

# Опановування основами Go: Практичний посібник з освоєння мови Go

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

### Фреймворк Fiber



Фреймворк Fiber для Go — це веб-фреймворк, натхненний Express.js і побудований на основі Fasthttp, який є найшвидшим HTTP двигуном для Go. Основна мета Fiber - спростити процес розробки, забезпечуючи при цьому високу продуктивність і низьке використання пам'яті.

Основні особливості Fiber включають:

- Робастна Маршрутизація:** Маршрутизація у Fiber схожа на Express, що робить її зрозумілою та легкою для роботи. Це дозволяє легко налаштовувати маршрути для вашого додатку.
- Обслуговування Статичних Файлів:** Fiber дозволяє легко обслуговувати статичні файли, такі як HTML, CSS і JavaScript, визначаючи статичні маршрути.
- Висока Продуктивність:** Завдяки побудові на Fasthttp, додатки, створені за допомогою Fiber, мають високу продуктивність.
- Готовність до Роботи з API:** Fiber є відмінним вибором для створення REST API на Go. Він спрощує приймання та відправлення даних.
- Гнучка Підтримка Мідлварів:** У Fiber є широкий вибір існуючих мідлварів, а також можливість створення власних.
- Низьке Використання Пам'яті:** Fiber має низький відбиток пам'яті, що дозволяє реалізовувати функціональність, не переймаючись про використання пам'яті вашого додатку.

7. **Швидке Програмування:** API Fiber добре спроектоване і легке для вивчення, особливо якщо ви маєте досвід роботи з Express.js.
8. **Підтримка Шаблонних Двигунів:** Fiber підтримує кілька шаблонних двигунів, таких як Handlebars і Pug.
9. **Підтримка WebSocket:** Fiber дозволяє використовувати можливості WebSocket для створення інтерактивних користувацьких досвідів.
10. **Обмежувач Частоти Запитів:** З Fiber легко обмежувати повторні запити до публічних API та кінцевих точок, уникнення зловмисних запитів.

Fiber є потужним інструментом для розробки веб-додатків на Go, надаючи гнучкість, продуктивність та низьке використання ресурсів.

Як почати ?

Ось приклад простого веб-сервісу, реалізованого з використанням фреймворку Fiber для Go:

### Встановлення Fiber

Спочатку вам потрібно встановити Fiber. Це можна зробити за допомогою команди `go get`:

```
go get github.com/gofiber/fiber/v2
```

### Створення основного файлу

Створіть файл `main.go` і додайте до нього наступний код:

```
package main

import (
    fiber "github.com/gofiber/fiber/v2"
    "github.com/gofiber/fiber/v2/middleware/logger"
    "log"
)

func main() {
    // Створення нового Fiber додатку
    app := fiber.New()
    // Використання логера
    app.Use(logger.New(logger.Config{
        Format: "[$ip]:$port $status - $method $path\n",
    }))
    // Визначення GET маршруту
    app.Get("/", func(c *fiber.Ctx) error {
        // Відправлення відповіді
        return c.JSON(fiber.Map{"message": "Hello, World!"})
    })
}
```

```
// Запуск сервера на порту 8080
if err := app.Listen(":8080"); err != nil {
    log.Fatal(err)
}
```

Цей код створює простий Fiber сервер, який слухає на порту 8080 і відповідає "Hello, World!" на HTTP GET запити до кореневого шляху (/).

