# Golanguage API

# Setup

1. Download and Install Go from URL <https://golang.org/dl/>
2. Set environment variables

set PATH environment variable like C:\Program Files\Go\bin.

Set GOPATH to Go project workspace directory, e.g., `C:\Users\Tanvi\go

1. Install git from URL <https://git-scm.com/download/>
2. Setup the go project
   1. Run below command to create project directory
      1. mkdir -p %GOPATH%\src\github.com\sirisha\api-go-project
      2. cd %GOPATH%\src\github.com\sirisha\api-go-project
   2. Initialize a new Go module
      1. go mod init github.com/sirisha/api-go-project
   3. Install Required Packages
      1. Install Gin (web framework) : run below commands in project directory
         1. go get -u github.com/gin-gonic/gin
            1. <https://go.dev/doc/tutorial/web-service-gin>
         2. Install GORM (ORM library) : run below commands in project directory
            1. go get -u gorm.io/gorm
            2. go get -u gorm.io/driver/sqlite

<https://www.tutorialspoint.com/sqlite/index.htm>

# API development

1. Switch to project directory and create go file like sirisha\_go\_api.go and write code here.
2. Few details about go file.

**a) Import packages**

(

"github.com/gin-gonic/gin"

"gorm.io/driver/sqlite"

"gorm.io/gorm"

"net/http"

)

- **package main:** Defines the package name. In Go, `main` denotes an executable package.

- `**import`:** Imports necessary packages:

- `**github.com/gin-gonic/gin`:** Gin framework for building HTTP web services.

- `**gorm.io/driver/sqlite`:** SQLite driver for Gorm ORM.

- `**gorm.io/gorm`:** Gorm ORM library.

- `**net/http`:** Provides HTTP client and server implementations.

**b) Global Variable for Database**

var db \*gorm.DB

- `db`: A global variable to hold the database connection

**c) Database Initialization**

func initDatabase() {

var err error

db, err = gorm.Open(sqlite.Open("records.db"), &gorm.Config{})

if err != nil {

panic("failed to connect database")

}

db.AutoMigrate(&Record{})

}

- `initDatabase`: Initializes the database connection.

- `gorm.Open`: Opens a connection to the SQLite database named `records.db`.

- `db.AutoMigrate(&Record{})`: Automatically migrates the `Record` struct into a table.

d) Record struct

type Record struct {

ID uint `json:"id" gorm:"primaryKey"`

Name string `json:"name"`

IsActive bool `json:"is\_active"`

CreatedAt int64 `json:"created\_at" gorm:"autoCreateTime"`

UpdatedAt int64 `json:"updated\_at" gorm:"autoUpdateTime"`

}

- `Record`: Defines the structure of a record.

- `ID`: Primary key.

- `Name`: Name of the record.

- `IsActive`: Status of the record (active/inactive).

- `CreatedAt` and `UpdatedAt`: Timestamps for creation and update times.

e) Main Function

func main() {

initDatabase()

router := gin.Default()

router.POST("/records", addRecord)

router.GET("/records", getRecords)

router.PUT("/records/:id", updateRecord)

router.DELETE("/records/:id", deleteRecord)

router.PATCH("/records/:id/activate", activateRecord)

router.PATCH("/records/:id/deactivate", deactivateRecord)

router.Run(":8080")

}

```

- `main`: Entry point of the application.

- `initDatabase`: Initializes the database.

- `gin.Default()`: Creates a Gin router with default middleware (logger and recovery).

- `router.POST`, `router.GET`, `router.PUT`, `router.DELETE`, `router.PATCH`: Define routes for CRUD operations.

- `router.Run(":8080")`: Starts the web server on port 8080.

**f) Handler Functions**

1. Add Record

func addRecord(c \*gin.Context) {

var record Record

if err := c.ShouldBindJSON(&record); err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})

return

}

record.IsActive = true

db.Create(&record)

c.JSON(http.StatusOK, record)

}

```

- `addRecord`: Handles POST requests to add a new record.

- `c.ShouldBindJSON`: Binds JSON payload to the `record`.

- `record.IsActive = true`: Sets the record to active by default.

- `db.Create(&record)`: Creates a new record in the database.

- `c.JSON`: Responds with the created record in JSON format.

