Go & Gin & Gorm

Introduction à Go

https://medium.com/@nidhigahlawat2002/learn-go-programming-fast-and-easy-basics-for-beginners-175f133a40e

https://devopssec.fr/article/cours-apprendre-langage-programmation-go

https://geekflare.com/fr/getting-started-with-golang/

TP - Bases

Écrivez un programme Go qui demande à l'utilisateur de saisir une température en degrés Celsius. Le programme doit convertir cette température en degrés Fahrenheit et afficher le résultat. Utilisez la formule de conversion suivante : Fahrenheit = Celsius * 9/5 + 32.

Votre programme devrait afficher le message suivant :
"Entrez une température en degrés Celsius :"

Après que l'utilisateur ait entré la température, le programme doit afficher le résultat de la conversion dans le format suivant :
"La température en degrés Fahrenheit est : [résultat]"

Solution

```
package main

import "fmt"

func main() {
    var celsius float64

    fmt.Print("Entrez une température en degrés Celsius : ")
    fmt.Scan(&celsius)

    // Conversion de la température en degrés Fahrenheit
    fahrenheit := celsius*9/5 + 32

    // Affichage du résultat
    fmt.Printf("La température en degrés Fahrenheit est : %.2f", fahrenheit)
}
```

Explication du code :

- La première ligne `package main` indique que le fichier est un programme exécutable et non une bibliothèque.
- `import "fmt"` est utilisé pour importer le package fmt qui fournit des fonctions pour la saisie/sortie de base.
- La fonction `main()` est la fonction principale qui est exécutée lorsque le programme est lancé.

- Une variable `celsius` de type `float64` est déclarée pour stocker la température en degrés Celsius saisie par l'utilisateur.
- `fmt.Print` est utilisé pour afficher le message d'invite demandant à l'utilisateur d'entrer la température en degrés Celsius.
- `fmt.Scan(&celsius)` est utilisé pour lire la valeur entrée par l'utilisateur et la stocker dans la variable `celsius`.
- La conversion de la température en degrés Fahrenheit est effectuée à l'aide de la formule `fahrenheit := celsius*9/5 + 32`.
- `fmt.Printf` est utilisé pour afficher le résultat de la conversion en degrés Fahrenheit. La directive de format "%.2f" est utilisée pour afficher le résultat avec 2 décimales.
- L'exécution du programme se termine et affiche le résultat de la conversion de température.

Manipulation avancée de données en Go

Structures

https://www.simplilearn.com/tutorials/golang-tutorial/golang-struct

https://devopssec.fr/article/structures-et-methodes-golang

Fichiers

https://devopssec.fr/article/structures-et-methodes-golang

https://www.golinuxcloud.com/golang-os/

https://gobyexample.com/reading-files

TP - Structures et Tableaux

Vous devez développer un programme Go pour gérer une liste de tâches. Chaque tâche est représentée par un nom et un statut (complet ou incomplet). Le programme doit permettre à l'utilisateur de :

1. Ajouter une tâche à la liste en saisissant son nom.
2. Marquer une tâche comme complète en saisissant son nom.
3. Afficher la liste des tâches, en indiquant leur nom et leur statut.

Votre programme devrait afficher le message suivant : "Bienvenue dans le gestionnaire de tâches !"

Ensuite, il devrait afficher un menu avec les options suivantes :

1. Ajouter une tâche
2. Marquer une tâche comme complète
3. Afficher la liste des tâches
4. Quitter

Après chaque action effectuée par l'utilisateur, le menu devrait être réaffiché jusqu'à ce que l'utilisateur choisisse l'option "Quitter".

Solution

```
package main

import "fmt"

type Task struct {
    Name   string
    Status string
}

func main() {
    fmt.Println("Bienvenue dans le gestionnaire de tâches !")

    tasks := make([]Task, 0)

    for {
```

```
fmt.Println("\nMenu:")
fmt.Println("1. Ajouter une tâche")
fmt.Println("2. Marquer une tâche comme complète")
fmt.Println("3. Afficher la liste des tâches")
fmt.Println("4. Quitter")

var choice int
fmt.Print("Votre choix : ")
fmt.Scan(&choice)

switch choice {
case 1:
    var name string
    fmt.Print("Nom de la tâche à ajouter : ")
    fmt.Scan(&name)

    task := Task{
        Name:   name,
        Status: "Incomplet",
    }

    tasks = append(tasks, task)
    fmt.Println("Tâche ajoutée avec succès !")

case 2:
    var name string
    fmt.Print("Nom de la tâche à marquer comme complète : ")
    fmt.Scan(&name)

    for i := range tasks {
        if tasks[i].Name == name {
            tasks[i].Status = "Complet"
            fmt.Println("Tâche marquée comme complète avec succès !")
            break
        }
    }

case 3:
    fmt.Println("Liste des tâches :")
    for _, task := range tasks {
```

```
            fmt.Printf("- %s : %s\n", task.Name, task.Status)
        }

    case 4:
        fmt.Println("Au revoir !")
        return

    default:
        fmt.Println("Choix invalide. Veuillez sélectionner une option valide.")
    }
  }
}
```

