Taller de Lenguajes de Programación

API Rest & ORM

Javier Villegas Lainas

07 de Julio 2025

Introducción

Desarrollo de APIs con GO

Go es un lenguaje de programación moderno desarrollado por Google que está ganando popularidad entre los desarrolladores para crear API. Go es un lenguaje compilado, lo que significa que produce código rápido y eficiente que puede ejecutarse en diversas plataformas. Go también está diseñado para ser fácil de leer y escribir, por lo que es una gran opción para la construcción de sistemas de software grandes y complejos.

Una de las principales ventajas de utilizar Go para crear API es su excelente compatibilidad con la concurrencia. La concurrencia es la capacidad de ejecutar múltiples tareas al mismo tiempo, y Go facilita la escritura de código que puede aprovechar los procesadores multinúcleo y otros recursos de hardware. Esto significa que las APIs de Go pueden manejar altos niveles de tráfico y peticiones sin ralentizarse o bloquearse.

Otro beneficio de usar Go para construir APIs es su soporte integrado para JSON (JavaScript Object Notation), un formato de datos ligero que se usa ampliamente para el intercambio de datos en aplicaciones web. El soporte de Go para JSON facilita la creación de APIs que pueden aceptar y devolver datos en este formato, simplificando el proceso de construcción e integración de APIs con otros sistemas de software.

En general, Go es un lenguaje potente y flexible, muy adecuado para crear APIs. Su velocidad, soporte de concurrencia y soporte JSON incorporado lo convierten en una gran opción para los desarrolladores que buscan crear API escalables y de alto rendimiento.

Introducción a net/http y Estructura Básica

Para entender cómo funciona un servidor HTTP en Go, piensen en un restaurante. El net/http es como el edificio del restaurante y la cocina. Nosotros definimos las recetas (handlers) para cada plato (ruta) que los clientes (peticiones HTTP) pueden pedir.

Cuando un cliente hace una petición (por ejemplo, GET /tasks), esa petición llega a nuestro servidor. El servidor, actuando como un anfitrión, mira la "orden" y la dirige a la "cocina" (el handler) que sabe cómo preparar ese plato.

Vamos a ver cómo se ve esto en código. Crearemos un servidor simple que responde a algunas rutas.

```Go
// main.go
package main

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

func main() {
	// Definir rutas manualmente
	http.HandleFunc("/", homeHandler)
	http.HandleFunc("/tasks", tasksHandler)
	http.HandleFunc("/health", healthHandler)

	fmt.Println("🚀 Servidor iniciado en puerto 8080")
	log.Fatal(http.ListenAndServe(":8080", nil))
}

// Handler para la ruta principal
func homeHandler(w http.ResponseWriter, r *http.Request) {
	w.WriteHeader(http.StatusOK)
```

```
	fmt.Fprintf(w, "¡Bienvenido a la API de Gestión de Tareas!")
}

// Handler básico para tareas
func tasksHandler(w http.ResponseWriter, r *http.Request) {
	switch r.Method {
	case "GET":
		fmt.Fprintf(w, "Listando todas las tareas")
	case "POST":
		fmt.Fprintf(w, "Creando nueva tarea")
	default:
		w.WriteHeader(http.StatusMethodNotAllowed)
		fmt.Fprintf(w, "Método no permitido")
	}
}

// Handler para verificar salud del servidor
func healthHandler(w http.ResponseWriter, r *http.Request) {
	w.Header().Set("Content-Type", "application/json")
	w.WriteHeader(http.StatusOK)
	fmt.Fprintf(w, `{"status": "ok", "message": "Servidor funcionando
correctamente"}`)
}
```

Ejecutamos nuestro proyecto con

```shell
go run main.go
```

Para testear nuestra api, abrimos otra sesión de nuestra línea de comando y ejecutamos lo siguiente:

```shell
curl http://localhost:8080/

curl http://localhost:8080/tasks

curl http://localhost:8080/health
```

Manejo de Peticiones y Respuestas

Cuando construimos APIs, necesitamos interactuar con los datos que nos envía el cliente y enviarles datos de vuelta. http.Request y http.ResponseWriter son nuestras herramientas principales aquí.

