DWARVES FOUNDATION

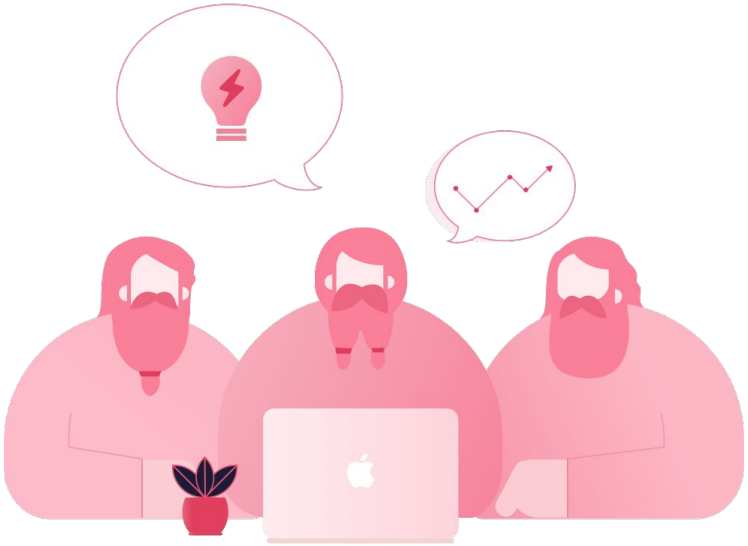
Web Development with Go

Author Name

Hieu Phan

@hieupq

andy@dwarvesv.com



Agenda

Topic summary

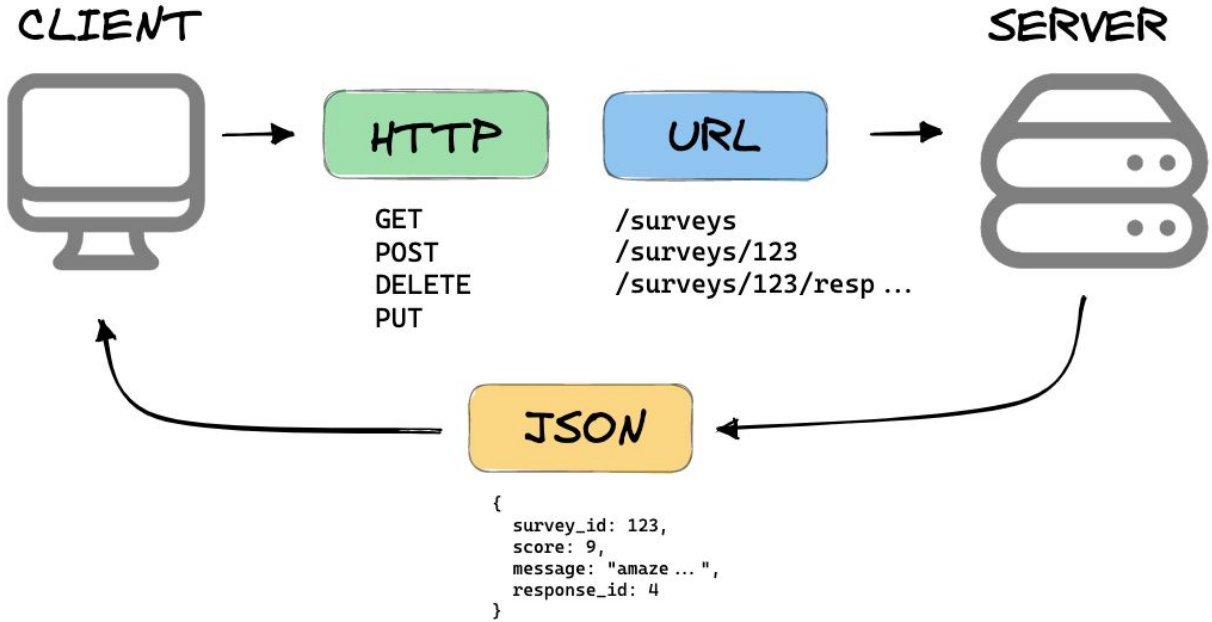
1. Introduction
2. Simple HTTP Server
3. Handling routing, request parsing, and response rendering
4. Working with middleware and authentication
5. Demo