Explication du code :

- Une structure `Task` est définie avec deux champs : `Name` pour le nom de la tâche et `Status` pour le statut de la tâche (complet ou incomplet).
- La fonction `main()` est la fonction principale qui est exécutée lorsque le programme est lancé.
- Un slice vide `tasks` est créée pour stocker les tâches ajoutées par l'utilisateur.
- Le programme utilise une boucle `for` pour afficher le menu et traiter les choix de l'utilisateur jusqu'à ce que l'option "Quitter" soit sélectionnée.
- Selon le choix de l'utilisateur, différentes actions sont effectuées :
  - Pour l'option 1, l'utilisateur est invité à saisir le nom de la tâche à ajouter. Une nouvelle tâche est créée avec le statut "Incomplet" et ajoutée au slice `tasks`.
  - Pour l'option 2, l'utilisateur est invité à saisir le nom de la tâche à marquer comme complète. Le statut de la tâche correspondante dans une `tasks` est mis à jour.
  - Pour l'option 3, toutes les tâches dans les `tasks` sont affichées avec leur nom et leur statut.
  - Pour l'option 4, le programme se termine et affiche "Au revoir !".
  - Si l'utilisateur choisit une option invalide, un message d'erreur est affiché.

TP - Fichiers

Vous devez développer un programme Go pour gérer une liste de contacts. Chaque contact est représenté par un nom et un numéro de téléphone. Le programme doit permettre à l'utilisateur de :

1. Ajouter un contact en saisissant son nom et son numéro de téléphone.
2. Rechercher un contact par son nom et afficher son numéro de téléphone.
3. Afficher la liste complète des contacts, en indiquant leur nom et leur numéro de téléphone.
4. Enregistrer les contacts dans un fichier.
5. Charger les contacts à partir d'un fichier lors du démarrage du programme.

Votre programme devrait afficher le message suivant : "Bienvenue dans le gestionnaire de contacts !"

Ensuite, il devrait afficher un menu avec les options suivantes :

1. Ajouter un contact
2. Rechercher un contact
3. Afficher la liste des contacts
4. Enregistrer les contacts dans un fichier
5. Charger les contacts à partir d'un fichier
6. Quitter

Après chaque action effectuée par l'utilisateur, le menu devrait être réaffiché jusqu'à ce que l'utilisateur choisisse l'option "Quitter".