- http.Request: Piensen en él como el formulario de pedido que el cliente llena.
    - Método (GET, POST, PUT, DELETE): r.Method - Qué tipo de operación quiere el cliente.
    - Ruta/URL: r.URL.Path - A qué recurso se dirige la petición.
    - Parámetros de URL: Partes de la URL que actúan como identificadores (ej. /tasks/123 donde 123 es el ID). net/http no los parsea automáticamente, veremos cómo hacerlo manualmente o con frameworks.
    - Query Parameters: Datos adicionales en la URL después de un ? (ej. /tasks?status=completed). r.URL.Query() devuelve un mapa de estos.
    - Headers: Información meta sobre la petición (ej. Content-Type, Authorization). r.Header.Get("Content-Type").

    - Body: El "contenido" de la petición, común en POST/PUT para enviar JSON, XML, etc. r.Body.

- http.ResponseWriter: Es el plato de entrega de su restaurante.
    - Escribir el cuerpo de la respuesta: fmt.Fprintf(w, "mensaje") o w.Write([]byte("mensaje")).
    - Establecer encabezados: w.Header().Set("Content-Type", "application/json").
    - Establecer código de estado: w.WriteHeader(http.StatusOK) (200 OK), http.StatusCreated (201 Created), http.StatusBadRequest (400 Bad Request), etc. Importante: WriteHeader debe llamarse antes de escribir el cuerpo.