Introduction to HTTP and RESTful APIs

HTTP and REST

- HTTP (HyperText Transfer Protocol) is a client-server protocol that forms the foundation of data exchange on the web
- Clients, such as web browsers, initiate requests, and servers respond with the requested resources, which can include HTML documents, images, videos, and other types of data
- REST (REpresentational State Transfer) is an architectural style for distributed hypermedia systems
- In REST, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs)

HTTP and REST



HTTP and REST

- Go's `net/http` package provides a full set of functions and types for building HTTP clients, servers, and other HTTP-based operations.
- Many popular Go libraries and frameworks, like the Gin Web Framework, build on top of the `net/http` package to simplify the development of RESTful APIs

Simple HTTP Server

Simple HTTP server with **net/http**

- We first import the `net/http` package.
- In main(), we call http.HandleFunc() to handle all requests "/" with our handler.
- This will start the server and print "Hello World!" for any requests to localhost:8080.

```go
package main

import (
  "fmt"
  "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintf(w, "Hello World!")
}

func main() {
  http.HandleFunc("/", handler)
  http.ListenAndServe(":8080", nil)
}
```

HTTP server with Gin

- We can create a simple router with `gin.Default()`.
- We define some routes:
  - GET /user/:name - returns a string greeting the name
  - POST /user - takes a JSON user, validates, and echoes back the name/age
- In practice, patterns seen here would be separate into handlers, include interfaces and structs elsewhere, etc.

```go
package main

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

type JSON struct {
  Name string `json:"name"`
  Age  int    `json:"age"`
}

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

  r.GET("/user/:name", func(c *gin.Context) {
    name := c.Param("name")
    c.String(200, "Hello %s", name)
  })

  r.POST("/user", func(c *gin.Context) {
    var json JSON

    if err := c.ShouldBindJSON(&json); err != nil {
      c.JSON(400, gin.H{"error": err.Error()})
      return
    }

    c.JSON(200, gin.H{"name": json.Name, "age": json.Age})
  })

  r.Run()
}
```

DWARVES FOUNDATION

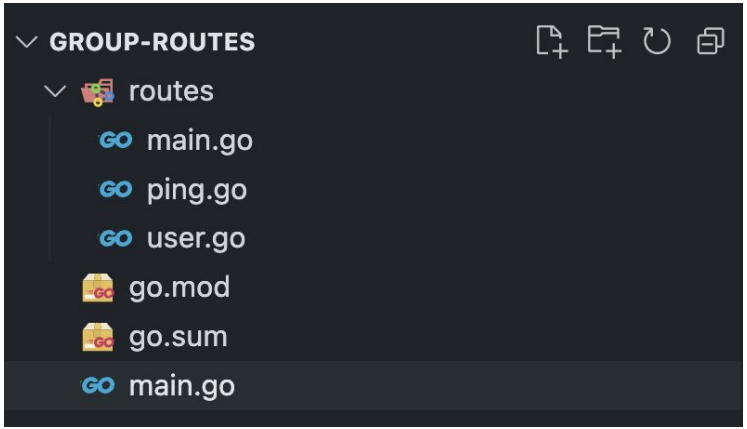
Routes grouping

- Gin does support routes grouping, that helps organize our code in an elegant way
  (e.g: versioning with many routes)

```go
package routes

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

// Default router
var router = gin.Default()

// Run will start the server
func Run() {
    getRoutes()
    router.Run("localhost:9000")
}

// getRoutes will create our routes of our entire application
// this way every group of routes will be in a single file
func getRoutes() {
    v1 := router.Group("/v1")
    addUserRoutes(v1)
    addPingRoutes(v1)

    v2 := router.Group("/v2")
    addPingRoutes(v2)
}
```