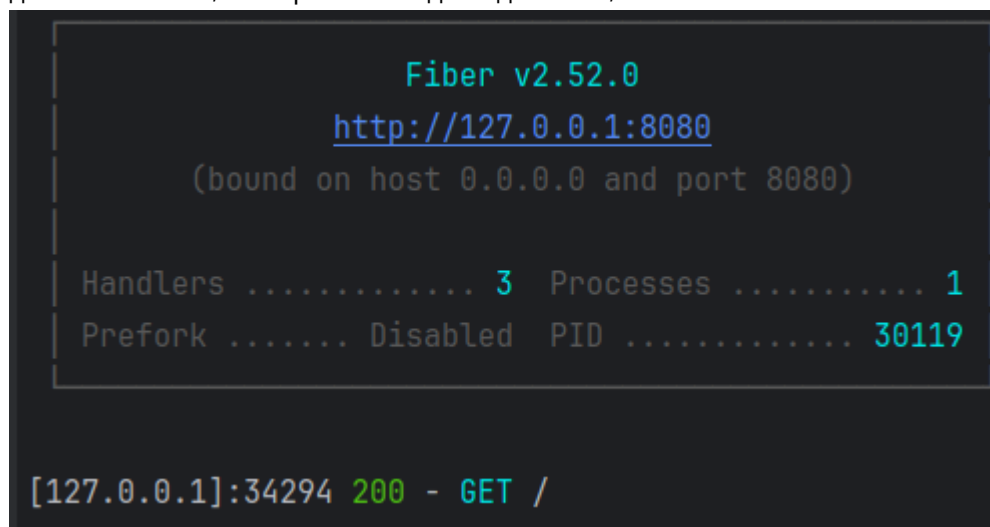
Зверніть увагу, що Fiber не має автоматичного логгера. Для моніторингу та відлагодження веб-додатків можна використати готовий middleware фреймворку Fiber.

## Запуск серверу

Щоб запустити сервер, виконайте файл `main.go` за допомогою Go:

```
go run main.go
```

Тепер, коли ви відкриєте `http://localhost:8080` у вашому веб-браузері або зробите запит за допомогою curl, ви отримаєте відповідь "Hello, World!".



```
Fiber v2.52.0
http://127.0.0.1:8080
(bound on host 0.0.0.0 and port 8080)

Handlers ..... 3  Processes ..... 1
Prefork ..... Disabled  PID ..... 30119

[127.0.0.1]:34294 200 - GET /
```

Цей приклад демонструє основи створення веб-сервісу з використанням Fiber, показуючи його простоту і зручність для швидкої розробки. Fiber надає багато можливостей для більш складних веб-додатків, включаючи роботу з міدلварами, маршрутизацією, виконанням запитів і відповідей та багато іншого.

## Робастна маршрутизація

Робастна маршрутизація у фреймворку Fiber для Go дозволяє легко визначати маршрути та параметри маршруту. Ось приклад того, як можна створити декілька маршрутів з параметрами і без них:

```
package main
```

```
import (
    "fmt"
    "log"

    "github.com/gofiber/fiber/v2"
)

func main() {
    app := fiber.New()

    // Основний маршрут
    app.Get("/", func(c *fiber.Ctx) error {
        return c.SendString("Головна сторінка")
    })

    // Маршрут з параметром
    app.Get("/greeting/:name", func(c *fiber.Ctx) error {
        name := c.Params("name") // Отримання параметра маршруту
        return c.SendString(fmt.Sprintf("Привіт, %s!", name))
    })

    // Маршрут з необов'язковим параметром
    app.Get("/hello/:name?", func(c *fiber.Ctx) error {
        name := c.Params("name", "Невідомий") // Значення за замовчуванням, якщо
        параметр відсутній
        return c.SendString(fmt.Sprintf("Привіт, %s!", name))
    })

    // Маршрут з декількома параметрами
    app.Get("/greet/:name/:age/:city", func(c *fiber.Ctx) error {
        name := c.Params("name")
        age := c.Params("age")
        city := c.Params("city")
        return c.SendString(fmt.Sprintf("%s з міста %s, тобі %s років", name,
city, age))
    })

    // Запуск сервера
    log.Fatal(app.Listen(":3000"))
}
```

У цьому прикладі:

- `/`: Основний маршрут.
- `/greeting/:name`: Маршрут з обов'язковим параметром `name`.
- `/hello/:name?`: Маршрут з необов'язковим параметром `name`.
- `/greet/:name/:age/:city`: Маршрут з декількома параметрами.

