Опановування основами Go: Практичний посібник з освоєння мови Go

Розділ 10: Мережеве Програмування та Розробка Веб-Застосунків в Go

Фрейворк Gin

Gin - це високопродуктивний веб-фреймворк для мови програмування Go, створений для побудови RESTful API з мінімальною кількістю шаблонного коду. Він вирізняється завдяки своїй простоті, ефективності та мінімалістичному дизайну, пропонуючи функції, такі як маршрутизація, підтримка проміжного програмного забезпечення та відображення даних【5†source】【7†source】.

Одна з основних переваг використання Gin - це його продуктивність. Він відомий як швидкий фреймворк з малим використанням пам'яті, переважно тому, що він побудований на основі HttpRouter. Цей аспект робить його відмінним вибором для застосунків, де швидкість і ефективність є критичними【6†source】.

Gin сприяє організації вашого коду через групування маршрутів, дозволяючи вам ефективно управляти різними частинами вашої системи бекенду. Ця функція особливо корисна для створення чітких розрізнень між різними версіями API або між публічними та приватними кінцевими точками【5†source】.

Що стосується контролерів та обробників, Gin спонукає до відокремлення бізнес-логіки від обробників маршрутів, підвищуючи підтримку коду та читабельність. Визначивши контролери, ви можете управляти складною бізнес-логікою окремо, роблячи ваш код більш чистим і організованим【5†source】.

Для початківців або тих, хто новачок у Go та Gin, налаштування проекту є досить простим. Після встановлення Go та налаштування вашого робочого простору, ви можете легко встановити Gin та почати створювати свій додаток. Gin дозволяє швидко визначати моделі даних, налаштовувати маршрути та писати функції контролерів для різних операцій CRUD. Це робить його доступним фреймворком для розробників різного рівня навичок【7†source】.

На відміну від інших фреймворків Go, таких як Echo, Gin забезпечує баланс між простотою та контролем. Хоча Echo може пропонувати більш простий синтаксис, синтаксис Gin, натхненний мовою API Martini, надає розробникам

більше контролю та гнучкості. Це робить Gin підходящим вибором для тих, хто віддає перевагу трохи більшій складності заради більшої налаштування та контролю над функціональністю своєї програми【6†source】.

Загалом, Gin є міцним та ефективним вибором для створення веб-додатків та API на Go, особливо для розробників, які шукають фреймворк, що поєднує продуктивність з мінімалістичним підходом.

Створення простого сервісу на мові Go з використанням фреймворку Gin можна продемонструвати на прикладі простого API для відповіді на запити. Ось кроки, які потрібно виконати для створення такого сервісу:

Як почати ?

Крок 1: Встановлення Gin

Інінцалізація модуля (необхідно для використання "зовнішних" залежностей)

```
go mod init ginsample
```

Встановіть Gin за допомогою команди:

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

Крок 2: Створення основного файлу

Створіть файл `main.go` та введіть наступний код:

```
package main

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

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

    router.GET("/ping", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "message": "pong",
        })
    })

    router.Run(":8080")
}
```

Цей код ініціалізує новий сервер Gin та налаштовує маршрут `/ping`, який відповідає на GET запити з повідомленням у форматі JSON.

Крок 3: Запуск серверу

Запустіть ваш сервер, виконавши наступну команду у вашому терміналі:

```
go run main.go
```

Після запуску, ваш сервер буде слухати на порту 8080.

Крок 4: Тестування API

Ви можете протестувати ваш API, відправивши запит GET до `http://localhost:8080/ping`. Ви повинні отримати відповідь у вигляді JSON: `{"message":"pong"}`.

Цей базовий приклад демонструє створення простого веб-сервера з використанням Go та Gin. Ви можете розширити цей приклад, додавши додаткові маршрути, контролери та логіку для обробки різних HTTP запитів. Gin дозволяє легко створювати складніші веб-додатки та API з використанням мови Go.

Групування маршрутів та middleware

Ось приклад створення простого веб-сервера на Go з використанням фреймворку Gin, де я використаю групування маршрутів і проміжне програмне забезпечення (middleware) для базової автентифікації (BasicAuth).

Крок 1: Підготовка основного файлу

Створіть файл `main.go` та введіть наступний код:

```
package main

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

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

    // Створення групи маршрутів для API
    apiRoutes := router.Group("/api")

    // Конфігурація BasicAuth Middleware
    authorized := apiRoutes.Group("/", gin.BasicAuth(gin.Accounts{
        "username1": "password1",
        "username2": "password2",
    }))

    // Маршрути для авторизованих користувачів
    authorized.GET("/secure-data", func(c *gin.Context) {
        c.JSON(200, gin.H{"message": "Ви маєте доступ до захищених даних!"})
    })

    // Звичайний маршрут без автентифікації
    apiRoutes.GET("/open-data", func(c *gin.Context) {
        c.JSON(200, gin.H{"message": "Це відкриті дані, доступні без
автентифікації."})
    })

    router.Run(":8080")
}
```

Крок 2: Запуск серверу

Запустіть ваш сервер, виконавши наступну команду у вашому терміналі:

```
go run main.go
```

Після запуску, ваш сервер буде слухати на порту 8080.