Routes grouping

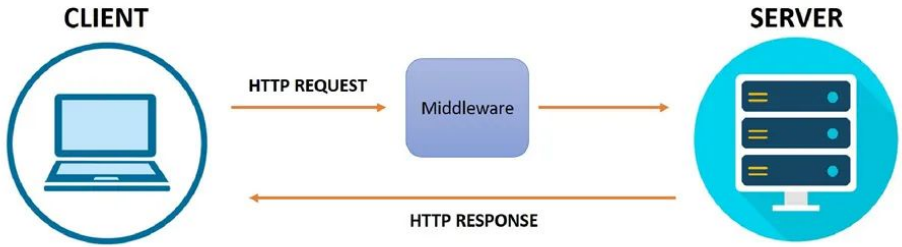
```go
package routes

import (
    "net/http"

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

// addUserRoutes will add the routes for the user group
func addUserRoutes(rg *gin.RouterGroup) {
    user := rg.Group("/user")

    user.GET("/", func(c *gin.Context) {
        c.JSON(http.StatusOK, gin.H{"message": "users"})
    })
    user.GET("/comments", func(c *gin.Context) {
        c.JSON(http.StatusOK, gin.H{"message": "user comments"})
    })
    user.GET("/posts", func(c *gin.Context) {
        c.JSON(http.StatusOK, gin.H{"message": "user posts"})
    })
}
```




```go
package main

import "go-2023/group-routes/routes"

func main() {
    // Our server will live in the routes package
    routes.Run()
}
```

Middleware and Auth

- Here are some common reasons to use middlewares in Gin:
  - Authentication - Validate tokens, auth headers, etc to protect routes
  - Authorization - Check user roles/permissions to allow or deny access
  - Logging - Log requests, add request IDs, etc
  - CORS - Enable CORS for cross-origin requests
  - Rate limiting - Limit requests to prevent abuse/DDoS



```go
package main

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

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

  // Authentication middleware
  r.Use(authMiddleware())

  // Auth API
  r.GET("/auth", func(c *gin.Context) {
    c.JSON(200, gin.H{
      "message": "auth successful",
    })
  })

  // Protected API
  r.GET("/data", func(c *gin.Context) {
    c.JSON(200, gin.H{
      "data": "secret",
    })
  })

  r.Run()
}
```

```go
func authMiddleware() gin.HandlerFunc {
  return func(c *gin.Context) {
    // Validate auth token
    if isValid := validateToken(c.GetHeader("Authorization")); !isValid {
      c.AbortWithStatus(http.StatusUnauthorized)
      return
    }

    c.Next()
  }
}

func validateToken(token string) bool {
  // token validation logic
  return true
}
```

Gracefully shutdown

- Notify interruption/ termination signal without context

```go
package main

import (
    "context"
    "log"
    "net/http"
    "os"
    "os/signal"
    "syscall"
    "time"

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

func main() {
    router := gin.Default()
    router.GET("/", func(c *gin.Context) {
        time.Sleep(5 * time.Second)
        c.String(http.StatusOK, "Welcome Gin Server")
    })

    srv := &http.Server{
        Addr:    ":8080",
        Handler: router,
    }

    // Initializing the server in a goroutine so that
    // it won't block the graceful shutdown handling below
    go func() {
        if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
            log.Fatalf("listen: %s\n", err)
        }
    }()

    // Wait for interrupt signal to gracefully shutdown the server with
    // a timeout of 5 seconds.
    quit := make(chan os.Signal, 1)
    // kill (no param) default send syscall.SIGTERM
    // kill -2 is syscall.SIGINT
    // kill -9 is syscall.SIGKILL but can't be catch, so don't need add it
    signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM)
    <-quit
    log.Println("Shutting down server...")

    // The context is used to inform the server it has 5 seconds to finish
    // the request it is currently handling
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()
    if err := srv.Shutdown(ctx); err != nil {
        log.Fatal("Server forced to shutdown: ", err)
    }

    log.Println("Server exiting")
}
```

