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.

Revision #3
Created 2023-05-19 10:36:52 UTC by Noé Larrieu-Lacoste
Updated 2023-05-19 13:37:56 UTC by Noé Larrieu-Lacoste