Veamos esto con un ejemplo:

```go
// handlers/tasks.go
package main

import (
    "encoding/json"
    "fmt"
    "log"
    "net/http"
    "strconv"
    "strings"
    "time"
)

// Estructura de datos para una tarea
```

```go
type Task struct {
    ID          int       `json:"id"`
    Title       string    `json:"title"`
    Description string    `json:"description"`
    Completed   bool      `json:"completed"`
    CreatedAt   time.Time `json:"created_at"`
}

// Almacén temporal en memoria
var tasks []Task
var nextID = 1

// Handler mejorado para tareas con manejo completo
func tasksHandler(w http.ResponseWriter, r *http.Request) {
    // Configurar headers comunes
    w.Header().Set("Content-Type", "application/json")

    switch r.Method {
    case "GET":
        handleGetTasks(w, r)
    case "POST":
        handleCreateTask(w, r)
    default:
        w.WriteHeader(http.StatusMethodNotAllowed)
        json.NewEncoder(w).Encode(map[string]string{
```

```
            "error": "Método no permitido",
        })
    }
}

// GET /tasks - Listar tareas con filtros opcionales
func handleGetTasks(w http.ResponseWriter, r *http.Request) {
    // Leer query parameters
    queryParams := r.URL.Query()
    completed := queryParams.Get("completed")
    limit := queryParams.Get("limit")

    // Aplicar filtros
    filteredTasks := tasks

    // Filtrar por estado completado
    if completed != "" {
        isCompleted, err := strconv.ParseBool(completed)
        if err == nil {
            var filtered []Task
            for _, task := range tasks {
                if task.Completed == isCompleted {
                    filtered = append(filtered, task)
                }
            }
```

```
            filteredTasks = filtered
        }
    }

    // Aplicar límite
    if limit != "" {
        if limitNum, err := strconv.Atoi(limit); err == nil && limitNum > 0 {
            if limitNum < len(filteredTasks) {
                filteredTasks = filteredTasks[:limitNum]
            }
        }
    }

    // Leer headers importantes
    userAgent := r.Header.Get("User-Agent")
    log.Printf("Cliente conectado: %s", userAgent)

    w.WriteHeader(http.StatusOK)
    json.NewEncoder(w).Encode(map[string]interface{}{
        "tasks": filteredTasks,
        "total": len(filteredTasks),
    })
}

// POST /tasks - Crear nueva tarea
```

```go
func handleCreateTask(w http.ResponseWriter, r *http.Request) {
    // Verificar Content-Type
    contentType := r.Header.Get("Content-Type")
    if !strings.Contains(contentType, "application/json") {
        w.WriteHeader(http.StatusBadRequest)
        json.NewEncoder(w).Encode(map[string]string{
            "error": "Content-Type debe ser application/json",
        })
        return
    }

    // Leer y parsear body
    var newTask Task
    decoder := json.NewDecoder(r.Body)
    decoder.DisallowUnknownFields() // Rechazar campos desconocidos

    if err := decoder.Decode(&newTask); err != nil {
        w.WriteHeader(http.StatusBadRequest)
        json.NewEncoder(w).Encode(map[string]string{
            "error": "JSON inválido: " + err.Error(),
        })
        return
    }

    // Validar datos
```

```
    if strings.TrimSpace(newTask.Title) == "" {
        w.WriteHeader(http.StatusBadRequest)
        json.NewEncoder(w).Encode(map[string]string{
            "error": "El título es obligatorio",
        })
        return
    }

    // Crear tarea
    newTask.ID = nextID
    newTask.CreatedAt = time.Now()
    newTask.Completed = false // Siempre inicia como no completada
    nextID++

    tasks = append(tasks, newTask)

    w.WriteHeader(http.StatusCreated)
    json.NewEncoder(w).Encode(map[string]interface{}{
        "message": "Tarea creada exitosamente",
        "task":    newTask,
    })
}

// Handler para manejar una tarea específica por ID
func taskByIDHandler(w http.ResponseWriter, r *http.Request) {
```

```
w.Header().Set("Content-Type", "application/json")