Solution

Code solution expliqué :

```
package main


import (
	"bufio"
	"fmt"
	"os"
	"strings"
)
```

```
type Contact struct {
	Name  string
	Phone string
}

func main() {
	fmt.Println("Bienvenue dans le gestionnaire de contacts !")

	contacts := make(map[string]string)

	loadContactsFromFile(contacts)

	for {
		fmt.Println("\nMenu:")
		fmt.Println("1. Ajouter un contact")
		fmt.Println("2. Rechercher un contact")
		fmt.Println("3. Afficher la liste des contacts")
		fmt.Println("4. Enregistrer les contacts dans un fichier")
		fmt.Println("5. Charger les contacts à partir d'un fichier")
		fmt.Println("6. Quitter")

		var choice int
		fmt.Print("Votre choix : ")
		fmt.Scan(&choice)

		switch choice {
		case 1:
			var name, phone string
			fmt.Print("Nom du contact : ")
			scanner := bufio.NewScanner(os.Stdin)
			scanner.Scan()
			name = scanner.Text()
			fmt.Print("Numéro de téléphone : ")
			scanner.Scan()
			phone = scanner.Text()
			contacts[name] = phone
			fmt.Println("Contact ajouté avec succès !")

		case 2:
			var name string
```

```
□□□fmt.Print("Nom du contact à rechercher : ")
□□□scanner := bufio.NewScanner(os.Stdin)
□□□scanner.Scan()
□□□name = scanner.Text()
□□□phone, found := contacts[name]
□□□if found {
□□□□fmt.Printf("Numéro de téléphone de %s : %s\n", name, phone)
□□□} else {
□□□□fmt.Println("Contact non trouvé.")
□□□}

□□case 3:
□□□fmt.Println("Liste des contacts :")
□□□for name, phone := range contacts {
□□□□fmt.Printf("- %s : %s\n", name, phone)
□□□}

□□case 4:
□□□saveContactsToFile(contacts)
□□□fmt.Println("Contacts enregistrés avec succès !")

□□case 5:
□□□loadContactsFromFile(contacts)
□□□fmt.Println("Contacts chargés avec succès !")

□□case 6:
□□□fmt.Println("Au revoir !")
□□□return

□□default:
□□□fmt.Println("Choix invalide. Veuillez sélectionner une option valide.")
□□}
□}
}

func loadContactsFromFile(contacts map[string]string) {
□file, err := os.Open("contacts.txt")
□if err != nil {
□□return
□}
```

```
	defer file.Close()

	scanner := bufio.NewScanner(file)
	for scanner.Scan() {
		line := scanner.Text()
		parts := strings.Split(line, ":")
		if len(parts) == 2 {
			name := strings.TrimSpace(parts[0])
			phone := strings.TrimSpace(parts[1])
			contacts[name] = phone
		}
	}

	if err := scanner.Err(); err != nil {
		fmt.Println("Erreur lors du chargement des contacts :", err)
	}
}

func saveContactsToFile(contacts map[string]string) {
	file, err := os.Create("contacts.txt")
	if err != nil {
		fmt.Println("Erreur lors de l'enregistrement des contacts :", err)
		return
	}
	defer file.Close()

	writer := bufio.NewWriter(file)
	for name, phone := range contacts {
		line := fmt.Sprintf("%s: %s\n", name, phone)
		_, err := writer.WriteString(line)
		if err != nil {
			fmt.Println("Erreur lors de l'écriture dans le fichier :", err)
		}
	}

	writer.Flush()
}
```

Explication du code :

- Une structure `Contact` est définie avec deux champs : `Name` pour le nom du contact et `Phone` pour son numéro de téléphone.
- La fonction `main()` est la fonction principale qui est exécutée lorsque le programme est lancé.
- Une map `contacts` est utilisée pour stocker les contacts, où la clé est le nom du contact et la valeur est son numéro de téléphone.
- La fonction `loadContactsFromFile()` est utilisée pour charger les contacts à partir d'un fichier `contacts.txt`. Chaque ligne du fichier doit être au format "Nom: Numéro".
- La fonction `saveContactsToFile()` est utilisée pour enregistrer les contacts dans le fichier `contacts.txt`. Chaque contact est écrit sur une ligne séparée avec le format "Nom: Numéro".
- Le programme utilise une boucle `for` pour afficher le menu et traiter les choix de l'utilisateur jusqu'à ce que l'option "Quitter" soit sélectionnée.
- Selon le choix de l'utilisateur, différentes actions sont effectuées :
  - Pour l'option 1, l'utilisateur est invité à saisir le nom et le numéro de téléphone du contact, puis le contact est ajouté à la map `contacts`.
  - Pour l'option 2, l'utilisateur est invité à saisir le nom du contact à rechercher. Si le contact est trouvé dans la map `contacts`, son numéro de téléphone est affiché.
  - Pour l'option 3, tous les contacts dans la map `contacts` sont affichés avec leur nom et leur numéro de téléphone.
  - Pour l'option 4, les contacts sont enregistrés dans le fichier `contacts.txt` en utilisant la fonction `saveContactsToFile()`.
  - Pour l'option 5, les contacts sont chargés à partir du fichier `contacts.txt` en utilisant la fonction `loadContactsFromFile()`.
  - Pour l'option 6, le programme se termine et affiche "Au revoir !".
  - Si l'utilisateur choisit une option invalide, un message d'erreur est affiché.

Introduction à Gin

https://go.dev/doc/tutorial/web-service-gin

https://medium.com/@wattanai.tha/go-tutorial-series-ep-1-building-rest-api-with-gin-7c17c7ab1d5b

TP : Création d'une API REST basique avec Gin

Vous devez développer une API REST pour gérer une liste de tâches. Chaque tâche est représentée par un identifiant unique, un titre et un statut (complet ou incomplet). L'API doit prendre en charge les opérations CRUD (création, lecture, mise à jour et suppression) pour les tâches.