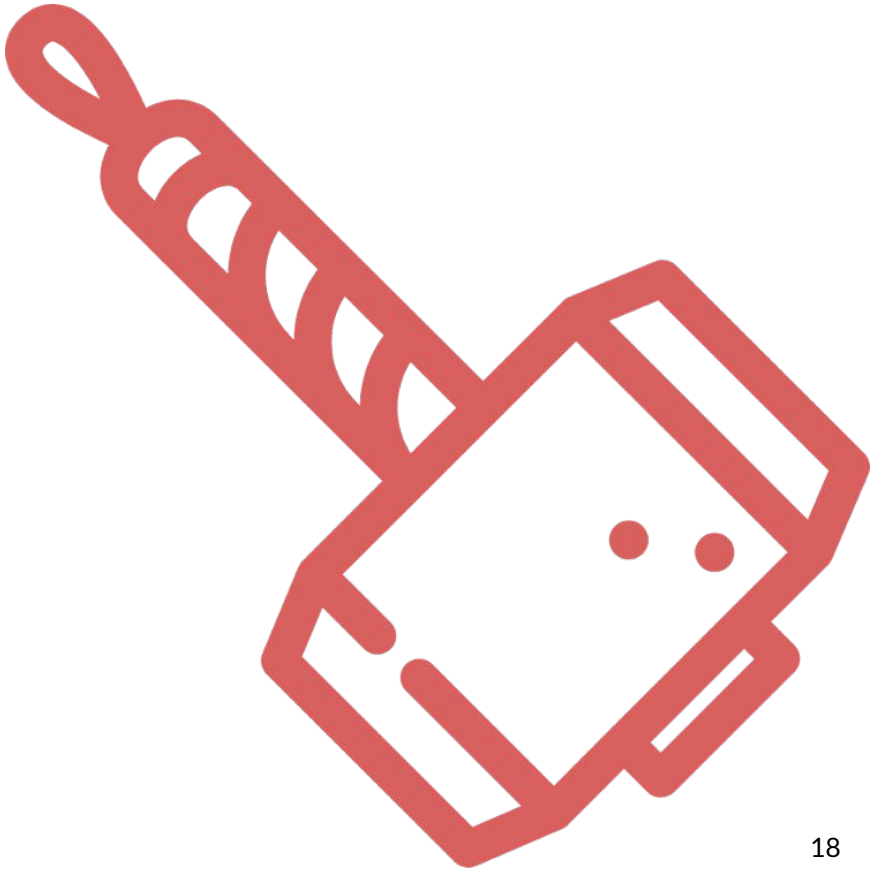
Gracefully shutdown

- Notify interruption/ termination signal with context

```go
package main

import (
    "context"
    "log"
    "net/http"
    "os/signal"
    "syscall"
    "time"

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

func main() {
    // Create context that listens for the interrupt signal from the OS.
    ctx, stop := signal.NotifyContext(context.Background(), syscall.SIGINT, syscall.SIGTERM)
    defer stop()

    router := gin.Default()
    router.GET("/", func(c *gin.Context) {
        time.Sleep(10 * time.Second)
        c.String(http.StatusOK, "Welcome Gin Server")
    })

    srv := &http.Server{
        Addr:    ":8080",
        Handler: router,
    }

    // Initializing the server in a goroutine so that
    // it won't block the graceful shutdown handling below
    go func() {
        if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
            log.Fatalf("listen: %s\n", err)
        }
    }()

    // Listen for the interrupt signal.
    <-ctx.Done()

    // Restore default behavior on the interrupt signal and notify user of shutdown.
    stop()
    log.Println("shutting down gracefully, press Ctrl+C again to force")

    // The context is used to inform the server it has 5 seconds to finish
    // the request it is currently handling
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()
    if err := srv.Shutdown(ctx); err != nil {
        log.Fatal("Server forced to shutdown: ", err)
    }

    log.Println("Server exiting")
}
```