// Extraer ID de la URL (routing manual básico)
path := strings.TrimPrefix(r.URL.Path, "/tasks/")
if path == "" {
    w.WriteHeader(http.StatusBadRequest)
    json.NewEncoder(w).Encode(map[string]string{
        "error": "ID de tarea requerido",
    })
    return
}

id, err := strconv.Atoi(path)
if err != nil {
    w.WriteHeader(http.StatusBadRequest)
    json.NewEncoder(w).Encode(map[string]string{
        "error": "ID debe ser un número válido",
    })
    return
}

// Buscar tarea
var foundTask *Task
for i := range tasks {
    if tasks[i].ID == id {
```

```
            foundTask = &tasks[i]
            break
        }
    }

    if foundTask == nil {
        w.WriteHeader(http.StatusNotFound)
        json.NewEncoder(w).Encode(map[string]string{
            "error": "Tarea no encontrada",
        })
        return
    }

    switch r.Method {
    case "GET":
        w.WriteHeader(http.StatusOK)
        json.NewEncoder(w).Encode(foundTask)
    case "PUT":
        handleUpdateTask(w, r, foundTask)
    case "DELETE":
        handleDeleteTask(w, r, id)
    default:
        w.WriteHeader(http.StatusMethodNotAllowed)
        json.NewEncoder(w).Encode(map[string]string{
            "error": "Método no permitido",
```

```
        })
    }
}

// PUT /tasks/{id} - Actualizar tarea
func handleUpdateTask(w http.ResponseWriter, r *http.Request, task *Task) {
    var updates Task
    if err := json.NewDecoder(r.Body).Decode(&updates); err != nil {
        w.WriteHeader(http.StatusBadRequest)
        json.NewEncoder(w).Encode(map[string]string{
            "error": "JSON inválido",
        })
        return
    }

    // Actualizar campos (manteniendo ID y fecha de creación)
    if updates.Title != "" {
        task.Title = updates.Title
    }
    if updates.Description != "" {
        task.Description = updates.Description
    }
    task.Completed = updates.Completed

    w.WriteHeader(http.StatusOK)
```

```
    json.NewEncoder(w).Encode(map[string]interface{}{
        "message": "Tarea actualizada exitosamente",
        "task":    task,
    })
}

// DELETE /tasks/{id} - Eliminar tarea
func handleDeleteTask(w http.ResponseWriter, r *http.Request, id int) {
    for i, task := range tasks {
        if task.ID == id {
            // Eliminar tarea del slice
            tasks = append(tasks[:i], tasks[i+1:]...)
            w.WriteHeader(http.StatusOK)
            json.NewEncoder(w).Encode(map[string]string{
                "message": "Tarea eliminada exitosamente",
            })
            return
        }
    }
}

// main.go actualizado
func main() {
    // Inicializar con algunas tareas de ejemplo
    tasks = []Task{
```

```
        {
            ID:          1,
            Title:       "Estudiar Go",
            Description: "Completar el tutorial de APIs",
            Completed:   false,
            CreatedAt:   time.Now().Add(-24 * time.Hour),
        },
        {
            ID:          2,
            Title:       "Hacer ejercicio",
            Description: "30 minutos de cardio",
            Completed:   true,
            CreatedAt:   time.Now().Add(-12 * time.Hour),
        },
    }
    nextID = 3

    // Registrar rutas
    http.HandleFunc("/", homeHandler)
    http.HandleFunc("/tasks", tasksHandler)
    http.HandleFunc("/tasks/", taskByIDHandler)
    http.HandleFunc("/health", healthHandler)

    fmt.Println("🚀 Servidor iniciado en puerto 8080")
    fmt.Println("📖 Rutas disponibles:")
```

```
    fmt.Println("  GET    /tasks           - Listar tareas")
    fmt.Println("  POST   /tasks           - Crear tarea")
    fmt.Println("  GET    /tasks/{id}      - Obtener tarea por ID")
    fmt.Println("  PUT    /tasks/{id}      - Actualizar tarea")
    fmt.Println("  DELETE /tasks/{id}      - Eliminar tarea")

    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Ahora validemos que nuestra API responda correctamente a nuestras peticiones, para esto en la línea de comando ejecutamos lo siguiente:

Shell