1. Définir une structure `Task` qui représente une tâche avec les champs suivants :
   - `ID` : un identifiant unique de la tâche (un nombre entier)
   - `Title` : le titre de la tâche (une chaîne de caractères)
   - `Status` : le statut de la tâche (une chaîne de caractères, soit "complet" ou "incomplet")
2. Créer une variable `tasks` de type slice `[]Task` pour stocker les tâches.
3. Définir les routes de l'API REST avec Gin :
   - GET `/tasks` : récupère la liste des tâches
   - POST `/tasks` : crée une nouvelle tâche
   - GET `/tasks/:id` : récupère les détails d'une tâche spécifique par son identifiant
   - PUT `/tasks/:id` : met à jour une tâche spécifique par son identifiant
   - DELETE `/tasks/:id` : supprime une tâche spécifique par son identifiant
4. Implémenter les handlers (fonctions) pour chaque route :
   - GET `/tasks` : renvoie la liste des tâches en tant que réponse JSON
   - POST `/tasks` : crée une nouvelle tâche à partir des données JSON envoyées et l'ajoute à la liste des tâches
   - GET `/tasks/:id` : récupère les détails d'une tâche spécifique par son identifiant et renvoie les données JSON correspondantes
   - PUT `/tasks/:id` : met à jour les données d'une tâche spécifique par son identifiant à partir des données JSON envoyées
   - DELETE `/tasks/:id` : supprime une tâche spécifique par son identifiant de la liste des tâches
5. Installez le framework Gin en utilisant la commande `go get -u github.com/gin-gonic/gin`.
6. Utilisez la méthode `gin.Run()` pour exécuter l'API sur un port spécifique.

Solution

```
package main


import (
	"github.com/gin-gonic/gin"
	"net/http"
```

```
    "strconv"
)

// Structure pour représenter une tâche
type Task struct {
    ID     int    `json:"id"`
    Title  string `json:"title"`
    Status string `json:"status"`
}

// Variable pour stocker les tâches
var tasks []Task

func main() {
    router := gin.Default()

    // Route pour récupérer la liste des tâches
    router.GET("/tasks", getTasks)

    // Route pour créer une nouvelle tâche
    router.POST("/tasks", createTask)

    // Route pour récupérer les détails d'une tâche spécifique
    router.GET("/tasks/:id", getTask)

    // Route pour mettre à jour une tâche spécifique
    router.PUT("/tasks/:id", updateTask)

    // Route pour supprimer une tâche spécifique
    router.DELETE("/tasks/:id", deleteTask)

    // Exécute l'API sur le port 8080
    router.Run(":8080")
}

// Handler pour récupérer la liste des tâches
func getTasks(c *gin.Context) {
    c.JSON(http.StatusOK, tasks)
}
```

```
// Handler pour créer une nouvelle tâche
func createTask(c *gin.Context) {
    var task Task

    // Bind les données JSON envoyées à la structure Task
    if err := c.ShouldBindJSON(&task); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }

    // Génère un nouvel identifiant unique pour la tâche
    task.ID = len(tasks) + 1

    // Ajoute la tâche à la liste des tâches
    tasks = append(tasks, task)

    c.JSON(http.StatusCreated, task)
}

// Handler pour récupérer les détails d'une tâche spécifique
func getTask(c *gin.Context) {
    id, err := strconv.Atoi(c.Param("id"))
    if err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": "Invalid task ID"})
        return
    }

    for _, task := range tasks {
        if task.ID == id {
            c.JSON(http.StatusOK, task)
            return
        }
    }

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

// Handler pour mettre à jour une tâche spécifique
func updateTask(c *gin.Context) {
    id, err := strconv.Atoi(c.Param("id"))
```

```
    if err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": "Invalid task ID"})
        return
    }

    var updatedTask Task
    if err := c.ShouldBindJSON(&updatedTask); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }

    for index, task := range tasks {
        if task.ID == id {
            // Met à jour les champs de la tâche avec les nouvelles valeurs
            tasks[index].Title = updatedTask.Title
            tasks[index].Status = updatedTask.Status

            c.JSON(http.StatusOK, tasks[index])
            return
        }
    }

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

// Handler pour supprimer une tâche spécifique
func deleteTask(c *gin.Context) {
    id, err := strconv.Atoi(c.Param("id"))
    if err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": "Invalid task ID"})
        return
    }

    for index, task := range tasks {
        if task.ID == id {
            // Supprime la tâche de la liste des tâches
            tasks = append(tasks[:index], tasks[index+1:]...)

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

```
□□}
□}

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

Ce code utilise le framework Gin pour créer un routeur qui définit les routes de l'API REST. Chaque route est associée à un handler (fonction) qui est exécuté lorsque la route est appelée.

