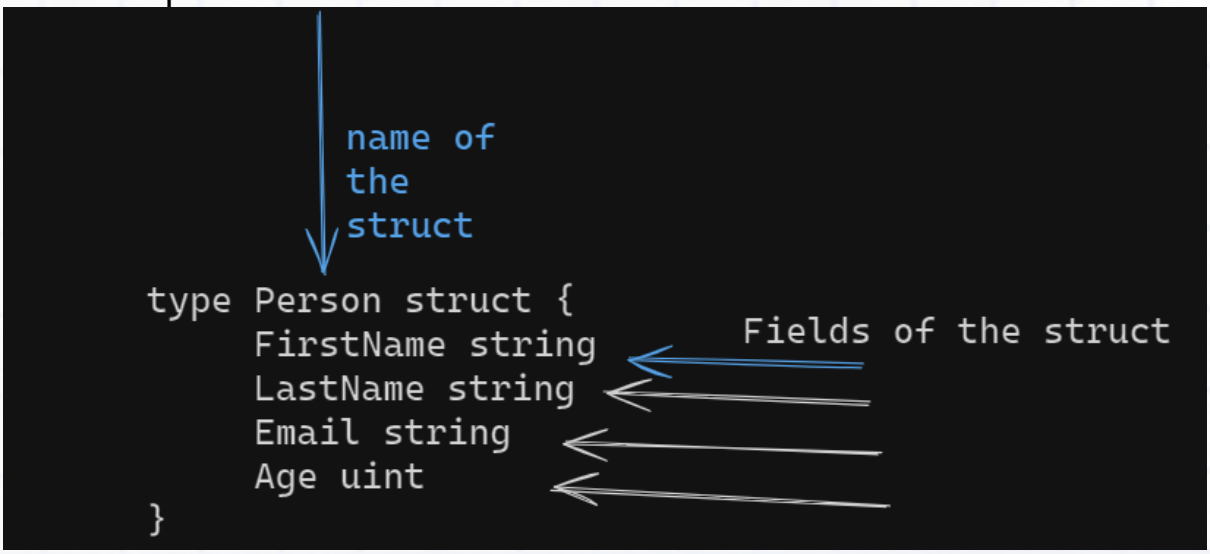
8. **Struct** in go [External Reference Link]

In the Go programming language, a struct is a composite data type used to define a collection of fields grouped together under a single name. Each field within a struct has a name and a type, and they are accessed using dot notation. Structs in Go provide a way to create custom data types that can represent complex data structures.

Template of the struct: -

```
type <nameofthestruct> struct {

  <fieldname>  <fieldtype>
  <fieldname>  <fieldtype>

}
```

For Example: -



Struct.go file

```
package main;
import "fmt"

// Structs are user-defined data types that are used to store a collection of data fields.
// Person is a struct that has four fields: FirstName, LastName, Email, and Age.
// The type of each field is specified after the field name.
// The fields of a struct can be accessed using the dot operator.
type Person struct {
    FirstName string
    LastName  string
    Email     string
    Age       uint
}

// NewPerson is a function that creates a new Person struct and returns it.
// The function takes four parameters: firstName, lastName, email, and age.
// The function returns a Person struct and an error.
func NewPerson(firstName, lastName, email string, age uint) (Person , error) {
```

```
    return Person{
        FirstName: firstName,
        LastName:  lastName,
        Email:     email,
        Age:       age,
    }, nil;
}
func Structs() {
    person1 := Person{FirstName: "John", LastName: "Doe", Email: "johndoe@mail.com", Age:
43}
    person2 := Person{FirstName: "Alex", LastName: "Carey", Email: "carrybusinessmail.com",
Age: 56}

    fmt.Printf("FirstName of person1 is %s \n", person1.FirstName)
    fmt.Printf("LastName of the person2 is %s\n", person2.LastName)

    // Structs can also be initialized without specifying the field names.
    person3 := Person{"Jane", "Doe", "jane@gmail.com", 30}
    fmt.Printf("Email of person3 is %s\n", person3.Email)

    // Can Change the value of a field
    person3.Email = "janedoe@gmail.com";
    fmt.Printf("Updated Email of person3 is %s\n", person3.Email)

    // Create a new Person struct using the NewPerson function.
    person4, _ := NewPerson("Alice", "Smith", "alice@mial.com", 25)
    fmt.Printf("FirstName of person4 is %s\n", person4.FirstName)
}
```

PS: - 1. Use main.go file to call Structs function
2. Use the same command used above in the terminal to run the code.

9. **Pointers** in go. [External Reference Link]

Pointers are one of the most powerful and fundamental features of the Go programming language. They allow you to manipulate memory directly, create dynamic data structures, and improve program efficiency. However, pointers can also be tricky to use, and incorrect usage can lead to memory leaks and other errors. In this article, we will explore the basics of pointers in Go, including syntax, memory management, pass by reference, pointer arithmetic, arrays, and functions. We will also cover advanced concepts in pointers and best practices for using pointers in Go. By the end of this article, you should have a solid understanding of how to use pointers in Go and how to avoid common pitfalls.

pointers.go file

```go
package main

import "fmt"
// syntax for declaring pointers
// var pointer *datatype
// var ptr *int
// var ptr1 *string
// In functions we can pass the address of a variable as an argument to a function
// and the function can change the value of the variable at that address
// syntax for passing pointers to a function
// func functionName(pointer *datatype)
func swap(x *int, y *int) {
    temp := *x
    *x = *y
    *y = temp
}

func double(x *int) *int {
    result := *x * 2
    return &result
}
// pointer with struct
```

```go
type person struct {
    name string
    age  int
}

// function that takes a pointer to a struct as an argument
func changeName(p *person) {
    p.name = "Alice"
}
func Pointers() {
    x := 1
    y := 2
    fmt.Println(x, y) // output: 1 2
    swap(&x, &y)
    fmt.Println(x, y) // output: 2 1
    ptr := double(&x)
    fmt.Println(*ptr) // output: 4

    // pointer with struct
    p := person{name: "Bob", age: 30}
    fmt.Println(p) // output: {Bob 30}
    changeName(&p)
    fmt.Println(p) // output: {Alice 30}
}
```

10. **Receiver** functions in go [External Reference Link]

In Go, a receiver function (or method) is a function that has a special receiver argument, allowing it to be called on instances of a specific type. The receiver can be a value or a pointer to a value of the type. This is similar to methods in object-oriented programming languages where functions are associated with objects.

Defining a Method with a Receiver

To define a method with a receiver, you specify the receiver type in parentheses between the func keyword and the method name. Here are examples with both value and pointer receivers:

Value Receiver

- A value receiver operates on a copy of the value, meaning any modifications made inside the method do not affect the original value.

Pointer Receiver

- A pointer receiver operates on the actual value, so modifications made inside the method affect the original value.

receiver.go file

```
package main
import "fmt"

type Rectangle struct {
    Width  float64
    Height float64
}

//Area method with a value receiver of type Rectangle struct
// The receiver is a value receiver because the method does not modify the struct
func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}
// Scale method with a pointer receiver
// The receiver is a pointer receiver because the method modifies the struct
func (r *Rectangle) Scale(factor float64) {
    r.Width *= factor
```

```
    r.Height *= factor
}

func Receiver() {
    rect := Rectangle{Width: 10, Height: 5}
    fmt.Println("Area:", rect.Area())
    rect.Scale(2)
    fmt.Println("Area:", rect.Area())
    fmt.Println("Width:", rect.Width)
    fmt.Println("Height:", rect.Height)
}
```

11. **Arrays and Slice** in go

arrayandslice.go file

```
package main

import "fmt"

func ArraySlices() {
    // Declaring an array of integers with a fixed length of 5
    var a [5]int
    // Initializing an array with values
    b := [5]int{1, 2, 3, 4, 5}
    // Partial initialization (remaining elements are zero-valued)
    c := [5]int{1, 2, 3}
    // Accessing and modifying elements
    a[0] = 10
    fmt.Println(a[0])
    fmt.Println(b)
    fmt.Println(c)
```

```
    // slices
    // Declaring a slice
    var d []int
    // Initializing a slice using make
    e := make([]int, 5)
    // Initializing a slice using a slice literal
    f := []int{1, 2, 3, 4, 5}
    // Accessing and modifying elements
    d = append(d, 10)

    fmt.Println(d[0])
    fmt.Println(e)
    fmt.Println(f)
}
```

12. **Packages** in go [External Reference Link]

- A package is made up of Go files that live in the same directory and have the same package statement at the beginning. You can include additional functionality from packages to make your programs more sophisticated. Some packages are available through the Go Standard Library and are therefore installed with your Go installation. Others can be installed with Go's go get command. You can also build your own Go packages by creating Go files in the same directory across which you want to share code by using the necessary package statement.

- As in our previous codes, we were importing the "fmt" package to print Hello World into the console, so the package fmt comes with golang installation.

- Way of importing package import <modulename/relative-path from the root directory to the package>

calculatorPackage/main.go file.

```
package calculatorpackage

func Add(a int, b int) int {
    return a + b;
}

func Subtract(a int, b int) int {
    return a - b;
}

func Multiply(a int, b int) int {
    return a * b;
}

func divide(a int, b int) int {
    if b == 0 {
        return -1;
    } else {
        return a / b;
    }
}
```

main.go file

```
package main
import "fmt"
import "mymodule/calculatorpackage"

func main() {
    fmt.Println("Hello, World!")
    fmt.Println("Addition: ", calculatorpackage.Add(10, 5))
```

```
    fmt.Println("Subtraction: ", calculatorpackage.Subtract(10, 5))
    fmt.Println("Multiplication: ", calculatorpackage.Multiply(10, 5))
}
```

PS: - **if you try to import the divide function you will get the error**
undefined: calculator.divide

And the reason is the visibility of functions outside the package.

1. Exported Identifiers:
    - Start with an uppercase letter.
    - Accessible from other packages.
2. Unexported Identifiers:
    - Start with a lowercase letter.
    - Not accessible from other packages.
    - But can be accessible in the same package
    - Examples: divide function in calculatorPackage package.
3. Packages:
    - Go programs are organized into packages.
    - Each package can contain multiple files.
    - Visibility rules apply across package boundaries.
4. Fields and Methods in Structs:
    - Fields or methods in a struct follow the same capitalization rules (first letter uppercase).
5. Exported functions in calculatorPackage: add,subtract,multiply.
6. Unexported function in calculatorPackage : divide

13. Go Routines and Channels(Go Concurrency) [External Reference Link]

1. Go Routine

- A goroutine is an independent function that executes simultaneously in some separate lightweight threads managed by Go. GoLang provides it to support concurrency in Go.

- Goroutines let you do things at the same time (concurrently). Think of it like having two people cooking in the kitchen at the same time, each doing a different task.
- Here, go cook("Pizza") starts cooking pizza in a new go routine, and immediately after, the main go routine starts cooking pasta. They happen at the same time.
- So In this example, we have two goroutines one is the main routines which is the main thread obviously and the other one is we just created using the go keyword.
- **Here's an example of how a goroutine looks like:**

```
package main


import (
    "fmt"
    "time"
)


func cook(name string) {
    fmt.Println("Cooking Routine started:", name)
    // Sleep pauses the current goroutine for at least the duration d.
    // A negative or zero duration causes Sleep to return immediately.
    time.Sleep(2 * time.Second /** duration d**/)
    fmt.Println("Cooking Routine finished:", name)
}



func GoRoutines() {
    fmt.Printf("Starting of Main Routine\n")
    fmt.Println("Cooking started: Pasta")
    go cook("Pizza") // Start cooking pizza in a new go routine
    fmt.Printf("Completing the Main Routine\n")
}
```

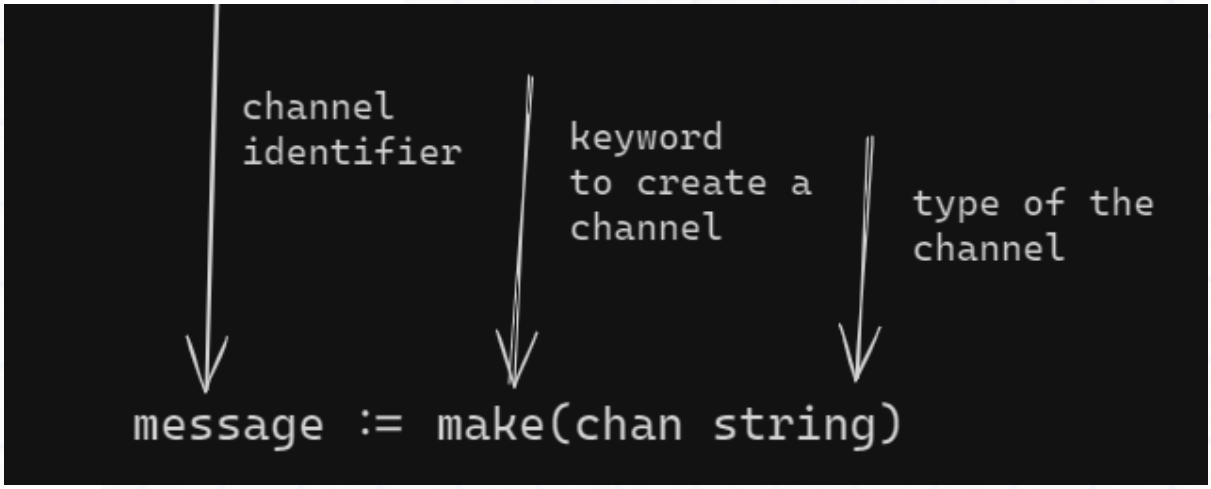
- Here, we are not able to see the output of the cooking function, Do you have any guesses??
- The Reason is Main thread is executing faster as compared to the cooking thread, which takes 2 seconds to complete.
- So there is no communication b/w routines (main and cooking).
- In order to make communication b/w routines, channels are introduced.

2. Channel

- A way to send data from one task (go routine) to another. Like passing notes between these chefs to coordinate.

- Channels are like passing notes between people. If one person writes a note (sends data) and gives it to another person (another go routine), the second person can read this note (receives data).

→ Let's create channel

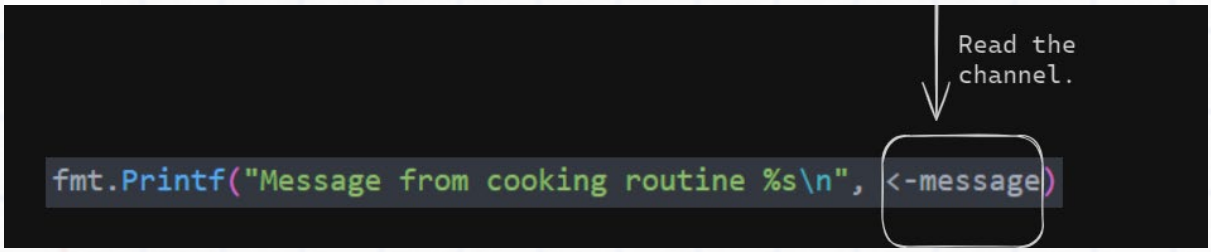


- **Here's an example of how a channel with go routine looks like:**

```
package main


import (
    "fmt"
```

```
    "time"
)


func cook(name string,message chan string) {
    fmt.Println("Cooking Routine started:", name)
    // Sleep pauses the current goroutine for at least the duration d.
    // A negative or zero duration causes Sleep to return immediately.
    time.Sleep(2 * time.Second /** duration d**/)
    fmt.Println("Cooking Routine finished:", name)
    message <- "Pizza is ready"
}


func GoRoutines() {
    fmt.Printf("Starting of Main Routine\n")
    fmt.Println("Cooking started: Pasta")
    message := make(chan string)
    go cook("Pizza",message) // Start cooking pizza in a new go routine
    fmt.Println("Message from the cook:", <-message)
    fmt.Printf("Completing the Main Routine\n")
}
```

→ Read the channel



→ Write the channel

```
message <- "Hii I'm from cooking routine"
```

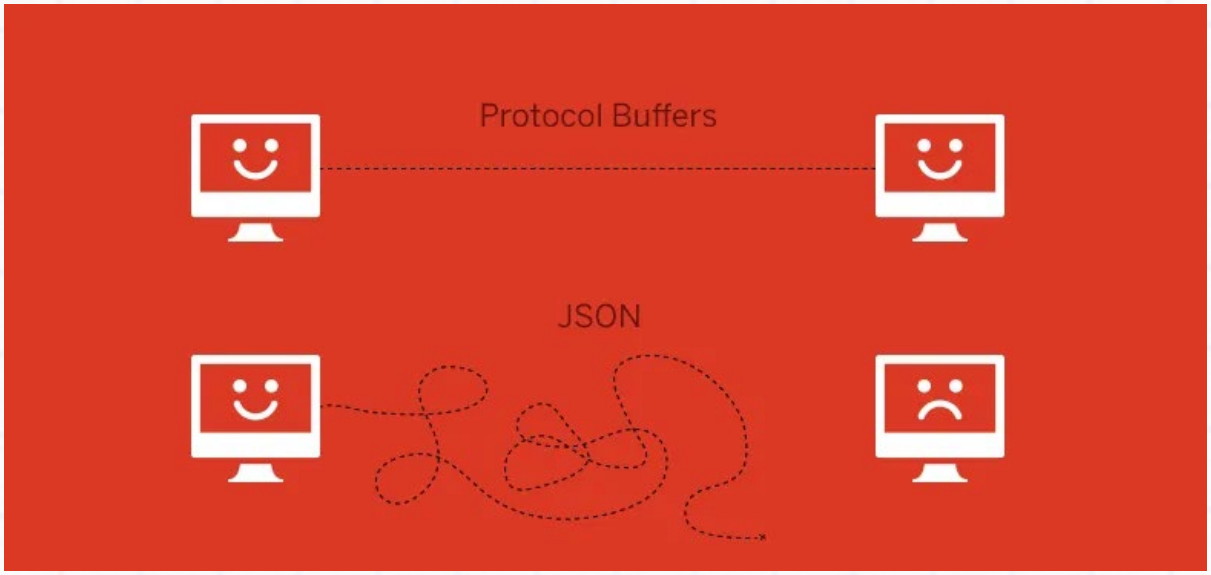
14. Looping and Conditional in go

[External Reference Link]

gRPC(gRPC Remote Procedure Calls)

1. Introduction to gRPC

→ gRPC is a high-performance RPC framework/technology built by Google. It uses Google's own "Protocol Buffers", which is an opensource message format for data serialization, as the default method of communication between the client and the server. Similar to REST APIs mostly use JSON as the message format, gRPC uses Protocol Buffer (ProtoBuf for short) format as the message format and the IDL (interface definition language) to describe the payload parameters and response parameters.
As for the method of communication between the client stub and the server code, gRPC uses HTTP/2 as the default protocol. Since gRPC uses HTTP/2 features it supports 4 types of APIs.



Here are the main types of gRPC:-

- gRPC Unary RPC: Unary RPC is the simplest form of gRPC. In a unary RPC, the client sends a single request to the server and receives a single response in return. This is similar to a traditional synchronous function call. Unary RPC is suitable for scenarios where a single request and response are sufficient.

- gRPC Server Streaming RPC: Server Streaming RPC allows the client to send a single request to the server and receive a stream of responses in return. This is useful when the server needs to send a sequence of data to the client, such as real-time updates or a large set of results.

- gRPC Client Streaming RPC: Client Streaming RPC is the opposite of Server Streaming RPC. In this case, the client sends a stream of requests to the server and receives a single response in return. This is useful when the client has a continuous flow of data to send to the server, and the server processes it and responds when it's done.

- gRPC Bidirectional Streaming RPC: Bidirectional Streaming RPC allows both the client and server to send a stream of messages to each other. This creates a full-duplex communication channel where both sides can send and receive data independently. Bidirectional Streaming RPC is suitable for scenarios requiring interactive and continuous communication.

2. gRPC Server

gRPC (gRPC Remote Procedure Calls) is a high-performance, open-source, universal remote procedure call (RPC) framework initially developed by Google. A gRPC server is a server-side implementation that handles incoming gRPC requests, processes them, and sends back the appropriate responses. Here's a deeper look into what a gRPC server is and how it functions:

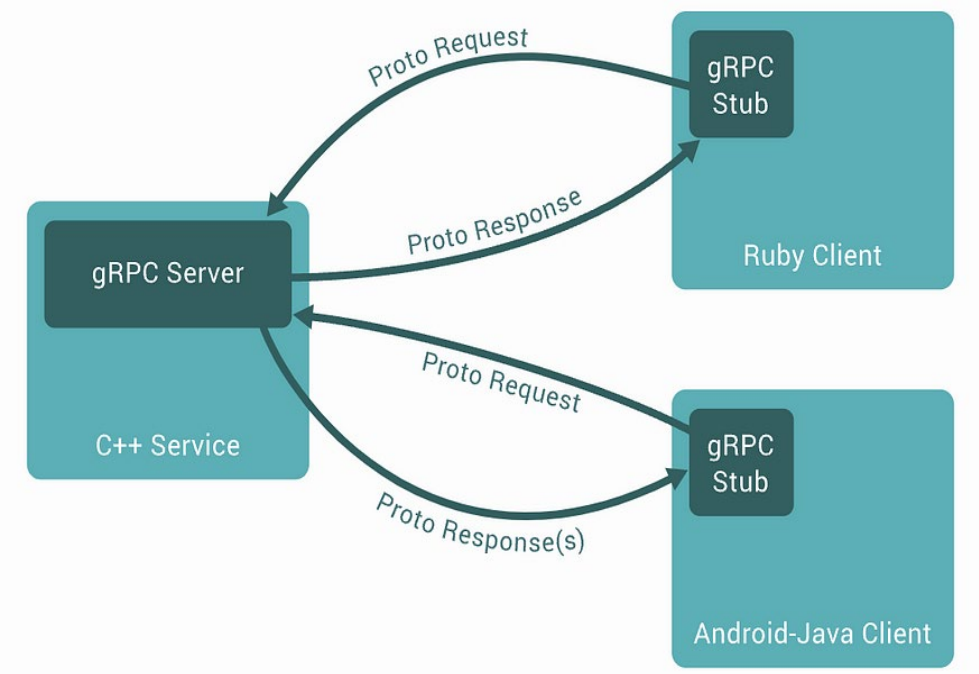
Key Components of a gRPC Server

- **Service Definition:**
  Services in gRPC are defined using Protocol Buffers (protobuf), a language-agnostic binary serialization format. You define your service methods and messages in a .proto file.

- **Server Implementation:**
  After defining the service in a .proto file, you generate server-side code using the protoc compiler. This code provides the necessary boilerplate to implement the service.
  You implement the service by extending the generated base class and overriding its methods to provide the actual business logic.

- **Server Initialization:**
  You create a gRPC server instance and register the implemented service with the server.
  The server is then bound to a specific port and starts listening for incoming RPC calls.

- **Handling Requests:**
  When a request comes in, the server routes it to the appropriate method implementation.
  The method processes the request, performs the necessary operations (such as querying a database or performing computations), and returns a response.



3. Protocol Buffer (Protobuf)

- Protocol Buffers, often referred to as Protobuf, is a method developed by Google for serializing structured data. It's designed to be a fast, efficient, and language-independent mechanism for encoding data to send over the wire or to store for later use. Here's a brief introduction to the key concepts of Protocol Buffers:

- protoc compiler installation: -
  i. Window → Link
  ii. Mac → Link
  iii. Linux → Link

  Make sure to install the latest version of the compiler.

  After installation, you can verify the version of the proto compiler.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\dell> protoc --version
libprotoc 25.3
PS C:\Users\dell>
```

Resources for proto buffer syntax and some examples also:- Readme file

4. Hands on gRPC

- Clone the repository >> Link to the repository.
- For the Reference this is the final source code.

Final Source Code Link >> grpc-introduction

- Open the folder grpc-introduction in visual studio code.
- Install the dependencies.

```
go get -u google.golang.org/grpc

go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@v1.2

go install google.golang.org/protobuf/cmd/protoc-gen-go@v1.28
```

- Compile the **greet.proto** file, by running the command(find this command in Makefile (Makefile reference link)).

```
PROBLEMS 3  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

rahulchhabra@192 grpc-introduction % make proto-generate
protoc -I ./proto \
        --go_out ./proto --go_opt paths=source_relative \
        --go-grpc_out ./proto --go-grpc_opt paths=source_relative \
        ./proto/greet/greet.proto
rahulchhabra@192 grpc-introduction %
```

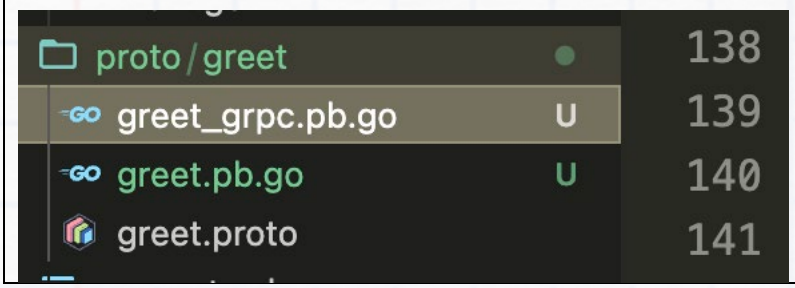
```
proto-generate:
```

```
protoc -I ./proto \
--go_out ./proto --go_opt paths=source_relative \
--go-grpc_out ./proto --go-grpc_opt paths=source_relative \
./proto/greet/greet.proto
```

Let's down break this command: -

- **-I—To specify the import path where imported packages in .proto files are searched**
- **--go_out—To specify where to put generated Go code for messages**
- **--go_opt—To configure options for Go source code generation, such as paths=source_relative, to keep the same folder structure after source code generation**
- **--go-grpc_out—To define the destination folder of gRPC-specific Go source code, such as calling a service function**
- **--go-grpc_opt—To configure options for gRPC-related operations, such as paths=source_relative, to have the same folder structure after source code generation.**

After the Compilation of greet.proto file , two new files will be added to into proto/greet folder.



Inside greet_grpc.pb.go file, you can find the rpc function(without body) that we have added in greet.proto file

>> greet.proto file.

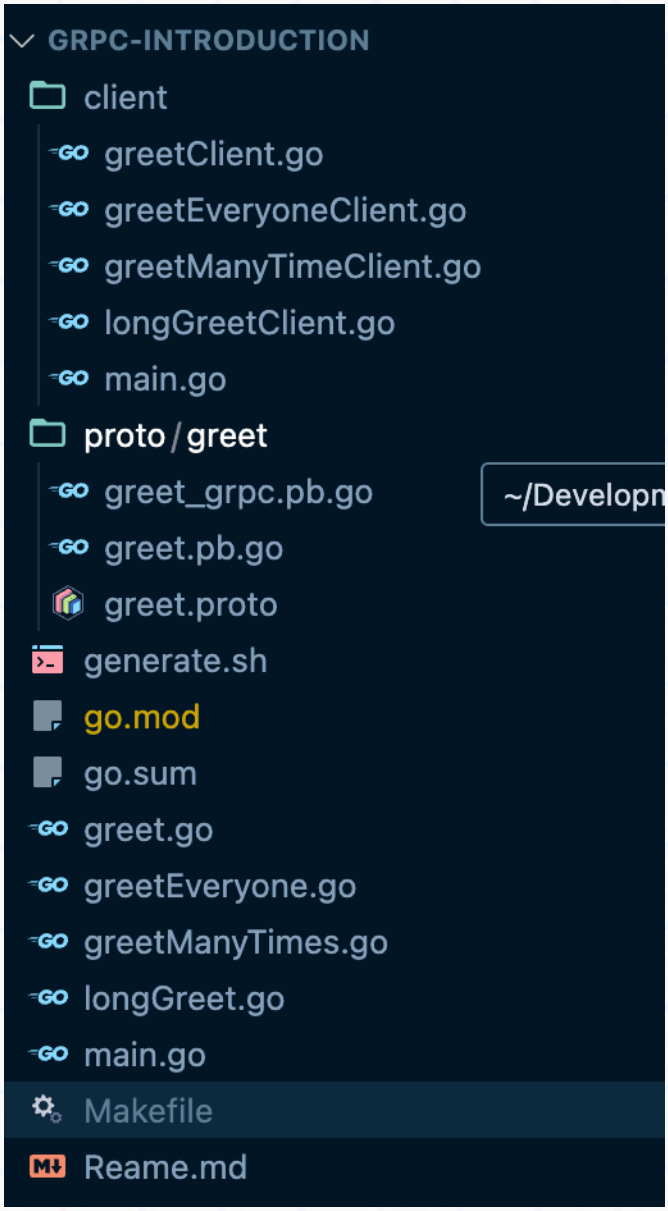
```
45
46  service GreetService {
47      // Unary
48      rpc Greet(GreetRequest) returns (GreetResponse) {};
49
50      // Server Streaming
51      rpc GreetManyTimes(GreetManyTimesRequest) returns (stream GreetManyTimesResponse) {};
52
53      // Client Streaming
54      rpc LongGreet (stream LongGreetRequest) returns (LongGreetResponse) {};
55
56      // BiDi Streaming
57      rpc GreetEveryone (stream GreetEveryoneRequest) returns (stream GreetEveryoneResponse) {};
58        You, 2 months ago • Initial Code
59  }
```

>> greet_grpc.pb.go file.