```
# Obtener todas las tareas
curl http://localhost:8080/tasks

# Obtener tareas completadas
curl "http://localhost:8080/tasks?completed=true"

# Crear nueva tarea
curl -X POST http://localhost:8080/tasks \
  -H "Content-Type: application/json" \
  -d '{"title":"Nueva tarea","description":"Descripción de la tarea"}'

# Obtener tarea específica
curl http://localhost:8080/tasks/1
```

```
# Actualizar tarea
curl -X PUT http://localhost:8080/tasks/1 \
  -H "Content-Type: application/json" \
  -d '{"completed":true}'

# Eliminar tarea
curl -X DELETE http://localhost:8080/tasks/2
```

Middlewares

En Go, el middleware actúa como una capa que intercepta y procesa las peticiones HTTP antes de que lleguen a los manejadores de la aplicación. Se utiliza habitualmente para añadir funcionalidades como autenticación, registro y validación de peticiones de forma modular. Las funciones de middleware en Go toman un http.Handler (o http.HandlerFunc) como entrada y devuelven un http.Handler modificado. Esto permite encadenar múltiples funciones de middleware, creando una cadena de pasos de procesamiento de peticiones.

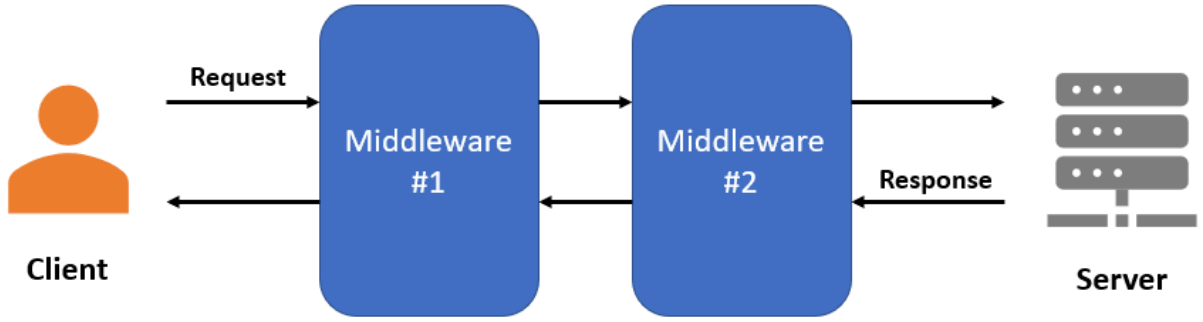
Los middlewares son uno de los conceptos más importantes de la ingeniería de backend. **Son piezas de software independientes y reutilizables que enlazan diferentes sistemas.**

En el desarrollo web, es habitual colocar uno o varios middlewares entre el cliente y el servidor. Esencialmente, esto crea un puente entre los datos y las interfaces de usuario.

Cada middleware actúa de forma independiente sobre una petición o respuesta HTTP. La salida de un middleware puede ser la entrada de otro middleware. Se forma así una cadena de middlewares.

¿Por qué middlewares? En las aplicaciones web, los middlewares nos permiten centralizar y reutilizar funcionalidades comunes en cada petición o respuesta. Por ejemplo, puedes tener un middleware que registre cada petición HTTP.

Para diseñar Go middlewares, existen ciertas reglas y patrones que debemos seguir. Vamos a ir explorando los conceptos de un middleware Go y creará dos de ellos para una aplicación sencilla.



Veamos con un ejemplo práctico lo básico de un middleware. Primero creamos un directorio para alojar nuestro mini proyecto Go. Luego abrimos nuestra línea de comandos o terminal y creamos un directorio para nuestro proyecto y a continuación ejecutamos lo siguiente

```shell
mkdir go-middleware

cd go-middlevare

code .
```

Luego en vscode agregamos el archivo main.go y codeamos lo siguiente

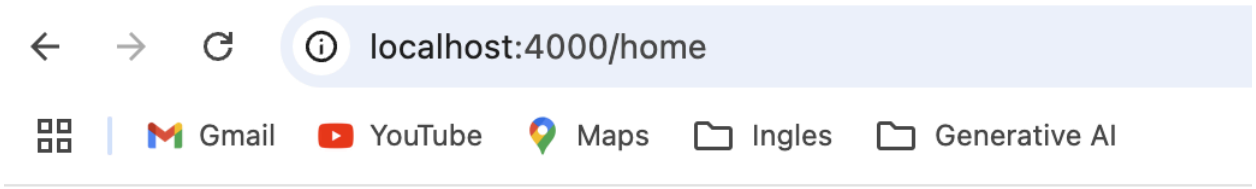
```go
package main


import (

	"log"
```

```
        "net/http"
)

func main() {
        // Initialize a new ServeMux
        mux := http.NewServeMux()

        // Register home function as handler for /home
        mux.HandleFunc("/home", home)

        // Run HTTP server with custom ServeMux at localhost:4000
        err := http.ListenAndServe(":4000", mux)
        // Exit if any errors occur
        log.Fatal(err)
}

// Handles HTTP requests to /home
func home(w http.ResponseWriter, r *http.Request) {
        // Writes a byte slice with the text "Welcome to Go Middleware"
        // in the response body
        w.Write([]byte("Welcome to Go Middleware"))
}
```

Hemos declarado un nuevo **ServeMux** (un router en la terminología de Go) con la función http.NewServeMux(). Luego, se ha registrado la función home como manejadora de cualquier

petición HTTP a la ruta /home. Por último, se inicia un servidor HTTP en localhost:4000 con el enrutador personalizado.

Una vez que hayas terminado, en el terminal, ejecuta go run main.go y visita localhost:4000/home en tu navegador favorito. Deberías ver una respuesta como la siguiente