La structure `Task` est définie avec les champs ID, Title et Status correspondant aux données d'une tâche.

La variable `tasks` est uneslice (tranche) qui est utilisée pour stocker les tâches.

Les handlers implémentent les opérations CRUD pour les tâches :

- `getTasks` renvoie la liste des tâches en tant que réponse JSON.
- `createTask` crée une nouvelle tâche à partir des données JSON envoyées et l'ajoute à la liste des tâches.
- `getTask` récupère les détails d'une tâche spécifique par son identifiant et renvoie les données JSON correspondantes.
- `updateTask` met à jour les données d'une tâche spécifique par son identifiant à partir des données JSON envoyées.
- `deleteTask` supprime une tâche spécifique par son identifiant de la liste des tâches.

L'API est exécutée sur le port 8080 en utilisant `router.Run(":8080")`.

Assurez-vous d'installer le framework Gin en utilisant la commande `go get -u github.com/gin-gonic/gin` avant d'exécuter le code.

Introduction à Gorm

https://gorm.io/fr_FR/docs/index.html

https://adlerhsieh.com/blog/gorm:-a-simple-guide-on-crud

TP : Intégration de Gorm avec Gin pour une API REST

Vous devez développer une API REST pour gérer une liste de livres. Chaque livre est représenté par un identifiant unique, un titre, un auteur et une année de publication. L'API doit prendre en charge les opérations CRUD (création, lecture, mise à jour et suppression) pour les livres.

Utilisez la bibliothèque Gorm, un ORM (Object-Relational Mapping), pour interagir avec la base de données. Gorm facilite l'accès et la manipulation des données en utilisant des requêtes SQL dans le code Go.

Définir une structure `Book` qui représente un livre avec les champs suivants :

- ID : un identifiant unique du livre (un nombre entier)
- Title : le titre du livre (une chaîne de caractères)
- Author : l'auteur du livre (une chaîne de caractères)
- Year : l'année de publication du livre (un nombre entier)

Créer une connexion à la base de données à l'aide de Gorm et configurer le modèle `Book` pour qu'il corresponde à une table dans la base de données.

Définir les routes de l'API REST avec Gin :

- GET /books : récupère la liste des livres
- POST /books : crée un nouveau livre
- GET /books/:id : récupère les détails d'un livre spécifique par son identifiant
- PUT /books/:id : met à jour un livre spécifique par son identifiant
- DELETE /books/:id : supprime un livre spécifique par son identifiant

Implémenter les handlers (fonctions) pour chaque route en utilisant Gorm pour interagir avec la base de données :

- GET /books : renvoie la liste des livres en tant que réponse JSON en utilisant Gorm pour récupérer tous les enregistrements de la table `books`.
- POST /books : crée un nouveau livre à partir des données JSON envoyées et l'ajoute à la base de données en utilisant Gorm pour créer un nouvel enregistrement dans la table `books`.
- GET /books/:id : récupère les détails d'un livre spécifique par son identifiant et renvoie les données JSON correspondantes en utilisant Gorm pour récupérer un enregistrement spécifique de la table `books`.

- PUT /books/:id : met à jour les données d'un livre spécifique par son identifiant à partir des données JSON envoyées en utilisant Gorm pour mettre à jour l'enregistrement correspondant dans la table `books`.
- DELETE /books/:id : supprime un livre spécifique par son identifiant de la base de données en utilisant Gorm pour supprimer l'enregistrement correspondant de la table `books`.

Installez les dépendances nécessaires, y compris Gin et Gorm, en utilisant les commandes suivantes :

```
go get -u github.com/gin-gonic/gin
go get -u gorm.io/gorm
go get -u gorm.io/driver/mysql
```