Крок 3: Тестування API

1. Відправте запит GET до `http://localhost:8080/api/open-data`. Ви повинні отримати відповідь без необхідності автентифікації.
2. Відправте запит GET до `http://localhost:8080/api/secure-data`, використовуючи одне з імен користувача та паролів, заданих у BasicAuth Middleware. Цей запит потребує автентифікації.

Цей приклад демонструє, як використовувати групування маршрутів і проміжне програмне забезпечення для реалізації базової автентифікації в додатку на основі Gin.

Найкращі приктики

При розробці веб-додатків з використанням фреймворку Gin для мови програмування Go, важливо дотримуватися кращих практик, щоб забезпечити безпеку, продуктивність та легкість утримання вашого додатку. Ось кілька ключових порад:

1. **Використання стандартного логера Gin**: Стандартний логер Gin надає детальну інформацію про запити та відповіді, що полегшує відлагодження та виявлення "вузьких місць" продуктивності.

2. **Перевага мідлварів над власними обробниками**: Мідлварі дозволяють повторно використовувати компоненти, спрощуючи підтримку та оновлення додатку.

3. **Уникнення глобальних змінних**: Глобальні змінні можуть ускладнити відлагодження та привести до непередбачуваних результатів. Використовуйте локальні змінні для кращого контролю.

4. **Простота маршрутів**: Утримуйте маршрути якомога простішими, щоб уникнути помилок і забезпечити легшу підтримку.

5. **Обмеження кількості параметрів у маршруті**: Використовуйте мінімальну кількість параметрів для уникнення складності і плутанини.

6. **Використання об'єкта контексту для передачі даних між мідлварами**: Це забезпечує ефективне зберігання та доступ до даних.

7. **Перевірка вхідних даних за допомогою структурних тегів і пакетів валідаторів**: Використовуйте ці інструменти для швидкої та легкої валідації вхідних даних користувачів.

8. **Повернення значущих повідомлень про помилки з HTTP статус-кодами**: Це дозволяє передавати детальну інформацію про помилки користувачам вашого API.

9. **Додавання логування для відлагодження**: Логування допомагає виявляти та вирішувати проблеми в додатку.

10. **Ретельне тестування коду**: Це допомагає виявити та виправити помилки, забезпечуючи надійність та якість вашого додатку.

> Пам'ятайте, що ефективна реалізація цих практик може значно вплинути на успіх вашого проекту на Gin.

Корисне та цікаве

Фреймворк Gin для Go включає в себе багато корисних і цікавих функцій, які роблять його вибором для розробки веб-додатків та API. Ось деякі з них:

Корисні Функції

1. **Маршрутизація (Routing)**: Gin дозволяє легко створювати маршрути для обробки HTTP запитів. Ви можете визначити маршрути для різних HTTP методів (GET, POST, PUT, DELETE тощо).

   Приклад:

   ```
   router.GET("/someGet", getting)
   router.POST("/somePost", posting)
   ```

2. **Мідлварі (Middleware)**: Gin дозволяє додавати мідлварі, які виконуються перед або після обробників запитів. Це може бути використано для логування, аутентифікації, обробки помилок тощо.

   Приклад:

   ```
   router.Use(gin.Logger())
   router.Use(gin.Recovery())
   ```

3. **Групування Маршрутів (Route Grouping)**: Групування маршрутів дозволяє організувати схожі маршрути разом, спрощуючи управління кодом.

   Приклад:

   ```
   v1 := router.Group("/v1")
   {
       v1.GET("/login", loginEndpoint)
       v1.GET("/submit", submitEndpoint)
   }
   ```

Цікаві Функції

1. **Вбудована Підтримка JSON, XML та HTML**: Gin дозволяє легко відповідати на запити в форматах JSON, XML або HTML, автоматично встановлюючи відповідні заголовки відповіді.

Приклад JSON відповіді:

```
router.GET("/someJSON", func(c *gin.Context) {
    c.JSON(http.StatusOK, gin.H{"message": "hey", "status": http.StatusOK})
})
```

2. **Підтримка WebSockets**: Хоча Gin не має вбудованої підтримки WebSockets, ви можете інтегрувати його з зовнішніми пакетами, такими як "gorilla/websocket", для створення інтерактивних додатків в реальному часі.

3. **Параметри Маршруту і Query Строки**: Gin дозволяє легко отримувати доступ до параметрів маршруту та query строк, що робить обробку HTTP запитів більш гнучкою.

   Приклад:

```
router.GET("/user/:name", func(c *gin.Context) {
    name := c.Param("name")
    c.String(http.StatusOK, "Hello %s", name)
})
```

> Ці функції роблять Gin потужним і гнучким інструментом для розробки веб-додатків на Go. Gin поєднує в собі легкість у використанні з широкими можливостями для розширення та налаштування, що робить його популярним вибором серед розробників Go.