```
Welcome to Go Middleware
```

Handler and ServeHTTP

Antes de empezar a crear middleware Go, es bueno que conozcamos algunas teorías. En particular, ¿qué es exactamente un handler en Go?

En Go, un handler no es más que un objeto (struct) que satisface la interfaz http.Handler.

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

http.Handler es una interfaz , que tiene un método ServeHTTP . Lo que significa que cualquier tipo que tenga el método ServeHTTP con la interfaz ResponseWriter y un puntero a Request puede ser tratado como un http.Handler .

Veamos con un ejemplo más amplio donde se pueda observar cómo trabajan los handlers

```Go
// middleware/middleware.go

package main


import (
```

```
    "fmt"
    "log"
    "net/http"
    "time"
)

// Tipo para funciones middleware
type Middleware func(http.HandlerFunc) http.HandlerFunc

// Middleware de Logging
func LoggingMiddleware(next http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        start := time.Now()

        // Crear un wrapper del ResponseWriter para capturar el status code
        wrapper := &responseWrapper{
            ResponseWriter: w,
            statusCode:     http.StatusOK,
        }

        // Ejecutar el siguiente handler
        next(wrapper, r)

        // Log después de la ejecución
        duration := time.Since(start)
```

```
        log.Printf(
            "%s %s %d %v %s",
            r.Method,
            r.URL.Path,
            wrapper.statusCode,
            duration,
            r.RemoteAddr,
        )
    }
}

// Wrapper para capturar el status code
type responseWrapper struct {
    http.ResponseWriter
    statusCode int
}

func (rw *responseWrapper) WriteHeader(code int) {
    rw.statusCode = code
    rw.ResponseWriter.WriteHeader(code)
}

// Middleware de CORS
func CORSMiddleware(next http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
```

```
        // Configurar headers CORS
        w.Header().Set("Access-Control-Allow-Origin", "*")
        w.Header().Set("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE,
OPTIONS")
        w.Header().Set("Access-Control-Allow-Headers", "Content-Type,
Authorization")

        // Manejar preflight requests
        if r.Method == "OPTIONS" {
            w.WriteHeader(http.StatusOK)
            return
        }

        next(w, r)
    }
}

// Middleware de autenticación simple
func AuthMiddleware(next http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        // Verificar header de autorización
        authHeader := r.Header.Get("Authorization")

        // Para este ejemplo, aceptamos cualquier Bearer token
        if authHeader == "" || len(authHeader) < 7 || authHeader[:7] != "Bearer
" {
```

```
            w.Header().Set("Content-Type", "application/json")
            w.WriteHeader(http.StatusUnauthorized)
            fmt.Fprintf(w, `{"error": "Token de autorización requerido"}`)
            return
        }

        // Simular validación de token
        token := authHeader[7:]
        if token != "secret123" {
            w.Header().Set("Content-Type", "application/json")
            w.WriteHeader(http.StatusUnauthorized)
            fmt.Fprintf(w, `{"error": "Token inválido"}`)
            return
        }

        next(w, r)
    }
}

// Middleware de Rate Limiting simple
func RateLimitMiddleware(next http.HandlerFunc) http.HandlerFunc {
    // Mapa para contar requests por IP
    requestCounts := make(map[string]int)

    return func(w http.ResponseWriter, r *http.Request) {
```

```
        clientIP := r.RemoteAddr

        // Incrementar contador
        requestCounts[clientIP]++

        // Límite simple: máximo 10 requests por cliente
        if requestCounts[clientIP] > 10 {
            w.Header().Set("Content-Type", "application/json")
            w.WriteHeader(http.StatusTooManyRequests)
            fmt.Fprintf(w, `{"error": "Demasiados requests"}`)
            return
        }

        next(w, r)
    }
}

// Función para encadenar middlewares
func ChainMiddlewares(handler http.HandlerFunc, middlewares ...Middleware)
http.HandlerFunc {
    for i := len(middlewares) - 1; i >= 0; i-- {
        handler = middlewares[i](handler)
    }
    return handler
}
```

