Опановування основами Go: Практичний посібник з освоєння мови Go

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

Будування Базового HTTP-Сервера

Створення базового HTTP-сервера в Go можна зробити використовуючи стандартний пакет `net/http`. У цьому розділі ми розглянемо, як побудувати простий HTTP-сервер без використання сторонніх пакетів маршрутизації. Основною ідеєю буде розгалуження шляхів URL для обробки різних запитів.

Приклад HTTP-Сервера з Розгалуженням Шляхів

```
package main

import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        switch r.URL.Path {
        case "/":
            fmt.Fprintf(w, "Головна сторінка")
        case "/about":
            fmt.Fprintf(w, "Про нас")
        case "/contact":
            fmt.Fprintf(w, "Контакти")
        default:
            http.NotFound(w, r)
        }
    })

    fmt.Println("Сервер запущено на http://localhost:8080")
    http.ListenAndServe(":8080", nil)
}
```

Пояснення

У цьому прикладі:

1. Сервер слухає на порту `8080`.
2. Функція `http.HandleFunc` встановлює обробник для кореневого шляху URL (`"/"`).

3. Всередині обробника використовується конструкція `switch` для розгалуження в залежності від `r.URL.Path`. Це дозволяє обробляти різні шляхи URL, такі як `"/"`, `"/about"` та `"/contact"`.

4. Для будь-якого іншого шляху використовується `http.NotFound` для повідомлення, що сторінка не знайдена.

Цей метод є простим способом ручного розгалуження шляхів URL, який може бути корисним для простих веб-серверів або коли використання повноцінного фреймворку для маршрутизації є надмірним.

File server

Створення простого файлового сервера в Go є досить простим завданням завдяки вбудованим функціям у пакеті `net/http`. Ви можете використовувати `http.FileServer` для створення базового сервера, який обслуговує статичні файли з вказаної директорії.

Ось приклад простого файлового сервера, який обслуговує файли з локальної директорії:

```
package main

import (
    "log"
    "net/http"
)

func main() {
    // Вказуємо директорію, яка буде кореневою для файлового сервера
    fs := http.FileServer(http.Dir("./public"))

    // Налаштовуємо маршрутизацію
    http.Handle("/", fs)

    // Виводимо інформаційне повідомлення та запускаємо сервер
    log.Println("Файловий сервер запущено на http://localhost:8080/")
    err := http.ListenAndServe(":8080", nil)
    if err != nil {
        log.Fatal(err)
    }
}
```

Пояснення:

1. `http.Dir("./public")` вказує, що файловий сервер буде використовувати директорію `public` як корінь для обслуговування файлів. Ви можете змінити шлях на іншу директорію, звідки ви хочете обслуговувати файли.

2. `http.FileServer(http.Dir("./public"))` створює новий файловий сервер.

3. `http.Handle("/", fs)` налаштовує основний маршрут для обслуговування файлів з файлового сервера.

4. `http.ListenAndServe(":8080", nil)` запускає HTTP-сервер на порту 8080.

Коли ви запустите цей код, сервер почне обслуговувати файли з директорії `public`. Ви можете звертатися до файлів у цій директорії, відкривши `http://localhost:8080/` у вашому веб-браузері.

Embeded file server

Використання пакету `embed` у Go для створення файлового сервера дозволяє вбудовувати статичні файли прямо у ваш бінарний файл. Це особливо корисно для розподілу додатків, які включають свої веб-ресурси, як-от HTML, CSS, JavaScript, зображення тощо. Ось приклад, як можна створити файловий сервер, використовуючи `embed` для вбудовування директорії з ресурсами:

```
package main

import (
    "embed"
    "net/http"
    "log"
)

//go:embed public/*
var content embed.FS

func main() {
    // Створення файлового сервера з використанням вбудованої файлової системи
    fileServer := http.FileServer(http.FS(content))

    // Налаштовуємо обробку запитів до кореня на наш файловий сервер
    http.Handle("/", fileServer)

    // Запускаємо сервер
    log.Println("Сервер запущено на http://localhost:8080/")
    err := http.ListenAndServe(":8080", nil)
    if err != nil {
        log.Fatal(err)
    }
}
```

Пояснення:

1. `//go:embed public/*` - ця директива вказує компілятору Go вбудувати вміст директорії `public` у бінарний файл. Усі файли з цієї директорії будуть доступні в бінарнику.
2. `var content embed.FS` - оголошуємо змінну `content`, яка буде використовуватися як файлова система.
3. `http.FileServer(http.FS(content))` - створюємо новий файловий сервер, що використовує вбудовані файли.

4. `http.Handle("/", fileServer)` - налаштовуємо маршрутизацію так, щоб всі запити до кореня обслуговувалися нашим файловим сервером.

5. `http.ListenAndServe(":8080", nil)` - запускаємо HTTP-сервер на порту 8080.

Цей код створює простий HTTP-сервер, який обслуговує статичні файли з вбудованої директорії `public`. Тепер, запустивши цей сервер, ви зможете доступити до статичних файлів, відкривши `http://localhost:8080/` у вашому веб-браузері.