```
148
149  // GreetServiceServer is the server API for GreetService service.
150  // All implementations must embed UnimplementedGreetServiceServer
151  // for forward compatibility
152  type GreetServiceServer interface {
153      // Unary
154      Greet(context.Context, *GreetRequest) (*GreetResponse, error)
155      // Server Streaming
156      GreetManyTimes(*GreetManyTimesRequest, GreetService_GreetManyTimesServer) error
157      // Client Streaming
158      LongGreet(GreetService_LongGreetServer) error
159      // BiDi Streaming
160      GreetEveryone(GreetService_GreetEveryoneServer) error
161      mustEmbedUnimplementedGreetServiceServer()
162  }
163
```

>> Let’s start implementing all these rpc function one by one.

5. Directory Structure



6. main.go file

```
package main

import (
    "fmt"
    greetpb "grpc-dev/proto/greet"
    "log"
    "net"

    "google.golang.org/grpc"
    "google.golang.org/grpc/reflection"
)

type GreetService struct {
    greetpb.UnimplementedGreetServiceServer
}

func main() {

    fmt.Println("Hello, I'm the server")

    // Create a listener on TCP port 50055
    lis, err := net.Listen("tcp", "0.0.0.0:50055")
    if err != nil {
        log.Fatalf("Failed to listen: %v ", err)
    }

    // Create a gRPC server
    s := grpc.NewServer()
    // Register the greet service
    greetpb.RegisterGreetServiceServer(s, &GreetService{})
```

```
    // Register reflection service on gRPC server.
    reflection.Register(s)

    // Serve the gRPC server
    if err = s.Serve(lis); err != nil {
        log.Fatalf("Failed to serve: %v", err)
    }
}
```

- The main function starts by printing a message to the console.
- It then creates a TCP listener on port 50055. If there's an error (like the port is already in use), it logs the error and exits the program.
- A new gRPC server is created using grpc.NewServer().
- The GreetService is registered to the gRPC server. This service is defined by the greetpb package, which is likely generated from a .proto file. The GreetService struct is an implementation of the UnimplementedGreetServiceServer interface from the greetpb package.
- The reflection service is registered on the gRPC server. This is used for service discovery and can be useful for debugging.
- Finally, the gRPC server is started with s.Serve(lis). If there's an error (like a networking issue), it logs the error and exits the program.
- Run the gRPC server by running the below command into the terminal

```
go run.
```

7. greet.go file (Unary RPC)

```
package main

import (
    "context"
    "fmt"
    greetpb "grpc-dev/proto/greet"
)

func (*GreetService) Greet(ctx context.Context, req *greetpb.GreetRequest)
(*greetpb.GreetResponse, error) {

    fmt.Printf("Greet func was invoked with %v: \n", req)

    // Get the first name from the client request
    firstName := req.GetGreeting().GetFirstName()
    // Get the last name from the client request
    lastName := req.GetGreeting().GetLastName()
    // Create the response
    result := "Hello, " + firstName + " " + lastName
    res := &greetpb.GreetResponse{
        Result: result,
    }
    // Return the response
    return res, nil
}
```

8. greetManyTimes.go file (Server Streaming RPC)

```
package main

import (
    "fmt"
    greetpb "grpc-dev/proto/greet"
    "strconv"
    "time"
)

func (*GreetService) GreetManyTimes(req *greetpb.GreetManyTimesRequest, stream
greetpb.GreetService_GreetManyTimesServer) error {

    fmt.Println("GreetManyTimes function was invoked with a request", req)
    // Get the first name from the client request
    firstName := req.GetGreeting().GetFirstName()
    lastName := req.GetGreeting().GetLastName()

    // Send a response to the client for 10 times
    for i := 0; i < 10; i++ {
        result := "Hello, " + firstName + " " + lastName + " | " + strconv.Itoa(i)
        res := &greetpb.GreetManyTimesResponse{
            Result: result,
        }
        stream.Send(res)
        time.Sleep(1000 * time.Millisecond)
    }
    return nil
}
```

- The GreetManyTimes function is defined on the GreetService struct. It takes two parameters: a req of type *greetpb.GreetManyTimesRequest and a stream of type greetpb.GreetService_GreetManyTimesServer. The stream is an interface that provides methods for reading from and writing to the client stream.
- The function starts by printing a message to the console, indicating that it has been invoked and showing the request it received.
- It then extracts the first name and last name from the request.
- The function enters a loop where it constructs a greeting message for the client, which includes the first name, last name, and the current iteration number. This message is then sent to the client via the stream.Send(res) method.
- After sending each message, the function pauses for 1 second (time.Sleep(1000 * time.Millisecond)).
- This loop continues for 10 iterations, meaning the client will receive 10 messages.
- After sending all the messages, the function returns nil to indicate that it has finished without errors.

8. longGreet.go file (Client Streaming RPC)

```
package main

import (
    "fmt"
    "io"
    "log"
    greetpb "grpc-dev/proto/greet"
)

func (*GreetService) LongGreet(stream greetpb.GreetService_LongGreetServer) error {

    fmt.Printf("LongGreet func was invoked\n")
    result := ""
    // Receive the client stream of requests from the client and respond to each request
```

```
    for {
        req, err := stream.Recv()
        if err == io.EOF {
            log.Printf("Reached end of stream EOF: %v ", err)
            return stream.SendAndClose(&greetpb.LongGreetResponse{
                Result: result,
            })
        }
        if err != nil {
            log.Fatalf("Error receiving from the client stream: %v ", err)
        }
        // Get the first name from the client request
        firstName := req.GetGreeting().GetFirstName()
        result += "Hello " + firstName + "!\n"
        fmt.Printf("Received message: %v\n", firstName)
    }
}
```

- The LongGreet function is defined on the GreetService struct. It takes a stream parameter which is of type greetpb.GreetService_LongGreetServer. This is an interface that provides methods for reading from and writing to the client stream.
- The function starts by printing a message to the console.
- It then enters a loop where it continuously tries to receive messages from the client stream.
- If it receives an io.EOF error, this means the client has finished sending messages. It logs a message, sends a final response to the client with stream.SendAndClose(), and returns from the function.
- If it receives any other error, it logs the error and exits the program.
- If it successfully receives a message, it extracts the first name from the message, appends a greeting to the result string, and prints a message to the console.

- This loop continues until the client stops sending messages or an error occurs.

9. greetEveryone.go file (Bi-directional Streaming RPC)

```
package main

import (
    "fmt"
    greetpb "grpc-dev/proto/greet"
    "io"
    "log"
)

func (*GreetService) GreetEveryone(stream greetpb.GreetService_GreetEveryoneServer) error
{
    fmt.Printf("GreetEveryone func invoked\n")
    // Receive the client stream of requests from the client and respond to each request
    // This is a server streaming function
    for {
        req, err := stream.Recv()
        if err == io.EOF {
            log.Println("Reached EOF")
            return nil
        }
        if err != nil {
            log.Fatalf("Error while reading client stream: %v ", err)
            return err
        }
        firstName := req.GetGreeting().GetFirstName()
        result := "Hello, " + firstName + ". "

        sendErr := stream.Send(&greetpb.GreetEveryoneResponse{
```

```
            Result: result,
        })
        if sendErr != nil {
            log.Fatalf("Error while sending data to client: %v ", err)
            return err
        }
    }
}
```

- The GreetEveryone function is defined on the GreetService struct. It takes a stream parameter which is of type greetpb.GreetService_GreetEveryoneServer. This is an interface that provides methods for reading from and writing to the client stream.
- The function starts by printing a message to the console.
- It then enters a loop where it continuously tries to receive messages from the client stream.
- If it receives an io.EOF error, this means the client has finished sending messages. It logs a message and returns from the function.
- If it receives any other error, it logs the error and returns it, which will cause the function to exit.
- If it successfully receives a message, it extracts the first name from the message and constructs a greeting.
- It then tries to send the greeting back to the client. If there's an error, it logs the error and returns it, which will cause the function to exit.
- This loop continues until the client stops sending messages or an error occurs.

11. client/greetClient.go file

- The Greet function is defined. It takes a greetpb.GreetServiceClient as a parameter, which is used to call the Greet RPC on the server.

- The function starts by printing a message to the console, indicating that a unary RPC has been initiated.
- It then creates a greetpb.GreetRequest with a greetpb.Greeting containing a first name and last name.
- It calls the Greet method on the client, passing in a background context and the request. This sends the request to the server and waits for the response.
- If there's an error calling Greet, it logs the error and exits the function.
- Finally, it logs the response from the server to the console.

12. client/longGreetClient.go file

- The LongGreet function is defined. It takes a greetpb.GreetServiceClient as a parameter, which is used to call the LongGreet RPC on the server.
- The function starts by printing a message to the console, indicating that client streaming has been initiated.
- It then creates a slice of greetpb.LongGreetRequest pointers, each containing a greetpb.Greeting with a first name and last name.
- It calls the LongGreet method on the client, passing in a background context. This returns a client stream for sending requests and receiving the response.
- If there's an error calling LongGreet, it logs the error and exits the function.
- It then enters a loop where it sends each request to the server via the client stream. After sending each request, it pauses for 1 second (time.Sleep(1000 * time.Millisecond)).
- After sending all the requests, it calls CloseAndRecv on the client stream. This closes the stream from the client side and waits for the response from the server.
- If there's an error receiving the response, it logs the error and exits the function.
- Finally, it prints the response from the server to the console.

- In a real-world application, the LongGreet function might send more complex requests or handle the response in a different way.

13. client/greetManyTimesClient.go file

- The GreetManyTimes function is defined. It takes a greetpb.GreetServiceClient as a parameter, which is used to call the GreetManyTimes RPC on the server.
- The function starts by printing a message to the console, indicating that server streaming has started.
- It then creates a greetpb.GreetManyTimesRequest with a greetpb.Greeting containing a first name and last name.
- It calls the GreetManyTimes method on the client, passing in a background context and the request. This sends the request to the server and returns a stream for receiving the responses.
- If there's an error calling GreetManyTimes, it logs the error and exits the function.
- It then enters a loop where it continuously tries to receive messages from the server stream.
- If it receives an io.EOF error, this means the server has finished sending messages. It prints a message and breaks out of the loop.
- If it receives any other error, it logs the error and exits the function.
- If it successfully receives a message, it logs the message to the console.
- This loop continues until the server stops sending messages or an error occurs.

14. client/greetEveryoneClient.go

- The GreetEveryone function is defined. It takes a greetpb.GreetServiceClient as a parameter, which is used to call the GreetEveryone RPC on the server.
- The function starts by printing a message to the console, indicating that bidirectional streaming has been initiated.

- It then creates a slice of greetpb.GreetEveryoneRequest pointers, each containing a greetpb.Greeting with a first name and last name.
- It calls the GreetEveryone method on the client, passing in a background context. This returns a stream for sending requests and receiving responses.
- If there's an error calling GreetEveryone, it logs the error and exits the function.
- It creates a channel waitChannel that will be used to synchronize the sending and receiving goroutines.
- It starts a goroutine for sending requests. In this goroutine, it loops over each request, sends it to the server via the stream, and then sleeps for 1 second. After sending all the requests, it closes the send side of the stream.
- It starts another goroutine for receiving responses. In this goroutine, it continuously tries to receive messages from the stream. If it receives an io.EOF error, this means the server has finished sending messages and it breaks out of the loop. If it receives any other error, it logs the error and breaks out of the loop. If it successfully receives a message, it logs the message to the console. After the loop, it closes the waitChannel.
- It waits for the waitChannel to close. This ensures that the function doesn't return until both the sending and receiving goroutines have finished.

15. client/main.go

- The main function starts by printing a message to the console, indicating that it's a client.
- It then attempts to establish a connection to a gRPC server running on localhost at port 50055 using grpc.Dial. The grpc.WithInsecure option is used because the server isn't using TLS.
- If there's an error connecting to the server, it logs the error and exits the function.
- It defers the closing of the connection until the main function returns. This ensures that the connection is always closed, even if an error occurs.
- It creates a new greetpb.GreetServiceClient using the connection. This client can be used to call the RPCs defined in the GreetService service.

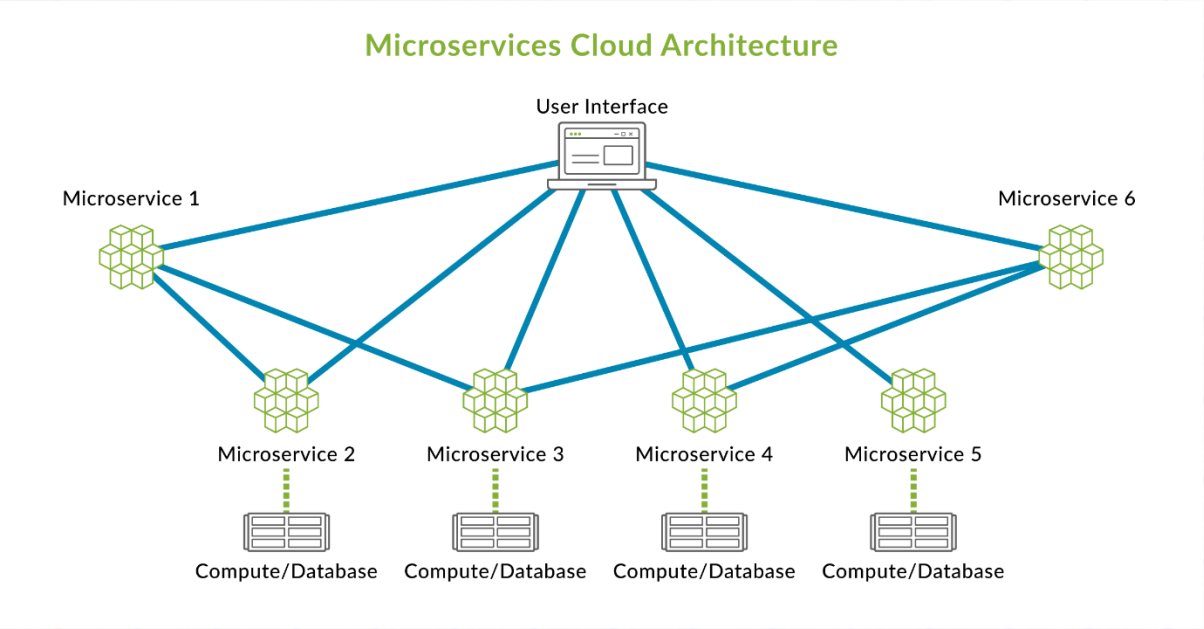
- It calls the Greet function, passing in the client. This function calls the Greet RPC on the server.
- The other function calls (GreetManyTimes, GreetEveryone, LongGreet) are commented out. If you uncomment these lines one by one (one function at a time invoked), you can see the different types of RPCs in action:
- GreetManyTimes demonstrates server-side streaming. The server will send multiple responses for a single request.
- GreetEveryone demonstrates bidirectional streaming. The client and server send multiple messages to each other independently.
- LongGreet demonstrates client-side streaming. The client sends multiple requests and gets a single response from the server.
- You can run the client by running the below commands in the terminal (Integrated VSC Terminal)

```
cd client
go run .
```

Final Source Code >> Link

Meal Mingle

1. Introduction to Microservices.



Microservices are an architectural and organizational approach to software development where software is composed of small independent services that communicate over well-defined APIs. These services are owned by small, self-contained teams.

Microservices architectures make applications easier to scale and faster to develop, enabling innovation and accelerating time-to-market for new features.

Key Points of Microservices:

- Single Responsibility: Each microservice handles one specific function.

- Independent Deployment: Services can be deployed and updated independently.
- Communication: Services communicate via APIs using protocols like HTTP/REST or messaging queues.
- Decentralized Data Management: Each service manages its own database.
- Service Discovery: Tools like Eureka or Consul help services locate each other.
- API Gateway: Acts as a single entry point for client requests, routing them to the appropriate services.
- Resilience: Patterns like circuit breakers and retries handle failures gracefully.
- Monitoring and Logging: Centralized systems track performance and health of services.

Example Workflow:

- Client Request: A client sends a request to the API Gateway.
- API Gateway: Routes the request to the appropriate microservice.
- Service Interaction: The microservice processes the request, possibly interacting with other services.
- Database Access: The microservice accesses its own database.
- Response: The microservice sends the response back through the API Gateway to the client.

Benefits:

- Scalability: Individual services can be scaled independently.

- Flexibility: Teams can use different technologies and tools best suited for each service.
- Resilience: The system remains operational even if some services fail.

For more reference → External Refernce about microservices

Clone the Meal-Mingle Respository >> Link
Final Source Code of Auth-Microservice >> Link

Final Source Code of Restaurant-Microservice >> Link

Final Source Code of Order-Microservice >> Link

2. Auth Microservice.

The "auth-micro" service is typically responsible for both authentication and authorization. It verifies user and admin credentials, manages session information, and handles roles and permissions.

For authentication, it checks if the provided credentials are valid. If they are, it issues a JWT (JSON Web Token) that the user or admin can use for subsequent requests.

For authorization, it checks the roles and permissions associated with the JWT provided in a request. This determines what resources and operations the user or admin has access to.

Additionally, the "auth-micro" service may also handle the creation and management of admin details, ensuring that only authorized personnel can perform administrative tasks.

Step 1. Setup (Installation)

1. Install the VSC and setup the go installation
2. Open Visual studio code integrated terminal.

3. Go to the auth-microservice
    a. **cd auth-micro**
4. Run the commands given below in the terminal to add dependencies.

```
go get -u gorm.io/gorm
go get -u gorm.io/driver/mysql
go get -u google.golang.org/grpc
go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@v1.2
go install google.golang.org/protobuf/cmd/protoc-gen-go@v1.28
go get -u golang.org/x/crypto/bcrypt
go get github.com/grpc-ecosystem/grpc-gateway/v2/protoc-gen-grpc-gateway
go get github.com/grpc-ecosystem/grpc-gateway/v2/runtime
go get github.com/joho/godotenv
go get github.com/dgrijalva/jwt-go
go get -u go.uber.org/zap
go get github.com/gorilla/handlers
go get github.com/stretchr/testify@v1.8.0
```

- **go get -u gorm.io/gorm**: This command fetches the GORM library, which is a popular ORM (Object-Relational Mapper) for Go. The -u flag ensures that the latest version is fetched.

- **go get -u gorm.io/driver/mysql**: This command fetches the MySQL driver for GORM. This allows GORM to interact with MySQL databases.

- **go get -u google.golang.org/grpc**: This command fetches the gRPC library for Go. gRPC is a high-performance, open-source universal RPC framework.

- **go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@v1.2**: This command installs the specific version (v1.2) of the Go gRPC plugin for the Protocol Buffers compiler.

- **go install google.golang.org/protobuf/cmd/protoc-gen-go@v1.28**: This command installs the specific version (v1.28) of the Go plugin for the Protocol Buffers compiler.
- **go get -u golang.org/x/crypto/bcrypt**: This command fetches the bcrypt library from the golang.org/x/crypto package. Bcrypt is a password hashing algorithm.
- **go get github.com/grpc-ecosystem/grpc-gateway/v2/protoc-gen-grpc-gateway**: This command fetches the gRPC Gateway plugin for the Protocol Buffers compiler. gRPC Gateway generates a reverse-proxy server which translates a RESTful JSON API into gRPC.
- **go get github.com/grpc-ecosystem/grpc-gateway/v2/runtime**: This command fetches the runtime package for the gRPC Gateway.
- **go get github.com/joho/godotenv**: This command fetches the godotenv library, which allows you to read from .env files.
- **go get github.com/dgrijalva/jwt-go**: This command fetches the jwt-go library, which is used for encoding and decoding JSON Web Tokens (JWT) in Go.
- **go get -u go.uber.org/zap**: This command fetches the Zap logging library from Uber. Zap is a fast, structured, leveled logging library for Go.
- **go get github.com/gorilla/handlers**: This command fetches the handlers package from the Gorilla web toolkit. This package provides useful handlers for Go's net/http package.

5. Compile the **user.proto** file

```
cd auth-micro
make proto-generate
```

6. Run the command

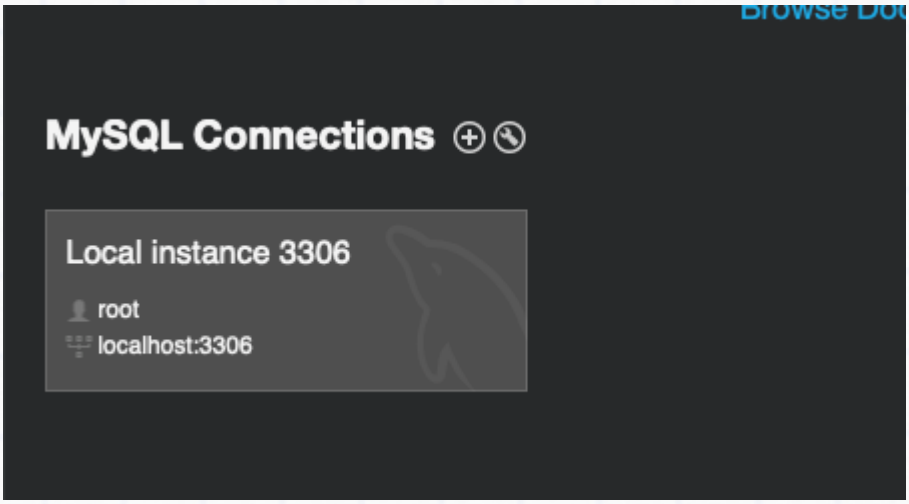
```
go mod tidy
```

PS: - Don't worry if you are getting some errors after cloning the code from the GitHub, all the errors will be gone after adding and importing the dependencies.

1. If you are getting any error while running the command (make proto-generate), make sure the environment variable is set.
2. If the error is "protoc-gen-go-grpc: program not found or is not executable, Please specify a program using absolute path or make sure the program is available in your PATH system variable"

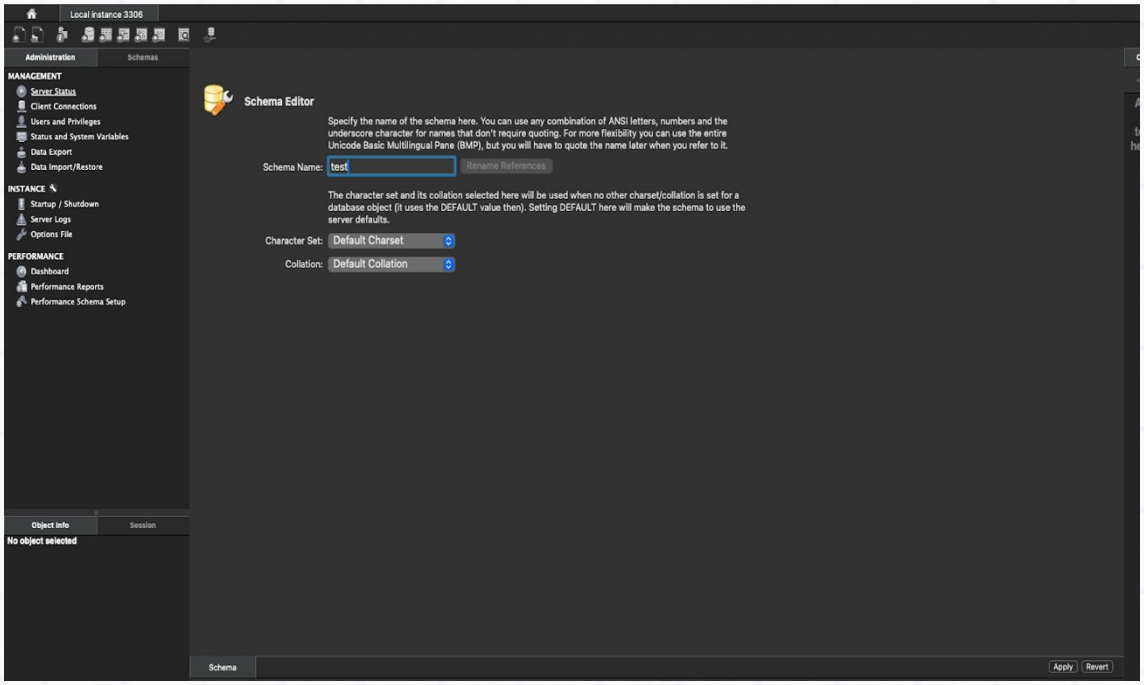
   Use this command "export PATH=$PATH:$(go env GOPATH)/bin " for mac without double quotes.

Open Workbench and click on the Local instance

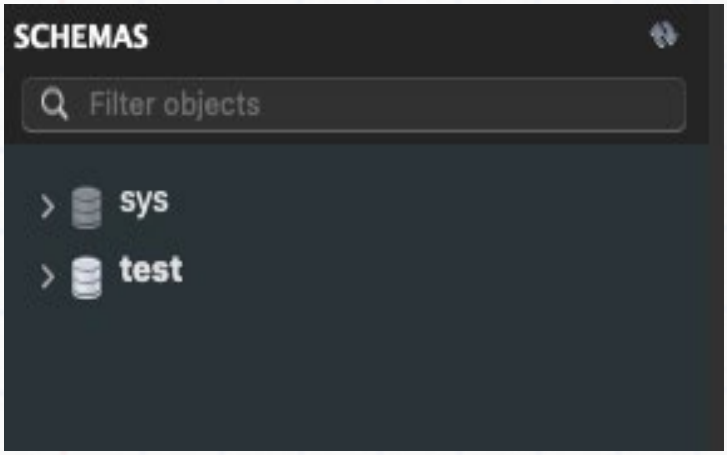


Create a schema test.

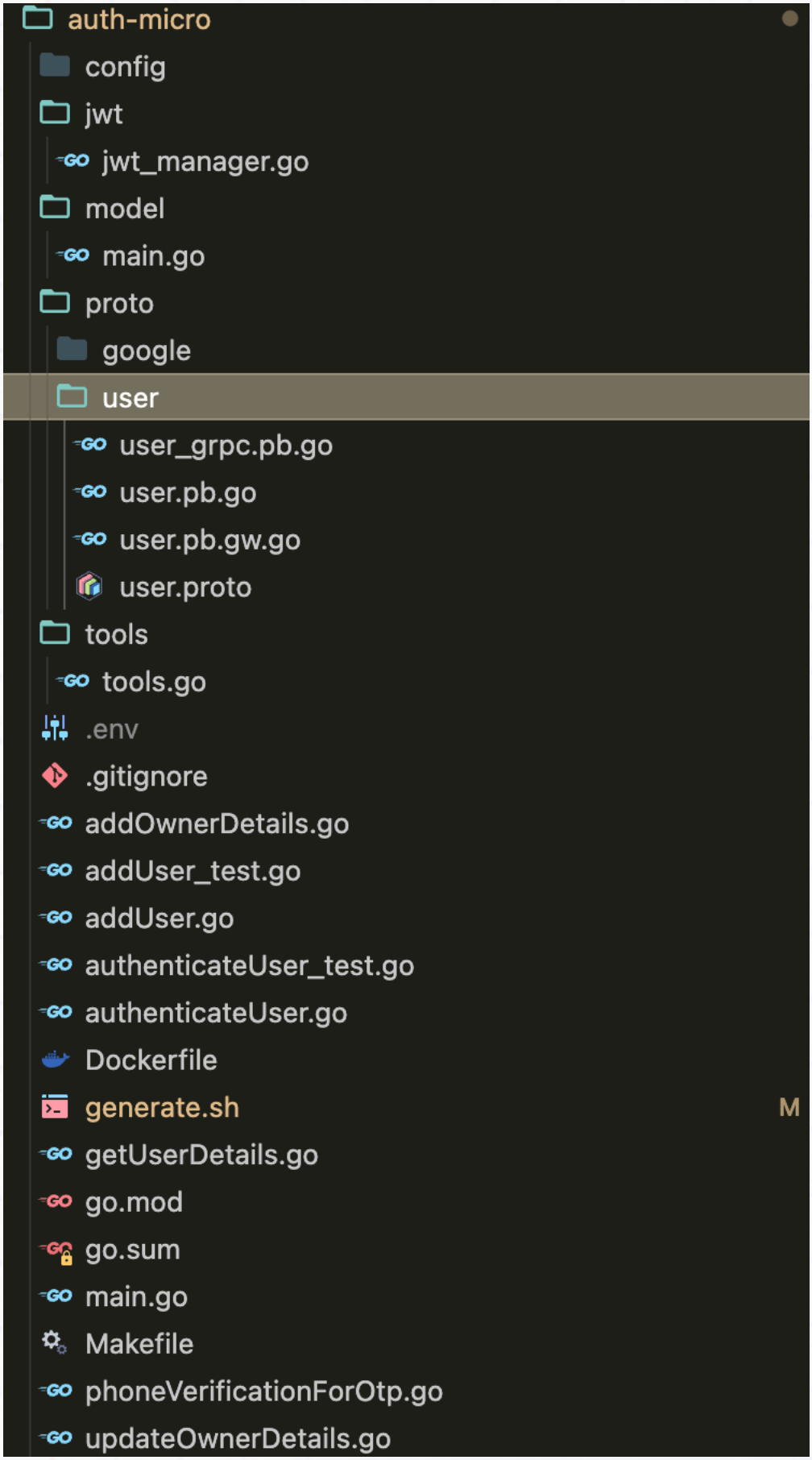
7. Integrating with the SQL Database
   1. Install the SQL Server Link
   2. Install the Workbench Link
   3. Open the workbench and create a schema test on the connected server.



The test schema will be available in the schema list (top left corner).



Step 2. The Directory Structure of the auth-microservice will like this: -


auth-micro
config
jwt
jwt_manager.go
model
main.go
proto
google
user
user_grpc.pb.go
user.pb.go
user.pb.gw.go
user.proto
tools
tools.go
.env
.gitignore
addOwnerDetails.go
addUser_test.go
addUser.go
authenticateUser_test.go
authenticateUser.go
Dockerfile
generate.sh
M
getUserDetails.go
go.mod
go.sum
main.go
Makefile
phoneVerificationForOtp.go
updateOwnerDetails.go

ep 3. main.go/model

gorm:"unique" is a tag used with the GORM (Go Object-Relational Mapper) library. This tag is used to set a unique constraint on a field in a struct that represents a database table. When GORM creates the table in the database, it will ensure that this field has a unique constraint, meaning that no two rows in the table can have the same value for this field. If you try to insert a row with a duplicate value in this field, the database will return an error.

`gorm:"foreignKey:UserID;unique"` This is a GORM (Go Object-Relational Mapper) tag used in Go struct field declarations. It's used to specify database-related properties for the struct field. Here's a breakdown:

- foreignKey:UserID: This tells GORM that the field is a foreign key that references the UserID field in another table.
- unique: This tells GORM that the values in this field should be unique across all records in the table.

Step 4. Create a .env file for environment variables.

```
SECRET_KEY="SECRET_KEY_JWT_TOKEN"
MYSQL_USER="root"
MYSQL_HOST="127.0.0.1"
MYSQL_PASSWORD="<password>"
MYSQL_DATABASE="test"
MYSQL_PORT="3306"
```

PS: - Make sure to replace <password> with the password entered during DB installation.

Step 5. main.go/config

```
package config

import (
    model "auth-microservice/model"
    "fmt"
    "log"
    "os"
    "strings"
    "unicode"
```

```
    "github.com/joho/godotenv"
    "golang.org/x/crypto/bcrypt"
    "gorm.io/driver/mysql"
    "gorm.io/gorm"
)

func DatabaseDsn() string {
    return fmt.Sprintf("%s:%s@tcp(%s:%s)/%s?charset=utf8&parseTime=True&loc=Local",
        os.Getenv("MYSQL_USER"),
        os.Getenv("MYSQL_PASSWORD"),
        os.Getenv("MYSQL_HOST"),
        os.Getenv("MYSQL_PORT"),
        os.Getenv("MYSQL_DATABASE"),
    )
}
func ValidatePhone(userPhone string) bool {
    if len(userPhone) != 10 {
        return false
    }

    for _, char := range userPhone {
        if char < '0' || char > '9' {
            return false
        }
    }
    return true
}
func ValidateFields(userEmail string, userPassword string, userName string, userPhone string)
bool {
    // Responsible for validating the fields
    if userEmail == "" || userPassword == "" || userName == "" || userPhone == "" {
```

```
        return false
    }
    if !strings.Contains(userEmail, "@") || !strings.Contains(userEmail, ".") {
        return false
    }
    // password validation also
    return len(userPassword) >= 6 && ValidatePhone(userPhone)
}
func GoDotEnvVariable(key string) string {
    err := godotenv.Load(".env")
    if err != nil {
        log.Fatalf("Error loading .env file")
    }
    return os.Getenv(key)
}
func ConnectDB() (*gorm.DB, *gorm.DB) {
    // Responsible for connecting to the database
    userdb, err := gorm.Open(mysql.Open(DatabaseDsn()), &gorm.Config{})
    if err != nil {
        panic("failed to connect database")
    }
    // Migrate the schema
    userdb.AutoMigrate(&model.User{})

    ownerDetailsdb, err := gorm.Open(mysql.Open(DatabaseDsn()), &gorm.Config{})
    if err != nil {
        panic("failed to connect database")
    }
    // Migrate the schema
    ownerDetailsdb.AutoMigrate(&model.Details{})
    return userdb, ownerDetailsdb
}
```

```
func GenerateHashedPassword(password string) string {
    // Responsible for generating a hashed password
    hashedPassword, err := bcrypt.GenerateFromPassword([]byte(password),
bcrypt.DefaultCost)
    if err != nil {
        log.Fatalf("Couldn't hash password and the error is %s", err)
    }
    return string(hashedPassword)
}
func ComparePasswords(hashedPassword string, password string) error {
    return bcrypt.CompareHashAndPassword([]byte(hashedPassword), []byte(password))
}

func ValidateOwnerDeatils(AccountNumber string, IFSCCode string,
    BankName string, BranchName string, PanNumber string,
    AdharNumber string, GstNumber string) bool {
    // Responsible for validating the fields
    if AccountNumber == "" || IFSCCode == "" || BankName == "" ||
        BranchName == "" || PanNumber == "" || AdharNumber == "" || GstNumber == "" {
        return false
    }
    if len(AccountNumber) != 12 {
        return false
    }
    // Check if IFSC code is 11 characters and starts with a letter
    if len(IFSCCode) != 11 || !unicode.IsLetter(rune(IFSCCode[0])) {
        return false
    }

    // Check if GST number is 15 characters and matches the pattern
    return true
```

```
}
```

1. **DatabaseDsn()**: This function constructs a MySQL data source name (DSN) from environment variables.

2. **ValidateFields(userEmail, userPassword, userName, userPhone string):** This function validates user input fields. It checks if any of the fields are empty, if the email contains "@" and ".", if the phone number is exactly 10 digits long and only contains digits, and if the password is at least 6 characters long.

3. **GoDotEnvVariable(key string):** This function loads environment variables from a .env file and returns the value of the specified key.

4. **ConnectDB()**: This function connects to a MySQL database using the DSN from DatabaseDsn(). It returns two *gorm.DB instances after migrating the User and Details models.

5. **GenerateHashedPassword(password string)**: This function generates a hashed password using the bcrypt algorithm.

6. **ComparePasswords(hashedPassword, password string):** This function compares a hashed password with a plaintext password. It returns an error if they don't match.

7. **ValidateOwnerDetails(AccountNumber, IFSCCode, BankName, BranchName, PanNumber, AdharNumber, GstNumber string):** This function validates owner details. It checks if any of the fields are empty, if the account number is exactly 10 digits long, if the IFSC code is 11 characters long and starts with a letter, and if the GST number is 15 characters long and matches a specific pattern.

Ps: If you are getting this error “Could not import the dependency”.
Then reinstall the same dependency again, generate.sh file contains all the dependencies command.

Step 6. jwt_manager.go/jwt

```
package jwt
import (
   "context"
```

```
    "fmt"
    "os"
    "strings"
    "time"

    "auth-microservice/model"

    "github.com/dgrijalva/jwt-go"
    "google.golang.org/grpc"
    "google.golang.org/grpc/metadata"
    "google.golang.org/grpc/status"
)

type JWTManager struct {
    secretKey     string
    tokenDuration time.Duration
}
type UserClaims struct {
    jwt.StandardClaims
    UserEmail string
    UserRole  string
}

func NewJWTManager(secretKey string, tokenDuration time.Duration) (*JWTManager, error) {
    return &JWTManager{
        secretKey:     secretKey,
        tokenDuration: tokenDuration,
    }, nil
}
func (manager *JWTManager) GenerateToken(user *model.User) (string, error) {
    claims := UserClaims{
        StandardClaims: jwt.StandardClaims{
```