Така маршрутизація дуже зручна для створення RESTful API або веб-додатків, де потрібно гнучко обробляти різні запити. Fiber дозволяє легко управляти маршрутами та їх параметрами, роблячи код чистішим та зрозумілішим.

## Групування маршрутів (використання middleware для груп)

Групування маршрутів у фреймворку Fiber дозволяє організувати маршрути, які поділяють спільні характеристики, наприклад, загальні префікси шляху або middleware. Нижче наведено приклад групування маршрутів з використанням middleware:

```
package main

import (
    "github.com/gofiber/fiber/v2"
)

func main() {
    app := fiber.New()

    // Middleware для всього додатку
    app.Use(func(c *fiber.Ctx) error {
        // Логіка загального middleware
        return c.Next()
    })

    // Група API маршрутів
    api := app.Group("/api", func(c *fiber.Ctx) error {
        // Middleware, специфічне для групи "/api"
        return c.Next()
    })

    // Додавання маршрутів до групи
    api.Get("/user", func(c *fiber.Ctx) error {
        // Обробка запиту
        return c.SendString("API User Endpoint")
    })

    api.Get("/product", func(c *fiber.Ctx) error {
        // Обробка запиту
        return c.SendString("API Product Endpoint")
    })

    // Запуск сервера
    app.Listen(":3000")
}
```

У цьому прикладі:

- **app.Use**: Визначає middleware, який застосовується до всіх маршрутів у додатку.
- **app.Group**: Створює групу маршрутів. У цьому випадку, створюється група для API зі своїм middleware, яке виконується перед маршрутами в цій групі.
- **api.Get**: Додає конкретні маршрути до групи `/api`, такі як `/api/user` і `/api/product`.

Такий підхід дозволяє організувати маршрутизацію вашого додатку більш структуровано та чисто, особливо коли в додатку багато маршрутів.

## Групування маршрутів (використання middleware для груп)

Групування маршрутів у фреймворку Fiber дозволяє організувати маршрути, які поділяють спільні характеристики, наприклад, загальні префікси шляху або middleware. Нижче наведено приклад групування маршрутів з використанням middleware:

```
package main

import (
    "github.com/gofiber/fiber/v2"
)

func main() {
    app := fiber.New()

    // Middleware для всього додатку
    app.Use(func(c *fiber.Ctx) error {
        // Логіка загального middleware
        return c.Next()
    })

    // Група API маршрутів
    api := app.Group("/api", func(c *fiber.Ctx) error {
        // Middleware, специфічне для групи "/api"
        return c.Next()
    })

    // Додавання маршрутів до групи
    api.Get("/user", func(c *fiber.Ctx) error {
        // Обробка запиту
        return c.SendString("API User Endpoint")
    })

    api.Get("/product", func(c *fiber.Ctx) error {
        // Обробка запиту
        return c.SendString("API Product Endpoint")
    })

    // Запуск сервера
    app.Listen(":3000")
}
```

У цьому прикладі:

- **app.Use**: Визначає middleware, який застосовується до всіх маршрутів у додатку.
- **app.Group**: Створює групу маршрутів. У цьому випадку, створюється група для API зі своїм middleware, яке виконується перед маршрутами в цій групі.
- **api.Get**: Додає конкретні маршрути до групи `/api`, такі як `/api/user` і `/api/product`.

Такий підхід дозволяє організувати маршрутизацію вашого додатку більш структуровано та чисто, особливо коли в додатку багато маршрутів.