1. Get Records

func getRecords(c \*gin.Context) {

var records []Record

db.Find(&records)

c.JSON(http.StatusOK, records)

}

```

- `getRecords`: Handles GET requests to fetch all records.

- `db.Find(&records)`: Retrieves all records from the database.

- `c.JSON`: Responds with the list of records in JSON format.

1. Update Record

func updateRecord(c \*gin.Context) {

var record Record

id := c.Param("id")

if err := db.First(&record, id).Error; err != nil {

c.JSON(http.StatusNotFound, gin.H{"error": "Record not found"})

return

}

if err := c.ShouldBindJSON(&record); err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})

return

}

db.Save(&record)

c.JSON(http.StatusOK, record)

}

- `updateRecord`: Handles PUT requests to update a record.

- `c.Param("id")`: Retrieves the record ID from the URL.

- `db.First(&record, id)`: Finds the record by ID.

- `c.ShouldBindJSON`: Binds the updated JSON payload to the `record` struct.

- `db.Save(&record)`: Saves the updated record in the database.

- `c.JSON`: Responds with the updated record in JSON format.

1. Delete Record

func deleteRecord(c \*gin.Context) {

var record Record

id := c.Param("id")

if err := db.First(&record, id).Error; err != nil {

c.JSON(http.StatusNotFound, gin.H{"error": "Record not found"})

return

}

db.Delete(&record)

c.JSON(http.StatusOK, gin.H{"message": "Record deleted"})

}

```

- `deleteRecord`: Handles DELETE requests to delete a record

.

- `db.First(&record, id)`: Finds the record by ID.

- `db.Delete(&record)`: Deletes the record from the database.

- `c.JSON`: Responds with a success message in JSON format.

1. Active Record

func activateRecord(c \*gin.Context) {

var record Record

id := c.Param("id")

if err := db.First(&record, id).Error; err != nil {

c.JSON(http.StatusNotFound, gin.H{"error": "Record not found"})

return

}

record.IsActive = true

db.Save(&record)

c.JSON(http.StatusOK, record)

}

```

- `activateRecord`: Handles PATCH requests to activate a record.

- `db.First(&record, id)`: Finds the record by ID.

- `record.IsActive = true`: Sets the record to active.

- `db.Save(&record)`: Saves the updated record in the database.

- `c.JSON`: Responds with the updated record in JSON format.

1. Deactive Record

func deactivateRecord(c \*gin.Context) {

var record Record

id := c.Param("id")

if err := db.First(&record, id).Error; err != nil {

c.JSON(http.StatusNotFound, gin.H{"error": "Record not found"})

return

}

record.IsActive = false

db.Save(&record)

c.JSON(http.StatusOK, record)

}

```

- `deactivateRecord`: Handles PATCH requests to deactivate a record.

- `db.First(&record, id)`: Finds the record by ID.

- `record.IsActive = false`: Sets the record to inactive.

- `db.Save(&record)`: Saves the updated record in the database.

- `c.JSON`: Responds with the updated record in JSON format.

# Run the API Go Project

Run below command from project directory.

go run sirisha\_go\_api.go

# Verification

Using postman tool to run the individual API’s

Steps

1. Download and Install postman tool using <https://www.postman.com/downloads/>
2. Create collection in postman and add all API’s
   1. addRecord API

A screenshot of a computer

Description automatically generated

* 1. getRecord API

A screenshot of a computer

Description automatically generated

* 1. updateRecord API

A screenshot of a computer

Description automatically generated

* 1. activeRecord API

A screenshot of a computer

Description automatically generated

* 1. deactivateRecord API

A screenshot of a computer

Description automatically generated

* 1. deleteRecord API

A screenshot of a computer

Description automatically generated

Github Repository

1. Create new repository in <https://github.com/> by making public.
2. Use below commands to fetch and commit the code.
3. Initialize Local Repository

switch to project directory

git init

git config user.email "maddinenisirishatmk@gmail.com"

git config user.name "sirimaddineni12"

git config --global user.email"maddinenisirishatmk@gmail.com"

git config --global user.name "sirimaddineni12"

1. Add Files to the Repository

git add .

1. Commit Changes

git commit -m "Initial commit"

1. Link Your Local Repository to GitHub

git remote add origin https://github.com/your-username/your-repository-name.git

1. Push Project to GitHub

git push -u origin master

6. Fetch files from github

git pull

**adhoc information helps for git**

Password authentication not supporting , so follow below steps to generate PAT.

Personal Access Tokens (PAT)

A Personal Access Token (PAT) can be used in place of your password.

**Creating a Personal Access Token**

* Go to GitHub and log in.
* Navigate to Settings by clicking on your profile picture in the top right corner and selecting Settings.
* Go to Developer settings on the left sidebar.
* Select Personal access tokens.
* Click on Generate new token.
* Give your token a descriptive name, select the scopes or permissions you need (for pushing code, you typically need repo), and then click Generate token.
* Copy the token. This is the only time you'll be able to see it.
* Using the Personal Access Token
* When you push your code, you will be prompted for a username and password. Use your GitHub username as the username and the Personal Access Token as the password.