Solution

```
package main

import (
	"github.com/gin-gonic/gin"
	"gorm.io/driver/mysql"
	"gorm.io/gorm"
	"net/http"
	"strconv"
)

type Book struct {
	ID     int    `gorm:"primaryKey" json:"id"`
	Title  string `json:"title"`
	Author string `json:"author"`
	Year   int    `json:"year"`
}

var (
	db    *gorm.DB
	err   error
	books []Book
)

func main() {
	dsn := "user:password@tcp(localhost:3306)/dbname?charset=utf8mb4&parseTime=True&loc=Local"
```

```
    db, err = gorm.Open(mysql.Open(dsn), &gorm.Config{})
    if err != nil {
        panic("failed to connect to database")
    }

    db.AutoMigrate(&Book{})

    router := gin.Default()

    router.GET("/books", getBooks)
    router.POST("/books", createBook)
    router.GET("/books/:id", getBook)
    router.PUT("/books/:id", updateBook)
    router.DELETE("/books/:id", deleteBook)

    router.Run(":8080")
}

func getBooks(c *gin.Context) {
    db.Find(&books)
    c.JSON(http.StatusOK, books)
}

func createBook(c *gin.Context) {
    var book Book
    if err := c.ShouldBindJSON(&book); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }

    db.Create(&book)

    c.JSON(http.StatusCreated, book)
}

func getBook(c *gin.Context) {
    id, _ := strconv.Atoi(c.Param("id"))

    var book Book
    db.First(&book, id)
```

```
    if book.ID == 0 {
        c.JSON(http.StatusNotFound, gin.H{"error": "Book not found"})
        return
    }

    c.JSON(http.StatusOK, book)
}

func updateBook(c *gin.Context) {
    id, _ := strconv.Atoi(c.Param("id"))

    var book Book
    db.First(&book, id)

    if book.ID == 0 {
        c.JSON(http.StatusNotFound, gin.H{"error": "Book not found"})
        return
    }

    if err := c.ShouldBindJSON(&book); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }

    db.Save(&book)

    c.JSON(http.StatusOK, book)
}

func deleteBook(c *gin.Context) {
    id, _ := strconv.Atoi(c.Param("id"))

    var book Book
    db.First(&book, id)

    if book.ID == 0 {
        c.JSON(http.StatusNotFound, gin.H{"error": "Book not found"})
        return
    }
```

```
db.Delete(&book)

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

1. **Création de la connexion à la base de données :** Avant de pouvoir interagir avec la base de données, nous devons créer une connexion en utilisant Gorm et le pilote MySQL. Cela se fait en utilisant la fonction `gorm.Open()` avec la chaîne de connexion `dsn` qui contient les informations de connexion telles que le nom d'utilisateur, le mot de passe, l'adresse du serveur et le nom de la base de données.
2. **Définition du modèle `Book` et migration de la table :** Nous définissons la structure `Book` qui représente un livre et utilisons les tags de struct Gorm pour configurer les détails de la table dans la base de données. Dans cet exemple, nous utilisons le tag `primaryKey` pour spécifier que le champ `ID` est la clé primaire de la table. Après avoir défini le modèle, nous utilisons `db.AutoMigrate(&Book{})` pour créer ou mettre à jour automatiquement la table dans la base de données.
3. **Définition des routes de l'API REST avec Gin :** Nous utilisons le framework Gin pour définir les routes de l'API REST. Nous créons un routeur avec `gin.Default()` et définissons les routes et les méthodes correspondantes. Les routes sont définies avec les chemins `/books`, `/books/:id`, etc., et les méthodes HTTP associées (`GET`, `POST`, `PUT`, `DELETE`).
4. **Implémentation des handlers pour chaque route :** Nous implémentons les handlers (fonctions) qui sont exécutés lorsque les routes sont appelées. Chaque handler utilise Gorm pour effectuer des opérations sur la base de données en fonction de l'action CRUD correspondante.
   - Le handler `getBooks` récupère tous les enregistrements de la table `books` en utilisant `db.Find(&books)` et renvoie la liste des livres en tant que réponse JSON.
   - Le handler `createBook` crée un nouveau livre à partir des données JSON envoyées en utilisant `c.ShouldBindJSON(&book)` pour lier les données JSON à la structure `Book`. Ensuite, nous utilisons `db.Create(&book)` pour créer un nouvel enregistrement dans la table `books` et renvoyons le livre créé en tant que réponse JSON.
   - Le handler `getBook` récupère les détails d'un livre spécifique par son identifiant en utilisant `db.First(&book, id)` pour trouver le premier enregistrement correspondant dans la table `books`. Si le livre n'est pas trouvé, nous renvoyons une réponse JSON indiquant que le livre n'a pas été trouvé.
   - Le handler `updateBook` met à jour les données d'un livre spécifique par son identifiant en utilisant `db.First(&book, id)` pour récupérer l'enregistrement correspondant, puis `c.ShouldBindJSON(&book)` pour lier les données JSON à la structure `Book`. Ensuite, nous utilisons `db.Save(&book)` pour mettre à jour l'enregistrement dans la table `books` et renvoyons le livre mis à jour en tant que réponse JSON.
   - Le handler `deleteBook` supprime un livre spécifique par son identifiant en utilisant `db.First(&book, id)` pour récupérer l'enregistrement correspondant, puis `db.Delete(&book)` pour supprimer l'enregistrement de la table `books`. Nous renvoyons ensuite une réponse JSON indiquant que le livre a été supprimé.

Pour chaque handler, nous vérifions également si l'opération de recherche (`First()` ou `Find()`) a réussi en vérifiant la valeur de `book.ID`. Si `book.ID` est égal à 0, cela signifie que l'enregistrement correspondant n'a pas été trouvé et nous renvoyons une réponse JSON appropriée.

5. **Exécution de l'API sur un port spécifique :** Nous utilisons `router.Run(":8080")` pour exécuter l'API sur le port 8080. Vous pouvez modifier le numéro de port en fonction de vos besoins.

Middleware et fonctionnalités avancées avec Gin

https://gin-gonic.com/docs/examples/using-middleware/

https://sosedoff.com/2014/12/21/gin-middleware.html

https://www.youtube.com/watch?v=6UBD54YPX2A

TP : Développement avancé d'une API REST avec Gin et Gorm

Dans ce TP, vous allez enrichir votre API REST existante en utilisant les fonctionnalités avancées de Gin et Gorm. Vous allez ajouter des middlewares pour la gestion des erreurs, l'authentification, etc. De plus, vous utiliserez Gorm pour la persistance des données en travaillant avec des modèles de données complexes et des associations. N'hésitez pas à explorer les fonctionnalités supplémentaires de Gin et Gorm pour améliorer votre API REST.

Étape 1 : Mise en place

1. Créez un nouveau projet Go et initialisez un module à l'aide de la commande `go mod init`.
2. Installez les dépendances nécessaires, y compris Gin et Gorm, en utilisant les commandes suivantes :