```
            ExpiresAt: time.Now().Add(manager.tokenDuration).Unix(),
        },
        UserEmail: user.Email,
        UserRole:  user.Role,
    }
    // creating new token...
    token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
    return token.SignedString([]byte(manager.secretKey))
}
func VerifyToken(accessToken string) (*UserClaims, error) {
    token, err := jwt.ParseWithClaims(
        accessToken,
        &UserClaims{},
        func(token *jwt.Token) (interface{}, error) {
            _, ok := token.Method.(*jwt.SigningMethodHMAC)
            if !ok {
                return nil, fmt.Errorf("unexpected token signing method")
            }
            return []byte(os.Getenv("SECRET_KEY")), nil
        },
    )
    if err != nil {
        return nil, fmt.Errorf("invalid token: %w", err)
    }
    claims, ok := token.Claims.(*UserClaims)
    if !ok {
        return nil, fmt.Errorf("invalid token claims")
    }

    return claims, nil
}
```

```
func UnaryInterceptor(ctx context.Context, req any, info *grpc.UnaryServerInfo, handler
grpc.UnaryHandler) (resp any, err error) {
    // skip the authentication for the health check endpoint
    if info.FullMethod == "/userpb.UserService/AddUser" || info.FullMethod ==
"/userpb.UserService/AuthenticateUser" {
        return handler(ctx, req)
    }
    if info.FullMethod == "/userpb.UserService/PhoneVerification" {
        return handler(ctx, req)
    }
    md, ok := metadata.FromIncomingContext(ctx)
    if !ok {
        return nil, status.Errorf(401, "metadata is not provided")
    }
    tokenString := md.Get("authorization")
    if len(tokenString) == 0 {
        return nil, status.Errorf(401, "authorization token is not provided")
    }
    token := strings.Split(tokenString[0], " ")
    // Parse JWT token
    claims, err := VerifyToken(token[1])
    if err != nil {
        return nil, status.Errorf(401, "token is invalid: %v", err)
    }
    // Pass useremail to context for further use
    ctx = context.WithValue(ctx, "userEmail", claims.UserEmail)
    ctx = context.WithValue(ctx, "userRole", claims.UserRole)
    // Proceed with the request
    return handler(ctx, req)
}
```

1. **JWTManager struct**: This struct holds the secret key for signing the JWT and the duration for which the token is valid.

2. **UserClaims struct**: This struct extends the standard JWT claims with UserEmail and UserRole fields.

3. **NewJWTManager(secretKey string, tokenDuration time.Duration):** This function creates a new JWT manager with the provided secret key and token duration.

4. **GenerateToken(user *model.User):** This method of JWTManager generates a new JWT for a user. The token's claims include the user's email, role, and an expiration time.

5. **VerifyToken(accessToken string):** This function verifies a JWT and returns the claims if the token is valid. It checks the signing method and the secret key.

6. **UnaryInterceptor(ctx context.Context, req any, info *grpc.UnaryServerInfo, handler grpc.UnaryHandler):** This function is a gRPC unary interceptor. It checks the incoming request for an authorization token in the metadata. If the token is valid, it adds the user's email and role to the context and passes the request to the handler. If the token is invalid or not provided, it returns an error. It also skips authentication for the AddUser and AuthenticateUser endpoints.

Guide on Jwt-Token >> Link
Guide on Authentication and Authorization >> Link

Step 7. Creating a gRPC Server with gRPC Gateway More on gRPC Gateway in main.go file

```
package main

import (
   "auth-microservice/config"
```

```
    "auth-microservice/jwt"
    userpb "auth-microservice/proto/user"
    "context"
    "log"
    "net"
    "net/http"
    "os"
    "time"

    "github.com/gorilla/handlers"
    "github.com/grpc-ecosystem/grpc-gateway/v2/runtime"
    "github.com/joho/godotenv"
    "go.uber.org/zap"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials/insecure"
    "gorm.io/gorm"
)

const (
    StatusBadRequest      = 400
    StatusConflict        = 409
    StatusInternalServerError = 500
    StatusOK              = 200
    StatusCreated         = 201
    StatusNotFound        = 404
    StatusUnauthorized    = 401
    StatusForbidden       = 403
)
var logger *zap.Logger

func init() {
    var err error
```

```
    logger, err = zap.NewDevelopment()
    if err != nil {
        panic(err)
    }
    defer logger.Sync()
}

var userDbConnector *gorm.DB
var ownerDetailsDbConector *gorm.DB

type UserService struct {
    userpb.UnimplementedUserServiceServer
    jwtManager *jwt.JWTManager
}

// Responsible for starting the server
func startServer() {
    // Log a message

    logger.Info("Starting server...")
    // Initialize the gotenv file..
    err := godotenv.Load()
    if err != nil {
        logger.Fatal("Error loading .env file", zap.Error(err))
    }

    // Create a new context
    userDbConnector, ownerDetailsDbConector = config.ConnectDB()

    // Start the server on port 50051
    listener, err := net.Listen("tcp", "localhost:50051")
    if err != nil {
```

```
    logger.Fatal("Failed to start server", zap.Error(err))
}

// Creating a new JWT Manager
JwtManager, err := jwt.NewJWTManager(os.Getenv("SECRET_KEY"), 5*time.Hour)
if err != nil {
    logger.Fatal("Failed to create JWT manager", zap.Error(err))
}

// Create a new gRPC server
grpcServer := grpc.NewServer(
    grpc.UnaryInterceptor(jwt.UnaryInterceptor),
)

// Register the service with the server
userpb.RegisterUserServiceServer(grpcServer, &UserService{jwtManager: JwtManager})

// Start the server in a new goroutine
go func() {
    if err := grpcServer.Serve(listener); err != nil {
        logger.Fatal("Failed to serve", zap.Error(err))
    }
}()

// Create a new gRPC-Gateway server
connection, err := grpc.DialContext(
    context.Background(),
    "localhost:50051",
    grpc.WithBlock(),
    grpc.WithTransportCredentials(insecure.NewCredentials()),
)
if err != nil {
```

```
        logger.Fatal("Failed to dial server", zap.Error(err))
    }

    // Create a new gRPC-Gateway mux
    gwmux := runtime.NewServeMux()

    // Register the service with the gRPC-Gateway
    err = userpb.RegisterUserServiceHandler(context.Background(), gwmux, connection)
    if err != nil {
        logger.Fatal("Failed to register gateway", zap.Error(err))
    }

    // Enable CORS
    corsOrigins := handlers.AllowedOrigins([]string{"http://localhost:3000"})
    corsMethods := handlers.AllowedMethods([]string{"GET", "POST", "PUT", "DELETE",
"OPTIONS"})
    corsHeaders := handlers.AllowedHeaders([]string{"Content-Type", "Authorization"})
    corsHandler := handlers.CORS(corsOrigins, corsMethods, corsHeaders)
    wrappedGwmux := corsHandler(gwmux)

    // Create a new HTTP server
    gwServer := &http.Server{
        Addr:    ":8090",
        Handler: wrappedGwmux,
    }
    logger.Info("Serving gRPC-Gateway", zap.String("address", "http://0.0.0.0:8090"))
    if err := gwServer.ListenAndServe(); err != http.ErrServerClosed {
        log.Fatalf("Failed to listen and serve: %v", err)
    }
}

func main() {
```

```
    // Start the server
    startServer()
}
```

This Go code is the main entry point for an authentication microservice. It sets up and starts a gRPC server and a gRPC-Gateway server. Here's a breakdown:

1. **init()**: This function initializes a logger using the zap library.

2. **UserService struct**: This struct implements the UserServiceServer interface from the userpb package. It includes a JWT manager for handling JSON Web Tokens.

3. **startServer()**: This function starts the gRPC and gRPC-Gateway servers. It does the following:

    i. Loads environment variables from a .env file using godotenv.
    ii. Connects to the database using config.ConnectDB().
    iii. Starts a gRPC server on localhost:50051. It registers the UserService with the server and uses a JWT unary interceptor for handling tokens.
    iv. Starts a gRPC-Gateway server on localhost:8090. This server acts as a HTTP/JSON proxy to the gRPC server. It also sets up CORS (Cross-Origin Resource Sharing) using the handlers package from gorilla.

4. **main():** This function simply calls startServer() to start the servers.

5. The gRPC server and gRPC-Gateway server run concurrently in separate goroutines. The gRPC server handles gRPC requests, while the gRPC-Gateway server translates HTTP/JSON requests into gRPC requests.

Go to auth-microservice :- cd auth-micro
Run Command (to start the server) :- **go run .**

Step 8. Adding addUser RPC in addUser.go file.

```
package main

import (
    "auth-microservice/config"
    "auth-microservice/model"
    userpb "auth-microservice/proto/user"
    "context"
    "fmt"
    "strconv"

    "go.uber.org/zap"
    "gorm.io/gorm"
)

// AddUser is a RPC that adds a new user to the database
func (userServiceManager *UserService) AddUser(ctx context.Context, request
*userpb.AddUserRequest) (*userpb.AddUserResponse, error) {
    userEmail := request.UserEmail
    userPassword := request.UserPassword
    userName := request.UserName
    userPhone := request.UserPhone
    userRole := request.UserRole

    logger.Info("Received AddUser request",
    zap.String("userEmail", userEmail), zap.String("userName", userName),
    zap.String("userPhone", userPhone), zap.String("userRole", userRole))
```

```
    if userRole != model.UserRole && userRole != model.AdminRole {
        logger.Warn("Invalid user role", zap.String("userRole", userRole))
        return &userpb.AddUserResponse{
            Data:       nil,
            Message:    "Invalid user role. User role can only be user or admin",
            Error:      "Invalid Role",
            StatusCode: StatusBadRequest,
        }, nil
    }
    if !config.ValidateFields(userEmail, userPassword, userName, userPhone) {
        logger.Warn("Invalid request fields",
        zap.String("userEmail", userEmail),
        zap.String("userName", userName),
        zap.String("userPhone", userPhone))

        return &userpb.AddUserResponse{
            Data:       nil,
            Message:    "The request contains missing or invalid fields. Make sure Phone number is
10 digits long.",
            Error:      "Invalid Request",
            StatusCode: int64(StatusBadRequest),
        }, nil
    }
    var existingUser model.User
    userNotFoundError := userDbConnector.Where("email = ?",
userEmail).First(&existingUser).Error
    // If the user is not found, create a new user
    if userNotFoundError == gorm.ErrRecordNotFound {
        hashedPassword := config.GenerateHashedPassword(userPassword)
        newUser := &model.User{Name: userName, Email: userEmail,
            Phone: userPhone, Password: hashedPassword, Role: userRole}
```

```
    // Create a new user in the database and return the primary key if successful or an error if
it fails
    primaryKey := userDbConnector.Create(newUser)
    if primaryKey.Error != nil {
        logger.Error("Failed to create user", zap.String("userPhone",existingUser.Phone),
zap.Error(primaryKey.Error))
        return &userpb.AddUserResponse{
            Data:       nil,
            Message:    "The phone number is already registered.",
            Error:      "Conflict",
            StatusCode: int64(StatusConflict),
        }, nil
    }
    // Gennerating the the jwt token.
    token, err := userServiceManager.jwtManager.GenerateToken(newUser)
    if err != nil {
        logger.Error("Error in generating token")
        return &userpb.AddUserResponse{
            Data:       nil,
            Error:      "Internal Server Error",
            StatusCode: int64(StatusInternalServerError),
            Message:    "Security Issues, Please try again later.",
        }, nil
    }
    logger.Info(fmt.Sprintf("User %s created successfully", newUser.Name))
    return &userpb.AddUserResponse{
        Message: "User created successfully",
        Error:   "", StatusCode: int64(StatusOK),
        Data: &userpb.Responsedata{
            User: &userpb.User{
                UserId:    strconv.FormatUint(uint64(newUser.ID), 10),
                UserName:  newUser.Name,
```

```
                UserEmail: newUser.Email,
                UserPhone: newUser.Phone,
            },
            Token: token,
        },
    }, nil
}
logger.Warn("User email already registered", zap.String("userEmail", userEmail))
return &userpb.AddUserResponse{
    Data:       nil,
    Message:    "User Email is already registered",
    Error:      "Conflict",
    StatusCode: int64(StatusConflict),
}, nil
}
```

1. The method AddUser takes in a context and a request of type AddUserRequest from the userpb package. It returns a response of type AddUserResponse and an error.

2. It first extracts the user's email, password, name, phone, and role from the request and logs the received request.

3. It then checks if the user role is valid (either UserRole or AdminRole). If not, it logs a warning and returns a response with an error message and a StatusBadRequest status code.

4. It then validates the request fields (email, password, name, phone) using the ValidateFields function from the config package. If any of these fields are invalid, it logs a warning and returns a response with an error message and a StatusBadRequest status code.

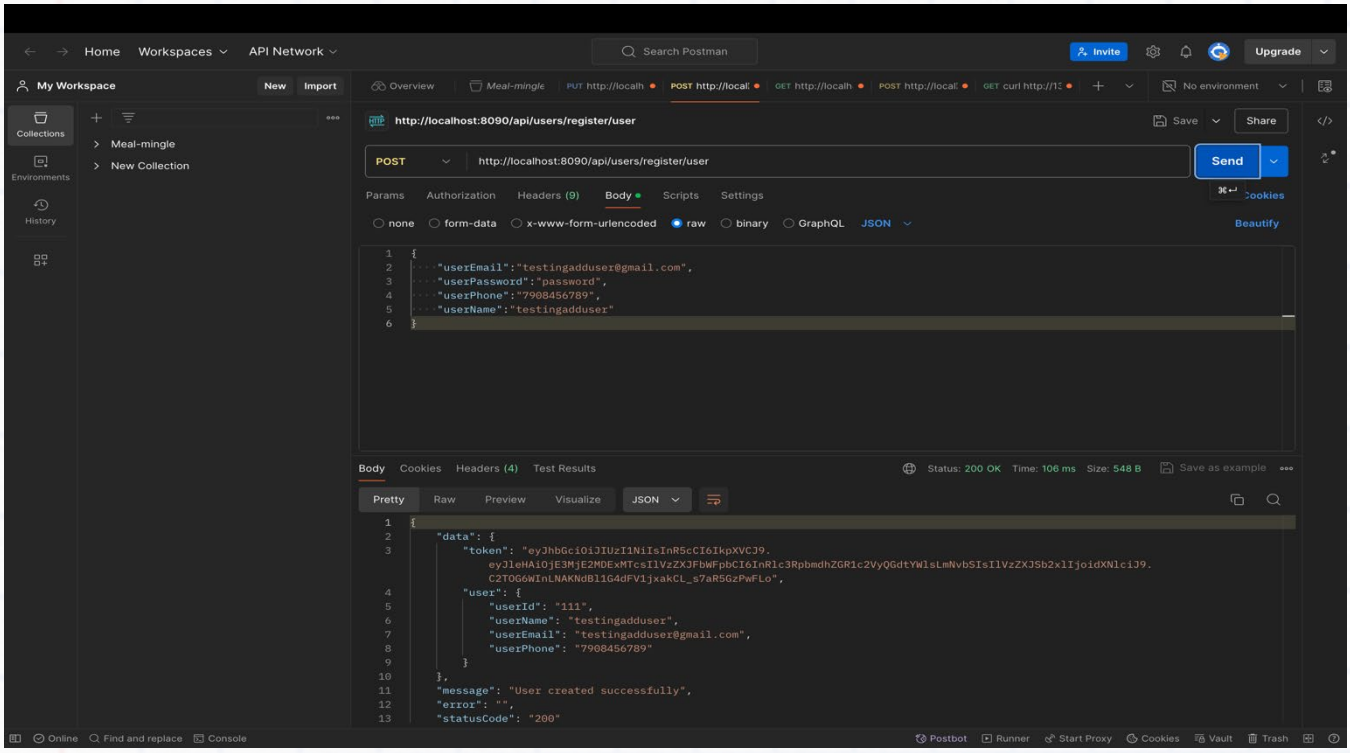
5. It then tries to find a user with the provided email in the database. If the user is found, it logs a warning and returns a response with an error message and a StatusConflict status code.
6. If the user is not found, it hashes the provided password using the GenerateHashedPassword function from the config package, creates a new User struct with the provided details, and tries to add the new user to the database.

7. If there's an error in adding the user to the database, it logs the error and returns a response with an error message and a StatusConflict status code.

8. If the user is added successfully, it generates a JWT token for the user using the GenerateToken method of the jwtManager in the UserService. If there's an error in generating the token, it logs the error and returns a response with an error message and a StatusInternalServerError status code.

9. If the token is generated successfully, it logs that the user was created successfully and returns a response with the user's details and the generated token, and a StatusOK status code.

Testing addUser gRpc route through postman, as gprc gateway is already impelmented.
You can find the request url in the rpc implementation inside user.proto file


Home
Workspaces
API Network
Search Postman
Invite
Upgrade
My Workspace
New
Import
Collections
Environments
History
Meal-mingle
New Collection
Overview
Meal-mingle
No environment
http://localhost:8090/api/users/register/user
Save
Share
POST
http://localhost:8090/api/users/register/user
Send
Params
Authorization
Headers (9)
Body
Scripts
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{
"userEmail":"testingadduser@gmail.com",
"userPassword":"password",
"userPhone":"7908456789",
"userName":"testingadduser"
}
Body
Cookies
Headers (4)
Test Results
Status: 200 OK
Time: 106 ms
Size: 548 B
Save as example
Pretty
Raw
Preview
Visualize
JSON
{
"data": {
"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJleHAiOjE3MjE2MDExMTcsIlVzZXJFbWFpbCI6InRlc3RpbmdhZGR1c2VyQGdtYWlsLmNvbSIsIlVzZXJSb2xlIjoidXNlciJ9.
C2TOG6WInLNAKNdBl1G4dFV1jxakCL_s7aR5GzPwFLo",
"user": {
"userId": "111",
"userName": "testingadduser",
"userEmail": "testingadduser@gmail.com",
"userPhone": "7908456789"
}
},
"message": "User created successfully",
"error": "",
"statusCode": "200"
Online
Find and replace
Console
Postbot
Runner
Start Proxy
Cookies
Vault
Trash

```
rpc AddUser(AddUserRequest) returns (AddUserResponse){
    option (google.api.http) = {
        post: "/api/users/register/{userRole}"
        body: "*"
    };
};
message user {
    string userId = 1;
    string userName = 2;
    string userEmail = 3;
    string userPhone = 4;
}
message Responsedata{
    string token = 1;
    user user = 2;
}
message AddUserRequest {
    string userName = 1;
    string userEmail = 2;
    string userPhone = 3;
    string userPassword = 4;
    string userRole = 5;
}
message AddUserResponse {
    Responsedata data = 1;
    string message = 2;
    string error = 3;
    int64 statusCode = 4;
}
```

Request Body

Response body

More on Http Request >> Link
More on Postman >> Link

Step 9. Adding authenticateUser.go file

```
package main

import (
  "auth-microservice/config"
  "auth-microservice/model"
  userpb "auth-microservice/proto/user"
```

```
    "context"
    "strconv"
    "strings"

    "go.uber.org/zap"
)


func (UserServiceManager *UserService) AuthenticateUser(ctx context.Context, request
*userpb.AuthenticateUserRequest) (*userpb.AuthenticateUserResponse, error) {
    userEmail := request.UserEmail
    userPassword := request.UserPassword
    logger.Info("Received AuthenticateUser request",
        zap.String("userEmail", userEmail))

    // Check if the request contains the required fields
    if userEmail == "" || userPassword == "" || !strings.Contains(userEmail, "@") ||
    !strings.Contains(userEmail, ".") || len(userPassword) < 6 {

        logger.Warn("Invalid request fields",
            zap.String("userEmail", userEmail),
            zap.String("userPaasword", userPassword))

        return &userpb.AuthenticateUserResponse{
            Data:       nil,
            Message:    "The request contains missing or invalid fields.",
            Error:      "Invalid fields!",
            StatusCode: StatusBadRequest,
        }, nil
    }
    var existingUser model.User
    userNotFoundError := userDbConnector.Where("email = ?",
userEmail).First(&existingUser).Error
```

```
    // If the user is not found, create a new user with the provided details
    if userNotFoundError != nil {
        logger.Warn("Authentication failed",
            zap.String("userEmail", userEmail),
            zap.String("userRole", existingUser.Role),
            zap.Error(userNotFoundError))
        return &userpb.AuthenticateUserResponse{
            Data:       nil,
            Message:    "Authentication Failed, User not found OR Invalid role",
            Error:      "Not Found",
            StatusCode: StatusNotFound,
        }, nil
    }
    if existingUser.Role != request.Role {
        logger.Warn("Invalid role for user", zap.String("userEmail", userEmail),
zap.String("userRole", existingUser.Role))
        return &userpb.AuthenticateUserResponse{
            Data:       nil,
            Message:    "Invalid role",
            Error:      "Unauthorized",
            StatusCode: StatusUnauthorized,
        }, nil
    }
    if config.ComparePasswords(existingUser.Password, userPassword) != nil {
        logger.Warn("Authentication failed due to wrong password",
            zap.String("userEmail", userEmail))
        return &userpb.AuthenticateUserResponse{
            Message: "Authentication Failed,Wrong Password",
            Error:   "Unauthorized", StatusCode: StatusUnauthorized,
        }, nil
    }
    // Gennerating the the jwt token.
```

```
    token, err := UserServiceManager.jwtManager.GenerateToken(&existingUser)
    if err != nil {
        logger.Error("Error in generating token",
            zap.String("userEmail", userEmail),
            zap.Error(err))
        return &userpb.AuthenticateUserResponse{
            Data:       nil,
            Error:      "Internal Server Error",
            StatusCode: StatusInternalServerError,
            Message:    "Security Issues, Please try again later.",
        }, nil

    }
    logger.Info("User authenticated successfully",
        zap.String("userEmail", existingUser.Email),
        zap.String("userName", existingUser.Name))

    return &userpb.AuthenticateUserResponse{Error: "",
        Message:    "User authenticated successfully",
        StatusCode: StatusCreated,
        Data: &userpb.Responsedata{
            User: &userpb.User{
                UserId:    strconv.FormatUint(uint64(existingUser.ID), 10),
                UserName:  existingUser.Name,
                UserEmail: existingUser.Email,
                UserPhone: existingUser.Phone,
            },
            Token: token,
        },
    }, nil
}
```

1. The method AuthenticateUser takes in a context and a request of type AuthenticateUserRequest from the userpb package. It returns a response of type AuthenticateUserResponse and an error.

2. It first extracts the user's email and password from the request and logs the received request.

3. It then checks if the request contains the required fields (email and password). If any of these fields are missing or invalid, it logs a warning and returns a response with an error message and a StatusBadRequest status code.

4. It then tries to find a user with the provided email in the database. If the user is not found, it logs a warning and returns a response with an error message and a StatusNotFound status code.

5. If the user is found, it checks if the user's role matches the role in the request. If they don't match, it logs a warning and returns a response with an error message and a StatusUnauthorized status code.

6. It then compares the provided password with the user's password in the database using the ComparePasswords function from the config package. If the passwords don't match, it logs a warning and returns a response with an error message and a StatusUnauthorized status code.

7. If the passwords match, it generates a JWT token for the user using the GenerateToken method of the jwtManager in the UserService. If there's an error in generating the token, it logs the error and returns a response with an error message and a StatusInternalServerError status code.

8. If the token is generated successfully, it logs that the user was authenticated successfully and returns a response with the user's details and the generated token, and a StatusCreated status code.

Step 10. Adding addOwnerDetails.go file

```
package main

import (
    "auth-microservice/config"
    "auth-microservice/model"
    userpb "auth-microservice/proto/user"
    "context"
    "strconv"

    "go.uber.org/zap"
)

func (*UserService) AddOwnerDetails(ctx context.Context, request
*userpb.AddOwnerDetailsRequest) (*userpb.AddOwnerDetailsResponse, error) {
    logger.Info("Received AddOwnerDetails request",
        zap.String("accountNumber", request.AccountNumber),
        zap.String("bankName", request.BankName),
        zap.String("branchName", request.BranchName),
        zap.String("ifscCode", request.IfscCode),
        zap.String("panNumber", request.PanNumber),
        zap.String("adharNumber", request.AdharNumber),
        zap.String("gstNumber", request.GstNumber))

    userEmail, emailCtxError := ctx.Value("userEmail").(string)
    userRole, roleCtxError := ctx.Value("userRole").(string)
```

```
    if !emailCtxError || !roleCtxError {
        logger.Error("Failed to get user email and role from context")
        return &userpb.AddOwnerDetailsResponse{
            Message:    "Failed to get user email and role from context",
            Error:      "Internal Server Error",
            StatusCode: StatusInternalServerError,
        }, nil
    }
    logger.Info("Context values retrieved", zap.String("userEmail", userEmail),
zap.String("userRole", userRole))
    if userRole != model.AdminRole {
        logger.Warn("Permission denied", zap.String("userRole", userRole))
        return &userpb.AddOwnerDetailsResponse{
            Data:       nil,
            Message:    "You do not have permission to perform this action. Only admin can add
owner details",
            StatusCode: StatusForbidden,
            Error:      "Forbidden",
        }, nil
    }
    var ownerDetails model.Details
    ownerDetails.AccountNumber = request.AccountNumber
    ownerDetails.BankName = request.BankName
    ownerDetails.BrachName = request.BranchName
    ownerDetails.IfscCode = request.IfscCode
    ownerDetails.PanNumber = request.PanNumber
    ownerDetails.AdharNumber = request.AdharNumber
    ownerDetails.GstNumber = request.GstNumber

    // validate fields here
    if !config.ValidateOwnerDeatils(request.AccountNumber, request.IfscCode,
```

```
        request.BankName, request.BranchName, request.PanNumber, request.AdharNumber,
request.GstNumber) {
        logger.Warn("Invalid owner details", zap.String("userEmail", userEmail))
        return &userpb.AddOwnerDetailsResponse{
            Data:       nil,
            Message:    "Invalid owner details make sure to use mentioned format.",
            Error:      "Invalid Fields",
            StatusCode: StatusBadRequest,
        }, nil
    }
    // check if the user is owner or not
    var user model.User
    userDbConnector.Where("email = ?", userEmail).First(&user)
    ownerDetails.UserId = strconv.FormatUint(uint64(user.ID), 10)

    // check if the owner details already exists
    var existingOwnerDetails model.Details
    ownerDetailsNotFoundError := ownerDetailsDbConector.Where("user_id = ?",
ownerDetails.UserId).First(&existingOwnerDetails).Error
    if ownerDetailsNotFoundError != nil {
        // create a new owner details
        primaryKey := ownerDetailsDbConector.Create(&ownerDetails)
        if primaryKey.Error != nil {
            logger.Error("Failed to create owner details", zap.String("userId", ownerDetails.UserId),
zap.Error(primaryKey.Error))
            return &userpb.AddOwnerDetailsResponse{
                Data:       nil,
                Message:    "Owner details already exist",
                Error:      "Failed to create owner details",
                StatusCode: StatusConflict,
            }, nil
        }
```

```
    logger.Info("Owner details added successfully", zap.String("userId", ownerDetails.UserId))
    return &userpb.AddOwnerDetailsResponse{
        Data: &userpb.AddOwnerDetailsResponseData{
            UserId: ownerDetails.UserId,
        },
        Message:    "Owner details added successfully",
        Error:      "",
        StatusCode: StatusCreated,
    }, nil
  }
  logger.Warn("Owner details already exist", zap.String("userId", ownerDetails.UserId))
  return &userpb.AddOwnerDetailsResponse{
    Message:    "Owner details already exists",
    Error:      "Conflict",
    StatusCode: StatusConflict,
  }, nil
}
```

1. This Go code is a method of the UserService struct that adds owner details to the database. Here's a breakdown:

2. The method AddOwnerDetails takes in a context and a request of type AddOwnerDetailsRequest from the userpb package. It returns a response of type AddOwnerDetailsResponse and an error.

3. It first logs the received request, which includes the account number, bank name, branch name, IFSC code, PAN number, Aadhar number, and GST number.

4. It then tries to extract the user's email and role from the context. If it fails, it logs an error and returns a response with an error message and a StatusInternalServerError status code.

5. If the user's role is not AdminRole, it logs a warning and returns a response with an error message and a StatusForbidden status code.

6. It then creates a new Details struct with the provided details and validates these details using the ValidateOwnerDetails function from the config package. If the details are invalid, it logs a warning and returns a response with an error message and a StatusBadRequest status code.

7. It then tries to find a user with the provided email in the database and sets the UserId of the Details struct to the ID of the found user.

8. It then tries to find existing owner details for the user in the database. If the details are not found, it tries to add the new owner details to the database.

9. If there's an error in adding the owner details to the database, it logs the error and returns a response with an error message and a StatusConflict status code.

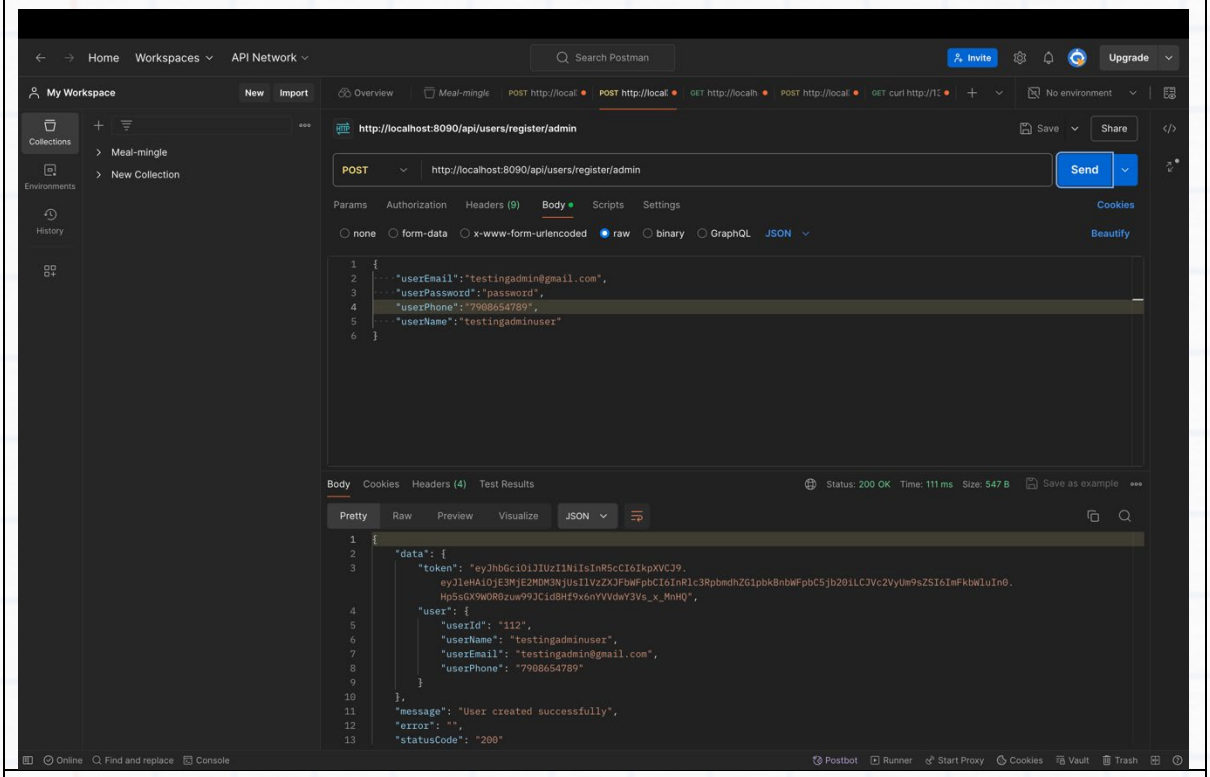
10. If the owner details are added successfully, it logs that the details were added successfully and returns a response with the user's ID and a StatusCreated status code.

11. If the owner details already exist in the database, it logs a warning and returns a response with an error message and a StatusConflict status code.

>> Testing addOwner rpc routes is little bit diff. As compared to testing addUser rpc, because it include jwt token in the authorization, bearer token (part of request body)

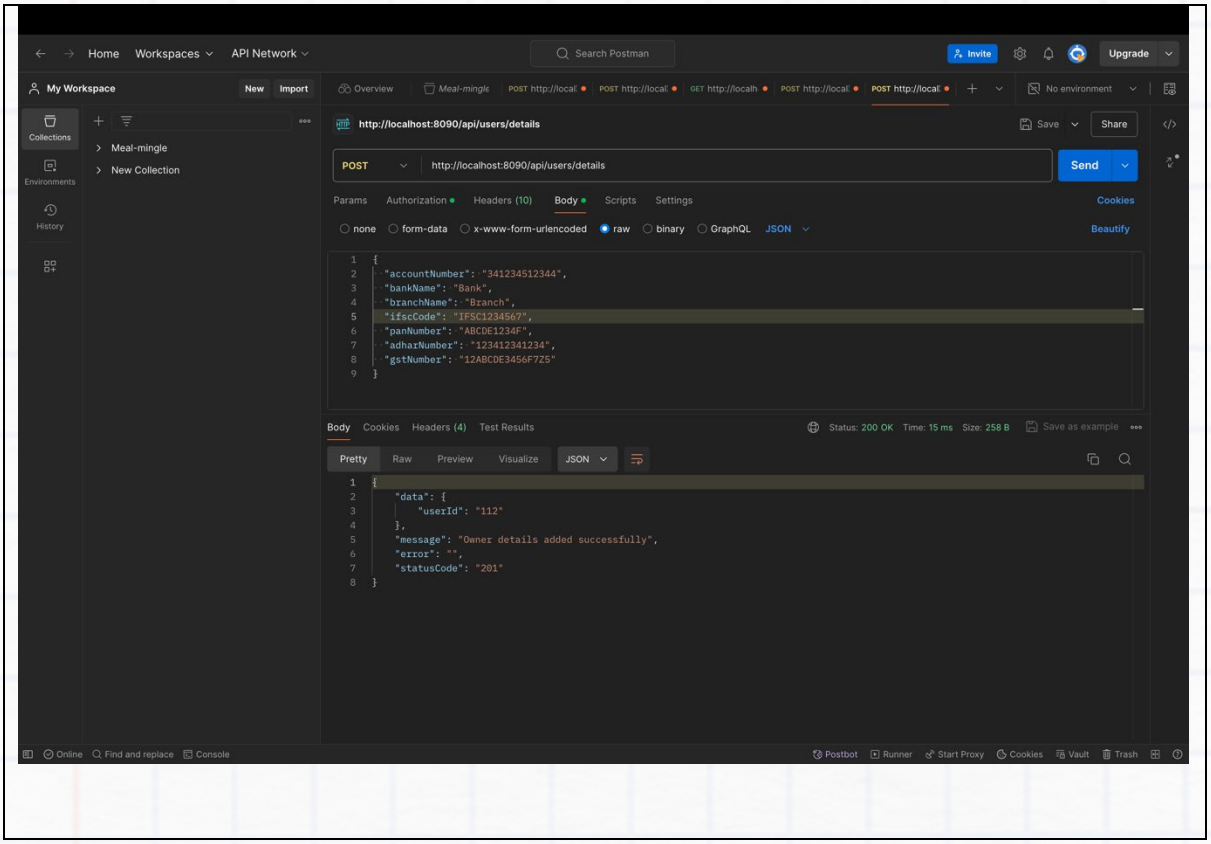
>> Here is the step by step guide.

1. Register the admin or if already registered login the admin.



2. Grab the token, click on the Authorization (between Params and Headers), select Bearer Token and paste the token inside the textbox.

3. Now, we can test the addOwnerDetails rpc.



PS: With the same steps we can test routes that are protected by jwt-token.
More information on routes >> Link
Refer this to get more understanding on how to convert protobuff to their corresponding json >> Link

Step 11. Adding updateOwnerDetails.go file