Використання JWT автрізації та пакету GORM у сервісі на Gin

Почнемо з створення пакета `models` який описує данні для нашого сервісу.

```
package models

import "gorm.io/gorm"

// User - модель користувача
type User struct {
    gorm.Model
    Email    string `gorm:"unique" json:"email"`
    Password string `json:"password"`
}
// Profile - данні користувача
type Profile struct {
    gorm.Model
    UserID   uint   `json:"user_id" gorm:"unique;not null"`
    FullName string `json:"full_name" gorm:"not null"`
    Age      int    `json:"age" gorm:"default:30"`
}

// ProfileWithUserEmail - модель профілю з електронною адресою користувача
type ProfileWithUserEmail struct {
```

```
    Profile
    Email string `json:"email"`
}

// ProfileWithUser - модель профілю з користувачем
type ProfileWithUser struct {
    Profile
    User User `json:"user" gorm:"foreignKey:UserID;references:ID"`
}
```

Цей код використовує два популярних пакети у світі Go: GORM (ORM-бібліотека для Go) та Gin (веб-фреймворк). Ось детальний опис функціоналу кожної моделі, визначеної у цьому коді:

Модель `User`

Ця модель представляє користувача в системі. Використовуючи GORM, описані такі поля:

- `gorm.Model`: Це вбудована структура, яка включає поля `ID`, `CreatedAt`, `UpdatedAt`, `DeletedAt`. Вона забезпечує базові можливості ORM для кожної сутності.
- `Email`: Поле для зберігання електронної пошти користувача. Атрибут `gorm:"unique"` гарантує, що кожна електронна пошта в базі даних буде унікальною.
- `Password`: Поле для зберігання пароля користувача. Воно серіалізується у формат JSON.

Модель `Profile`

Ця модель представляє профіль користувача. Вона включає:

- `gorm.Model`: Так само, як і в `User`, використовується для базових ORM функцій.
- `UserID`: Унікальний ідентифікатор користувача, пов'язаний з профілем. `gorm:"unique;not null"` гарантує, що кожен профіль буде мати унікальний та не пустий `UserID`.
- `FullName`: Повне ім'я користувача. `gorm:"not null"` означає, що це поле не може бути пустим.
- `Age`: Вік користувача, зі стандартним значенням 30 (`gorm:"default:30"`).

Модель `ProfileWithUserEmail`

Ця модель розширює `Profile`, додаючи додаткове поле:

- `Email`: Електронна пошта користувача. Це поле не входить безпосередньо у `Profile`, але додається тут для зручності представлення даних.

Модель `ProfileWithUser`

Ця модель також розширює `Profile`, але додає цілу структуру `User`:

- `User`: Повна сутність користувача. Використовується відношення на основі внешнього ключа (`foreignKey:UserID;references:ID`), що дозволяє GORM автоматично зв'язувати записи `User` з відповідними записами `Profile`.

> Кожна з цих моделей може бути використана для різних цілей, наприклад, для роботи з базою даних, створення API кінцевих точок у Gin тощо. GORM забезпечує гнучкість у роботі з базою

> даних, а Gin - у створенні веб-додатків та API.

Створення кортроллерів для роботи з данними

```
package controllers

import (
    "github.com/gin-gonic/gin"
    "jwt_gorm/initializers"
    "jwt_gorm/models"
    "net/http"
)

type profile struct {
    FullName string `json:"full_name" binding:"required"`
    Age      int    `json:"age"`
}
```

Цей код з пакету `controllers` виконує декілька ключових функцій у контексті веб-додатку, розробленого з використанням фреймворків Gin та GORM у Go. Кожна функція відповідає за певну частину CRUD (Створення, Читання, Оновлення, Видалення) операцій над профілями користувачів. Ось детальний опис функціоналу кожної функції:

Функція `CreateProfile`

- **Завдання**: Створює новий профіль користувача.
- **Логіка**:
  - Перевіряє, чи є інформація про користувача у токені (JWT або інший механізм аутентифікації).
  - Витягує дані з тіла запиту (JSON) щодо повного імені та віку.
  - Створює новий профіль у базі даних за допомогою GORM.
  - Повертає створений профіль або повідомлення про помилку.

```
func CreateProfile(c *gin.Context) {
    // Get the user id from the token
    tmp, ok := c.Get("user")
    if !ok {
        c.AbortWithStatus(http.StatusUnauthorized)
        return
    }
    user := tmp.(models.User)
    // Get the full name and age off req body
    var body profile
    if c.ShouldBindJSON(&body) != nil {
        c.JSON(http.StatusBadRequest, gin.H{
            "error": "Failed to read body",
        })
        return
    }
```

```
    // Create the profile
    profile := models.Profile{UserID: user.ID, FullName: body.FullName, Age:
body.Age}
    result := initializers.DB.Create(&profile)

    if result.Error != nil {
        c.JSON(http.StatusBadRequest, gin.H{
            "error":    "Failed to create profile.",
            "database": result.Error,
        })
        return
    }
    // Respond
    c.JSON(http.StatusOK, profile)
}
```

Функція `GetProfile`

- **Завдання**: Витягує профіль користувача, базуючись на його ID.
- **Логіка**:
  - Перевіряє автентифікацію користувача.
  - Використовує GORM для пошуку профілю в базі даних.
  - Повертає профіль або помилку, якщо профіль не знайдено.

```
func GetProfile(c *gin.Context) {
    // Get the user id from the token
    tmp, ok := c.Get("user")
    if !ok {
        c.AbortWithStatus(http.StatusUnauthorized)
        return
    }
    user := tmp.(models.User)
    // Find the profile
    var profile models.Profile
    result := initializers.DB.Where("user_id = ?", user.ID).First(&profile)

    if result.Error != nil {
        c.JSON(http.StatusNotFound, gin.H{
            "error": "Profile not found.",
        })
        return
    }
    // Respond
    c.JSON(http.StatusOK, profile)
}
```