```
go get -u github.com/gin-gonic/gin
go get -u gorm.io/gorm
go get -u gorm.io/driver/mysql
```

Étape 2 : Ajout de middlewares

1. Ajoutez un middleware de gestion des erreurs à votre routeur Gin. Ce middleware sera responsable de la gestion des erreurs globales et renverra une réponse JSON appropriée en cas d'erreur.
2. Ajoutez un middleware d'authentification pour sécuriser certaines routes de votre API. Ce middleware vérifiera la présence d'un jeton d'authentification dans les en-têtes de la requête et autorisera ou rejettera l'accès en conséquence.

Étape 3 : Utilisation de Gorm avec des modèles complexes

1. Définissez de nouveaux modèles de données complexes pour votre application, tels que `Author` (auteur) et `Category` (catégorie). Les modèles peuvent avoir des relations entre eux, par exemple un livre peut avoir un auteur et appartenir à une catégorie.
2. Utilisez les fonctionnalités de Gorm pour définir ces relations et associer les modèles. Par exemple, vous pouvez utiliser les tags de struct Gorm tels que `ForeignKey`, `AssociationForeignKey`

, `Many2Many` pour définir les relations entre les modèles.
3. Mettez à jour vos handlers pour prendre en compte les nouvelles relations et les fonctionnalités avancées de Gorm. Par exemple, vous pouvez utiliser les méthodes `Preload` et `Joins` de Gorm pour récupérer les données associées lors de la récupération des livres, des auteurs, etc.

Étape 4 : Exploration des fonctionnalités supplémentaires de Gin et Gorm

1. Explorez les fonctionnalités supplémentaires de Gin et Gorm pour améliorer votre API REST. Par exemple, vous pouvez utiliser les groupes de routes de Gin pour organiser vos routes, ou utiliser des requêtes avancées avec Gorm pour effectuer des opérations complexes sur la base de données.
2. Implémentez au moins une fonctionnalité supplémentaire de votre choix en utilisant Gin et Gorm. Par exemple, vous pouvez ajouter la pagination des résultats, la recherche de livres par titre, l'ajout de commentaires aux livres, etc.

Solution