```
package main

import (
    "auth-microservice/config"
    "auth-microservice/model"
    userpb "auth-microservice/proto/user"
```

```
    "context"
    "strconv"

    "go.uber.org/zap"
)

func (*UserService) UpdateOwnerDetails(ctx context.Context, request
*userpb.UpdateOwnerDetailsRequest) (*userpb.UpdateOwnerDetailsResponse, error) {
    // get the user email from the context
    userEmail, ok := ctx.Value("userEmail").(string)
    if !ok {
        logger.Error("Failed to get user email from context")
        return &userpb.UpdateOwnerDetailsResponse{
            Data:       nil,
            Message:    "Failed to get user email from context",
            StatusCode: StatusInternalServerError,
            Error:      "Internal Server Error",
        }, nil
    }
    logger.Info("Received UpdateOwnerDetails request", zap.String("userEmail", userEmail))
    // get the user email from the database
    var user model.User
    var ownerDetails model.Details
    err := userDbConnector.Where("email = ?", userEmail).First(&user)
    if err.Error != nil {
        logger.Warn("Admin does not exist", zap.String("userEmail", userEmail),
zap.Error(err.Error))
        return &userpb.UpdateOwnerDetailsResponse{
            Data:       nil,
            Message:    "Admin does not exist",
            Error:      "Not authorized",
            StatusCode: StatusNotFound,
```

```
        }, nil
    }
    logger.Info("Retrieved user details successfully", zap.String("userEmail", userEmail))
    // validate fields here
    if !config.ValidateOwnerDeatils(request.AccountNumber, request.IfscCode,
        request.BankName, request.BranchName, request.PanNumber,
        request.AdharNumber, request.GstNumber) {
        logger.Warn("Invalid owner details", zap.String("userEmail", userEmail))
        return &userpb.UpdateOwnerDetailsResponse{
            Data:       nil,
            Message:    "You do not have permission to perform this action. Invalid owner details.",
            Error:      "Invalid Fields",
            StatusCode: StatusBadRequest,
        }, nil
    }
    // check if owner details already exists
    ownerDetailsNotFoundError := ownerDetailsDbConector.Where("user_id = ?",
user.ID).First(&ownerDetails).Error
    if ownerDetailsNotFoundError != nil || user.Role != model.AdminRole {
        logger.Warn("Owner details not found", zap.String("userEmail", userEmail),
zap.Error(ownerDetailsNotFoundError))
        return &userpb.UpdateOwnerDetailsResponse{
            Data:       nil,
            Message:    "Owner details not found",
            Error:      "Not authorized",
            StatusCode: StatusNotFound,
        }, nil
    }

    ownerDetails.AccountNumber = request.AccountNumber
    ownerDetails.BankName = request.BankName
    ownerDetails.BrachName = request.BranchName
```

```
    ownerDetails.IfscCode = request.IfscCode
    ownerDetails.PanNumber = request.PanNumber
    ownerDetails.AdharNumber = request.AdharNumber
    ownerDetails.GstNumber = request.GstNumber
    ownerDetails.UserId = strconv.FormatUint(uint64(user.ID), 10)

    // Save updated owner details
    if err := ownerDetailsDbConector.Where("user_id = ?", user.ID).Save(&ownerDetails).Error;
err != nil {
        logger.Error("Failed to update owner details", zap.String("userEmail", userEmail),
zap.Error(err))
        return &userpb.UpdateOwnerDetailsResponse{
            Data:       nil,
            Message:    "Failed to update owner details",
            Error:      "Internal Server Error",
            StatusCode: StatusInternalServerError,
        }, nil
    }
    logger.Info("Owner details updated successfully", zap.String("userEmail", userEmail))
    return &userpb.UpdateOwnerDetailsResponse{
        Data: &userpb.UpdateOwnerDetailsResponseData{
            UserId: ownerDetails.UserId,
        },
        Message:    "Owner details updated successfully",
        StatusCode: StatusOK,
        Error:      "",
    }, nil
}
```

1. The method UpdateOwnerDetails takes in a context and a request of type UpdateOwnerDetailsRequest from the userpb package. It returns a response of type UpdateOwnerDetailsResponse and an error.

2. It first tries to extract the user's email from the context. If it fails, it logs an error and returns a response with an error message and a StatusInternalServerError status code.

3. It then tries to find a user with the provided email in the database. If the user is not found, it logs a warning and returns a response with an error message and a StatusNotFound status code.

4. It then validates the provided owner details using the ValidateOwnerDetails function from the config package. If the details are invalid, it logs a warning and returns a response with an error message and a StatusBadRequest status code.

5. It then tries to find existing owner details for the user in the database. If the details are not found or the user's role is not AdminRole, it logs a warning and returns a response with an error message and a StatusNotFound status code.

6. It then updates the owner details with the provided details and tries to save the updated details to the database.

7. If there's an error in saving the updated details to the database, it logs the error and returns a response with an error message and a StatusInternalServerError status code.

8. If the owner details are updated successfully, it logs that the details were updated successfully and returns a response with the user's ID and a StatusOK status code.

Step 12. phoneVerificationForOtp.go

```
package main

import (
    "auth-microservice/config"
    "auth-microservice/model"
    userpb "auth-microservice/proto/user"
    "context"

    "go.uber.org/zap"
)

func (*UserService) PhoneVerification(ctx context.Context, request
*userpb.PhoneVerificationRequest) (*userpb.PhoneVerificationResponse, error) {
    logger.Info("Received PhoneVerification request", zap.String("phone", request.Phone))
    phone := request.Phone
    if !config.ValidatePhone(phone) {
        logger.Warn("Invalid phone number", zap.String("phone", phone))
        return &userpb.PhoneVerificationResponse{
            Data:       nil,
            Message:    "Invalid phone number. Phone number can only be 10 digits long",
            Error:      "Invalid Phone",
            StatusCode: StatusBadRequest,
        }, nil
    }
    // check if the phone number is already registered
    var existingUser model.User
    userNotFoundError := userDbConnector.Where("phone = ?",
phone).First(&existingUser).Error
    if userNotFoundError == nil {
        logger.Warn("Phone number already registered", zap.String("phone", phone))
        return &userpb.PhoneVerificationResponse{
```

```
            Data: &userpb.PhoneVerificationResponseData{
                Phone: phone,
            },
            Message:    "The phone number is already registered.",
            Error:      "Phone number already registered",
            StatusCode: StatusOK,
        }, nil
    }
    return &userpb.PhoneVerificationResponse{
        Data:       nil,
        Message:    "Phone number is not registered",
        Error:      "Not Found",
        StatusCode: StatusNotFound,
    }, nil
}
```

- A method of the UserService struct that verifies a phone number.
- The method PhoneVerification is defined with a context and a PhoneVerificationRequest as parameters. It returns PhoneVerificationResponse and an error.
- It logs that it received a PhoneVerification request.
- It retrieves the phone number from the request.
- It validates the phone number using the ValidatePhone function from the config package. If the phone number is invalid, it logs a warning and returns a response with a status code indicating a bad request.
- It initializes a User struct and attempts to find a user in the database where the phone number matches the one from the request. If it finds a user, it logs a warning and returns a response with a status code indicating success.
- If it doesn't find a user, it returns a response with a status code indicating not found.

Step 13. authenticateUser_test.go file.

```
// user_service_test.go
package main

import (
    "auth-microservice/model"
    userpb "auth-microservice/proto/user"
    "context"
    "testing"

    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/mock"
    "github.com/stretchr/testify/suite"
)

func (m *MockJWTManager) GenerateToken(user *model.User) (string, error) {
    args := m.Called(user)
    return args.String(0), args.Error(1)
}

// MockUserService is a mock implementation of the UserService interface.
type MockUserService struct {
    mock.Mock
}

func (m *MockUserService) AuthenticateUser(ctx context.Context, request
*userpb.AuthenticateUserRequest) (*userpb.AuthenticateUserResponse, error) {
    args := m.Called(ctx, request)
    return args.Get(0).(*userpb.AuthenticateUserResponse), args.Error(1)
}

type UserServiceTestSuite struct {
```

```
    suite.Suite
    userService *MockUserService
    jwtManager  *MockJWTManager
}

func (suite *UserServiceTestSuite) SetupTest() {
    // Initialize mocks
    suite.jwtManager = new(MockJWTManager)
    suite.userService = new(MockUserService)

    // Initialize UserService with the mock JWT manager

}

func (suite *UserServiceTestSuite) TestAuthenticateUser_Success() {
    request := &userpb.AuthenticateUserRequest{
        UserEmail:    "validuser@example.com",
        UserPassword: "correctpassword",
        Role:         model.UserRole,
    }

    expectedUser := &model.User{
        Name:     "Valid User",
        Email:    "validuser@example.com",
        Phone:    "9876543210",
        Role:     model.UserRole,
        Password: "hashedpassword", // Assume this is the hashed password
    }

    token := "mocked_token"

    // Set up expectations
```

```
    suite.jwtManager.On("GenerateToken", expectedUser).Return(token, nil)

    suite.userService.On("AuthenticateUser", context.Background(), request).Return(
        &userpb.AuthenticateUserResponse{
            StatusCode: 200,
            Message:    "User authenticated successfully",
            Data: &userpb.Responsedata{
                User: &userpb.User{
                    UserId:    "1",
                    UserName:  expectedUser.Name,
                    UserEmail: expectedUser.Email,
                    UserPhone: expectedUser.Phone,
                },
                Token: token,
            },
            Error: "",
        }, nil)

    // Call the AuthenticateUser function
    response, err := suite.userService.AuthenticateUser(context.Background(), request)

    // Assert the results
    assert.Nil(suite.T(), err)
    assert.Equal(suite.T(), int64(200), response.StatusCode)
    assert.Equal(suite.T(), "User authenticated successfully", response.Message)
    assert.NotNil(suite.T(), response.Data)
    assert.Equal(suite.T(), token, response.Data.Token)
}

func (suite *UserServiceTestSuite) TestAuthenticateUser_UserNotFound() {
    request := &userpb.AuthenticateUserRequest{
        UserEmail:    "nonexistentuser@example.com",
```

```
        UserPassword: "password",
        Role:         model.UserRole,
    }

    // Set up expectations
    suite.userService.On("AuthenticateUser", context.Background(), request).Return(
        &userpb.AuthenticateUserResponse{
            StatusCode: 404,
            Message:    "Authentication Failed, User not found OR Invalid role",
            Error:      "Not Found",
            Data:       nil,
        }, nil)

    // Call the AuthenticateUser function
    response, err := suite.userService.AuthenticateUser(context.Background(), request)

    // Assert the results
    assert.Nil(suite.T(), err)
    assert.Equal(suite.T(), int64(404), response.StatusCode)
    assert.Equal(suite.T(), "Authentication Failed, User not found OR Invalid role",
response.Message)
    assert.Nil(suite.T(), response.Data)
}

func TestUserServiceTestSuite(t *testing.T) {
    suite.Run(t, new(UserServiceTestSuite))
}
```

The unit tests for a UserService using the testify package in Go. It includes two test cases: TestAuthenticateUser_Success and TestAuthenticateUser_UserNotFound.

- MockJWTManager and MockUserService are mock structures that implement the methods GenerateToken and AuthenticateUser respectively. These methods return predefined results when called, which is useful for testing.

- UserServiceTestSuite is a test suite that includes the mock structures as fields. The SetupTest method is called before each test to initialize these fields.

- TestAuthenticateUser_Success is a test case that simulates successful user authentication. It sets up the mock methods to return successful results, calls the AuthenticateUser method, and then checks that the returned results are as expected.

- TestAuthenticateUser_UserNotFound is a test case that simulates a failed authentication due to a non-existent user. It sets up the mock AuthenticateUser method to return a failure result, calls the AuthenticateUser method, and then checks that the returned results are as expected.

- TestUserServiceTestSuite is the main test function that runs the test suite.

Step 13. addUser_test.go

```
// user_service_test.go
package main

import (
    "auth-microservice/model"
    userpb "auth-microservice/proto/user"
    "context"
    "testing"

    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/mock"
    "github.com/stretchr/testify/suite"
    "gorm.io/gorm"
)
```

```
// MockJWTManager is a mock implementation of the JWTManager interface.
type MockJWTManager struct {
    mock.Mock
}

// MockUserServiceAddUser is a mock implementation of the UserService interface.
type MockUserServiceAddUser struct {
    mock.Mock
    jwtManager *MockJWTManager
}

func (m *MockUserServiceAddUser) AddUser(ctx context.Context, request
*userpb.AddUserRequest) (*userpb.AddUserResponse, error) {
    args := m.Called(ctx, request)
    return args.Get(0).(*userpb.AddUserResponse), args.Error(1)
}

// MockDBConnector is a mock implementation of the GORM DB connector.
type MockDBConnector struct {
    mock.Mock
}

func (m *MockDBConnector) Where(query string, args ...interface{}) *gorm.DB {
    args = append([]interface{}{query}, args...)
    return m.Called(args...).Get(0).(*gorm.DB)
}

func (m *MockDBConnector) Create(value interface{}) *gorm.DB {
    return m.Called(value).Get(0).(*gorm.DB)
}

type UserServiceTestSuiteAddUser struct {
```

```
    suite.Suite
    userService *MockUserServiceAddUser
    jwtManager  *MockJWTManager
    dbConnector *MockDBConnector
}

func (suite *UserServiceTestSuiteAddUser) SetupTest() {
    // Initialize mocks
    suite.jwtManager = new(MockJWTManager)
    suite.dbConnector = new(MockDBConnector)
    suite.userService = &MockUserServiceAddUser{
        jwtManager: suite.jwtManager,
    }
}

func (suite *UserServiceTestSuiteAddUser) TestAddUser_Success() {
    request := &userpb.AddUserRequest{
        UserEmail:    "newuser@example.com",
        UserPassword: "validpassword",
        UserName:     "New User",
        UserPhone:    "1234567890",
        UserRole:     model.UserRole,
    }

    expectedUser := &model.User{
        Name:     "New User",
        Email:    "newuser@example.com",
        Phone:    "1234567890",
        Role:     model.UserRole,
        Password: "hashedpassword",
    }
```


```
    token := "mocked_token"

    suite.dbConnector.On("Where", "email = ?", request.UserEmail).Return(suite.dbConnector)
    suite.dbConnector.On("Create",
mock.AnythingOfType("*model.User")).Return(&gorm.DB{Error: nil})
    suite.jwtManager.On("GenerateToken", expectedUser).Return(token, nil)

    suite.userService.On("AddUser", context.Background(), request).Return(
        &userpb.AddUserResponse{
            StatusCode: 200,
            Message:    "User created successfully",
            Data: &userpb.Responsedata{
                User: &userpb.User{
                    UserId:    "1",
                    UserName:  expectedUser.Name,
                    UserEmail: expectedUser.Email,
                    UserPhone: expectedUser.Phone,
                },
                Token: token,
            },
            Error: "",
        }, nil)

    response, err := suite.userService.AddUser(context.Background(), request)
    assert.Nil(suite.T(), err)
    assert.Equal(suite.T(), int64(200), response.StatusCode)
    assert.Equal(suite.T(), "User created successfully", response.Message)
    assert.NotNil(suite.T(), response.Data)
    assert.Equal(suite.T(), token, response.Data.Token)
}

func (suite *UserServiceTestSuiteAddUser) TestAddUser_UserAlreadyExists() {
```

```
    request := &userpb.AddUserRequest{
        UserEmail:    "existinguser@example.com",
        UserPassword: "validpassword",
        UserName:     "Existing User",
        UserPhone:    "1234567890",
        UserRole:     model.UserRole,
    }

    suite.dbConnector.On("Where", "email = ?", request.UserEmail).Return(suite.dbConnector)
    suite.dbConnector.On("Create",
mock.AnythingOfType("*model.User")).Return(&gorm.DB{Error: gorm.ErrDuplicatedKey})

    suite.userService.On("AddUser", context.Background(), request).Return(
        &userpb.AddUserResponse{
            StatusCode: 409,
            Message:    "User Email is already registered",
            Error:      "Conflict",
            Data:       nil,
        }, nil)

    response, err := suite.userService.AddUser(context.Background(), request)
    assert.Nil(suite.T(), err)
    assert.Equal(suite.T(), int64(409), response.StatusCode)
    assert.Equal(suite.T(), "User Email is already registered", response.Message)
    assert.Nil(suite.T(), response.Data)
}

func (suite *UserServiceTestSuiteAddUser) TestAddUser_InvalidRole() {
    request := &userpb.AddUserRequest{
        UserEmail:    "validuser@example.com",
        UserPassword: "validpassword",
        UserName:     "Valid User",
```

```
        UserPhone:    "1234567890",
        UserRole:     "invalidrole",
    }

    suite.userService.On("AddUser", context.Background(), request).Return(
        &userpb.AddUserResponse{
            StatusCode: 400,
            Message:    "Invalid user role. User role can only be user or admin",
            Error:      "Invalid Role",
            Data:       nil,
        }, nil)

    response, err := suite.userService.AddUser(context.Background(), request)
    assert.Nil(suite.T(), err)
    assert.Equal(suite.T(), int64(400), response.StatusCode)
    assert.Equal(suite.T(), "Invalid user role. User role can only be user or admin",
response.Message)
    assert.Nil(suite.T(), response.Data)
}

func (suite *UserServiceTestSuiteAddUser) TestAddUser_InvalidFields() {
    request := &userpb.AddUserRequest{
        UserEmail:    "invalidemail",
        UserPassword: "short",
        UserName:     "",
        UserPhone:    "123",
        UserRole:     model.UserRole,
    }

    suite.userService.On("AddUser", context.Background(), request).Return(
        &userpb.AddUserResponse{
            StatusCode: 400,
```

```
            Message:    "The request contains missing or invalid fields. Make sure Phone number is
10 digits long.",
            Error:      "Invalid Request",
            Data:       nil,
        }, nil)

    response, err := suite.userService.AddUser(context.Background(), request)
    assert.Nil(suite.T(), err)
    assert.Equal(suite.T(), int64(400), response.StatusCode)
    assert.Equal(suite.T(), "The request contains missing or invalid fields. Make sure Phone
number is 10 digits long.", response.Message)
    assert.Nil(suite.T(), response.Data)
}

func (suite *UserServiceTestSuiteAddUser) TestAddUser_DatabaseError() {
    request := &userpb.AddUserRequest{
        UserEmail:    "newuser@example.com",
        UserPassword: "validpassword",
        UserName:     "New User",
        UserPhone:    "1234567890",
        UserRole:     model.UserRole,
    }

    suite.dbConnector.On("Where", "email = ?", request.UserEmail).Return(suite.dbConnector)
    suite.dbConnector.On("Create",
mock.AnythingOfType("*model.User")).Return(&gorm.DB{Error: gorm.ErrInvalidData})

    suite.userService.On("AddUser", context.Background(), request).Return(
        &userpb.AddUserResponse{
            StatusCode: 409,
            Message:    "The phone number is already registered.",
            Error:      "Conflict",
```

```
        Data:       nil,
    }, nil)

    response, err := suite.userService.AddUser(context.Background(), request)
    assert.Nil(suite.T(), err)
    assert.Equal(suite.T(), int64(409), response.StatusCode)
    assert.Equal(suite.T(), "The phone number is already registered.", response.Message)
    assert.Nil(suite.T(), response.Data)
}

func (suite *UserServiceTestSuiteAddUser) TestAddUser_TokenGenerationError() {
    request := &userpb.AddUserRequest{
        UserEmail:    "newuser@example.com",
        UserPassword: "validpassword",
        UserName:     "New User",
        UserPhone:    "1234567890",
        UserRole:     model.UserRole,
    }

    expectedUser := &model.User{
        Name:     "New User",
        Email:    "newuser@example.com",
        Phone:    "1234567890",
        Role:     model.UserRole,
        Password: "hashedpassword",
    }

    suite.dbConnector.On("Where", "email = ?", request.UserEmail).Return(suite.dbConnector)
    suite.dbConnector.On("Create",
mock.AnythingOfType("*model.User")).Return(&gorm.DB{Error: nil})
    suite.jwtManager.On("GenerateToken", expectedUser).Return("", gorm.ErrInvalidData)
```

```
    suite.userService.On("AddUser", context.Background(), request).Return(
        &userpb.AddUserResponse{
            StatusCode: 500,
            Message:    "Security Issues, Please try again later.",
            Error:      "Internal Server Error",
            Data:       nil,
        }, nil)

    response, err := suite.userService.AddUser(context.Background(), request)
    assert.Nil(suite.T(), err)
    assert.Equal(suite.T(), int64(500), response.StatusCode)
    assert.Equal(suite.T(), "Security Issues, Please try again later.", response.Message)
    assert.Nil(suite.T(), response.Data)
}

func TestUserServiceTestSuiteAddUserAdddUser(t *testing.T) {
    suite.Run(t, new(UserServiceTestSuiteAddUser))
}
```

User service in an authentication microservice. It uses the testify package for assertions and mocking.

- A test suite UserServiceTestSuiteAddUser with several test cases for the AddUser method of the user service. Each test case sets up specific expectations on the mock objects, calls the AddUser method, and then asserts the expected results.
- TestAddUser_Success: Tests the successful creation of a new user. It sets up the mocks to return a successful response and asserts that the response is as expected.
- TestAddUser_UserAlreadyExists: Tests the case where the user already exists. It sets up the mocks to return a duplicate key error and asserts that the response status code is 409 (Conflict).

- TestAddUser_InvalidRole: Tests the case where an invalid role is provided. It sets up the mocks to return a bad request response and asserts that the response status code is 400 (Bad Request).
- TestAddUser_InvalidFields: Tests the case where invalid fields are provided in the request. It sets up the mocks to return a bad request response and asserts that the response status code is 400 (Bad Request).
- TestAddUser_DatabaseError: Tests the case where a database error occurs. It sets up the mocks to return a database error and asserts that the response status code is 409 (Conflict).
- TestAddUser_TokenGenerationError: Tests the case where there's an error generating the token. It sets up the mocks to return a token generation error and asserts that the response status code is 500 (Internal Server Error).
- The TestUserServiceTestSuiteAddUserAdddUser function at the end runs the test suite.

Step 14. Dockerfile

```
# Base image
FROM golang:1.22.2-alpine3.19

# Move to working directory /app
WORKDIR /app

RUN apk add --no-cache bash

# Copy the code into the container
COPY . .

# Download dependencies using go mod
RUN go mod tidy && go mod vendor

# Expose PORT 8090 to the outside world
EXPOSE 8090
```

```
# Command to run the application when starting the container
CMD ["go", "run", "."]
```

1. **FROM golang:1.22.2-alpine3.19:** This line specifies the base image to be used. In this case, it's an image with Go version 1.22.2 and Alpine Linux version 3.19.

2. **WORKDIR /app**: This line sets the working directory inside the Docker container to /app.

3. **COPY . .** : This line copies the current directory (where the Dockerfile is located) on your local machine into the current directory inside the Docker container (which is /app due to the previous WORKDIR command).

4. **RUN go mod tidy && go mod vendor:** This line runs two commands inside the Docker container. go mod tidy ensures that the go.mod file matches the source code in the module. It adds missing modules necessary to build the current module's packages and dependencies, and it removes unused modules. go mod vendor creates a vendor directory in the current module and populates it with copies of all packages needed to support builds and tests of packages in the current module.

5. **EXPOSE 8090:** This line tells Docker that the container listens on the specified network port at runtime. In this case, it's port 8090.
6. **CMD ["go","run","."]:** This line provides defaults for an executing container. In this case, it runs the command go run . which compiles and runs the Go application. The . means it will run the main package in the current directory.

7. Docker Desktop Installation >> Link
    a. Run **docker version** command to check if it is completely installed.

Terminal Commands: -

Go to auth-microservice: - **cd auth-micro**
Build the docker file: - **docker build -t auth-microservice.**
Run the Container: - **docker container run -p 8090:8090 -e MYSQL_HOST=host.docker.internal auth-microservice**

2. Restaurant Microservice

Step 1. Install Dependencies.

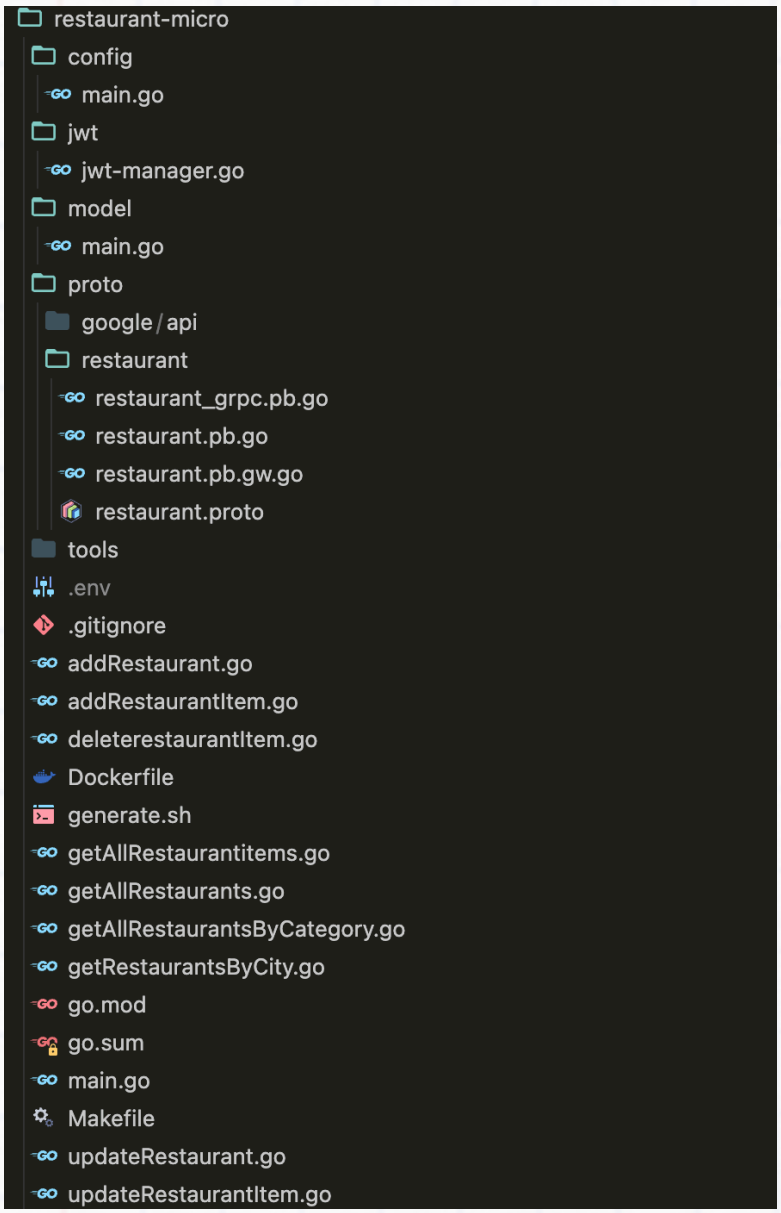
Dependencies are already installed, you can go to generate.sh file to check, what all dependencies are added.

Step 2. Compile restaurant.proto file

1. Go to the folder restaurant-micro and open VSC new integrated terminal.
2. Run the command to compile restaurant.proto file

```
cd restaurant-micro
make proto-generate
```

Step 3. The Directory Structure of the Restaurant microservice will look like this.



Step 4: main.go/ model

This file has three data models (Restaurant, Address, and Restaurant Item) using the GORM (Go Object-Relational Mapper) library for handling database operations.

Restaurant struct: Represents a restaurant with fields like name, phone, availability, rating, owner's email, image URL, operation days and hours, minimum order amount, and discount percentage.

Address struct: Represents an address with fields like street name, pincode, city, and country. It has a foreign key RestaurantId that references the ID field in the Restaurant struct, establishing a relationship between a restaurant and its address.

RestaurantItem struct: Represents a restaurant item with fields like item name, item price, image URL, category, cuisine type, and a boolean indicating if the item is vegetarian. It also has a foreign key RestaurantId that references the ID field in the Restaurant struct, establishing a relationship between a restaurant and its items.

The gorm.Model embedded in each struct includes fields ID, CreatedAt, UpdatedAt, DeletedAt for handling basic CRUD operations.

The struct field tags like gorm:"unique" and gorm:"default:open" are GORM specific tags that define constraints or default values for the fields in the database.

Step 5. Create a .env file for environment variables.