Функція `UpdateProfile`

- **Завдання**: Оновлює існуючий профіль користувача.
- **Логіка**:
    - Перевіряє, чи користувач аутентифікований.
    - Оновлює дані профілю (ім'я та вік), отримані з тіла запиту.
    - Зберігає оновлення в базі даних через GORM.
    - Повертає оновлений профіль або повідомлення про помилку.

```
func UpdateProfile(c *gin.Context) {
    // Get the user id from the token
    tmp, ok := c.Get("user")
    if !ok {
        c.AbortWithStatus(http.StatusUnauthorized)
        return
    }
    user := tmp.(models.User)
    // Get the full name and age off req body
    var body profile
    if c.ShouldBindJSON(&body) != nil {
        c.JSON(http.StatusBadRequest, gin.H{
            "error": "Failed to read body",
        })
        return
    }
    // Find the profile
    var profile models.Profile
    result := initializers.DB.Where("user_id = ?", user.ID).First(&profile)

    if result.Error != nil {
        c.JSON(http.StatusNotFound, gin.H{
            "error": "Profile not found.",
        })
        return
    }
    // Update the profile
    profile.FullName = body.FullName
    profile.Age = body.Age
    result = initializers.DB.Save(&profile)

    if result.Error != nil {
        c.JSON(http.StatusBadRequest, gin.H{
            "error": "Failed to update profile.",
        })
        return
    }
    // Respond
    c.JSON(http.StatusOK, profile)
}
```

Функція `GetProfileByID`

- **Завдання**: Витягує профіль за ID, включаючи електронну пошту з пов'язаної таблиці `users`.
- **Логіка**:
  - Виконує складний запит SQL з використанням GORM для отримання даних про профіль разом з електронною поштою користувача.
  - Повертає ці дані або помилку, якщо профіль не знайдено.

```
func GetProfileByID(c *gin.Context) {
    // Get the profile id from the req params
    id := c.Param("id")
    // Find the profile
    var profile models.ProfileWithUserEmail
    result := initializers.DB.Debug().Table("profiles").
        Select("profiles.*, users.email").
        Joins("left join users on profiles.user_id = users.id").
        Where("profiles.id = ?", id).
        First(&profile)

    if result.Error != nil {
        c.JSON(http.StatusNotFound, gin.H{
            "error": "Profile not found.",
        })
        return
    }
    // Respond
    c.JSON(http.StatusOK, profile)
}
```

Функція `GetDataByID`

- **Завдання**: Витягує профіль за ID, повертаючи деталі профілю разом з повною інформацією про користувача.
- **Логіка**:
  - Виконує запит через GORM, використовуючи з'єднання з таблицею `users` для отримання повної інформації про користувача.
  - Повертає деталі профілю разом з даними користувача або помилку,якщо профіль не знайдено. Ця функція демонструє більш складний запит з використанням з'єднання таблиць.

```
func GetDataByID(c *gin.Context) {
   // Get the profile id from the req params
   id := c.Param("profile_id")
   // Find the profile
   var profile models.ProfileWithUser
   result := initializers.DB.Debug().Table("profiles").
    Joins("User").
    Where("profiles.id = ?", id).
    First(&profile)

   if result.Error != nil {
    c.JSON(http.StatusNotFound, gin.H{
```

```
            "error": "Profile not found.",
        })
        return
    }
    // Respond
    c.JSON(http.StatusOK, profile)
}
```

> У цьому коді використовуються різні аспекти Gin та GORM для створення RESTful API, що обслуговує профілі користувачів. Важливі аспекти включають роботу з JSON даними, перевірку аутентифікації, взаємодію з базою даних через ORM, та обробку помилок. Код демонструє ефективні підходи до роботи з веб-запитами та базою даних у контексті сучасної розробки на Go.

controllers/main.go

```
package controllers

import (
    "jwt_gorm/initializers"
    "jwt_gorm/models"
    "net/http"
    "os"
    "time"

    "github.com/gin-gonic/gin"
    "github.com/golang-jwt/jwt/v5"
    "golang.org/x/crypto/bcrypt"
)

type request struct {
    Email    string `json:"email" binding:"required"`
    Password string `json:"password" binding:"required"`
}
```

Цей код з пакету `controllers` реалізує функціонал реєстрації, авторизації та валідації користувачів у веб-додатку, що використовує фреймворк Gin для Go. Ось опис основних функцій:

Функція `Signup`

- **Завдання**: Реєструє нового користувача.
- **Логіка**:
  - Витягує електронну пошту та пароль з тіла запиту.
  - Використовує `bcrypt` для хешування пароля.
  - Створює нового користувача в базі даних (GORM).
  - Повертає дані користувача з прихованим паролем.

```
func Signup(c *gin.Context) {
    // Get the email/pass off req Body
```

```
    var body request

    if err := c.ShouldBindJSON(&body); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{
            "error":   "Failed to read body",
            "message": err.Error(),
        })

        return
    }

    // Hash the password
    hash, err := bcrypt.GenerateFromPassword([]byte(body.Password), 10)

    if err != nil {
        c.JSON(http.StatusBadRequest, gin.H{
            "error": "Failed to hash password.",
        })
        return
    }

    // Create the user
    user := models.User{Email: body.Email, Password: string(hash)}

    result := initializers.DB.Create(&user)

    if result.Error != nil {
        c.JSON(http.StatusBadRequest, gin.H{
            "error": "Failed to create user.",
            "db":    result.Error,
        })
        return
    }
    user.Password = "<hidden>"
    // Respond
    c.JSON(http.StatusOK, user)
}
```

Функція `Login`

- **Завдання**: Авторизує користувача та створює JWT токен.
- **Логіка**:
    - Витягує електронну пошту та пароль з тіла запиту.
    - Перевіряє електронну пошту та пароль, порівнюючи з даними в базі.
    - Генерує JWT токен з використанням бібліотеки `github.com/golang-jwt/jwt/v5`.
    - Встановлює куки з токеном для авторизації.

```
func Login(c *gin.Context) {
    // Get email & pass off req body
    var body request
```

```
    if err := c.ShouldBindJSON(&body); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{
            "error":   "Failed to read body",
            "message": err.Error(),
        })

        return
    }

    // Look up for requested user
    var user models.User

    initializers.DB.First(&user, "email = ?", body.Email)

    if user.ID == 0 {
        c.JSON(http.StatusBadRequest, gin.H{
            "error": "Invalid email or password",
        })
        return
    }

    // Compare sent in password with saved users password
    err := bcrypt.CompareHashAndPassword([]byte(user.Password),
[]byte(body.Password))

    if err != nil {
        c.JSON(http.StatusBadRequest, gin.H{
            "error": "Invalid email or password",
        })
        return
    }

    // Generate a JWT token
    token := jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{
        "sub": user.ID,
        "exp": time.Now().Add(time.Hour * 24).Unix(),
    })

    // Sign and get the complete encoded token as a string using the secret
    tokenString, err := token.SignedString([]byte(os.Getenv("SECRET")))

    if err != nil {
        c.JSON(http.StatusBadRequest, gin.H{
            "error": "Failed to create token",
        })
        return
    }

    // Respond
    c.SetSameSite(http.SameSiteLaxMode)
    c.SetCookie("Authorization", tokenString, 3600*24, "", "", false, true)
    user.Password = "<hidden>"
    c.JSON(http.StatusOK, user)
}
```

Функція `Validate`

- **Завдання**: Перевіряє токен користувача та надає його дані.
- **Логіка**:
  - Витягує дані користувача з контексту (токена).
  - Повертає дані користувача як відповідь.

```
func Validate(c *gin.Context) {
    user, _ := c.Get("user")

    // user.(models.User).Email    -->   to access specific data

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

> Цей код демонструє стандартний підхід до реалізації системи реєстрації та аутентифікації в додатках, написаних на Go з використанням фреймворків Gin і GORM. Він охоплює основні аспекти безпеки, такі як хешування паролів та створення та валідація JWT токенів, а також використовує Gin для обробки HTTP запитів і GORM для взаємодії з базою даних.

Створюємо middleware

Цей код визначає проміжне програмне забезпечення (middleware) для налаштування CORS (Cross-Origin Resource Sharing) у веб-додатку, який використовує фреймворк Gin для Go. Давайте розглянемо його основні компоненти та функціональність:

```
package middleware

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

func CORSMiddleware() gin.HandlerFunc {
    return func(c *gin.Context) {
        c.Writer.Header().Set("Access-Control-Allow-Origin", "*")
        c.Writer.Header().Set("Access-Control-Allow-Credentials", "true")
        c.Writer.Header().Set("Access-Control-Allow-Headers", "Content-Type,
Content-Length, Accept-Encoding, X-CSRF-Token, Authorization, accept, origin,
Cache-Control, X-Requested-With")
        c.Writer.Header().Set("Access-Control-Allow-Methods", "POST, OPTIONS, GET,
PUT")

        if c.Request.Method == "OPTIONS" {
            c.AbortWithStatus(204)
            return
        }
```

```
            c.Next()
        }
    }
```

Функція `CORSMiddleware`

Ця функція створює налаштування CORS для обробки запитів із різних доменів. CORS є важливим механізмом безпеки, що дозволяє веб-серверам вказати, які домени можуть здійснювати запити до сервера. Ось деталі цієї функції:

- `Access-Control-Allow-Origin`: Встановлюється в "*", що дозволяє доступ до ресурсів з будь-якого домену. У виробничому середовищі рекомендується замість цього вказувати конкретні домени.
- `Access-Control-Allow-Credentials`: Встановлюється в "true", що дозволяє передачу облікових даних (наприклад, куків) у міждоменних запитах.
- `Access-Control-Allow-Headers`: Визначає, які HTTP-заголовки дозволені у запитах. У цьому прикладі включено різноманітні заголовки, включаючи `Authorization` для токенів аутентифікації.
- `Access-Control-Allow-Methods`: Визначає, які HTTP-методи дозволені при доступі до ресурсу. Тут вказано `POST`, `OPTIONS`, `GET`, `PUT`.
- Обробка `OPTIONS` запитів: Якщо метод запиту є `OPTIONS`, функція завершує обробку запиту, повертаючи статус 204 (No Content). Це типово для CORS, де `OPTIONS` запит використовується як "попередній" запит для перевірки, чи дозволений фактичний запит.

Інтеграція з Gin

Це проміжне програмне забезпечення можна інтегрувати у веб-додаток Gin, додавши його до ланцюга обробки запитів:

```
router := gin.Default()
router.Use(middleware.CORSMiddleware())
```

Використання цієї функції важливе для веб-додатків, що інтерактивно спілкуються з API, розміщеними на різних доменах, та є ключовим аспектом сучасного веб-розроблення.

Авторізаційне middleware

Цей код визначає проміжне програмне забезпечення (middleware) `RequireAuth` для фреймворку Gin у мові програмування Go, яке використовується для перевірки JWT (JSON Web Token) токенів автентифікації.

```
package middleware

import (
```

```
    "fmt"
    "jwt_gorm/initializers"
    "jwt_gorm/models"
    "net/http"
    "os"
    "time"

    "github.com/gin-gonic/gin"
    "github.com/golang-jwt/jwt/v5"
)

func RequireAuth(c *gin.Context) {
    // Get the cookie off the request
    tokenString, err := c.Cookie("Authorization")

    if err != nil {
        c.AbortWithStatus(http.StatusUnauthorized)
    }

    // Decode/validate it
    token, _ := jwt.Parse(tokenString, func(token *jwt.Token) (interface{}, error)
{
        // Don't forget to validate the alg is what you expect:
        if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
            return nil, fmt.Errorf("unexpected signing method: %v",
token.Header["alg"])
        }

        // hmacSampleSecret is a []byte containing your secret, e.g.
[]byte("my_secret_key")
        return []byte(os.Getenv("SECRET")), nil
    })

    if claims, ok := token.Claims.(jwt.MapClaims); ok && token.Valid {
        // Chec k the expiry date
        if float64(time.Now().Unix()) > claims["exp"].(float64) {
            c.AbortWithStatus(http.StatusUnauthorized)
        }

        // Find the user with token Subject
        var user models.User
        initializers.DB.First(&user, claims["sub"])

        if user.ID == 0 {
            c.AbortWithStatus(http.StatusUnauthorized)
        }

        // Attach the request
        c.Set("user", user)

        //Continue
        c.Next()
    } else {
        c.AbortWithStatus(http.StatusUnauthorized)
```

```
    }
}
```

Функціонал `RequireAuth`

- **Отримання токену**: Функція спочатку намагається отримати токен автентифікації з куків запиту.
- **Перевірка токену**: Використовуючи пакет `github.com/golang-jwt/jwt/v5`, функція розшифровує та перевіряє токен. Це включає перевірку алгоритму підпису та секретного ключа (`SECRET`).
- **Перевірка часу дії токену**: Функція перевіряє чи токен не прострочений.
- **Пошук користувача**: Якщо токен дійсний, функція шукає у базі даних користувача за ідентифікатором (`sub` в токені), використовуючи GORM.
- **Передача даних користувача**: Якщо користувач знайдений, його дані передаються до контексту Gin, що дозволяє наступним обробникам використовувати цю інформацію.
- **Переривання запиту**: Якщо токен недійсний, прострочений або користувач не знайдений, запит переривається зі статусом 401 (Неавторизований).

Інтеграція з Gin

Це проміжне програмне забезпечення можна інтегрувати у веб-додаток Gin, додавши його до ланцюга обробки запитів:

```
router := gin.Default()
router.Use(middleware.RequireAuth)
```

Важливість

Ця функція є ключовою для захисту захищених частин додатку, переконуючись, що лише автентифіковані користувачі мають доступ. Використання JWT є загальноприйнятим підходом для забезпечення безпеки API, оскільки вони дозволяють легко перевірити автентичність та повноваження користувача.

Цей код з пакету `initializers` в Go використовує бібліотеку `github.com/joho/godotenv` для завантаження змінних оточення з файлу `.env`. Файл `.env` часто використовується у розробці програмного забезпечення для зберігання конфігураційних параметрів, які не слід включати безпосередньо у вихідний код, особливо чутливих даних, таких як секретні ключі, паролі баз даних тощо. Ось детальний опис функціоналу цього коду:

Підготовка оточення та кофігурування

```
package initializers

import (
    "log"
```

```
    "github.com/joho/godotenv"
)

func LoadEnvVariables() {
    err := godotenv.Load()

    if err != nil {
        log.Fatal("Error loading .env file")
    }
}
```

Функція `LoadEnvVariables`

- **Завдання**: Завантажує змінні оточення з файлу `.env`.
- **Логіка**:
    - Використання `godotenv.Load()`: Ця функція шукає файл `.env` у поточній директорії та завантажує змінні оточення, визначені у ньому.
    - Обробка помилок: Якщо файл `.env` не знайдено або виникла інша помилка під час його завантаження, програма виведе помилку та завершить виконання. Це забезпечує, що програма не буде працювати без необхідних конфігураційних параметрів.

Важливість Використання `.env` файлів

- **Безпека**: Це дозволяє розробникам тримати конфігурацію окремо від коду, знижуючи ризик випадкового включення чутливих даних у вихідний код, особливо при роботі з відкритим кодом.
- **Гнучкість**: Різні середовища (розробка, тестування, виробництво) можуть мати різні конфігурації, які легко управляти через окремі `.env` файли.
- **Простота управління конфігурацією**: Змінні оточення можна легко змінювати без необхідності перекомпіляції додатку.

Завантаження `.env` файлу на початку виконання програми є поширеною практикою у багатьох проектах Go, що використовують змінні оточення для конфігурації.

Ініцація з'єднання з БД та автоміграція

Цей код з пакету `initializers` у Go виконує дві основні задачі для роботи з базою даних через GORM.