## Fiber websocket

```
package main

import (
    "log"

    "github.com/gofiber/contrib/websocket"
    "github.com/gofiber/fiber/v2"
)

func main() {
    app := fiber.New()

    app.Use("/ws", func(c *fiber.Ctx) error {
        // IsWebSocketUpgrade повертає true, якщо клієнт
        // запросив оновлення до протоколу WebSocket.
        if websocket.IsWebSocketUpgrade(c) {
            // c.Locals додається allowed: true до *fiber.Ctx для використання в
            наступних обробниках
            c.Locals("allowed", true)
            return c.Next()
        }
        return fiber.ErrUpgradeRequired
    })

    app.Get("/ws/:id", websocket.New(func(c *websocket.Conn) {
        // c.Locals додається до *websocket.Conn
        log.Println("allowed:", c.Locals("allowed")) // true
        log.Println("id:", c.Params("id"))           // 123
        log.Println("v:", c.Query("v"))               // 1.0
        log.Printf("session: [%s]\n", c.Cookies("session")) // ""

        // websocket.Conn зв'язки
        https://pkg.go.dev/github.com/fasthttp/websocket?tab=doc#pkg-index
        var (
            mt int
            msg []byte
            err error
        )
        for { // Нескінченний цикл для обробки повідомлень
            if mt, msg, err = c.ReadMessage(); err != nil {
                log.Println("читання:", err)
                break
            }
            log.Printf("отримано: %s", string(msg))
            msg = []byte("повернення: " + string(msg))
            // Повернення повідомлення
            if err = c.WriteMessage(mt, msg); err != nil {
```

```
        log.Println("запис:", err)
        break
    }
}
log.Printf("вихід\n")
}))

if err := app.Listen(":3000"); err != nil {
    log.Fatal()
}
// Доступ до сервера websocket: ws://localhost:3000/ws/123?v=1.0
}
```

Цей код демонструє створення простого WebSocket сервера з використанням фреймворку Fiber та пакету [github.com/gofiber/contrib/websocket](https://github.com/gofiber/contrib/websocket) для Go. Ось опис основних компонентів та їх функціональності:

### Структура коду

- Імпорт необхідних пакетів:** Імпортуються Fiber ([github.com/gofiber/fiber/v2](https://github.com/gofiber/fiber/v2)) та WebSocket ([github.com/gofiber/contrib/websocket](https://github.com/gofiber/contrib/websocket)).
- Ініціалізація Fiber додатку:** Створюється новий екземпляр Fiber.
- Використання Middleware для WebSocket:** Middleware перевіряє, чи запит є оновленням до WebSocket. Якщо так, він додає до контексту прапорець `allowed` і передає управління наступному обробнику.
- Маршрут WebSocket:** Встановлюється маршрут `/ws/:id` для обслуговування WebSocket з'єднань.
- Обробник WebSocket:** Ініціалізується обробник для з'єднань WebSocket, який виконує наступні функції:
  - Отримує та логує дані, такі як параметри запиту, куки, інформацію про доступ з `c.Locals`.
  - Читає повідомлення від клієнта в нескінченному циклі та логує їх.
  - Відправляє відповідь назад клієнту.
- Запуск сервера:** Сервер слухає на порту `3000`.

### Функціональність

- WebSocket Middleware:** Перед обробкою WebSocket з'єднання, middleware перевіряє, чи запит від клієнта є запитом на оновлення до WebSocket. Якщо ні, клієнту повертається помилка.
- Обробка WebSocket з'єднань:** Коли клієнт підключається до WebSocket на маршруті `/ws/:id`, сервер здійснює обмін повідомленнями з клієнтом. Сервер читає повідомлення, відправлені клієнтом, і відправляє відповідь.
- Логування:** Код логує різні аспекти з'єднання, такі як наявність доступу, параметри запиту, значення куки та вміст повідомлень.



## Використання

Цей сервер можна використовувати для створення додатків реального часу, таких як чати, ігри, або будь-які інші додатки, яким потрібна двостороння комунікація між клієнтом та сервером у реальному часі. WebSocket пропонує низьку затримку та високу ефективність для такого роду взаємодій.