```
SECRET_KEY="SECRET_KEY_JWT_TOKEN"
MYSQL_USER="root"
MYSQL_HOST="127.0.0.1"
MYSQL_PASSWORD="<password>"
MYSQL_DATABASE="test"
MYSQL_PORT="3306"
```

PS: - Make sure to replace **<password>** with the password entered during DB installation.

Step 6. main.go/config

1. DatabaseDsn(): This function constructs a Data Source Name (DSN) for connecting to a MySQL database. It uses environment variables for the database user, password, host, port, and database name.

2. GoDotEnvVariable(key string): This function loads a .env file using the godotenv package and returns the value of the environment variable specified by key.

3. ConnectDB(): This function connects to a MySQL database using the DSN from DatabaseDsn(). It returns three separate connections, each of which auto-migrates a different model (Restaurant, RestaurantItem, Address). This is likely a design mistake, as typically a single DB connection is shared.

4. ValidateRestaurantFields(...), ValidateRestaurantItemFields(...), ValidateRestaurantPhone(restaurantPhone string), and ValidateAddressFields(...): These functions validate various fields related to restaurants, restaurant items, phone numbers, and addresses. They return false if any field is invalid (e.g., empty strings, negative numbers, phone numbers not exactly 10 digits long), and true otherwise.

Step 7: main.go file [gRPC Server Logic]

1. This Go file is the main entry point for a restaurant microservice application. It uses gRPC for internal communication and gRPC-Gateway to expose a RESTful API to the outside world. Here's a breakdown:

2. init(): This function initializes a logger using the zap package. If there's an error, it panics.

3. startServer(): This function does the heavy lifting:

    a. It loads environment variables from a .env file.

    b. It connects to the database using the config.ConnectDB() function and stores the connections in global variables.
    c. It starts a gRPC server on localhost:50052 and registers the RestaurantService with it. The server runs in a separate goroutine.

    d. It creates a gRPC-Gateway server that connects to the gRPC server. It registers the RestaurantService with the gateway server and wraps the server's handler in a CORS handler.

e. It starts the gRPC-Gateway server on 0.0.0.0:8091.

4. main(): This function simply calls startServer().

5. The RestaurantService struct is empty and simply embeds the UnimplementedRestaurantServiceServer struct from the restaurantpb package. This suggests that the actual implementation of the service is in another file.

6. The jwt.UnaryInterceptor is used as a unary interceptor for the gRPC server, suggesting that the server uses JWT for authentication.

7. The handlers.CORS function is used to enable CORS for the gRPC-Gateway server. It allows requests from http://localhost:3000 and supports the GET, POST, PUT, DELETE, and OPTIONS methods. It allows the Content-Type and Authorization headers.

Step 8. addRestaurant.go file

```
package main

import (
    "context"
    "restaurant-micro/config"
    "restaurant-micro/model"
    restaurantpb "restaurant-micro/proto/restaurant"
    "strconv"

    "go.uber.org/zap"
)

func (*RestaurantService) AddRestaurant(ctx context.Context, request
*restaurantpb.AddRestaurantRequest) (*restaurantpb.AddRestaurantResponse, error) {
    logger.Info("Received AddRestaurant request")

    userEmail, emailCtxError := ctx.Value("userEmail").(string)
    userRole, roleCtxError := ctx.Value("userRole").(string)
```

```
    if !emailCtxError || !roleCtxError {
        logger.Error("Failed to get user email or role from context")
        return &restaurantpb.AddRestaurantResponse{
            Message:    "Failed to get user mail from context",
            Error:      "Internal Server Error",
            StatusCode: StatusInternalServerError,
        }, nil
    }
    logger.Info("Context values retrieved", zap.String("userEmail", userEmail),
zap.String("userRole", userRole))
    if userRole != model.AdminRole {
        logger.Warn("Permission denied", zap.String("userRole", userRole))
        return &restaurantpb.AddRestaurantResponse{
            Data:       nil,
            Message:    "You do not have permission to perform this action. Only admin can add a
restaurant",
            StatusCode: StatusForbidden,
            Error:      "Forbidden",
        }, nil
    }

    var restaurantAddress model.Address
    var restaurant model.Restaurant
    if request.Restaurant != nil && request.Restaurant.RestaurantAddress != nil {
        restaurantAddress.City = request.Restaurant.RestaurantAddress.City
        restaurantAddress.Country = request.Restaurant.RestaurantAddress.Country
        restaurantAddress.Pincode = request.Restaurant.RestaurantAddress.Pincode
        restaurantAddress.StreetName = request.Restaurant.RestaurantAddress.StreetName
    } else {
        logger.Warn("Invalid restaurant address data provided")
        return &restaurantpb.AddRestaurantResponse{
```

```
            Data:       nil,
            Message:    "Invalid restaurant address data provided. Some fields might be missing or
invalid",
            StatusCode: StatusBadRequest,
            Error:      "Bad Request",
        }, nil
    }
    restaurant.Name = request.Restaurant.RestaurantName
    restaurant.Availability = request.Restaurant.RestaurantAvailability
    restaurant.Phone = request.Restaurant.RestaurantPhoneNumber
    restaurant.Rating = request.Restaurant.RestaurantRating
    restaurant.ImageUrl = request.Restaurant.RestaurantImageUrl
    restaurant.OperationDays = request.Restaurant.RestaurantOperationDays
    restaurant.OperationHours = request.Restaurant.RestaurantOperationHours
    restaurant.RestaurantOwnerMail = userEmail
    restaurant.RestaurantMinimumOrderAmount =
request.Restaurant.RestaurantMinimumOrderAmount
    restaurant.RestaurantDiscountPercentage =
request.Restaurant.RestaurantDiscountPercentage

    logger.Info("Restaurant data populated", zap.String("restaurantName", restaurant.Name))

    if !config.ValidateRestaurantFields(restaurant.Name, restaurantAddress,
        restaurant.Phone, restaurant.Availability,
        restaurant.ImageUrl, restaurant.OperationDays,
        restaurant.OperationHours, restaurant.Rating,
restaurant.RestaurantMinimumOrderAmount,
        restaurant.RestaurantDiscountPercentage) {
        logger.Warn("Invalid restaurant data provided", zap.String("restaurantName",
restaurant.Name))

        return &restaurantpb.AddRestaurantResponse{
```

```
            Data:       nil,
            Message:    "Invalid restaurant data provided. Some fields might be missing or invalid",
            StatusCode: StatusBadRequest,
            Error:      "Bad Request",
        }, nil
    }

    if !config.ValidateRestaurantPhone(restaurant.Phone) {
        logger.Warn("Invalid phone number format", zap.String("phone", restaurant.Phone))
        return &restaurantpb.AddRestaurantResponse{
            Data:       nil,
            Message:    "Invalid phone number format",
            StatusCode: StatusBadRequest,
            Error:      "Bad Request",
        }, nil
    }
    var existingRestaurant model.Restaurant
    restaurantNotFoundErr := restaurantDBConnector.Where("name = ?",
restaurant.Name).First(&existingRestaurant).Error
    if restaurantNotFoundErr == nil {
        return &restaurantpb.AddRestaurantResponse{
            Data:       nil,
            Message:    "Same name restaurant exists. Please check the restaurant name and try
again.",
            StatusCode: StatusConflict,
            Error:      "Restaurant creation failed",
        }, nil
    }
    primaryKey := restaurantDBConnector.Create(&restaurant)
    if primaryKey.Error != nil {
        logger.Error("[ AddRestaurant ] Failed to add restaurant", zap.Error(primaryKey.Error))
        return &restaurantpb.AddRestaurantResponse{
```

```
            Data:       nil,
            Message:    "Failed to add restaurant",
            StatusCode: StatusConflict,
            Error:      "The provided phone number is already associated with an account",
        }, nil
    }
    restaurantAddress.RestaurantId = restaurant.ID
    err := restaurantAddressDBConnector.Create(&restaurantAddress)
    if err.Error != nil {
        logger.Error("[ AddRestaurant ] Failed to add restaurant address", zap.Error(err.Error))
        return &restaurantpb.AddRestaurantResponse{
            Data:       nil,
            Message:    "Failed to add restaurant address",
            StatusCode: StatusInternalServerError,
            Error:      err.Error.Error(),
        }, nil
    }
    logger.Info("Restaurant added successfully", zap.String("restaurantId",
strconv.FormatUint(uint64(restaurant.ID), 10)))
    RestaurantResponse := request.Restaurant
    RestaurantResponse.RestaurantId = strconv.FormatUint(uint64(restaurant.ID), 10)

    return &restaurantpb.AddRestaurantResponse{
        Data: &restaurantpb.AddRestaurantResponseData{
            Restaurant: RestaurantResponse,
        },
        Message:    "Restaurant added successfully",
        StatusCode: StatusOK,
        Error:      "",
    }, nil
}
```

Step 10. addRestaurantItem.go file

```
package main

import (
    "context"
    "restaurant-micro/config"
    "restaurant-micro/model"
    restaurantpb "restaurant-micro/proto/restaurant"
    "strconv"

    "go.uber.org/zap"
)

func (*RestaurantService) AddRestaurantItem(ctx context.Context, request
*restaurantpb.AddRestaurantItemRequest) (*restaurantpb.AddRestaurantItemResponse, error)
{
    logger.Info("Received AddRestaurantItem request")

    userEmail, emailCtxError := ctx.Value("userEmail").(string)
    userRole, roleCtxError := ctx.Value("userRole").(string)

    if !emailCtxError || !roleCtxError {
        logger.Error("Failed to get user email or role from context")
        return &restaurantpb.AddRestaurantItemResponse{
            Data:       nil,
            Message:    "Failed to get user mail from context",
            Error:      "Internal Server Error",
            StatusCode: StatusInternalServerError,
        }, nil
    }

    logger.Info("Context values retrieved", zap.String("userEmail", userEmail),
```

```
        zap.String("userRole", userRole))

    if userRole != model.AdminRole {
        logger.Warn("Permission denied", zap.String("userRole", userRole))
        return &restaurantpb.AddRestaurantItemResponse{
            Data:       nil,
            Message:    "You do not have permission to perform this action. Only admin can add a
restaurant item",
            StatusCode: StatusForbidden,
            Error:      "Forbidden",
        }, nil
    }
    if request.RestaurantItem == nil ||
        !config.ValidateRestaurantItemFields(request.RestaurantItem.RestaurantItemName,
            request.RestaurantItem.RestaurantItemImageUrl,
request.RestaurantItem.RestaurantItemPrice,
            request.RestaurantItem.RestaurantItemCategory,
            request.RestaurantItem.RestaurantItemCuisineType,
            request.RestaurantItem.RestaurantId) {
        logger.Warn("Invalid restaurant item data provided")
        return &restaurantpb.AddRestaurantItemResponse{
            Data:       nil,
            Message:    "Invalid restaurant item data provided.",
            StatusCode: StatusBadRequest,
            Error:      "Bad Request",
        }, nil
    }
    var restaurantItem model.RestaurantItem
    restaurantItem.ItemPrice = request.RestaurantItem.RestaurantItemPrice
    restaurantItem.ImageUrl = request.RestaurantItem.RestaurantItemImageUrl
    restaurantItem.ItemName = request.RestaurantItem.RestaurantItemName
    restaurantItem.Category = request.RestaurantItem.RestaurantItemCategory
```

```
    restaurantItem.CuisineType = request.RestaurantItem.RestaurantItemCuisineType
    restaurantItem.Veg = request.RestaurantItem.RestaurantItemVeg

    // fetch restaurant from restaurantDB
    var restaurant model.Restaurant
    restaurantID, err := strconv.ParseUint(request.RestaurantItem.RestaurantId, 10, 64)
    if err != nil {
        logger.Error("Failed to convert restaurant ID to uint", zap.Error(err))
        return nil, err
    }

    primaryKey := restaurantDBConnector.Where("id = ?", uint(restaurantID)).First(&restaurant)
    // check if the restaurant is exist or not.
    if primaryKey.Error != nil || restaurant.RestaurantOwnerMail != userEmail {
        logger.Warn("Restaurant does not exist or you are not authorized to modify this
restaurant's data",
            zap.String("restaurantName", request.RestaurantItem.GetRestaurantId()))
        return &restaurantpb.AddRestaurantItemResponse{
            Data:       nil,
            Message:    "You are not authorized to modify this restaurant's data Or Restaurant does
not exist",
            StatusCode: StatusForbidden,
            Error:      "Forbidden",
        }, nil
    }
    restaurantItem.RestaurantId = restaurant.ID
    // Create a new restaurant item in the database and return the primary key if successful or an
error if it fails
    result := restaurantItemDBConnector.Create(&restaurantItem)
    // Check if there is an error while creating the restaurant item
    if result.Error != nil {
```

```
        logger.Error("Failed to create restaurant item", zap.String("restaurantItemName",
request.RestaurantItem.RestaurantItemName), zap.Error(result.Error))
        return &restaurantpb.AddRestaurantItemResponse{
            Message:    "A food item with similar details might already exist on this restaurant's
menu.",
            StatusCode: StatusConflict,
            Error:      "Food item creation failed",
        }, nil
    }
    logger.Info("Restaurant item added successfully", zap.String("restaurantItemName",
request.RestaurantItem.RestaurantItemName))
    // Return a success message if the restaurant item is created successfully
    restaurantItemResponse := addRestaurantItemResponseData(restaurant.ID,
restaurantItem.ID, request.RestaurantItem)
    return &restaurantpb.AddRestaurantItemResponse{
        Data: &restaurantpb.AddRestaurantItemResponseData{
            RestaurantItem: restaurantItemResponse,
        },
        Message:    "Restaurant item added successfully",
        StatusCode: StatusCreated,
        Error:      "",
    }, nil
}

func addRestaurantItemResponseData(restaurantId uint, restaurantItemId uint,
restaurantItemData *restaurantpb.RestaurantItem) *restaurantpb.RestaurantItem {
    var restaurantItemResponse = &restaurantpb.RestaurantItem{
        RestaurantItemId:          strconv.FormatUint(uint64(restaurantItemId), 10),
        RestaurantItemName:        restaurantItemData.RestaurantItemName,
        RestaurantItemPrice:       restaurantItemData.RestaurantItemPrice,
        RestaurantItemImageUrl:    restaurantItemData.RestaurantItemImageUrl,
        RestaurantItemCategory:    restaurantItemData.RestaurantItemCategory,
```

```
        RestaurantItemCuisineType: restaurantItemData.RestaurantItemCuisineType,
        RestaurantItemVeg:         restaurantItemData.RestaurantItemVeg,
        RestaurantId:              strconv.FormatUint(uint64(restaurantId), 10),
    }
    return restaurantItemResponse
}
```

1. Logs the receipt of a new AddRestaurantItem request.
2. Extracts the user's email and role from the context. If it fails to do so, it logs an error and returns a response indicating an internal server error.
3. Checks if the user's role is AdminRole. If not, it logs a warning and returns a response indicating that the user does not have permission to perform this action.
4. Validates the fields of the restaurant item in the request. If any field is invalid, it logs a warning and returns a response indicating a bad request.
5. Creates a RestaurantItem model from the request data.
6. Fetches the restaurant from the database using the restaurant name from the request. If the restaurant does not exist or the user is not authorized to modify the restaurant's data, it logs a warning and returns a response indicating a forbidden action.
7. Sets the RestaurantId of the RestaurantItem to the ID of the fetched restaurant.
8. Attempts to create a new restaurant item in the database. If there's an error (for example, if a restaurant item with similar details already exists), it logs an error and returns a response indicating a conflict.
9. If the restaurant item is created successfully, it logs this success, creates a response data object, and returns a response indicating success.

The addRestaurantItemResponseData function is a helper function that creates a RestaurantItem protobuf message from the provided data. This message is included in the successful response from AddRestaurantItem.

Step 11. getAllRestaurantItem.go file

```
package main

import (
    "context"
    "restaurant-micro/model"
    restaurantpb "restaurant-micro/proto/restaurant"
    "strconv"

    "go.uber.org/zap"
)

func (*RestaurantService) GetAllRestaurantItems(ctx context.Context, request
*restaurantpb.GetAllRestaurantItemsRequest) (*restaurantpb.GetAllRestaurantItemsResponse,
error) {
    logger.Info("Received GetAllRestaurantItems request",
    zap.String("restaurantId", request.RestaurantId))

    // validate the restaurant id
    if request.RestaurantId == "" {
        logger.Warn("Invalid restaurant id provided")
        return &restaurantpb.GetAllRestaurantItemsResponse{
            Data: &restaurantpb.GetAllRestaurantItemsResponseData{
                TotalRestaurantItems: 0,
                RestaurantItems:      nil,
            },
            Message:    "Invalid restaurant id provided",
            StatusCode: StatusBadRequest,
            Error:      "Bad Request",
        }, nil
    }
    // fetch restaurant from restaurantDB.
```

```
    var restaurant model.Restaurant
    primaryKeyRes := restaurantDBConnector.Where("id = ?",
request.RestaurantId).First(&restaurant)

    // check if the restaurant is exist or not.
    if primaryKeyRes.Error != nil {
        logger.Warn("Restaurant does not exist", zap.String("restaurantId", request.RestaurantId))
        return &restaurantpb.GetAllRestaurantItemsResponse{
            Data: &restaurantpb.GetAllRestaurantItemsResponseData{
                TotalRestaurantItems: 0,
                RestaurantItems:      nil,
            },
            Message:    "Restaurant Does not exist",
            StatusCode: StatusBadRequest,
            Error:      "Bad Request",
        }, nil
    }
    var restaurantItems []model.RestaurantItem
    err := restaurantItemDBConnector.Where("restaurant_id = ?",
restaurant.ID).Find(&restaurantItems)
    if err.Error != nil {
        logger.Error("Failed to get restaurant items", zap.String("restaurantId",
request.RestaurantId), zap.Error(err.Error))
        return &restaurantpb.GetAllRestaurantItemsResponse{
            Data: &restaurantpb.GetAllRestaurantItemsResponseData{
                TotalRestaurantItems: 0,
                RestaurantItems:      nil,
            },
            Message:    "Failed to get restaurant items",
            StatusCode: StatusInternalServerError,
            Error:      "Internal Server Error",
        }, nil
```

```
    }
    restaurantItemsResponse := []*restaurantpb.RestaurantItem{}
    for _, restaurantItem := range restaurantItems {
        restaurantItemsResponse = append(restaurantItemsResponse,
&restaurantpb.RestaurantItem{
            RestaurantItemId:          strconv.FormatUint(uint64(restaurantItem.ID), 10),
            RestaurantItemName:        restaurantItem.ItemName,
            RestaurantItemPrice:       restaurantItem.ItemPrice,
            RestaurantItemImageUrl:    restaurantItem.ImageUrl,
            RestaurantItemCategory:    restaurantItem.Category,
            RestaurantItemCuisineType: restaurantItem.CuisineType,
            RestaurantItemVeg:         restaurantItem.Veg,
            RestaurantId:              request.RestaurantId,
        })
    }
    logger.Info("Restaurant items fetched successfully", zap.Int("totalItems",
len(restaurantItems)))
    return &restaurantpb.GetAllRestaurantItemsResponse{
        Data: &restaurantpb.GetAllRestaurantItemsResponseData{
            TotalRestaurantItems: int64(len(restaurantItems)),
            RestaurantItems:      restaurantItemsResponse,
        },
        Message:    "Restaurant items fetched successfully",
        StatusCode: StatusOK,
        Error:      "",
    }, nil
}
```

1. This file defines a method GetAllRestaurantItems on the RestaurantService type. This method is responsible for fetching all items of a specific restaurant's menu.

2. Logs the receipt of a new GetAllRestaurantItems request.

3. Validates the restaurant id from the request. If it's empty, it logs a warning and returns a response indicating a bad request.

4. Fetches the restaurant from the database using the restaurant id from the request. If the restaurant does not exist, it logs a warning and returns a response indicating a bad request.

5. Fetches all restaurant items from the database that are associated with the fetched restaurant. If there's an error, it logs an error and returns a response indicating an internal server error.

6. If the restaurant items are fetched successfully, it creates a slice of RestaurantItem protobuf messages from the fetched data.

7. Logs the successful fetching of restaurant items and the total number of items fetched.

8. Returns a response indicating success and containing the fetched restaurant items.

9. The RestaurantItem protobuf message is used to represent a restaurant item in the response. It includes fields for the item's id, name, price, image URL, category, cuisine type, whether it's vegetarian, and the id of the restaurant it belongs to.

Step 12. getAllRestaurants.go file

```
package main

import (
    "context"
    "restaurant-micro/model"
    restaurantpb "restaurant-micro/proto/restaurant"
    "strconv"
```

```
    "go.uber.org/zap"
)

func (*RestaurantService) GetAllRestaurants(ctx context.Context, request
*restaurantpb.GetAllRestaurantsRequest) (*restaurantpb.GetAllRestaurantsResponse, error) {
    logger.Info("Received GetAllRestaurants request")

    // get the user mail from the context
    userEmail, ok := ctx.Value("userEmail").(string)
    if !ok {
        logger.Error("Failed to get user email from context")
        return &restaurantpb.GetAllRestaurantsResponse{
            Data: &restaurantpb.GetAllRestaurantsResponseData{
                TotalRestaurants: 0,
                Restaurants:      nil,
            },
            Message:    "Failed to get user email from context",
            StatusCode: StatusInternalServerError,
            Error:      "Internal Server Error",
        }, nil
    }

    var restaurants []model.Restaurant
    err := restaurantDBConnector.Where("restaurant_owner_mail = ?",
userEmail).Find(&restaurants).Error

    if err != nil {
        logger.Error("Failed to get restaurants from the database", zap.Error(err))
        return &restaurantpb.GetAllRestaurantsResponse{
            Data: &restaurantpb.GetAllRestaurantsResponseData{
                TotalRestaurants: 0,
```

```
            Restaurants:      nil,
        },
        Message:    "Failed to get the restaurants to the database. Please try again later.",
        StatusCode: StatusInternalServerError,
        Error:      "Internal Server Error",
    }, nil
}
restaurantsResponse := []*restaurantpb.Restaurant{}
totalRestaurants := 0
for _, restaurant := range restaurants {
    totalRestaurants++
    // fetch all address of the restaurant from the database.
    var restaurantAddress model.Address
    restaurantAddressErr := restaurantAddressDBConnector.Where("restaurant_id = ?",
restaurant.ID).First(&restaurantAddress).Error
    if restaurantAddressErr != nil {
        logger.Error("Failed to get restaurant address from the database",
            zap.Error(restaurantAddressErr))

        return &restaurantpb.GetAllRestaurantsResponse{
            Data: &restaurantpb.GetAllRestaurantsResponseData{
                TotalRestaurants: 0,
                Restaurants:      nil,
            },
            Message:    "Failed to get the restaurant addresses to the database. Please try again
later.",
            StatusCode: StatusInternalServerError,
            Error:      "Internal Server Error",
        }, nil
    }
    restaurantsResponse = append(restaurantsResponse, &restaurantpb.Restaurant{
        RestaurantId:          strconv.FormatUint(uint64(restaurant.ID), 10),
```

```
            RestaurantName:           restaurant.Name,
            RestaurantAvailability:   restaurant.Availability,
            RestaurantRating:         restaurant.Rating,
            RestaurantImageUrl:       restaurant.ImageUrl,
            RestaurantPhoneNumber:    restaurant.Phone,
            RestaurantOperationDays:  restaurant.OperationDays,
            RestaurantOperationHours: restaurant.OperationHours,
            RestaurantOwnerMail:      restaurant.RestaurantOwnerMail,
            RestaurantMinimumOrderAmount: restaurant.RestaurantMinimumOrderAmount,
            RestaurantDiscountPercentage:  restaurant.RestaurantDiscountPercentage,

            RestaurantAddress: &restaurantpb.Address{
                City:       restaurantAddress.City,
                Country:    restaurantAddress.Country,
                Pincode:    restaurantAddress.Pincode,
                StreetName: restaurantAddress.StreetName,
            },
        })
    }
    logger.Info("Restaurants fetched successfully", zap.Int("totalRestaurants", totalRestaurants))

    return &restaurantpb.GetAllRestaurantsResponse{
        Data: &restaurantpb.GetAllRestaurantsResponseData{
            TotalRestaurants: int64(totalRestaurants),
            Restaurants:      restaurantsResponse,
        },
        Message:    "Restaurants fetched successfully",
        StatusCode: StatusOK,
        Error:      "",
    }, nil
}
```

- A method of the RestaurantService struct that fetches all restaurants owned by a user. Here's a step-by-step explanation:
- The method GetAllRestaurants is defined with a context and a request as parameters. It returns a response and an error.
- It logs that it received a GetAllRestaurants request.
- It attempts to retrieve the user's email from the context. If it fails, it logs an error and returns a response with a status code indicating an internal server error.
- It initializes a slice of model.Restaurant and attempts to find all restaurants in the database where the restaurant_owner_mail matches the user's email. If it encounters an error, it logs the error and returns a response with a status code indicating an internal server error.
- It initializes a slice of restaurantpb.Restaurant and a counter for the total number of restaurants.
- It iterates over the restaurants retrieved from the database. For each restaurant, it increments the counter and fetches the restaurant's address from the database. If it encounters an error, it logs the error and returns a response with a status code indicating an internal server error.
- It appends a new restaurantpb.Restaurant to the restaurantsResponse slice, populating its fields with the data from the current restaurant and its address.
- After iterating over all restaurants, it logs the total number of restaurants fetched.
- It returns a response containing the total number of restaurants and the restaurantsResponse slice, with a status code indicating success.

Step 13. updateRestaurant.go file

```
package main

import (
    "context"
    "restaurant-micro/config"
    "restaurant-micro/model"
```

```
    restaurantpb "restaurant-micro/proto/restaurant"

    "go.uber.org/zap"
)

func (*RestaurantService) UpdateRestaurant(ctx context.Context, request
*restaurantpb.UpdateRestaurantRequest) (*restaurantpb.UpdateRestaurantResponse, error) {
    logger.Info("Received UpdateRestaurant request",
        zap.Any("Request", request))

    userEmail, ok := ctx.Value("userEmail").(string)
    if !ok {
        logger.Error("Failed to get user email from context")

        return &restaurantpb.UpdateRestaurantResponse{
            Data:       nil,
            Message:    "",
            StatusCode: StatusInternalServerError,
            Error:      "Internal Server Error",
        }, nil
    }
    logger.Info("Fetched user email from context", zap.String("userEmail", userEmail))
    var restaurant model.Restaurant
    var restaurantAddress model.Address

    if request.Restaurant == nil || request.Restaurant.GetRestaurantAddress() == nil {
        logger.Warn("Invalid restaurant data provided")
        return &restaurantpb.UpdateRestaurantResponse{
            Message:    "Invalid restaurant data provided",
            StatusCode: StatusBadRequest,
            Error:      "Bad Request",
        }, nil
```

```
    } else {
        restaurantAddress.City = request.Restaurant.RestaurantAddress.City
        restaurantAddress.Country = request.Restaurant.RestaurantAddress.Country
        restaurantAddress.Pincode = request.Restaurant.RestaurantAddress.Pincode
        restaurantAddress.StreetName = request.Restaurant.RestaurantAddress.StreetName
    }
    if !config.ValidateRestaurantFields(
        request.Restaurant.RestaurantName, restaurantAddress,
        request.Restaurant.RestaurantPhoneNumber,
        request.Restaurant.RestaurantAvailability, request.Restaurant.RestaurantImageUrl,
        request.Restaurant.RestaurantOperationHours,
        request.Restaurant.RestaurantOperationDays,
        request.Restaurant.RestaurantRating,
        request.Restaurant.RestaurantMinimumOrderAmount,
        request.Restaurant.RestaurantDiscountPercentage) ||
        !config.ValidateRestaurantPhone(request.Restaurant.RestaurantPhoneNumber) {

        logger.Warn("Invalid restaurant fields", zap.Any("Restaurant", request.Restaurant))
        return &restaurantpb.UpdateRestaurantResponse{
            Message:    "Invalid restaurant data provided.Some fields might be missing, empty or
invalid, and make sure phone number contains only numbers and is 10 digits long",
            StatusCode: StatusBadRequest,
            Error:      "Invalid restaurant fields",
        }, nil
    }
    // fetch restaurant from restaurantDB
    primaryKeyRes := restaurantDBConnector.Where("id = ?",
request.Restaurant.RestaurantId).First(&restaurant)
    // check if the restaurant is exist or not
    if primaryKeyRes.Error != nil || restaurant.RestaurantOwnerMail != userEmail {
        logger.Warn("Unauthorized update attempt", zap.String("RestaurantOwnerMail",
            restaurant.RestaurantOwnerMail),
```

```
            zap.String("UserEmail", userEmail))
        return &restaurantpb.UpdateRestaurantResponse{
            Data:       nil,
            Message:    "You do not have permission to perform this action. Only restaurant owner
can update the restaurant",
            StatusCode: StatusUnauthorized,
            Error:      "Unauthorized",
        }, nil
    }
    err := restaurantAddressDBConnector.Where("restaurant_id = ?",
restaurant.ID).First(&restaurantAddress)
    if err.Error != nil {
        logger.Error("Failed to get restaurant address from the database", zap.Error(err.Error))
        return &restaurantpb.UpdateRestaurantResponse{
            Data:       nil,
            Message:    "Failed to get restaurant address from the database",
            StatusCode: StatusInternalServerError,
            Error:      "Internal Server Error",
        }, nil
    }
    request.Restaurant.RestaurantOwnerMail = userEmail
    restaurant.Name = request.Restaurant.RestaurantName
    restaurant.Availability = request.Restaurant.RestaurantAvailability
    restaurant.Phone = request.Restaurant.RestaurantPhoneNumber
    restaurant.Rating = request.Restaurant.RestaurantRating
    restaurant.ImageUrl = request.Restaurant.RestaurantImageUrl
    restaurant.OperationDays = request.Restaurant.RestaurantOperationDays
    restaurant.OperationHours = request.Restaurant.RestaurantOperationHours
    restaurant.RestaurantOwnerMail = userEmail
    restaurant.RestaurantMinimumOrderAmount =
request.Restaurant.RestaurantMinimumOrderAmount
```

```
    restaurant.RestaurantDiscountPercentage =
request.Restaurant.RestaurantDiscountPercentage
    restaurantAddress.City = request.Restaurant.RestaurantAddress.City
    restaurantAddress.Country = request.Restaurant.RestaurantAddress.Country
    restaurantAddress.Pincode = request.Restaurant.RestaurantAddress.Pincode
    restaurantAddress.StreetName = request.Restaurant.RestaurantAddress.StreetName

    updateRestaurantError := restaurantDBConnector.Save(&restaurant)
    updateAddressError := restaurantAddressDBConnector.Save(&restaurantAddress)

    if updateAddressError.Error != nil || updateRestaurantError.Error != nil {
        logger.Error("Failed to update restaurant", zap.Error(updateRestaurantError.Error))
        logger.Error("Failed to update restaurant address", zap.Error(updateAddressError.Error))
        return &restaurantpb.UpdateRestaurantResponse{
            Message:    "A restaurant with the same name or phone number already exists.",
            StatusCode: StatusConflict,
            Error:      "Conflict",
        }, nil
    }
    logger.Info("Restaurant updated successfully", zap.Any("Restaurant", request.Restaurant))
    return &restaurantpb.UpdateRestaurantResponse{
        Data: &restaurantpb.UpdateRestaurantData{
            Restaurant: request.Restaurant,
        },
        Message:    "Restaurant updated successfully",
        StatusCode: StatusOK,
        Error:      "",
    }, nil

}
```

1. Function named UpdateRestaurant that belongs to the RestaurantService struct. It's designed to update the details of a restaurant in a restaurant management.

    a. Logs the incoming UpdateRestaurant request.

    b. Extracts the user's email from the context. If it fails, it logs an error and returns an InternalServerError response.

    c. Checks if the restaurant data provided in the request is valid. If not, it returns a BadRequest response.

    d. Validates the restaurant fields and phone number. If they're invalid, it logs a warning and returns a BadRequest response.

    e. Fetches the restaurant from the database using the restaurant ID provided in the request.

    f. Checks if the restaurant exists and if the user is authorized to update it (i.e., the user is the owner of the restaurant). If not, it logs a warning and returns an Unauthorized response.

    g. Fetches the restaurant's address from the database. If it fails, it logs an error and returns an InternalServerError response.

    h. Updates the restaurant and address details with the new data provided in the request.

    i. Saves the updated restaurant and address details to the database. If it fails (e.g., a restaurant with the same name or phone number already exists), it logs an error and returns a Conflict response.

    j. If everything goes well, it logs a success message and returns an OKresponse with the updated restaurant data.

2. This function uses the zap package for structured logging and the gorm package (not directly imported but used via restaurantDBConnector and restaurantAddressDBConnector) for database operations.

Step 14. updateRestaurantItem.go file

```
package main

import (
    "context"
    "restaurant-micro/config"
    "restaurant-micro/model"
    restaurantpb "restaurant-micro/proto/restaurant"
    "strconv"

    "go.uber.org/zap"
)

func (*RestaurantService) UpdateRestaurantItem(ctx context.Context, request
*restaurantpb.UpdateRestaurantItemRequest) (*restaurantpb.UpdateRestaurantItemResponse,
error) {

    logger.Info("Received UpdateRestaurantItem request",
    zap.Any("Request", request))

    userEmail, ok := ctx.Value("userEmail").(string)
    if !ok {
        logger.Error("Failed to get user email from context")
        return &restaurantpb.UpdateRestaurantItemResponse{
            Message:    "Failed to get user email from context",
            StatusCode: StatusInternalServerError,
            Error:      "Internal Server Error",
        }, nil
    }
    // validate the restaurant item fields
    if request.RestaurantItem == nil ||
!config.ValidateRestaurantItemFields(request.RestaurantItem.RestaurantItemName,
```

```
        strconv.FormatInt(request.RestaurantItem.RestaurantItemPrice, 10),
        request.RestaurantItem.RestaurantItemPrice,
request.RestaurantItem.GetRestaurantItemCategory(),
        request.RestaurantItem.RestaurantItemCuisineType,
request.RestaurantItem.RestaurantItemImageUrl) {
            logger.Warn("Invalid restaurant item data provided",
            zap.Any("RestaurantItem", request.RestaurantItem))

        return &restaurantpb.UpdateRestaurantItemResponse{
            Message:    "Invalid restaurant item data provided.Some fields might be missing or
invalid",
            StatusCode: StatusBadRequest,
            Error:      "Invalid restaurant item fields",
        }, nil
    }
    // fetch restaurant from restaurantDB
    var restaurant model.Restaurant
    primaryKeyRes := restaurantDBConnector.Where("id = ?",
request.RestaurantItem.RestaurantId).First(&restaurant)
    // check if the restaurant is exist or nor
    if primaryKeyRes.Error != nil || restaurant.RestaurantOwnerMail != userEmail {
        logger.Warn("Unauthorized update attempt",
        zap.String("RestaurantOwnerMail", restaurant.RestaurantOwnerMail),
        zap.String("UserEmail", userEmail))

        return &restaurantpb.UpdateRestaurantItemResponse{
            Message:    "You do not have permission to perform this action. Only restaurant owner
can update the restaurant item",
            StatusCode: StatusForbidden,
            Error:      "Resource not found or forbidden",
        }, nil
    }
```