Аналіз та Обробка JSON

JSON (JavaScript Object Notation) є широко використовуваним форматом для обміну даними, особливо у веб-розробці. У Go, пакет `encoding/json` надає зручні інструменти для серіалізації (конвертації структур даних у JSON) та десеріалізації (конвертації JSON назад у структури даних).

Прийом та Відправлення JSON у HTTP-запитах

Розглянемо приклад створення HTTP-сервера, який приймає JSON у POST-запитах та відправляє JSON у відповідях.

Приклад Сервера:

```
package main

import (
    "encoding/json"
    "log"
    "net/http"
)

// Структура для запиту
type request struct {
    Name string `json:"name"`
}

// Структура для відповіді
type response struct {
    Greeting string `json:"greeting"`
}

func main() {
    http.HandleFunc("/greet", func(w http.ResponseWriter, r *http.Request) {
        if r.Method != "POST" {
            w.WriteHeader(http.StatusMethodNotAllowed)
            return
        }

        var req request
        if err := json.NewDecoder(r.Body).Decode(&req); err != nil {
            http.Error(w, err.Error(), http.StatusBadRequest)
            return
        }
```

```
        resp := response{Greeting: "Привіт, " + req.Name + "!"}
        w.Header().Set("Content-Type", "application/json")
        json.NewEncoder(w).Encode(resp)
    })

    log.Println("Сервер запущено на http://localhost:8080/")
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Пояснення:

1. **Обробник `/greet`:** Ми створюємо обробник для шляху `/greet`, який приймає лише POST-запити.
2. **Десеріалізація JSON:** Використовуючи `json.NewDecoder(r.Body).Decode(&req)`, ми десеріалізуємо тіло запиту JSON у структуру Go.
3. **Створення Відповіді:** Ми створюємо відповідь і серіалізуємо її назад у JSON за допомогою `json.NewEncoder(w).Encode(resp)`.
4. **Запуск Сервера:** Сервер слухає на порту 8080 і обслуговує запити на шляху `/greet`.

Тестування Сервера

Ви можете тестувати цей сервер, відправляючи POST-запити з JSON тілом, наприклад, використовуючи `curl`:

```
curl -X POST http://localhost:8080/greet -H "Content-Type: application/json" -d
'{"name":"Ім'я"}'
```

Цей запит має повернути відповідь у форматі JSON, таку як: `{"greeting":"Привіт, Ім'я!"}`.

Висновок

Робота з JSON є фундаментальною у створенні веб-сервісів. Go забезпечує прості та ефективні інструменти для обробки JSON, що робить мову відмінним вибором для розробки сучасних веб-додатків і API.

Створення RESTful API з Використанням `http.ServeMux` у Go

`http.ServeMux` є HTTP request multiplexer, який дозволяє легко визначати маршрути (routes) та їх обробники. Використання `http.ServeMux` є одним зі способів створення RESTful API у Go.

Приклад RESTful API

Ось базовий приклад створення RESTful API, який включає маршрути для обробки GET та POST запитів:

```go
package main

import (
    "encoding/json"
    "fmt"
    "log"
    "net/http"
)

type Product struct {
    ID    int    `json:"id"`
    Name  string `json:"name"`
}

var products = []Product{
    {ID: 1, Name: "Кава"},
    {ID: 2, Name: "Чай"},
}

// Обслуговує POST та GET запроси на /products
func productsHandler(w http.ResponseWriter, r *http.Request) {
    if r.Method == "GET" {
        json.NewEncoder(w).Encode(products)
    } else if r.Method == "POST" {
        var newProduct Product
        json.NewDecoder(r.Body).Decode(&newProduct)
        products = append(products, newProduct)
        w.WriteHeader(http.StatusCreated)
    } else {
        w.WriteHeader(http.StatusMethodNotAllowed)
    }
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/products", productsHandler)

    fmt.Println("Сервер запущено на http://localhost:8080/")
    log.Fatal(http.ListenAndServe(":8080", mux))
}
```

Пояснення:

1. **Структура `Product`:** Визначаємо базову структуру даних для нашого API.
2. **Список `products`:** Створюємо тестовий набір даних.
3. **Функція `productsHandler`:** Визначаємо обробник, який буде реагувати на GET та POST запити. GET запити повертають список продуктів, а POST запити додають новий продукт у список.
4. **Створення `ServeMux`:** `http.NewServeMux()` створює новий HTTP маршрутизатор.

5. **Реєстрація Обробника:** `mux.HandleFunc("/products", productsHandler)` реєструє обробник для маршруту `/products`.
6. **Запуск Сервера:** `http.ListenAndServe(":8080", mux)` запускає HTTP-сервер на порту 8080 з нашим маршрутизатором `mux`.

Тестування API:

- Виконайте GET запит до `http://localhost:8080/products` для отримання списку продуктів.
- Виконайте POST запит до `http://localhost:8080/products` з JSON-тілом, наприклад, `{"id": 3, "name": "Молоко"}`, для додавання нового продукту.

> Цей приклад демонструє базовий спосіб створення RESTful API у Go з використанням `http.ServeMux`. Однак, для більш складних застосунків може знадобитися використання більш потужних фреймворків маршрутизації чи мідлварів для кращого управління запитами, маршрутизацією та обробкою помилок.