**Dev of a TO-DO list API**

Developing a To-Do List API involves a series of steps, from setting up the project to implementing the **CRUD** (Create, Read, Update, Delete) operations. Here's a step-by-step guide on how to develop a simple To-Do List API:

Step 1:-

We will be using the Go programming language. We be using the gin framework.

***go mod init <Module name>*** :- “To steup your module file”

***go mod tidy*** :- to remove unused dependencies in the module file.

***go get github.com/gin-gonic/gin :-***  to download the gin framework and all the dependency.

**Define Data Structures:** Define the data structures you'll be working with, such as a **Task** struct to represent individual to-do tasks. Include fields like **ID**, **Title**, **Description**, **DueDate**, and **Completed**.

**type task stuct {**

**ID int**

**Title string**

**Description string**

**DueDate int**

**Completed string**

**}**

**How setup a router with gin framework**

Sure, I'd be happy to guide you through setting up a router using the Gin web framework for your To-Do List API project. Here's a high-level overview of the steps involved:

1. \*\*Install Gin:\*\*

First, you need to install the Gin framework using Go's package manager. Open your terminal and run:

```

go get -u github.com/gin-gonic/gin

```

2. \*\*Import Gin:\*\*

In your Go code, import the Gin package:

```go

import "github.com/gin-gonic/gin"

```

3. \*\*Create the Router:\*\*

Create a new Gin router instance:

```go

r := gin.Default() // Creates a new Gin router with default middleware

```

4. \*\*Define Routes:\*\*

Define your API routes using the router's methods (`GET`, `POST`, `PUT`, `DELETE`, etc.) and provide the respective handler functions.

```go

r.GET("/tasks", getTasks)

r.POST("/tasks", createTask)

r.PUT("/tasks/:id", updateTask)

r.DELETE("/tasks/:id", deleteTask)

```

5. \*\*Handler Functions:\*\*

Implement the handler functions for each route. These functions take a `gin.Context` parameter that provides access to the request and response objects.

```go

func getTasks(c \*gin.Context) {

// Retrieve tasks from your data storage and return them as JSON

}

func createTask(c \*gin.Context) {

// Parse the request body and create a new task

}

func updateTask(c \*gin.Context) {

// Parse the request body and update the specified task

}

func deleteTask(c \*gin.Context) {

// Delete the specified task

}

```

6. \*\*Run the Server:\*\*

Finally, start the Gin HTTP server to listen for incoming requests:

```go

r.Run(":8080") // Listen on port 8080

```

Remember that the handler functions should perform the specific CRUD operations you've defined, interacting with your data storage (like a slice or database). The context (`c`) object gives you access to the request parameters, headers, and more.

Additionally, Gin provides features for route grouping, middleware application, and error handling. You can enhance your API by adding logging, authentication, and other middleware as needed.

Keep in mind that this is a simplified overview. For more detailed information and examples, refer to the official Gin documentation: https://github.com/gin-gonic/gin

**The gist about r := gin.Default()**

Sure, I'd be happy to explain!

In the context of the Gin framework for Go (Golang), `gin.Default()` is a function that creates a new instance of a Gin router with some default middleware already configured. Let's break it down:

1. \*\*Gin Router\*\*: A router is an essential component in web frameworks that helps you define routes for your application. Routes specify what action should be taken when a particular URL is accessed. For instance, you might want to show a specific page when users visit the root URL "/", and another page when they visit "/about".

2. \*\*Middleware\*\*: Middleware are functions that are executed before or after the main handler for a particular route. They are used for tasks such as logging, authentication, request/response modification, etc. Middleware can intercept requests and responses, allowing you to perform certain actions on them before they reach the actual route handler or after the handler has processed the request.

Now, when you use `gin.Default()`, here's what's happening:

- `gin`: This is the package name of the Gin framework.

- `Default()`: This is a function provided by Gin that returns a new instance of the router with default middleware preconfigured.

By default, the `gin.Default()` function sets up a few middleware components for you. These middleware components include:

- \*\*Logger Middleware\*\*: This middleware logs information about each incoming request, including the HTTP method, the URL, and the time taken to process the request.

- \*\*Recovery Middleware\*\*: This middleware recovers from any panics that might occur during request processing and returns a 500 Internal Server Error response instead of crashing the server.

Using `gin.Default()` is a convenient way to quickly set up a router with these common middleware components without having to configure them manually. However, you can always customize your middleware by using the `gin.New()` function to create a router instance and then adding your own middleware functions as needed.

Here's how you might create a router with custom middleware:

```go

package main

import (

"github.com/gin-gonic/gin"

"net/http"

)

func customMiddleware(c \*gin.Context) {

// Your custom middleware logic here

c.Next()

}

func main() {

r := gin.New()

// Use your custom middleware

r.Use(customMiddleware)

// ... Define routes and handlers ...

r.Run(":8080")

}

```

In this example, the `customMiddleware` function is your own middleware function that you want to apply to all routes. You add it using `r.Use(customMiddleware)` to ensure it's executed for every incoming request.

**How to test the api**

first run the api :- ***go run main.go***

1. **Get request:- *Invoke-RestMethod -Uri*** [***http://localhost:8080/tasks***](http://localhost:8080/tasks)
2. **Post request:-**

***$uri = "http://localhost:8080/tasks"***

***$headers = @{***

***"Content-Type" = "application/json"***

***}***

***$jsonPayload = @{***

***"id" = "4"***

***"title" = "Write Report"***

***"description" = "Write a quarterly report"***

***"duedate" = "2023-09-15T00:00:00Z"***

***"complete" = $false***

***}***

***$response = Invoke-RestMethod -Uri $uri -Method Post -Headers $headers -Body ($jsonPayload | ConvertTo-Json)***

***$response***

1. **Update request:-**

***$uri = "http://localhost:8080/tasks/2"***

***$headers = @{***

***"Content-Type" = "application/json"***

***}***

***$jsonBody = @{***

***"id" = "8"***

***"title" = "Blender Retopology"***

***"description" = "Complete Blender retopology"***

***"duedate" = "2023-08-30T00:00:00Z"***

***"complete" = $true***

***} | ConvertTo-Json***

***$response = Invoke-RestMethod -Uri $uri -Method Put -Headers $headers -Body $jsonBody***

***$response***

1. **Delete request:-**

***Invoke-RestMethod -Uri http://localhost:8080/tasks/4 -Method Delete***