```
    var restaurantItem model.RestaurantItem
    primaryKey := restaurantItemDBConnector.Where("id = ? AND restaurant_id = ?",
request.RestaurantItem.RestaurantItemId,
        restaurant.ID).First(&restaurantItem)
    if primaryKey.Error != nil {
        logger.Error("Failed to fetch restaurant item from database",
        zap.Error(primaryKey.Error))
        return &restaurantpb.UpdateRestaurantItemResponse{
            Message:    "Restaurant Item does not exist",
            StatusCode: StatusNotFound,
            Error:      "Not Found",
        }, nil
    }
    restaurantItem.ItemName = request.RestaurantItem.RestaurantItemName
    restaurantItem.ItemPrice = request.RestaurantItem.RestaurantItemPrice
    restaurantItem.Category = request.RestaurantItem.GetRestaurantItemCategory()
    restaurantItem.CuisineType = request.RestaurantItem.RestaurantItemCuisineType
    restaurantItem.Veg = request.RestaurantItem.RestaurantItemVeg
    restaurantItem.ImageUrl = request.GetRestaurantItem().GetRestaurantItemImageUrl()

    err := restaurantItemDBConnector.Save(&restaurantItem)
    if err.Error != nil {
        logger.Error("Failed to update restaurant item", zap.Error(err.Error))
        return &restaurantpb.UpdateRestaurantItemResponse{
            Message:    "Failed to update restaurant item, this can be due to same item name
already exist in the restaurant or some other issue.",
            StatusCode: StatusInternalServerError,
            Error:      "Internal Server Error",
        }, nil
    }
```

```
    logger.Info("Restaurant item updated successfully", zap.Any("RestaurantItem",
request.RestaurantItem))
    return &restaurantpb.UpdateRestaurantItemResponse{
        Data: &restaurantpb.UpdateRestaurantItemResponseData{
            RestaurantItem: request.RestaurantItem,
        },
        Message:    "Restaurant item updated successfully",
        StatusCode: StatusOK,
        Error:      "",
    }, nil
}
```

1. Function named UpdateRestaurantItem that belongs to the RestaurantService struct. It's designed to update the details of a restaurant item in a restaurant management.

    a. Logs the incoming UpdateRestaurantItem request.

    b. Extracts the user's email from the context. If it fails, it logs an error and returns an InternalServerError response.

    c. Validates the restaurant item fields. If they're invalid, it logs a warning and returns a BadRequest response.

    d. Fetches the restaurant from the database using the restaurant ID provided in the request.

    e. Checks if the restaurant exists and if the user is authorized to update it (i.e., the user is the owner of the restaurant). If not, it logs a warning and returns a Forbidden response.

    f. Fetches the restaurant item from the database. If it fails, it logs an error and returns a NotFound response.

    g. Updates the restaurant item details with the new data provided in the request.

    h. Saves the updated restaurant item details to the database. If it fails (e.g., a restaurant item with the same name already exists), it logs an error and returns an InternalServerError response.

i. If everything goes well, it logs a success message and returns an OKresponse with the updated restaurant item data.

This function uses the zap package for structured logging and the gorm package (not directly imported but used via restaurantDBConnector and restaurantItemDBConnector) for database operations.

Step 15: deleteRestaurantItem.go file

```
package main

import (
    "context"
    "restaurant-micro/model"
    restaurantpb "restaurant-micro/proto/restaurant"
    "strconv"

    "go.uber.org/zap"
)

func (*RestaurantService) DeleteRestaurantItem(ctx context.Context, request
*restaurantpb.DeleteRestaurantItemRequest) (*restaurantpb.DeleteRestaurantItemResponse,
error) {
    logger.Info("Received DeleteRestaurantItem request",
        zap.String("restaurantId", request.RestaurantId),
        zap.String("restaurantItemId", request.RestaurantItemId))

    // Get the user email from the context
    userEmail, emailCtxError := ctx.Value("userEmail").(string)
    userRole, roleCtxError := ctx.Value("userRole").(string)

    if !emailCtxError || !roleCtxError {
```

```
        logger.Error("Failed to get user email or role from context")
        return &restaurantpb.DeleteRestaurantItemResponse{
            Message:    "Failed to get user email and role from context",
            Error:      "Internal Server Error",
            StatusCode: StatusInternalServerError,
        }, nil
    }

    logger.Info("Context values retrieved", zap.String("userEmail", userEmail),
zap.String("userRole", userRole))
    if userRole != model.AdminRole {
        logger.Warn("Permission denied", zap.String("userRole", userRole))
        return &restaurantpb.DeleteRestaurantItemResponse{
            Data:       nil,
            Message:    "You do not have permission to perform this action. Only admin can add a
delete restaurant item",
            StatusCode: StatusForbidden,
            Error:      "Forbidden",
        }, nil
    }
    //convert string to int64
    restaurantId, _ := strconv.ParseUint(request.RestaurantId, 10, 64)
    restaurantItemId, _ := strconv.ParseUint(request.RestaurantItemId, 10, 64)
    // validate the restaurant item fields
    if restaurantId <= 0 || restaurantItemId <= 0 {
        logger.Warn("Invalid restaurant item data provided")
        return &restaurantpb.DeleteRestaurantItemResponse{
            Data:       nil,
            Message:    "Invalid restaurant item data provided",
            StatusCode: StatusBadRequest,
            Error:      "Bad Request",
        }, nil
```

```
    }

    // check if user own's this restaurant
    var restaurant model.Restaurant
    primaryKeyRes := restaurantDBConnector.Where("id = ?", restaurantId).First(&restaurant)
    if primaryKeyRes.Error != nil || restaurant.RestaurantOwnerMail != userEmail {
        logger.Warn("Restaurant does not exist or user is not the owner",
            zap.String("userEmail", userEmail),
            zap.String("restaurantId", request.RestaurantId))
        return &restaurantpb.DeleteRestaurantItemResponse{
            Data:       nil,
            Message:    "Restaurant does not exist OR you are not the owner of this restaurant",
            StatusCode: StatusForbidden,
            Error:      "Resource not found or forbidden",
        }, nil
    }

    // delete the restaurant item
    var restaurantItem model.RestaurantItem
    primaryKey := restaurantItemDBConnector.Where("id = ? AND restaurant_id = ?",
        restaurantItemId, restaurant.ID).First(&restaurantItem)

    if primaryKey.Error != nil {
        logger.Warn("Restaurant item does not exist", zap.String("restaurantItemId",
request.RestaurantItemId))
        return &restaurantpb.DeleteRestaurantItemResponse{
            Data:       nil,
            Message:    "Restaurant item does not exist",
            StatusCode: StatusForbidden,
            Error:      "Resource not found or forbidden",
        }, nil
    }
```

```
    err := restaurantItemDBConnector.Delete(&restaurantItem)
    if err.Error != nil {
        logger.Error("Failed to delete restaurant item", zap.String("restaurantItemId",
request.RestaurantItemId), zap.Error(err.Error))
        return &restaurantpb.DeleteRestaurantItemResponse{
            Data:       nil,
            Message:    "Failed to delete restaurant item",
            StatusCode: StatusInternalServerError,
            Error:      "Internal Server Error",
        }, nil
    }
    logger.Info("Restaurant item deleted successfully", zap.String("restaurantItemId",
strconv.FormatUint(uint64(restaurantItem.ID), 10)))
    return &restaurantpb.DeleteRestaurantItemResponse{
        Data: &restaurantpb.DeleteRestaurantItemResponseData{
            RestaurantItemId: strconv.FormatUint(uint64(restaurantItem.ID), 10),
        },
        Message:    "Restaurant item deleted successfully",
        StatusCode: StatusOK,
        Error:      "",
    }, nil
}
```

1. This Go code is a function named DeleteRestaurantItem that belongs to the RestaurantService struct. It's designed to delete a restaurant item in a restaurant management system. Here's a step-by-step breakdown of what the function does:

2. Logs the incoming DeleteRestaurantItem request.

3. Extracts the user's email and role from the context. If it fails, it logs an error and returns an InternalServerError response.

4. Checks if the user role is admin. If not, it logs a warning and returns a Forbidden response.

5. Converts the restaurant ID and restaurant item ID from string to uint64.

6. Validates the restaurant item fields. If they're invalid, it logs a warning and returns a BadRequest response.

7. Fetches the restaurant from the database using the restaurant ID provided in the request.

8. Checks if the restaurant exists and if the user is the owner of the restaurant. If not, it logs a warning and returns a Forbidden response.

9. Fetches the restaurant item from the database. If it fails, it logs a warning and returns a Forbidden response.

10. Deletes the restaurant item from the database. If it fails, it logs an error and returns an InternalServerError response.

11. If everything goes well, it logs a success message and returns an OK response with the ID of the deleted restaurant item.

12. This function uses the zap package for structured logging and the gorm package (not directly imported but used via restaurantDBConnector and restaurantItemDBConnector) for database operations.

Step 16. getAllRestaurantsByCategory.go file

```
package main

import (
    "context"
    "restaurant-micro/model"
    restaurantpb "restaurant-micro/proto/restaurant"
    "strconv"
```

```
    "go.uber.org/zap"
)

func (*RestaurantService) GetRestaurantsByItemCategory(ctx context.Context,
    request *restaurantpb.GetRestaurantsByItemCategoryRequest)
(*restaurantpb.GetRestaurantsByItemCategoryResponse, error) {

    logger.Info("Received GetRestaurantsByItemCategory request",
    zap.String("Category", request.Category))

    if request.Category == "" {
        logger.Warn("Category field is required but not provided")

        return &restaurantpb.GetRestaurantsByItemCategoryResponse{
            Data:       nil,
            Message:    "Invalid Field, category field is required",
            StatusCode: StatusBadRequest,
            Error:      "Bad Request",
        }, nil
    }
    var restaurantItems []model.RestaurantItem
    err := restaurantItemDBConnector.Where("category = ?",
request.Category).Find(&restaurantItems)
    if err.Error != nil {
        logger.Error("Failed to get restaurant items from the database", zap.Error(err.Error))
        return &restaurantpb.GetRestaurantsByItemCategoryResponse{
            Data:       nil,
            Message:    "Failed to get restaurant items",
            StatusCode: StatusInternalServerError,
            Error:      "Internal Server Error",
        }, nil
    }
```

```
    logger.Info("Fetched restaurant items", zap.Int("Count", len(restaurantItems)))

    restaurantIds := make(map[uint]bool)
    var restaurantsResponse []*restaurantpb.Restaurant
    for _, restaurantItem := range restaurantItems {
        if restaurantIds[restaurantItem.RestaurantId] {
            continue
        }
        var restaurant model.Restaurant
        var restaurantAddress model.Address
        restaurantError := restaurantDBConnector.Where("id = ?",
restaurantItem.RestaurantId).First(&restaurant)
        restaurantAddressError := restaurantAddressDBConnector.Where("restaurant_id = ?",
restaurant.ID).First(&restaurantAddress)

        if restaurantError.Error != nil || restaurantAddressError.Error != nil {
            logger.Error("Failed to get restaurant or restaurant address from the database",
            zap.Error(restaurantError.Error),
            zap.Error(restaurantAddressError.Error))

            return &restaurantpb.GetRestaurantsByItemCategoryResponse{
                Data:       nil,
                Message:    "Failed to get restaurant or restaurant address",
                StatusCode: StatusInternalServerError,
                Error:      "Internal Server Error",
            }, nil
        }
        restaurantsResponse = append(restaurantsResponse, &restaurantpb.Restaurant{
            RestaurantId:           strconv.FormatUint(uint64(restaurant.ID), 10),
            RestaurantName:         restaurant.Name,
            RestaurantAvailability:  restaurant.Availability,
            RestaurantPhoneNumber:   restaurant.Phone,
```

```
            RestaurantRating:         restaurant.Rating,
            RestaurantImageUrl:       restaurant.ImageUrl,
            RestaurantOperationDays:  restaurant.OperationDays,
            RestaurantOperationHours: restaurant.OperationHours,
            RestaurantOwnerMail:      restaurant.RestaurantOwnerMail,
            RestaurantMinimumOrderAmount: restaurant.RestaurantMinimumOrderAmount,
            RestaurantDiscountPercentage: restaurant.RestaurantDiscountPercentage,
            RestaurantAddress: &restaurantpb.Address{
                City:       restaurantAddress.City,
                Country:    restaurantAddress.Country,
                Pincode:    restaurantAddress.Pincode,
                StreetName: restaurantAddress.StreetName,
            },
        })
        restaurantIds[restaurantItem.RestaurantId] = true
    }
    logger.Info("Fetched restaurants by item category successfully", zap.Int("TotalRestaurants",
len(restaurantsResponse)))
    return &restaurantpb.GetRestaurantsByItemCategoryResponse{
        Data: &restaurantpb.GetRestaurantsByItemCategoryResponseData{
            TotalRestaurants: int64(len(restaurantsResponse)),
            Restaurants:      restaurantsResponse,
        },
        Message:    "Restaurants fetched successfully",
        StatusCode: StatusOK,
        Error:      "",
    }, nil
}
```

Function named GetRestaurantsByItemCategory that belongs to the RestaurantService struct. It's designed to fetch restaurants based on the category of items they serve.

1. Logs the incoming GetRestaurantsByItemCategory request.
2. Checks if the category field in the request is empty. If it is, it logs a warning and returns a BadRequest response.
3. Fetches restaurant items from the database that match the category provided in the request.
4. If there's an error fetching the restaurant items, it logs an error and returns an InternalServerError response.
5. Logs the count of fetched restaurant items.
6. Initializes a map to keep track of unique restaurant IDs and a slice to store the response.
7. Iterates over the fetched restaurant items. For each item:
8. If the restaurant ID is already in the map, it skips to the next item.
9. Fetches the restaurant and its address from the database using the restaurant ID.
10. If there's an error fetching the restaurant or its address, it logs an error and returns an InternalServerError response.
11. Appends the restaurant to the response slice and adds its ID to the map.
12. Logs the total number of fetched restaurants.
13. Returns an OK response with the total number of restaurants and the restaurants themselves.
14. This function uses the zap package for structured logging and the gorm package (not directly imported but used via restaurantItemDBConnector, restaurantDBConnector, and restaurantAddressDBConnector) for database operations.

Step 17. getRestaurantsByCity.go file

```
package main
import (
    "context"
    "restaurant-micro/model"
    restaurantpb "restaurant-micro/proto/restaurant"
    "strconv"

    "go.uber.org/zap"
)

func (*RestaurantService) GetRestaurantsByCity(ctx context.Context, request
*restaurantpb.GetRestaurantsByCityRequest) (*restaurantpb.GetRestaurantsByCityResponse,
error) {

    logger.Info("Received GetRestaurantsByCity request",
        zap.String("City", request.City))

    if request.City == "" {
        logger.Warn("City field is required but not provided")
        return &restaurantpb.GetRestaurantsByCityResponse{
            Data:       nil,
            Message:    "Invalid Field, city field is required",
            StatusCode: StatusBadRequest,
            Error:      "Bad Request",
        }, nil
    }
    city := request.City
    var restaurantAddress []model.Address
    err := restaurantAddressDBConnector.Where("city = ?", city).Find(&restaurantAddress)
    if err.Error != nil {
        logger.Error("Failed to get restaurants from the database", zap.Error(err.Error))
```

```
        return &restaurantpb.GetRestaurantsByCityResponse{
            Data:       nil,
            Message:    "Failed to get restaurants from the database",
            StatusCode: StatusInternalServerError,
            Error:      "Internal Server Error",
        }, nil
    }
    logger.Info("Fetched restaurant addresses", zap.Int("Count", len(restaurantAddress)))

    var restaurantsResponse []*restaurantpb.Restaurant
    for _, address := range restaurantAddress {
        // fetch all restaurant details from the database filter by city.
        var restaurant model.Restaurant
        restaurantErr := restaurantDBConnector.Where("id = ?",
address.RestaurantId).First(&restaurant).Error
        if restaurantErr != nil {
            logger.Error("Failed to get restaurant from the database", zap.Error(restaurantErr))
            return &restaurantpb.GetRestaurantsByCityResponse{
                Data:       nil,
                Message:    "Failed to get restaurant from the database",
                StatusCode: StatusInternalServerError,
                Error:      "Internal Server Error",
            }, nil
        }
        restaurantsResponse = append(restaurantsResponse, &restaurantpb.Restaurant{
            RestaurantId:           strconv.FormatUint(uint64(restaurant.ID), 10),
            RestaurantName:         restaurant.Name,
            RestaurantAvailability: restaurant.Availability,
            RestaurantPhoneNumber:  restaurant.Phone,
            RestaurantRating:       restaurant.Rating,
            RestaurantImageUrl:     restaurant.ImageUrl,
            RestaurantOperationDays: restaurant.OperationDays,
```

```
            RestaurantOperationHours: restaurant.OperationHours,
            RestaurantOwnerMail:      restaurant.RestaurantOwnerMail,
            RestaurantMinimumOrderAmount: restaurant.RestaurantMinimumOrderAmount,
            RestaurantDiscountPercentage: restaurant.RestaurantDiscountPercentage,

            RestaurantAddress: &restaurantpb.Address{
                City:       address.City,
                Country:    address.Country,
                Pincode:    address.Pincode,
                StreetName: address.StreetName,
            },
        })
    }
    logger.Info("Fetched restaurants by city successfully", zap.Int("TotalRestaurants",
len(restaurantsResponse)))
    return &restaurantpb.GetRestaurantsByCityResponse{
        Data: &restaurantpb.GetRestaurantsByCityResponseData{
            TotalRestaurants: int64(len(restaurantsResponse)),
            Restaurants:      restaurantsResponse,
        },
        Message:    "Restaurants fetched successfully",
        StatusCode: StatusOK,
        Error:      "",
    }, nil
}
```

Function named GetRestaurantsByCity that belongs to the RestaurantService struct. It's designed to fetch restaurants based on the city they are located in.

1. Logs the incoming GetRestaurantsByCity request.

2. Checks if the city field in the request is empty. If it is, it logs a warning and returns a BadRequest response.

3. Fetches restaurant addresses from the database that match the city provided in the request.
4. If there's an error fetching the restaurant addresses, it logs an error and returns an InternalServerError response.
5. Logs the count of fetched restaurant addresses.
6. Initializes a slice to store the response.
7. Iterates over the fetched restaurant addresses. For each address:
8. Fetches the restaurant from the database using the restaurant ID in the address.
9. If there's an error fetching the restaurant, it logs an error and returns an InternalServerError response.
10. Appends the restaurant to the response slice, converting the restaurant's ID to a string and including the address in the response.
11. Logs the total number of fetched restaurants.
12. Returns an OK response with the total number of restaurants and the restaurants themselves.
13. This function uses the zap package for structured logging and the gorm package (not directly imported but used via restaurantAddressDBConnector and restaurantDBConnector) for database operations.

Step 18. Dockerfile [All about docker >> Link]

```
# Base image
FROM golang:1.22.2-alpine3.19

# Install bash
RUN apk add --no-cache bash
```

```
# Move to working directory /app
WORKDIR /app

# Copy the code into the container
COPY . .

# Download dependencies using go mod
RUN go mod tidy && go mod vendor

# Expose PORT 8091 to the outside world
EXPOSE 8091

# Command to run the application when starting the container
CMD ["go", "run", "."]
```

3. Order Microservice.

1. Install Dependencies.

A. Dependencies are already installed, you can go to generate.sh file to check, what all dependencies are added.

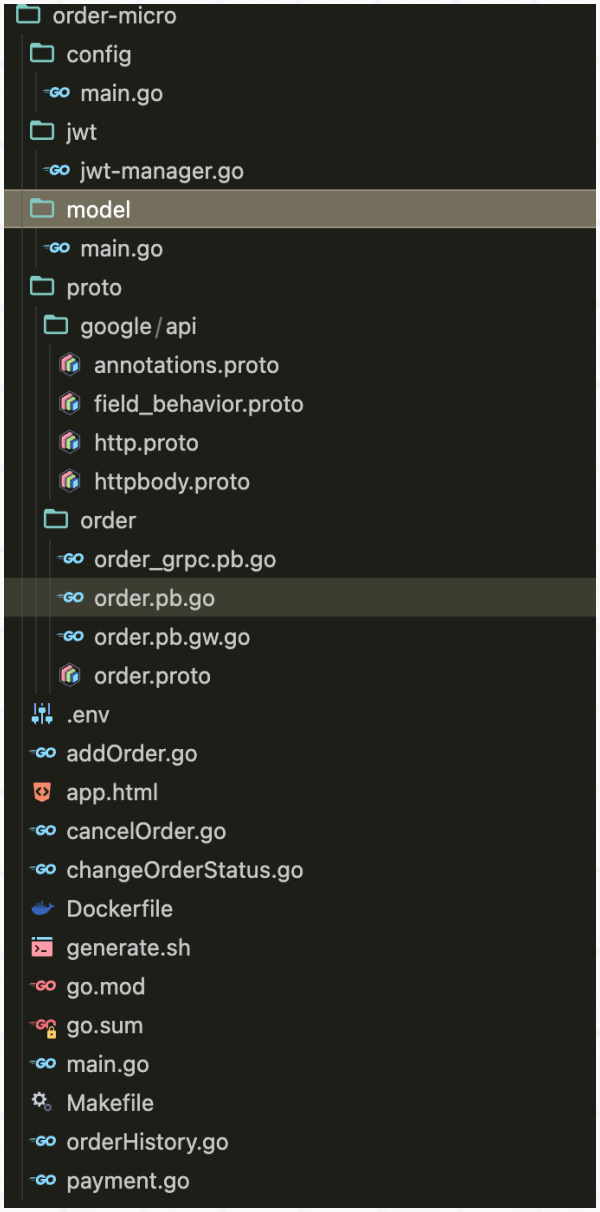
2. Compile order.proto file.

1. Go to the folder order-micro and open new integrated terminal.

2. Run the command to compile order.proto file

```
cd order-mico
make proto-generate
```

3. The Directory Structure of the order microservice will look like this: -



4.Create .env file for environment variables

```
DB_CONFIG="root:Aec19cs047@tcp(127.0.0.1:3306)/test?charset=utf8mb4&parseTime=True&loc=Local"
SECRET_KEY="SECRET_KEY_JWT_TOKEN"
MYSQL_USER="root"
MYSQL_HOST="127.0.0.1"
MYSQL_PASSWORD="<password>"
MYSQL_DATABASE="test"
MYSQL_PORT="3306"
```

Make sure to replace <password> with your db password.

1. main.go file

a. It defines a struct OrderService that embeds orderpb.UnimplementedOrderServiceServer to implement the OrderServiceServer interface.

b. It defines constants for HTTP status codes and global variables for database connectors and a logger.

c. In the init function, it initializes the logger. If it fails, it panics.

d. The startServer function does the following:

i. Loads environment variables from a .env file.
ii. Connects to the database and stores the connections in global variables.
iii. Starts a gRPC server on localhost:50053 and registers the OrderService with it.
iv. In a new goroutine, it starts serving the gRPC server.
v. Creates a gRPC-Gateway server that connects to the gRPC server.

vi. Registers the OrderService with the gRPC-Gateway server.
vii. Starts an HTTP server on port 8093 that serves the gRPC-Gateway server.

e. The main function calls startServer to start the server.

This code uses the zap library for structured, leveled logging, the gorm library for interacting with the database, the grpc and grpc-gateway libraries for the gRPC server and gateway, and the godotenv library for loading environment variables.

1.Go to order-microservice: - **cd order-micro**
2.Run Command (to start the server): - **go run .**

2. main.go/config

1. **DatabaseDsn**: This function constructs a data source name (DSN) for connecting to a MySQL database. It uses environment variables to get the necessary parameters such as the MySQL user, password, host, port, and database name.

2. **GoDotEnvVariable**: This function loads environment variables from a .env file and returns the value of a specified key. If there's an error in loading the .env file, it logs the error and terminates the program.

3. **ConnectDB**: This function connects to a MySQL database using the DSN from the DatabaseDsn function and returns two gorm.DB instances for the Order and OrderItem models. It also automatically migrates the Order and OrderItem models. If there's an error in connecting to the database, it panics.

4. **GetUserConnector**: This function connects to a MySQL database using the DSN from the DatabaseDsn function and returns a gorm.DB instance for the User model. It also automatically migrates the User model. If there's an error in connecting to the database, it panics.

5. **GetRestaurantConnector**: This function connects to a MySQL database using the DSN from the DatabaseDsn function and returns a gorm.DB instance for the Restaurant model. It also automatically migrates the Restaurant model. If there's an error in connecting to the database, it panics.

6. **ValidateOrderFields**: This function validates the fields of an order. It returns false if the shipping address or restaurant name is empty or the order amount is zero, and true otherwise.

7. main.go/model

```
package model

import "gorm.io/gorm"
const (
    AdminRole = "admin"
    UserRole = "user"
)

type OrderItem struct {
    gorm.Model
    Name     string
    Quantity int64
    Price    int64
    OrderId uint `gorm:"foreignKey:OrderId;references:ID"`
}

type Order struct {
    gorm.Model
    UserId uint `gorm:"foreignKey:UserId;references:ID"`
    TotalPrice int64
    Status string
    Discount string
    RestaurantId uint `gorm:"foreignKey:RestaurantId;references:ID"`
    ShippingAddress string
}
```

```
type User struct {
    gorm.Model
    Name string
    Password string
    Email  string `gorm:"unique"`
    Phone  string `gorm:"unique"`
    Address string
    City string
    Role string
}

type Restaurant struct {
    gorm.Model
    Name string `gorm:"unique"`
    Phone string `gorm:"unique"`
    Availability string `gorm:"default:open"`
    Rating float32 `gorm:"default:0"`
    RestaurantOwnerMail string
    ImageUrl string
    OperationDays string
    OperationHours string
}

type Address struct {
    gorm.Model
    RestaurantId uint
`gorm:"foreignKey:RestaurantId;references:ID;uniqueIndex:idx_restaurant_address"` // foreign
key referencing the primary key of the Restaurant table
    StreetName string
    Pincode string
    City string
    Country string
```

```
}
```

1. **OrderItem**: Represents an item in an order. It has fields for the item's name, quantity, price, and the ID of the order it belongs to.

2. **Order**: Represents an order. It has fields for the ID of the user who placed the order, the total price of the order, the status of the order, the discount applied to the order, the ID of the restaurant from which the order was placed, and the shipping address for the order.

3. **User**: Represents a user. It has fields for the user's name, password, email, phone number, address, city, and role.

4. **Restaurant**: Represents a restaurant. It has fields for the restaurant's name, phone number, availability, rating, owner's email, image URL, operation days, and operation hours.

5. **Address**: Represents an address. It has fields for the ID of the restaurant at the address, the street name, pin code, city, and country.

Each model uses the gorm.Model struct as an embedded field, which provides basic fields like ID, CreatedAt, UpdatedAt, and DeletedAt.

The gorm:"foreignKey:...;references:ID" tags are used to set up foreign key relationships between the models. The gorm:"unique" tags are used to ensure

that certain fields are unique. The gorm:"default:..." tags are used to set default values for certain fields.

8. payment.go file

```
package main

import (
    "context"
    "fmt"
    "log"
    "net/http"
    "os"
    "strconv"
    "time"

    "github.com/gin-gonic/gin"
    razorpay "github.com/razorpay/razorpay-go"
)

type PageVariables struct {
    OrderId string
    Email   string
    Name    string
    Amount  string
    Contact string
}

func Caller(amount int) string {
    statusChannel := make(chan string)
    router := gin.Default()
```

```
router.LoadHTMLGlob("*.html")
router.GET("/", func(c *gin.Context) {
    App(c, amount*100)
})
router.GET("/payment-fail", func(c *gin.Context) {
    fmt.Println("Payment Failed")
    PaymentFaliure(c, statusChannel)
})

router.GET("/payment-complete", func(c *gin.Context) {
    fmt.Println("Payment Page")
    PaymentSuccess(c, statusChannel)
})
srv := &http.Server{
    Addr:    ":8089",
    Handler: router,
}

go func() {
    if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
        log.Fatalf("listen: %s\n", err)
    }
}()

fmt.Println("Server is running on port 8089")

// create a timer and exit the server in 1 minute
time.AfterFunc(45*time.Second, func() {
    fmt.Println("Shutting down server...")
    if err := srv.Shutdown(context.Background()); err != nil {
        log.Fatal(err)
    }
```

```
    fmt.Println("Server gracefully stopped")
  })
  // Wait for a payment status to be sent to the channel or for a timeout
  select {
  case <-time.After(45 * time.Second):
    select {
    case status := <-statusChannel:
      return status
    default:
      return "no-status"
    }
  case <-time.After(1 * time.Minute):
    return "payment-failed"
  }

}

func App(c *gin.Context, amount int) {

  page := &PageVariables{}
  page.Amount = strconv.Itoa(amount)
  page.Email = "rahul.c@prograd.org"
  page.Name = "john Doe"
  page.Contact = "7906936789"
  //Create order_id from the server
  client := razorpay.NewClient("rzp_test_p4bR8DXSKNi8tJ", "JDcoQd2EgIcodZR1vGparqCq")

  data := map[string]interface{}{
    "amount":   page.Amount,
    "currency": "INR",
    "receipt":  "some_receipt_id",
  }
```

```
    body, err := client.Order.Create(data, nil)
    fmt.Println("////////////////reciept", body)
    if err != nil {
        fmt.Println("Problem getting the repository information", err)
        os.Exit(1)
    }

    value := body["id"]

    str := value.(string)
    fmt.Println("str////////////////", str)
    HomePageVars := PageVariables{ //store the order_id in a struct
        OrderId: str,
        Amount:  page.Amount,
        Email:   page.Email,
        Name:    page.Name,
        Contact: page.Contact,
    }

    c.HTML(http.StatusOK, "app.html", HomePageVars)
}

func PaymentSuccess(c *gin.Context, statusChannel chan<- string) {
    fmt.Println("Payment Successfull")
    paymentid := c.Query("paymentid")
    orderid := c.Query("orderid")
    signature := c.Query("signature")

    fmt.Println(paymentid, "paymentid")
    fmt.Println(orderid, "orderid")
    fmt.Println(signature, "signature")
```

```
    statusChannel <- "Payment Successfull"
    fmt.Println("Payment Successfull")
    c.Redirect(http.StatusFound, "http://localhost:3000/payment-success")
}

func PaymentFaliure(c *gin.Context, statusChannel chan<- string) {
    statusChannel <- "Payment Failed"
    c.Redirect(http.StatusFound, "http://localhost:3000/order-history")
}
```

1. It defines a struct PageVariables to hold the order details.
2. The Caller function does the following:

3. Creates a channel to communicate the payment status.
    a. Sets up a Gin router and loads HTML templates from the current directory.
    b. Defines routes for the home page, payment failure, and payment success. The home page route calls the App function with the payment amount. The failure and success routes call PaymentFailure and PaymentSuccess respectively, passing the status channel to them.
    c. Starts an HTTP server on port 8089 in a new goroutine.
    d. After 45 seconds, it shuts down the server.
    e. Waits for a payment status to be sent to the channel or for a timeout. If a status is received within 45 seconds, it returns the status. If no status is received within 1 minute, it returns "payment-failed".
    f. Payment-gateway url

        http://localhost:8089

4. The App function does the following:
    a. Creates a PageVariables instance with the payment amount and some hard-coded details.
    b. Creates a new Razorpay client with a test key and secret.
    c. Creates a new order with the payment amount in INR.

d. If the order creation is successful, it stores the order ID in PageVariables and renders an HTML page with these details.

5. The PaymentSuccess function reads the payment ID, order ID, and signature from the query parameters, sends "Payment Successful" to the status channel, and redirects to a success page.
6. The PaymentFailure function sends "Payment Failed" to the status channel and redirects to an orders page.

This uses the gin library for the HTTP server and routing, and the razorpay-go library to interact with the Razorpay API.

9. addOrder.go file

```
package main

import (
    "context"
    "order-microservice/config"
    "order-microservice/model"
    orderpb "order-microservice/proto/order"
    "strconv"

    "go.uber.org/zap"
)

func (*OrderService) AddOrder(ctx context.Context, request *orderpb.AddOrderRequest)
(*orderpb.AddOrderResponse, error) {
    userEmail, emailCtxError := ctx.Value("userEmail").(string)
    userRole, roleCtxError := ctx.Value("userRole").(string)

    if !emailCtxError || !roleCtxError {
        logger.Error("Failed to get user email and role from context",
            zap.Bool("emailCtxError", emailCtxError), zap.Bool("roleCtxError", roleCtxError))
        return &orderpb.AddOrderResponse{
```

```
            Data:       nil,
            Message:    "Failed to get user mail and role from context",
            Error:      "Internal Server Error",
            StatusCode: StatusInternalServerError,
        }, nil
    }
    logger.Info("Received AddOrder request",
    zap.String("userEmail", userEmail), zap.String("userRole", userRole))
    if userRole != model.UserRole {
        logger.Warn("Permission denied for non-user role",
        zap.String("userRole", userRole))

        return &orderpb.AddOrderResponse{
            Data:       nil,
            Message:    "You do not have permission to perform this action. Only user can add an
order",
            StatusCode: StatusForbidden,
            Error:      "Forbidden",
        }, nil
    }

    if len(request.OrderItems) == 0 || !config.ValidateOrderFields(request.ShippingAddress,
request.RestaurantName, request.OrderTotalPrice) {
        logger.Warn("Invalid request fields", zap.String("userEmail", userEmail))
        return &orderpb.AddOrderResponse{
            Data:       nil,
            Message:    "Invalid request. Please check the fields and try again.",
            StatusCode: StatusBadRequest,
            Error:      "The request contains invalid or missing fields.",
        }, nil
    }
    // fetching the user from user email.
```