Demo

Demo - Zer0 to Hero

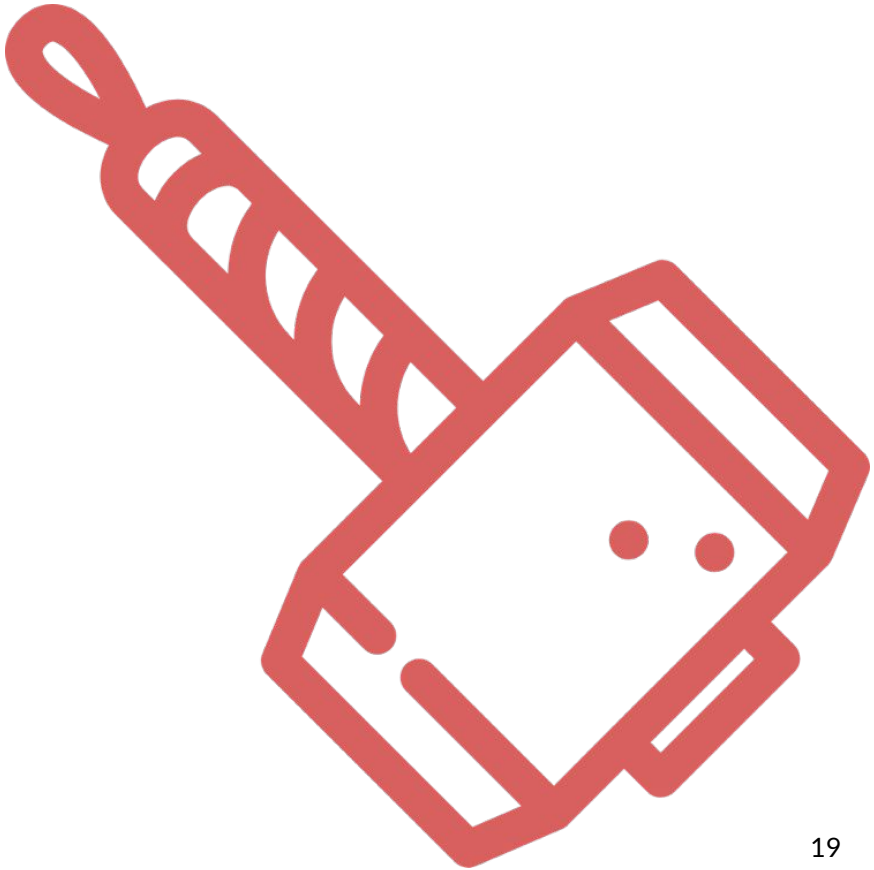


Assignment Day 6

Simple E-commerce: Add to Cart, Remove from Cart, and Checkout

Product management APIs with basic authentication

Cart management APIs



Assignment Day 6

Product Management - Auth BasicAuth, JWT Auth,...

POST /products: Create a new product. It receives product details as JSON input.

PUT /products/{product_id}: Update a product's details. It receives updated product details as JSON input.

DELETE /products/{product_id}: Delete a product by its ID.

GET /products: Retrieve a list of all products.

Assignment Day 6

Shopping Cart - Without Auth

POST /cart/add: Add items to the cart. It receives a product ID and quantity as JSON input.

DELETE /cart/remove: Remove items from the cart. It receives a product ID as JSON input.

POST /cart/checkout: Checkout and clear the cart. It returns a receipt with the total price.

Reference

Resources & Reference links

- https://gin-gonic.com/
- https://go.dev/doc/

DWARVES FOUNDATION

Thank You

DWARVES FOUNDATION

Q&A