```
package initializers

import (
    "fmt"
    "log"
    "os"

    "gorm.io/driver/sqlite"
    "gorm.io/gorm"
    "jwt_gorm/models"
)
```

```
var DB *gorm.DB
```

Функція `ConnectToDb`

```
func ConnectToDb() {
    var err error
    dsn := os.Getenv("DB")
    DB, err = gorm.Open(sqlite.Open(dsn), &gorm.Config{})

    if err != nil {
        log.Fatal("Failed to connect to DB")
    }
}
```

- **Завдання**: Встановлює з'єднання з базою даних.
- **Логіка**:
    - Витягує рядок з'єднання з базою даних (DSN - Data Source Name) з змінної оточення `DB`.
    - Використовує `gorm.Open` з драйвером SQLite для встановлення з'єднання.
    - У разі невдачі видає критичну помилку, яка призведе до зупинки програми.

Функція `SyncDatabase`

```
func SyncDatabase() error {
    err := DB.AutoMigrate(&models.User{})
    if err != nil {
        return fmt.Errorf("failed to sync database: %w", err)
    }
    err = DB.AutoMigrate(&models.Profile{})
    if err != nil {
        return fmt.Errorf("failed to sync database: %w", err)
    }
    return nil
}
```

- **Завдання**: Синхронізує структури даних моделей з базою даних.
- **Логіка**:
    - Використовує `DB.AutoMigrate` для створення або оновлення таблиць у базі даних на основі структур моделей (`models.User` та `models.Profile`).
    - У разі помилки повертає форматоване повідомлення про помилку.

Ці дві функції є ключовими для ініціалізації та підтримки бази даних у додатках Go, які використовують GORM. Вони дозволяють централізовано управляти з'єднанням з базою даних і структурою даних, спрощуючи роботу з базою даних та забезпечуючи консистентність між структурою даних у програмі та схемою бази даних.

Код сервісу та маршрутизація

```
package main

import (
    "github.com/gin-gonic/gin"
    "jwt_gorm/controllers"
    "jwt_gorm/initializers"
    "jwt_gorm/middleware"
    "log"
)

func main() {
    initializers.LoadEnvVariables()
    initializers.ConnectToDb()
    if err := initializers.SyncDatabase(); err != nil {
        log.Fatal(err)
    }
    r := gin.Default()
    r.Use(middleware.CORSMiddleware())
    r.POST("/signup", controllers.Signup)
    r.POST("/login", controllers.Login)
    authorized := r.Group("/", middleware.RequireAuth)
    authorized.GET("/validate", controllers.Validate)
    authorized.POST("/profile", controllers.CreateProfile)
    authorized.GET("/profile", controllers.GetProfile)
    authorized.PUT("/profile", controllers.UpdateProfile)
    authorized.GET("/profile/:id", controllers.GetProfileByID)  // This is an
admin route
    authorized.GET("/all/:profile_id", controllers.GetDataByID) // This is an
admin route
    if err := r.SetTrustedProxies([]string{"127.0.0.1"}); err != nil {
        log.Println(err)
    }
    if err := r.Run("127.0.0.1:8080"); err != nil {
        log.Fatal(err)
    }
}
```

Цей код є основним файлом (зазвичай `main.go`) для веб-додатку, створеного за допомогою фреймворку Gin у Go. Він інтегрує різні компоненти додатку, включаючи ініціалізацію, маршрутизацію та мідлварі. Ось ключові компоненти та їх ролі:

Ініціалізація

- `initializers.LoadEnvVariables()`: Завантажує змінні оточення з файлу `.env`.
- `initializers.ConnectToDb()`: Встановлює з'єднання з базою даних.
- `initializers.SyncDatabase()`: Синхронізує схему бази даних з моделями даних.

Налаштування Gin та Маршрутів

- `gin.Default()`: Створює новий екземпляр Gin.
- `r.Use(middleware.CORSMiddleware())`: Додає CORS мідлварі до всіх маршрутів.
- `r.POST("/signup", controllers.Signup)`: Визначає маршрут для реєстрації користувачів.
- `r.POST("/login", controllers.Login)`: Визначає маршрут для авторизації користувачів.

Авторизовані Маршрути

- `authorized := r.Group("/", middleware.RequireAuth)`: Створює групу маршрутів з вимогою авторизації.
- В групі `authorized` визначаються маршрути для профілів користувачів (створення, отримання, оновлення) та інші функції.

Додаткові Налаштування

- `r.SetTrustedProxies([]string{"127.0.0.1"})`: Налаштування довірених проксі.
- `r.Run("127.0.0.1:8080")`: Запуск сервера на вказаному порту і адресі.

Цей код є типовим прикладом RESTful API сервера, який використовує аутентифікацію на основі JWT, взаємодію з базою даних через GORM, і має захищені маршрути, доступні тільки авторизованим користувачам. Він демонструє гнучкість та потужність Gin і GORM у розробці сучасних веб-додатків на Go.

Gin - універсальні відповіді в сервісах

