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.

Revision #4
Created 19 May 2023 10:38:16 by Noé Larrieu-Lacoste
Updated 19 May 2023 13:35:34 by Noé Larrieu-Lacoste