```
    var user model.User
    var restaurant model.Restaurant
    userDBConnector, userDbErr := config.GetUserConnector()
    restaurantDBConnector, restaurantDbErr := config.GetRestaurantConnector()
    if userDbErr != nil || restaurantDbErr != nil {
        logger.Error("Failed to connect to database",
            zap.Error(userDbErr), zap.Error(restaurantDbErr))
        return &orderpb.AddOrderResponse{
            Message:    "Server Facing Issues",
            StatusCode: StatusInternalServerError,
            Error:      "Internal Server Error",
        }, nil
    }
    logger.Info("Connected to databases")
    if err := userDBConnector.Where("email = ?", userEmail).First(&user).Error; err != nil {
        logger.Warn("User not found", zap.String("userEmail", userEmail))
        return &orderpb.AddOrderResponse{
            Message:    "User not found",
            StatusCode: StatusNotFound,
            Error:      "Resource not found",
        }, nil
    }

    restaurantError := restaurantDBConnector.Where("name = ?",
request.RestaurantName).First(&restaurant).Error

    if restaurantError != nil {
        logger.Warn("Restaurant not found", zap.String("restaurantName",
request.RestaurantName))
        return &orderpb.AddOrderResponse{
            Message:    "Restaurant not found",
            StatusCode: StatusNotFound,
```

```
        Error:      "Resource not found",
    }, nil
}
// payment gateway logic goes here.
result := Caller(int(request.OrderTotalPrice))
if result != "Payment Successfull" {
    logger.Warn("Payment failed", zap.String("userEmail", userEmail))
    return &orderpb.AddOrderResponse{
        Message:    "Payment Required. Failed to add order.",
        StatusCode: StatusBadRequest,
        Error:      "Payment Required, Failed to add order",
    }, nil
}
// creating the order.
var order model.Order
order.RestaurantId = restaurant.ID
order.TotalPrice = request.OrderTotalPrice
order.UserId = user.ID
order.Status = "Pending"
order.Discount = request.OrderDiscount
order.ShippingAddress = request.ShippingAddress
// insert the order into the database.
orderDBConnector.Create(&order)
logger.Info("Order created", zap.Uint("orderId", order.ID))

// insert all the orderItems and using ordrid as foreign key.
for _, orderItem := range request.OrderItems {
    var orderItemModel model.OrderItem
    orderItemModel.OrderId = order.ID
    orderItemModel.Name = orderItem.OrderItemName
    orderItemModel.Price = orderItem.OrderItemPrice
    orderItemModel.Quantity = orderItem.OrderItemQuantity
```

```
        orderItemDBConnector.Create(&orderItemModel)
        logger.Info("Order item created", zap.Uint("orderId", order.ID), zap.String("itemName",
orderItem.OrderItemName))
    }
    logger.Info("Order added successfully", zap.Uint("orderId", order.ID))
    return &orderpb.AddOrderResponse{
        Message:    "Order added successfully",
        StatusCode: StatusOK,
        Error:      "",
        Data: &orderpb.Data{
            Order: []*orderpb.Order{
                {
                    OrderId:         strconv.FormatUint(uint64(order.ID), 10),
                    OrderItems:      request.OrderItems,
                    OrderTotalPrice: order.TotalPrice,
                    RestaurantName:  restaurant.Name,
                    OrderStatus:     "Pending 🕒",
                    ShippingAddress: request.ShippingAddress,
                },
            },
        },
    }, nil
}
```

1. The AddOrder method takes a context and an AddOrderRequest as parameters.
2. It retrieves the user's email and role from the context. If it fails, it logs an error and returns a response with an error message.

3. It checks if the user's role is not UserRole. If it's not, it logs a warning and returns a response indicating that the user doesn't have permission to add an order.

It validates the order fields. If they're invalid, it logs a warning and returns a response indicating that the request is invalid.

4. It connects to the user and restaurant databases. If it fails, it logs an error and returns a response indicating that the server is facing issues.
5. It fetches the user and restaurant from the databases. If it fails to find either, it logs a warning and returns a response indicating that the resource was not found.
6. It calls a payment gateway function. If the payment fails, it logs a warning and returns a response indicating that the payment is required.
7. It creates an order and inserts it into the database.
8. It loops through the order items in the request, creates an OrderItem for each, and inserts them into the database.
9. It logs that the order was added successfully and returns a response indicating that the order was added successfully.

This method uses the zap library for structured, leveled logging, and it uses the gorm library for interacting with the database.

10. app.html file

```
<html>
<button id="rzp-button1">Pay</button>
<style>
   @import
url('https://fonts.googleapis.com/css2?family=Roboto:wght@400;700&display=swap');

   body, html {
      height: 100%;
```

```
        margin: 0;
        display: flex;
        justify-content: center;
        align-items: center;
        background: linear-gradient(135deg, #6a11cb 0%, #2575fc 100%);
        font-family: 'Roboto', sans-serif;
    }

    button {
        padding: 15px 30px;
        font-size: 18px;
        font-weight: 700;
        cursor: pointer;
        background-color: #ff6b6b;
        color: white;
        border: none;
        border-radius: 50px;
        box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);
        transition: all 0.3s ease;
    }

    button:hover {
        background-color: #ff4757;
        box-shadow: 0 6px 10px rgba(0, 0, 0, 0.15);
        transform: translateY(-2px);
    }

    button:focus {
        outline: none;
    }
</style>
<script src="https://checkout.razorpay.com/v1/checkout.js"></script>
```

```
<script>
var options = {
    "key": "rzp_test_p4bR8DXSKNi8tJ", // Enter the Key ID generated from the Dashboard
    "amount": '{{.Amount}}', // Amount is in currency subunits. Default currency is INR. Hence,
50000 refers to 50000 paise
    "currency": "INR",
    "name": "Meal Mingle",
    "description": "Promote post",
    "image": "https://example.com/your_logo",
    "order_id": '{{.OrderId}}', //This is a sample Order ID. Pass the `id` obtained in the response of
Step 1
    "handler": function (response){
        a = (response.razorpay_payment_id);
        b =(response.razorpay_order_id);
        c = (response.razorpay_signature)
        window.location.replace("http://localhost:8089/payment-complete");
    },
    "prefill": {
        "name": '{{.Name}}',
        "email": '{{.Email}}',
        "contact": '{{.Contact}}'
    },
    "notes": {
        "address": "Razorpay Corporate Office"
    },
    "theme": {
        "color": "#3399cc"
    }
};
var rzp1 = new Razorpay(options);
rzp1.on('payment.failed', function (response){
        (response.error.code);
```

```
        (response.error.description);
        (response.error.source);
        (response.error.step);
        (response.error.reason);
        (response.error.metadata.order_id);
        (response.error.metadata.payment_id);
        window.location.replace("http://localhost:8089/payment-fail");

});
document.getElementById('rzp-button1').onclick = function(e){
    rzp1.open();
    e.preventDefault();
}
</script>
</html>
```

1. Abutton with the id rzp-button1 is created. When clicked, it will trigger the Razorpay payment process.

2. The CSS styles are defined for the button and the page.

3. The Razorpay checkout library is included using a script tag.

4. A JavaScript block is created where the Razorpay payment options are defined. These options include the key, amount, currency, name, description, image, order_id, and handler function.

5. The handler function is called when the payment is successful. It captures the payment id and order id from the response and redirects the user to the payment-complete page.

6. The prefill option is used to prefill the customer's information like name, email, and contact.

7. The notes option is used to add additional data to the payment.

8. The theme option is used to customize the color of the payment form.

9. A new Razorpay instance is created with the defined options.

10. An event listener is added to the payment.failed event. If the payment fails, it captures the error details from the response and redirects the user to the payment-fail page.
11. An onclick event is added to the button. When the button is clicked, the Razorpay payment form is opened.

> In the JavaScript block of the HTML code, there's an option called "key". This is your Razorpay key ID, which is used to authenticate your application with Razorpay's servers.
>
> To generate your own key:
>
> 1. Log in to your Razorpay Dashboard.
> 2. Navigate to Settings > API Keys > Generate New Key.
> 3. After the key is generated, the Key ID and Key Secret will be displayed. The Key Secret should be stored securely on your server and should not be shared with anyone.
> 4. Replace the "key" value in the JavaScript block with your own Key ID.

11. cancelOrder.go file

```
package main

import (
    "context"
    "order-microservice/config"
    "order-microservice/model"
    orderpb "order-microservice/proto/order"

    "go.uber.org/zap"
)

func (*OrderService) CancelOrder(ctx context.Context, request *orderpb.CancelOrderRequest)
(response *orderpb.CancelOrderResponse, err error) {
    // userEmail from the context
    userEmail, ok := ctx.Value("userEmail").(string)
    if !ok {
        logger.Error("Failed to get user email from context")
        return &orderpb.CancelOrderResponse{
            Message:    "Failed to get user email from context",
            Error:      "Internal Server Error",
            StatusCode: StatusInternalServerError,
        }, nil
    }
    // Validate the fields
    if request.OrderId == "" {
        logger.Warn("Invalid Order ID", zap.String("userEmail", userEmail))
        return &orderpb.CancelOrderResponse{
            Message:    "Invalid Order ID",
            Error:      "Invalid fields, make sure to use mentioned format.",
            StatusCode: StatusBadRequest,
        }, nil
```

```
}

//check if the order exists
var order model.Order
orderID := request.OrderId

res := orderDBConnector.Where("id = ?", orderID).First(&order)
if res.Error != nil {
    logger.Warn("Order not found", zap.String("orderID", orderID))
    return &orderpb.CancelOrderResponse{
        Message:    "Order not found with the given ID",
        Error:      "Not Found",
        StatusCode: StatusNotFound,
    }, nil
}

// check if the order exists or order belongs to the user
var user model.User
user.Email = userEmail
userDBConnector, userDbErr := config.GetUserConnector()
if userDbErr != nil {
    logger.Error("Failed to connect to user database", zap.Error(userDbErr))
    return &orderpb.CancelOrderResponse{
        Message:    "Server facing issues",
        StatusCode: StatusInternalServerError,
        Error:      "Internal Server Error",
    }, nil
}

userDBConnector.Where("email = ?", user.Email).First(&user)
if order.UserId != user.ID {
```

```
        logger.Warn("Unauthorized order cancellation attempt", zap.String("userEmail",
userEmail), zap.String("orderID", orderID))
        return &orderpb.CancelOrderResponse{
            Message:    "Order does not belong to the user, you are not authorized to cancel this
order.",
            Error:      "Unauthorized",
            StatusCode: StatusUnauthorized,
        }, nil
    }

    // delete the user..
    res = orderDBConnector.Delete(&order)
    if res.Error != nil {
        logger.Error("Failed to delete the order", zap.String("orderID", orderID))
        return &orderpb.CancelOrderResponse{
            Message:    "Failed to delete the order",
            Error:      "Internal server error",
            StatusCode: StatusInternalServerError,
        }, nil
    }
    logger.Info("Order cancelled successfully", zap.String("orderID", orderID))
    return &orderpb.CancelOrderResponse{
        Data:       &orderpb.CancelOrderData{OrderId: request.OrderId},
        Message:    "Order Cancelled Successfully",
        Error:      "",
        StatusCode: StatusOK,
    }, nil
}
```

1. The CancelOrder method takes a context and a CancelOrderRequest as parameters.

2. It retrieves the user's email from the context. If it fails, it logs an error and returns a response with an error message.

3. It validates the order ID field. If it's invalid, it logs a warning and returns a response indicating that the request is invalid.

4. It checks if the order exists in the database. If it doesn't, it logs a warning and returns a response indicating that the order was not found.

5. It connects to the user database. If it fails, it logs an error and returns a response indicating that the server is facing issues.

6. It fetches the user from the database. If the order doesn't belong to the user, it logs a warning and returns a response indicating that the user is not authorized to cancel the order.

7. It deletes the order from the database. If it fails, it logs an error and returns a response indicating that the server is facing issues.

8. It logs that the order was cancelled successfully and returns a response indicating that the order was cancelled successfully.

11. changeOrderStatus.go file

```
package main

import (
    "context"
    "order-microservice/config"
    "order-microservice/model"
    orderpb "order-microservice/proto/order"

    "go.uber.org/zap"
)
```

```
func (*OrderService) ChangeOrderStatus(ctx context.Context, request
*orderpb.ChangeOrderStatusRequest) (*orderpb.ChangeOrderStatusResponse, error) {
    // get the user mail from the context
    userEmail, ok := ctx.Value("userEmail").(string)
    if !ok {
        logger.Error("Failed to get user email from context")
        return &orderpb.ChangeOrderStatusResponse{
            Data:       nil,
            Message:    "Failed to get user email from context",
            StatusCode: StatusInternalServerError,
            Error:      "Internal Server Error",
        }, nil
    }

    // Validate the request
    if request.OrderId == "" || request.OrderStatus == "" {
        logger.Warn("Invalid request fields", zap.String("userEmail", userEmail))
        return &orderpb.ChangeOrderStatusResponse{
            Data:       nil,
            Message:    "Invalid request fields",
            StatusCode: StatusBadRequest,
            Error:      "Bad Request",
        }, nil
    }
    // get the user from the database
    var user model.User
    userDBConnector, userDbErr := config.GetUserConnector()
    if userDbErr != nil {
        logger.Error("Failed to connect to user database", zap.Error(userDbErr))
        return &orderpb.ChangeOrderStatusResponse{
            Data:       nil,
```

```
        Message:    "Server facing issues",
        StatusCode: StatusInternalServerError,
        Error:      "Internal Server Error",
    }, nil
}
if err := userDBConnector.Where("email = ?", userEmail).First(&user).Error; err != nil {
    logger.Warn("User not found", zap.String("userEmail", userEmail))
    return &orderpb.ChangeOrderStatusResponse{
        Data:       nil,
        Message:    "User not found",
        StatusCode: StatusNotFound,
        Error:      "Not Found",
    }, nil
}

// check if the order exists
var order model.Order
orderErr := orderDBConnector.Where("id = ?", request.OrderId).First(&order).Error

if orderErr != nil || order.UserId != user.ID {
    logger.Warn("Unauthorized attempt to change order status",
        zap.String("userEmail", userEmail),
        zap.String("orderID", request.OrderId))
    return &orderpb.ChangeOrderStatusResponse{
        Data:       nil,
        Message:    "Not Authorized to change the order status",
        StatusCode: StatusUnauthorized,
        Error:      "Unauthorized",
    }, nil
}
// change the status of the order
order.Status = request.OrderStatus
```

```
    saveErr := orderDBConnector.Save(&order).Error

    if saveErr != nil {
        logger.Error("Failed to save order status", zap.String("orderID", request.OrderId),
zap.Error(saveErr))
        return &orderpb.ChangeOrderStatusResponse{
            Data:       nil,
            Message:    "Failed to change order status",
            StatusCode: StatusInternalServerError,
            Error:      "Internal Server Error",
        }, nil
    }
    logger.Info("Order status changed successfully",
        zap.String("orderID", request.OrderId), zap.String("newStatus", request.OrderStatus))
    // return the response
    return &orderpb.ChangeOrderStatusResponse{
        Data: &orderpb.ChangeOrderStatusResponseData{
            OrderId:     request.OrderId,
            OrderStatus: request.OrderStatus,
        },
        Message:    "Order status changed successfully",
        StatusCode: StatusOK,
        Error:      "",
    }, nil
}
```

1. The method takes a context and a ChangeOrderStatusRequest as parameters. The request contains the order ID and the new status.

2. It tries to get the user's email from the context. If it fails, it logs an error and returns a ChangeOrderStatusResponse with an internal server error status.

3. It validates the request. If the order ID or the new status is empty, it logs a warning and returns a ChangeOrderStatusResponse with a bad request status.

4. It connects to the user database and retrieves the user with the given email. If it fails to connect to the database or find the user, it logs an error or warning and returns a ChangeOrderStatusResponse with an internal server error or not found status, respectively.

5. It checks if the order exists and belongs to the user. If it doesn't, it logs a warning and returns a ChangeOrderStatusResponse with an unauthorized status.

6. It changes the status of the order and saves it. If it fails to save the order, it logs an error and returns a ChangeOrderStatusResponse with an internal server error status.

7. If everything goes well, it logs an info message and returns a ChangeOrderStatusResponse with the order ID, the new status, a success message, and an OK status.

This uses the zap library for structured, leveled logging, and it seems to use a custom ORM for database operations. The ChangeOrderStatusRequest and ChangeOrderStatusResponse types are probably defined in the orderpb package, which is likely generated from a Protocol Buffers definition.

12. orderHistory.go file

```
package main

import (
    "context"
    "order-microservice/config"
    "order-microservice/model"
```

```
    orderpb "order-microservice/proto/order"
    "strconv"

    "go.uber.org/zap"
)

func (*OrderService) OrderHistory(ctx context.Context, request *orderpb.OrderHistoryRequest)
(*orderpb.OrderHistoryResponse, error) {
    // fetch the user email from the context
    userEmail, ok := ctx.Value("userEmail").(string)
    if !ok {
        logger.Error("Failed to get user email from context")
        return &orderpb.OrderHistoryResponse{
            Message:    "Server facing issues",
            StatusCode: StatusInternalServerError,
            Error:      "Internal Server Error",
        }, nil
    }
    // fetch the user from the user email
    var user model.User
    user.Email = userEmail
    userDBConnector, err := config.GetUserConnector()
    if err != nil {
        logger.Error("Failed to connect to user database", zap.Error(err))
        return &orderpb.OrderHistoryResponse{
            Message:    "Server facing issues",
            StatusCode: StatusInternalServerError,
            Error:      "Internal Server Error",
        }, nil
    }
    userDBConnector.Where("email = ?", user.Email).First(&user)
    restaurantDBConnector, err := config.GetRestaurantConnector()
```

```
    if err != nil {
        logger.Error("Failed to connect to user database", zap.Error(err))
        return &orderpb.OrderHistoryResponse{
            Message:    "Server facing issues",
            StatusCode: StatusInternalServerError,
            Error:      "Internal Server Error",
        }, nil
    }

    // fetch the orders from the user id
    var orders []model.Order
    orderDBConnector.Where("user_id = ?", user.ID).Find(&orders)
    // create the response
    var orderHistoryResponse orderpb.OrderHistoryResponse
    orderHistoryResponse.Data = &orderpb.Data{}
    for _, order := range orders {
        // for every order we are fetching the order items
        var orderItems []model.OrderItem
        var restaurant model.Restaurant
        orderItemDBConnector.Where("order_id = ?", order.ID).Find(&orderItems)
        restaurantDBConnector.Where("id = ?", order.RestaurantId).First(&restaurant)
        var orderItemsResponse []*orderpb.OrderItem
        for _, orderItem := range orderItems {
            orderItemsResponse = append(orderItemsResponse, &orderpb.OrderItem{
                OrderItemName:     orderItem.Name,
                OrderItemPrice:    orderItem.Price,
                OrderItemQuantity: orderItem.Quantity,
            })
        }
        orderHistoryResponse.Data.Order = append(orderHistoryResponse.Data.Order,
&orderpb.Order{
            OrderId:        strconv.FormatUint(uint64(order.ID), 10),
```

```
            OrderItems:      orderItemsResponse,
            OrderTotalPrice: order.TotalPrice,
            RestaurantName:  restaurant.Name,
            OrderStatus:     order.Status,
            ShippingAddress: order.ShippingAddress,
        })
    }
    // Set success message and status code
    orderHistoryResponse.Message = "Successfully fetched order history"
    orderHistoryResponse.StatusCode = StatusOK
    return &orderHistoryResponse, nil
}
```

1. The method takes a context and an OrderHistoryRequest as parameters.
2. It tries to get the user's email from the context. If it fails, it logs an error and returns an OrderHistoryResponse with an internal server error status.
3. It connects to the user database and retrieves the user with the given email. If it fails to connect to the database, it logs an error and returns an OrderHistoryResponse with an internal server error status.
4. It connects to the restaurant database. If it fails to connect, it logs an error and returns an OrderHistoryResponse with an internal server error status.
5. It retrieves all orders for the user from the order database.
6. It initializes an OrderHistoryResponse and its Data field.
7. For each order, it retrieves the order items from the order item database and the restaurant from the restaurant database.

8. It creates an OrderItem for each order item and appends it to the OrderItemsResponse.

9. It creates an Order with the order ID, the order items, the total price, the restaurant name, the order status, and the shipping address, and appends it to the Order field of the Data field of the OrderHistoryResponse.

10. It sets the message and the status code of the OrderHistoryResponse to indicate success.

11. It returns the OrderHistoryResponse.

13. Dockerfile

```
# Base image
FROM golang:1.22.2-alpine3.19

# Move to working directory /app
WORKDIR /app

# Copy the code into the container
COPY . .

# Download dependencies using go mod
RUN go mod tidy && go mod vendor

# Expose PORT 8093 for the order microservice grpc gateway
EXPOSE 8093
# Expose PORT 8089 for payment gateway.
EXPOSE 8089
# Command to run the application when starting the container
CMD ["go","run","."]
```

- **FROM golang:1.22.2-alpine3.19:** This line specifies the base image to be used. In this case, it's an image with Go version 1.22.2 and Alpine Linux version 3.19.

- **WORKDIR /app**: This line sets the working directory inside the Docker container to /app.

- **COPY . .:** This line copies the current directory (where the Dockerfile is located) on your local machine into the current directory inside the Docker container (which is /app due to the previous WORKDIR command).

- **RUN go mod tidy && go mod vendor**: This line runs two commands inside the Docker container. go mod tidy ensures that the go.mod file matches the source code in the module. It adds missing modules necessary to build the current module's packages and dependencies, and it removes unused modules. go mod vendor creates a vendor directory in the current module and populates it with copies of all packages needed to support builds and tests of packages in the current module.

- **EXPOSE 8093:** This line tells Docker that the container listens on the specified network port at runtime. In this case, it's port 8093 for the order microservice gRPC gateway.

- **EXPOSE 8089:** This line tells Docker that the container also listens on port 8089 for the payment gateway.

- **CMD ["go","run","."]:** This line provides defaults for an executing container. In this case, it runs the command go run . which compiles and runs the Go application. The . means it will run the main package in the current directory.

Go to order-microservice: - **cd order-micro**
Build docker file: - **docker build -t order-microservice .**

Run docker container: - **docker container run -p 8093:8093 -p 8089:8089 -e MYSQL_HOST=host.docker.internal order-microservice**

docker-compose-local.yml (Reference >> Link)

```yaml
version: '3.8'

services:
  database:
    image: mysql:8
    environment:
      MYSQL_ROOT_PASSWORD: <password>
      MYSQL_DATABASE: test
    volumes:
      - mysql-data:/var/lib/mysql
    ports:
      - "3307:3306"
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "--silent"]
      interval: 30s
      timeout: 10s
      retries: 5
    networks:
      - my_network

  auth-micro:
    build:
      context: ./auth-micro
    environment:
      MYSQL_HOST: database
      MYSQL_USER: root
```

```
    MYSQL_PASSWORD: <password>
    MYSQL_DATABASE: test
    MYSQL_PORT: 3306
  networks:
    - my_network
  ports:
    - "8090:8090"
  depends_on:
    database:
      condition: service_healthy

order-micro:
  build:
    context: ./order-micro
  environment:
    MYSQL_HOST: database
    MYSQL_USER: root
    MYSQL_PASSWORD: <password>
    MYSQL_DATABASE: test
    MYSQL_PORT: 3306
  networks:
    - my_network
  ports:
    - "8093:8093"
    - "8089:8089"
  depends_on:
    database:
      condition: service_healthy

restaurant-micro:
  build:
    context: ./restaurant-micro
```

```
    environment:
      MYSQL_HOST: database
      MYSQL_USER: root
      MYSQL_PASSWORD: <password>
      MYSQL_DATABASE: test
      MYSQL_PORT: 3306
    networks:
      - my_network
    ports:
      - "8091:8091"
    depends_on:
      database:
        condition: service_healthy

volumes:
  mysql-data:

networks:
  my_network:
    driver: bridge
```

Make sure to replace <password> with local mysql db password, Reasons are: -

Consistency Across Environments:

- Simplified Configuration: Easier management of database settings.
- Reduced Errors: Minimizes credential mismatch issues between environments.

Streamlined Development and Testing:

- Easier Switching: Quick transition between local and containerized MySQL.
- Simplified Automation: Unified credentials simplify scripts and CI/CD integration.

- Docker Compose file defines four services: database, auth-micro, order-micro, and restaurant-micro.

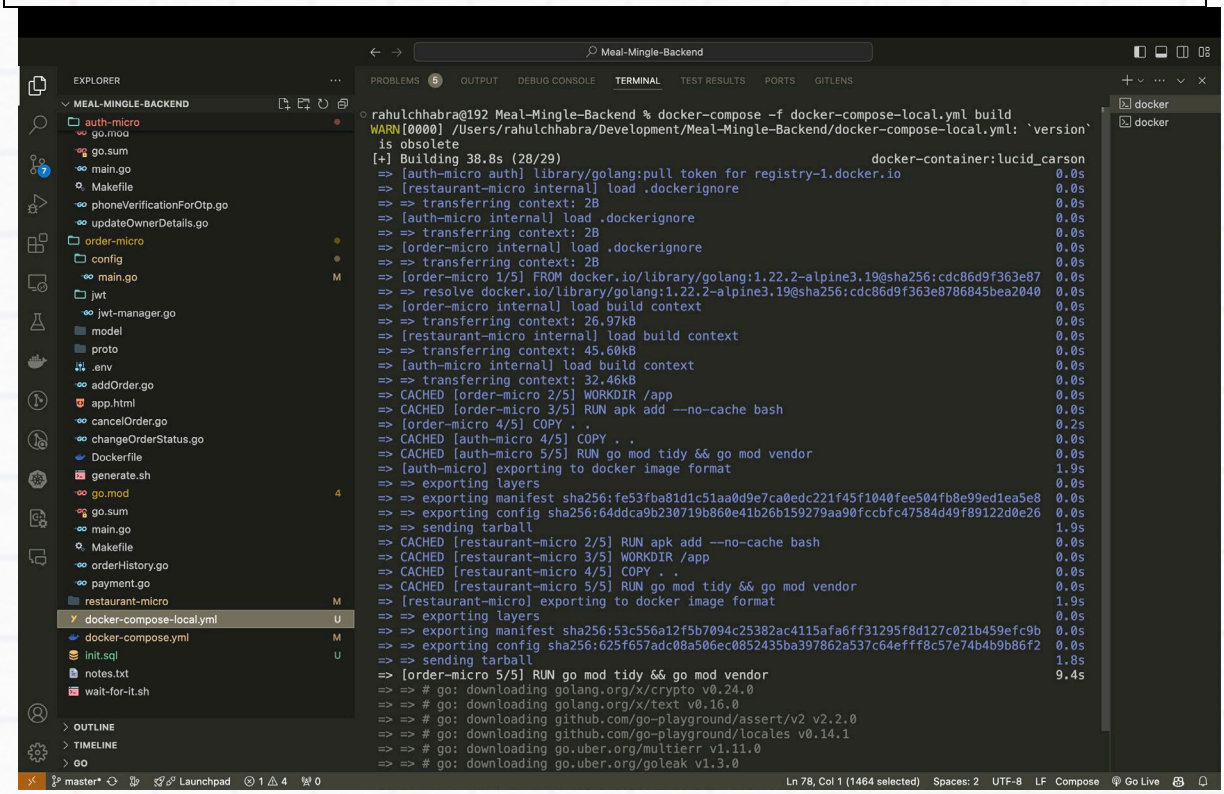
- **database**: This service uses the mysql:8 image to create a MySQL database container. The root password and database name are set through environment variables. The database data is stored in a Docker volume named mysql-data to persist data across container restarts. The container's MySQL port (3306) is mapped to port 3307 on the host machine. A health check is configured to periodically check if the MySQL server is running.

- auth-micro, order-micro, restaurant-micro: These services are built from Dockerfiles located in the ./auth-micro, ./order-micro, and ./restaurant-micro directories respectively. They connect to the database service using the provided MySQL credentials. Each service is exposed on a different port on the host machine (8090 for auth-micro, 8091 for order-micro, and 8093 for restaurant-micro). They all depend on the database service and will only start after the database service is healthy.

- All services are connected through a custom network named my_network, which is created with the bridge driver. This allows the services to communicate with each other.

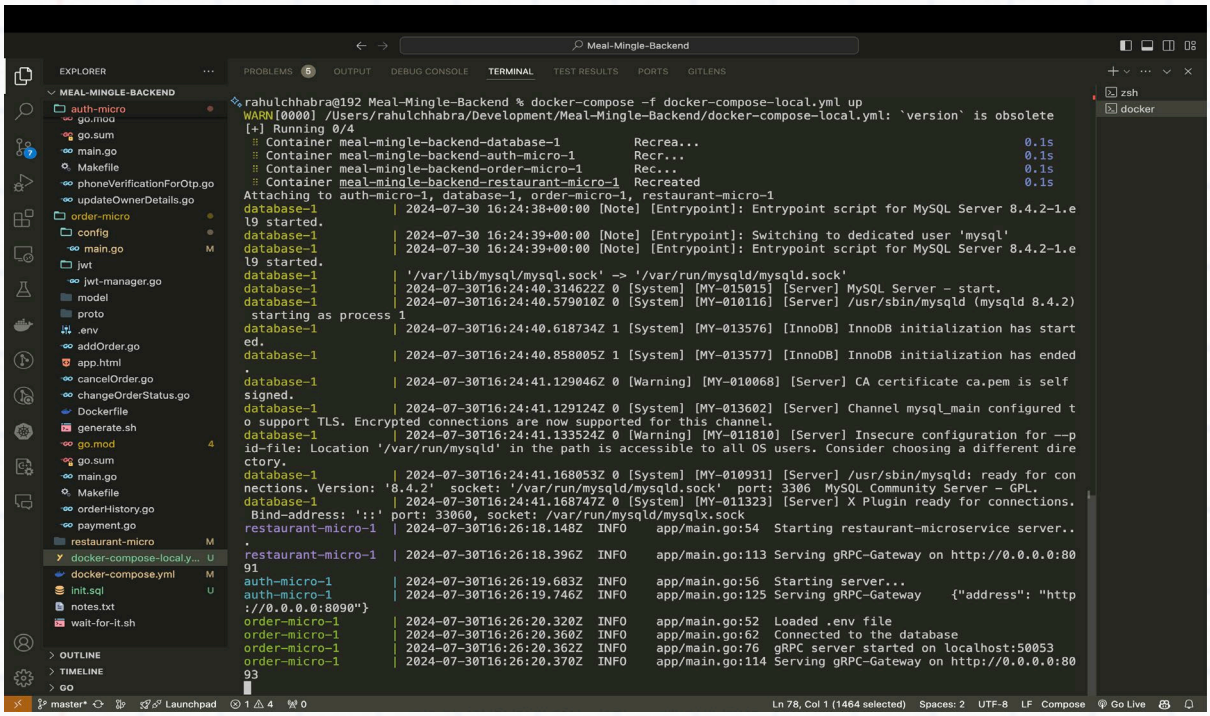
- The volumes section defines the mysql-data volume used by the database service, and the networks section defines the my_network network used by all services.

Build this docker compose file



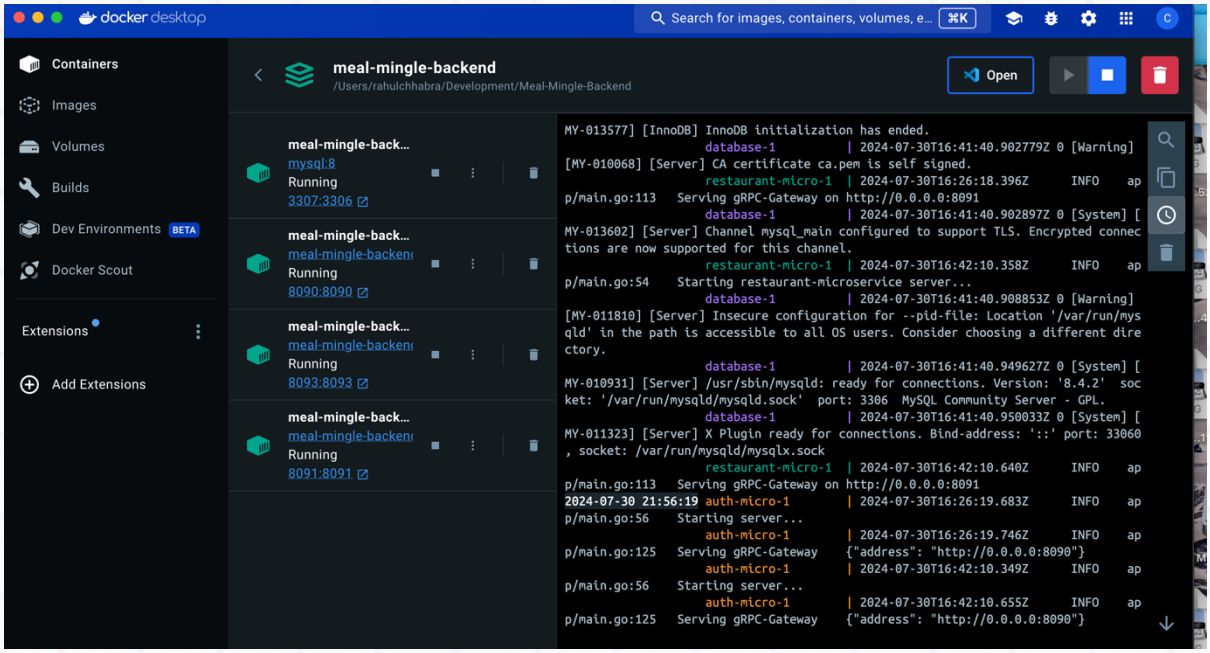
Command Used: -
docker-compose -f docker-compose-local.yml build

Run the containers

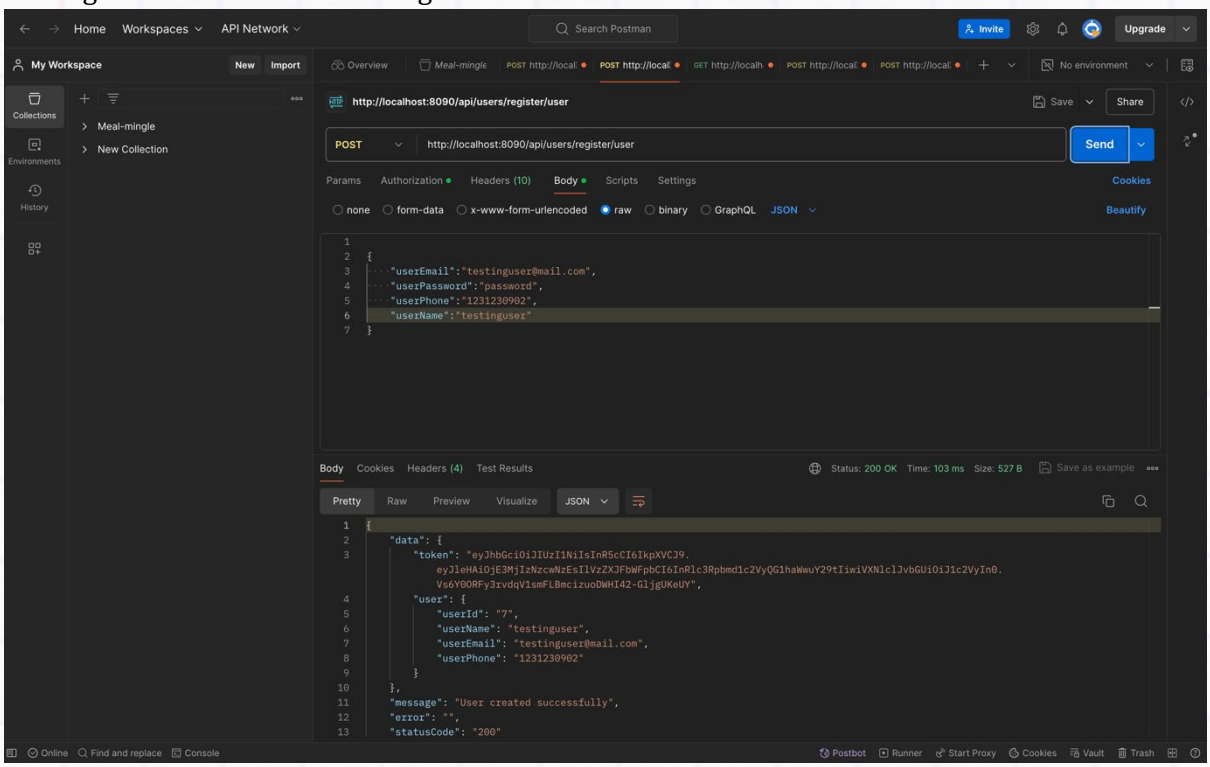


Command used: -
docker-compose -f docker-compose-local.yml up

We can verify all the running container from docker desktop



Testing the microservices running inside the container.



Deploying Containers to AWS

1. All About AWS.

   Reference >> Link

2. What is Cloud Computing

Reference >> Link

3. All About Amazon Elastic Container Service

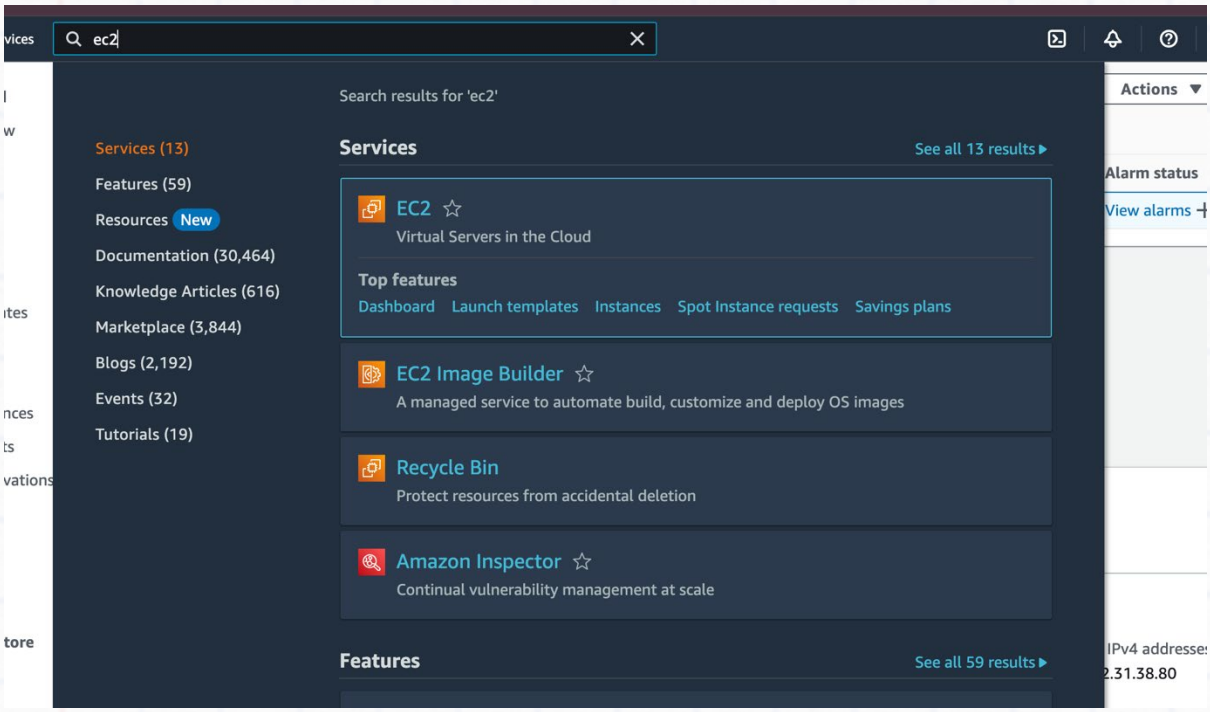
Reference >> Link

4.Create a new free aws account (step by step guide)

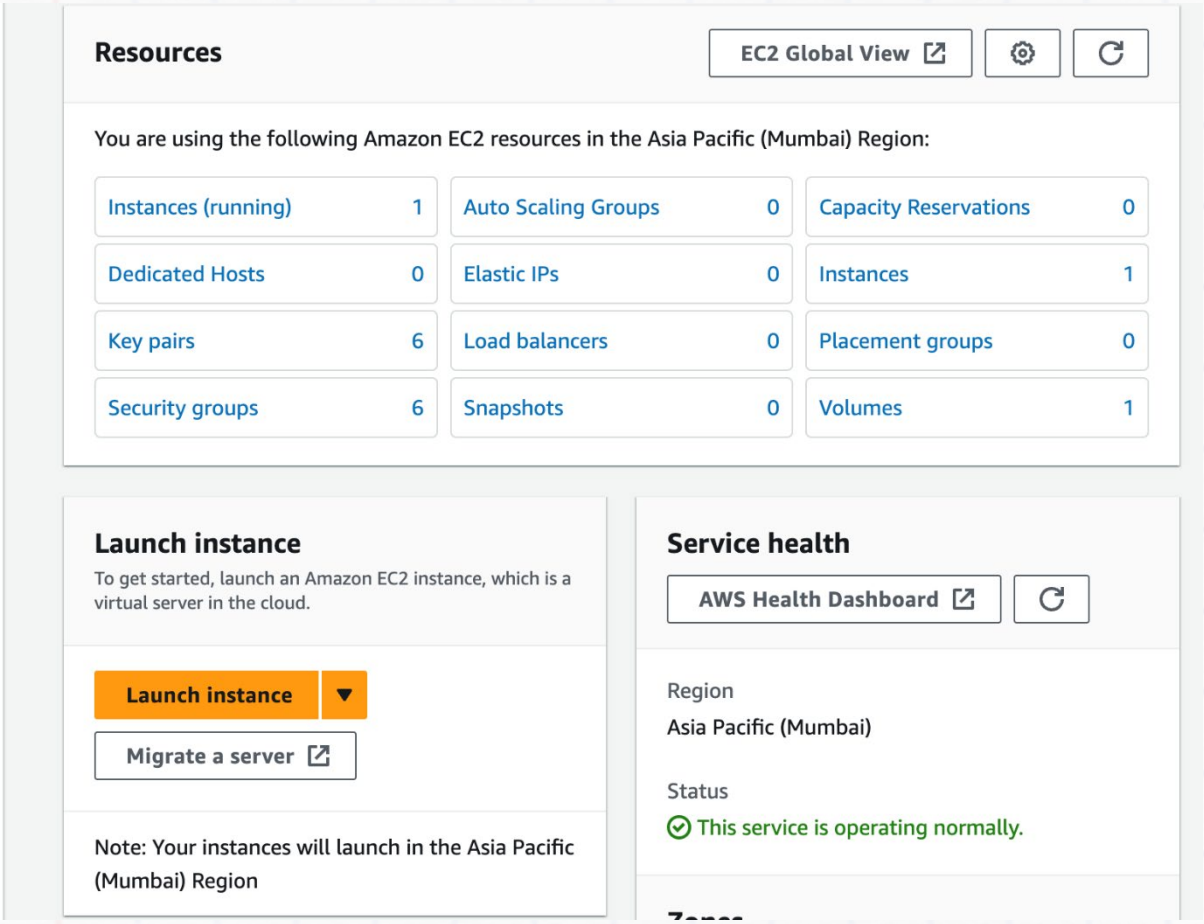
   Reference >> Link

5. Launch a new EC2 Instance.

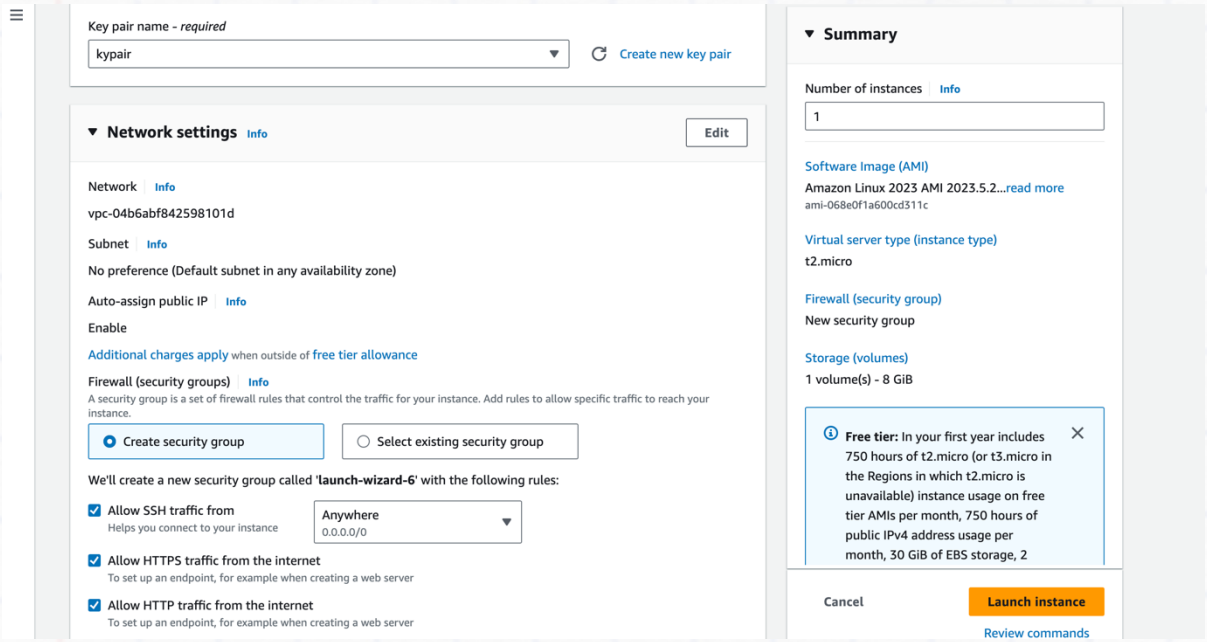
1. Go to Console Home and Search ec2.



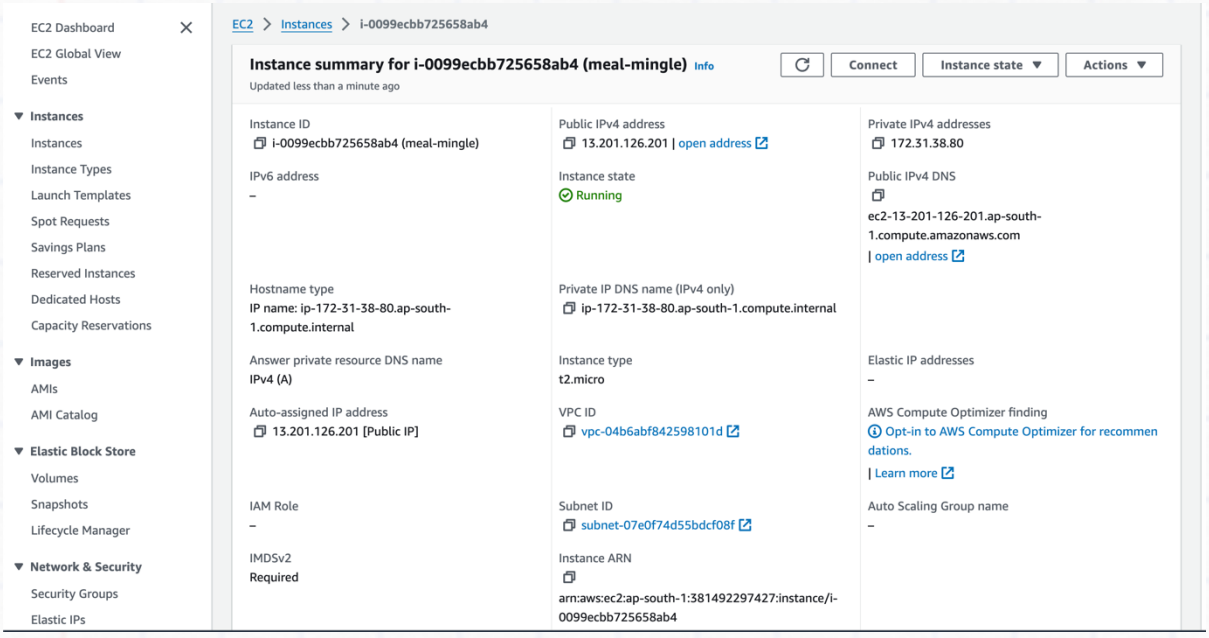
2. Go to EC2 and click on Launch Instance.



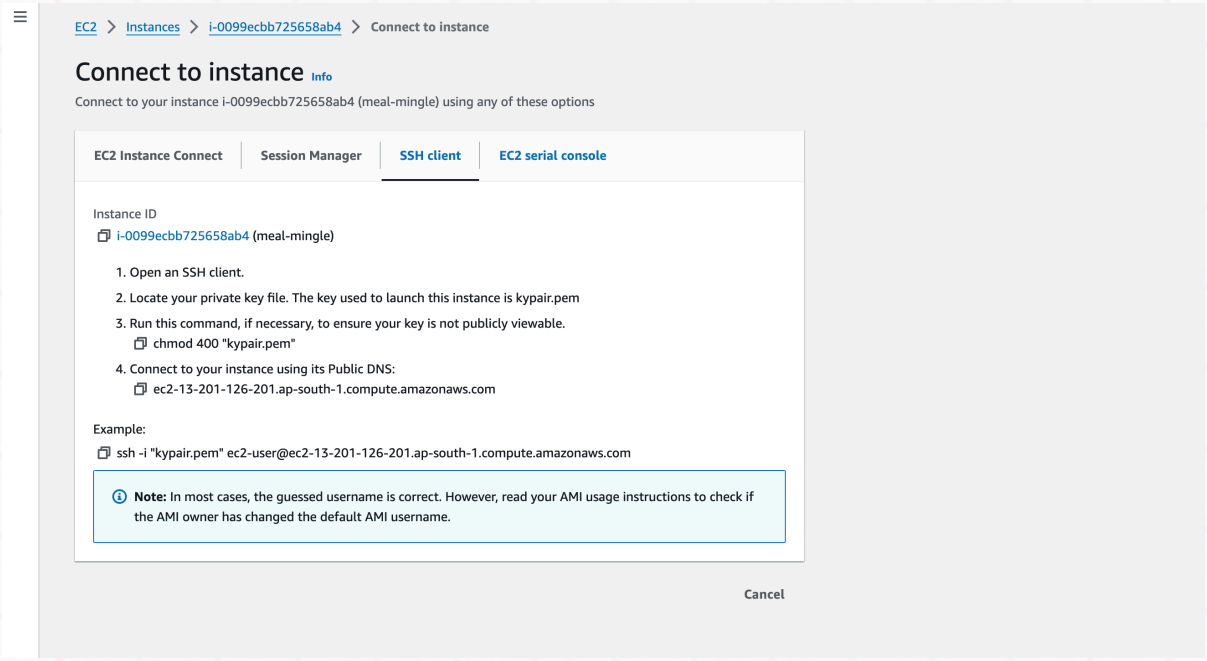
3. Leave all the setting as it is only generate key pair (In my case my named by key pair as kypair) and check all these below options then click on Launch Instance



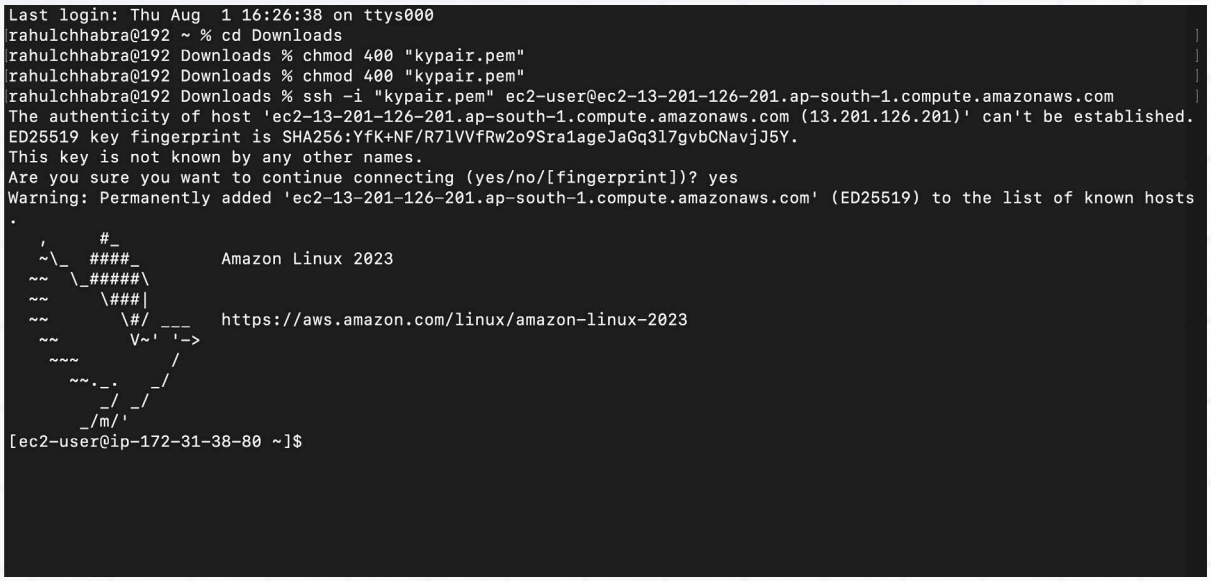
4. Click On connect



5. Go to SSH client



6. open the terminal and navigate to the folder where your kypair.pem file stored (In my case it is downloads folder) and paste the above two highlighted commands in the sequence.

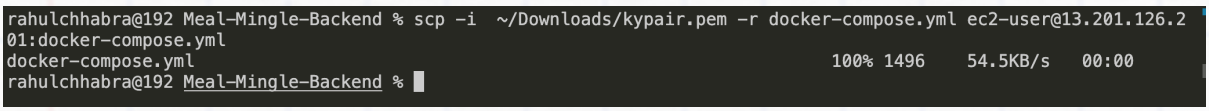


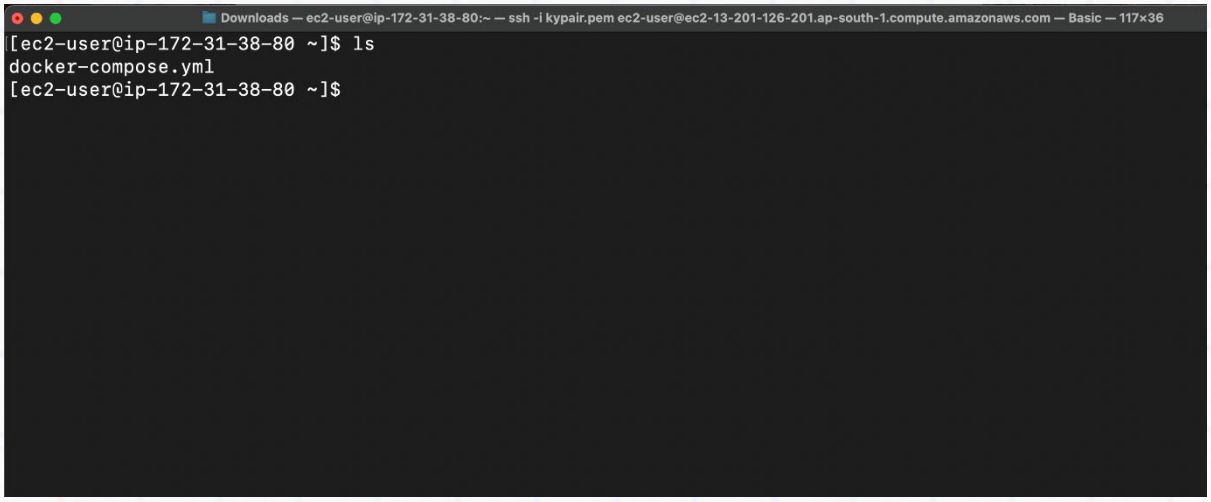
7. Go to the project and open vsc integrated terminal and paste the command

```
scp -i  <path-to-yourkypair.pem-file> -r docker-compose.yml ec2-user@<IPAddress>:docker-compose.yml
```

Replace <IPAddress> with the public IP address of your instance.

Replace <path-to-yourkypair.pem-file> with the path of your kypair.pem file.



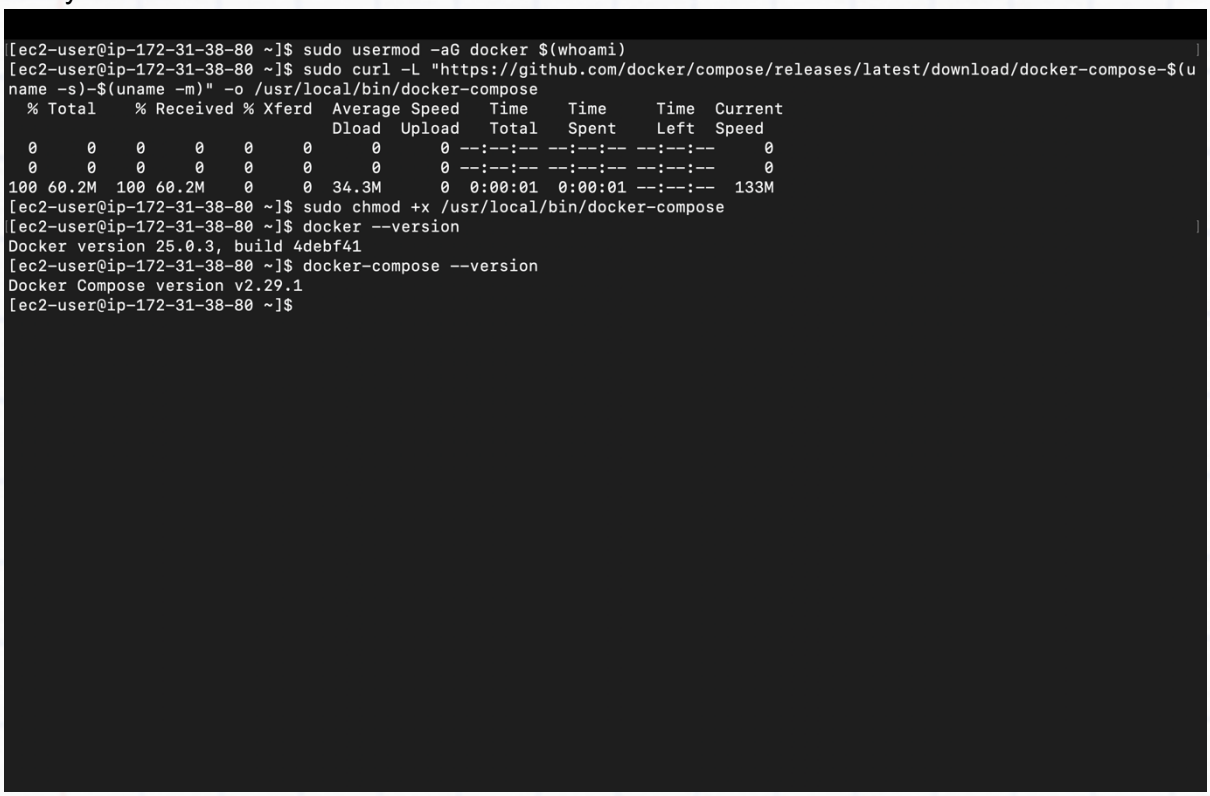


8. Install docker and docker-compose in ec2 instance.

```
1. Update the Package Repository
-------------------------------
sudo yum update -y

2. Install Docker Using the Yum Package Manager
----------------------------------------------
sudo yum install docker -y

3. Start Docker Service and Enable on Boot
-----------------------------------------
sudo service docker start
sudo systemctl enable docker

4. Add User to Docker Group (Optional)
-------------------------------------
sudo usermod -aG docker $(whoami)

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

6. Apply Executable Permissions
------------------------------
sudo chmod +x /usr/local/bin/docker-compose
```

7. Verify Installation
---
docker --version
docker-compose --version

Verify the Installation.

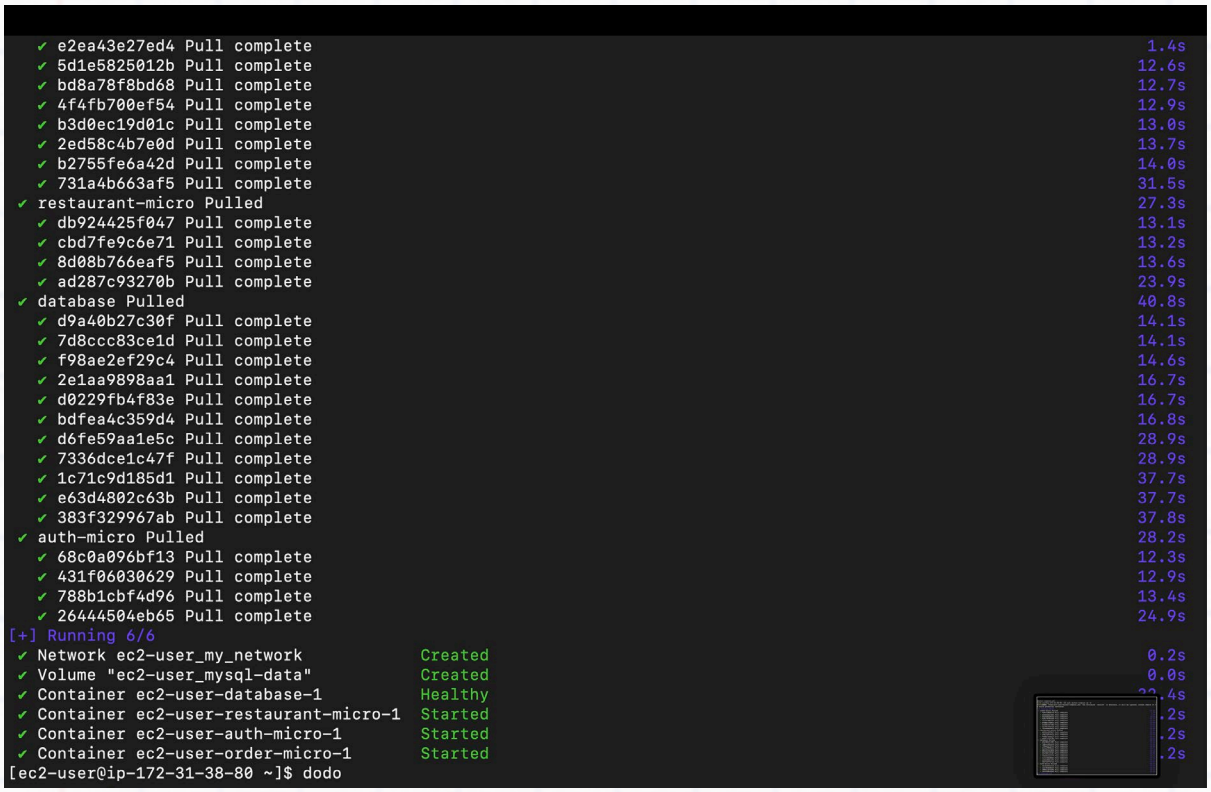


9. Final Step, Run the containers inside the ec2 Instance.

Command Used >> sudo docker-compose up -d

```
docker-compose.yml
[ec2-user@ip-172-31-38-80 ~]$ sudo docker-compose up -d
WARN[0000] /home/ec2-user/docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it t
o avoid potential confusion
[+] Running 32/32
 ✔ order-micro Pulled                                                                                              34.0s
   ✔ 4abcf2066143 Pull complete                                                                                     1.3s
   ✔ e2ea43e27ed4 Pull complete                                                                                     1.4s
   ✔ 5d1e5825012b Pull complete                                                                                    12.6s
   ✔ bd8a78f8bd68 Pull complete                                                                                    12.7s
   ✔ 4f4fb700ef54 Pull complete                                                                                    12.9s
   ✔ b3d0ec19d01c Pull complete                                                                                    13.0s
   ✔ 2ed58c4b7e0d Pull complete                                                                                    13.7s
   ✔ b2755fe6a42d Pull complete                                                                                    14.0s
   ✔ 731a4b663af5 Pull complete                                                                                    31.5s
 ✔ restaurant-micro Pulled                                                                                         27.3s
   ✔ db924425f047 Pull complete                                                                                    13.1s
   ✔ cbd7fe9c6e71 Pull complete                                                                                    13.2s
   ✔ 8d08b766eaf5 Pull complete                                                                                    13.6s
   ✔ ad287c93270b Pull complete                                                                                    23.9s
 ✔ database Pulled                                                                                                 40.8s
   ✔ d9a40b27c30f Pull complete                                                                                    14.1s
   ✔ 7d8ccc83ce1d Pull complete                                                                                    14.1s
   ✔ f98ae2ef29c4 Pull complete                                                                                    14.6s
   ✔ 2e1aa9898aa1 Pull complete                                                                                    16.7s
   ✔ d0229fb4f83e Pull complete                                                                                    16.7s
   ✔ bdfea4c359d4 Pull complete                                                                                    16.8s
   ✔ d6fe59aa1e5c Pull complete                                                                                    28.9s
   ✔ 7336dce1c47f Pull complete                                                                                    28.9s
   ✔ 1c71c9d185d1 Pull complete                                                                                    37.7s
   ✔ e63d4802c63b Pull complete                                                                                    37.7s
   ✔ 383f329967ab Pull complete                                                                                    37.8s
 ✔ auth-micro Pulled                                                                                               28.2s
   ✔ 68c0a096bf13 Pull complete                                                                                    12.3s
   ✔ 431f06030629 Pull complete                                                                                    12.9s
   ✔ 788b1cbf4d96 Pull complete                                                                                    13.4s
   ✔ 26444504eb65 Pull complete                                                                                    24.9s
[+] Running 6/6
```

10. You can see the containers up and running.



12. Make sure to stop the EC2 Instance, after using otherwise free tier will be exhauted earlier.