```
package main

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

func serveDifferentEncoding(c *gin.Context) {
    data := gin.H{
        "message": "Hello, World!",
    }
    switch c.GetHeader("Accept") {
    case "application/xml":
        c.XML(200, data)
    case "application/x-yaml":
        c.YAML(200, data)
    case "application/x-toml":
        c.TOML(200, data)
    case "text/plain":
        c.String(200, data["message"].(string))
    case "application/json":
        fallthrough
    default:
        c.JSON(200, data)
    }
}

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

```
    router.GET("/hello", serveDifferentEncoding)
    router.Run("127.0.0.1:8080")
}
```

Цей код використовує фреймворк Gin для створення простого веб-сервера, що може відповідати на запити з даними у різних форматах кодування залежно від заголовку `Accept` у запиті.

Функція `serveDifferentEncoding`

Ця функція реагує на HTTP запити, повертаючи відповідь у різних форматах кодування, таких як JSON, XML, YAML, TOML, або звичайний текст.

- Використання `switch`-конструкції для визначення типу відповіді на основі заголовка `Accept`.
- Застосування методів `c.XML`, `c.YAML`, `c.TOML`, `c.String`, та `c.JSON` з фреймворку Gin для форматування відповіді.

Функція `main`

Створює маршрутизатор Gin та визначає маршрут `/hello`, що обслуговується функцією `serveDifferentEncoding`.

- `router.GET("/hello", serveDifferentEncoding)`: Визначає маршрут GET запитів до шляху `/hello`.
- `router.Run("127.0.0.1:8080")`: Запускає веб-сервер на порту 8080.

Застосування

Цей код може бути використаний у веб-додатках, де потрібно підтримувати відповіді у різних форматах, наприклад, для API, яке взаємодіє з різними клієнтами, кожен з яких може мати свої вимоги до формату даних. Використання заголовка `Accept` дозволяє клієнтам вказувати, у якому форматі вони очікують отримати дані.

Ключові основи GORM

- Встановити з'єднання з базою даних за допомогою `DB, err = gorm.Open(драйвер.Open(dsn), &gorm.Config{...опції з'єднання})`
- Зробити автоміграції для всіх необхідних моделей даних. `err = DB.AutoMigrate(*модель даних)`
- Використати .First для отримання одного запису таблиці
- Використати .Find для отримання набору записів таблиці
- Використати .Create для додавання запису в таблицю
- Використати .Save для оновлення запису в таблиці

Для створення реляції один до одного достатньо вказати іншу модель (модель яка відповідає точно структурі таблиці в базі даних), наприклад:

```
package models
```

```
// User має одну CreditCard, UserID є зовнішнім ключем
type User struct {
  gorm.Model
  CreditCard CreditCard  // Вказуємо, що у користувача може бути одна кредитна
картка
}

type CreditCard struct {
  gorm.Model
  Number string  // Номер кредитної картки
  UserID uint    // ID користувача, якому належить кредитна картка
}

// Отримати список користувачів з жадібною загрузкою кредитної картки
func GetAll(db *gorm.DB) ([]User, error) {
  var users []User  // Створюємо змінну для зберігання списку користувачів
  // Використовуємо Preload для жадібної загрузки кредитних карток для кожного
користувача
  // Find заповнює змінну users списком користувачів з бази даних
  err := db.Model(&User{}).Preload("CreditCard").Find(&users).Error
  return users, err  // Повертаємо список користувачів та можливу помилку
}
```

Більше інформації на сторінках документації

Для створення реляції один до багатьох достатньо вказати як слайс іншу модель (модель яка відповідає точно структурі таблиці в базі даних), наприклад:

```
package models

// User має багато CreditCards, UserID є зовнішнім ключем
type User struct {
  gorm.Model
  CreditCards []CreditCard  // Слайс CreditCard, що представляє зв'язок "один до
багатьох" між User та CreditCard
}

// CreditCard представляє модель кредитної картки в базі даних
type CreditCard struct {
  gorm.Model
  Number string  // Номер кредитної картки
  UserID uint    // UserID є зовнішнім ключем, що зв'язує кредитну картку з
користувачем
}

// GetAll витягує список користувачів з бази даних з жадібною загрузкою кредитних
карток
func GetAll(db *gorm.DB) ([]User, error) {
    var users []User  // Створюємо змінну для зберігання списку користувачів
    // Використовуємо Preload для жадібної загрузки кредитних карток для кожного
користувача
```

```
    // Find заповнює змінну users списком користувачів з бази даних
    err := db.Model(&User{}).Preload("CreditCards").Find(&users).Error
    return users, err  // Повертаємо список користувачів та можливу помилку
}
```

Більше інформації на сторінках документації