```
package main

import (
	"github.com/gin-gonic/gin"
	"gorm.io/driver/mysql"
	"gorm.io/gorm"
	"net/http"
	"strconv"
)

type Book struct {
	ID       uint   `gorm:"primaryKey" json:"id"`
	Title    string `json:"title"`
	AuthorID uint   `json:"author_id"`
	CategoryID uint `json:"category_id"`
}

type Author struct {
	ID   uint   `gorm:"primaryKey" json:"id"`
	Name string `json:"name"`
}
```

```
type Category struct {
    ID   uint   `gorm:"primaryKey" json:"id"`
    Name string `json:"name"`
}

var (
    db       *gorm.DB
    err      error
    books    []Book
    authors  []Author
    categories []Category
)

func main() {
    dsn := "user:password@tcp(localhost:3306)/dbname?charset=utf8mb4&parseTime=True&loc=Local"

    db, err = gorm.Open(mysql.Open(dsn), &gorm.Config{})
    if err != nil {
        panic("failed to connect to database")
    }

    db.AutoMigrate(&Book{}, &Author{}, &Category{})

    router := gin.Default()

    router.Use(ErrorHandlerMiddleware)
    router.Use(AuthenticationMiddleware)

    router.GET("/books", getBooks)
    router.POST("/books", createBook)
    router.GET("/books/:id", getBook)
    router.PUT("/books/:id", updateBook)
    router.DELETE("/books/:id", deleteBook)

    router.Run(":8080")
}

func ErrorHandlerMiddleware(c *gin.Context) {
    c.Next()
```

```
    if len(c.Errors) > 0 {
        c.JSON(http.StatusInternalServerError, gin.H{"error": "Internal Server Error"})
    }
}

func AuthenticationMiddleware(c *gin.Context) {
    token := c.GetHeader("Authorization")

    if token != "my-secret-token" {
        c.JSON(http.StatusUnauthorized, gin.H{"error": "Unauthorized"})
        c.Abort()
        return
    }

    c.Next()
}

func getBooks(c *gin.Context) {
    db.Preload("Author").Preload("Category").Find(&books)
    c.JSON(http.StatusOK, books)
}

func createBook(c *gin.Context) {
    var book Book
    if err := c.ShouldBindJSON(&book); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }

    db.Create(&book)

    c.JSON(http.StatusCreated, book)
}

func getBook(c *gin.Context) {
    id, _ := strconv.Atoi(c.Param("id"))

    var book Book
    db.Preload("Author").Preload("Category").First(&book, id)
```

```
    if book.ID == 0 {
        c.JSON(http.StatusNotFound, gin.H{"error": "Book not found"})
        return
    }

    c.JSON(http.StatusOK, book)
}

func updateBook(c *gin.Context) {
    id, _ := strconv.Atoi(c.Param("id"))

    var book Book
    db.Preload("Author").Preload("Category").First(&book, id)

    if book.ID == 0 {
        c.JSON(http.StatusNotFound, gin.H{"error": "Book not found"})
        return
    }

    if err := c.ShouldBindJSON(&book); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }

    db.Save(&book)

    c.JSON(http.StatusOK, book)
}

func deleteBook(c *gin.Context) {
    id, _ := strconv.Atoi(c.Param("id"))

    var book Book
    db.First(&book, id)

    if book.ID == 0 {
        c.JSON(http.StatusNotFound, gin.H{"error": "Book not found"})
        return
    }
```

```
    db.Delete(&book)

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

Cette correction implémente les fonctionnalités avancées suivantes :

1. **Middlewares** :
   - Le middleware `ErrorHandlerMiddleware` gère les erreurs globales et renvoie une réponse JSON appropriée en cas d'erreur.
   - Le middleware `AuthenticationMiddleware` vérifie la présence d'un jeton d'authentification dans les en-têtes de la requête pour sécuriser certaines routes de l'API.
2. **Modèles complexes** :
   - Les modèles `Author` et `Category` sont ajoutés pour représenter les auteurs et les catégories respectivement. Les modèles `Book`, `Author` et `Category` ont des relations entre eux.
   - Gorm est utilisé pour définir les relations et associer les modèles avec les tags de struct appropriés.
3. **Fonctionnalités supplémentaires** :
   - Les handlers `getBooks`, `createBook`, `getBook`, `updateBook` et `deleteBook` sont mis à jour pour utiliser les fonctionnalités avancées de Gorm.
   - Les méthodes `Preload` de Gorm sont utilisées pour récupérer les données associées (auteur, catégorie) lors de la récupération des livres.
   - Les nouvelles relations et fonctionnalités avancées de Gorm sont prises en compte dans les opérations CRUD.