```
// main.go con middlewares

func main() {

    // Inicializar datos de ejemplo

    initSampleData()


    // Rutas públicas (solo con CORS y Logging)

    http.HandleFunc("/", ChainMiddlewares(homeHandler, CORSMiddleware,
LoggingMiddleware))

    http.HandleFunc("/health", ChainMiddlewares(healthHandler, CORSMiddleware,
LoggingMiddleware))


    // Rutas públicas de tareas (lectura)

    http.HandleFunc("/tasks", ChainMiddlewares(tasksHandler, CORSMiddleware,
LoggingMiddleware))


    // Rutas protegidas (requieren autenticación)

    http.HandleFunc("/tasks/create", ChainMiddlewares(

        createTaskHandler,

        AuthMiddleware,

        CORSMiddleware,

        LoggingMiddleware,

    ))


    http.HandleFunc("/tasks/", ChainMiddlewares(

        taskByIDHandler,

        AuthMiddleware,
```

```
        CORSMiddleware,
        LoggingMiddleware,
    ))

    fmt.Println("🚀 Servidor con middlewares iniciado en puerto 8080")
    fmt.Println("🔐 Rutas protegidas requieren: Authorization: Bearer secret123")
    log.Fatal(http.ListenAndServe(":8080", nil))
}

// Handler específico para crear tareas (versión protegida)
func createTaskHandler(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
        w.WriteHeader(http.StatusMethodNotAllowed)
        return
    }

    handleCreateTask(w, r)
}

func initSampleData() {
    tasks = []Task{
        {
            ID:          1,
            Title:       "Estudiar Go",
            Description: "Completar el tutorial de APIs",
```

```
            Completed:   false,
            CreatedAt:   time.Now().Add(-24 * time.Hour),
        },
    }
    nextID = 2
}
```

Frameworks de Routing: Gorilla Mux y Chi

Implementando Gorilla Mux

gorilla/mux es una biblioteca de gestión de enrutamiento del paquete de herramientas de desarrollo Web gorilla. El paquete de desarrollo Web gorilla es un conjunto de herramientas que ayuda a desarrollar servidores Web en el lenguaje Go. Cubre varios aspectos como el procesamiento de datos de formularios (gorilla/schema), comunicación websocket (gorilla/websocket), middleware (gorilla/handlers), gestión de sesiones (gorilla/sessions) y gestión segura de cookies (gorilla/securecookie).

mux tiene las siguientes ventajas:

- Implementa la interfaz estándar http.Handler y puede utilizarse en combinación con la biblioteca estándar net/http. Es muy ligero.
- Puede emparejar handlers basándose en el nombre de host de la petición, la ruta, el prefijo de la ruta, el protocolo, las cabeceras HTTP, la cadena de consulta y el método HTTP. También admite lógica de correspondencia personalizada.
- Se pueden utilizar variables en los nombres de host, las rutas y los parámetros de las peticiones, así como especificar expresiones regulares para ellos.
- Puede pasar parámetros a un manejador especificado para construir una URL completa.
- Admite la agrupación de rutas, lo que resulta conveniente para la gestión y el mantenimiento.

Para poder usar Gorilla Mux primero debemos descargarlo e instalarlo en nuestro proyecto para ello ejecutamos lo siguiente:

```shell
# Instalar Gorilla Mux

go get github.com/gorilla/mux
```

Ahora creemos un nuevo proyecto en go llamado go_gorilla y en el archivo main.go, codeamos lo siguiente:

```go
// main_gorilla.go

package main


import (

    "encoding/json"

    "fmt"

    "log"

    "net/http"

    "strconv"


    "github.com/gorilla/mux"

)


func main() {
```

```
    // Crear router
    r := mux.NewRouter()

    // Aplicar middlewares globales
    r.Use(CORSMiddleware)
    r.Use(LoggingMiddleware)

    // Rutas públicas
    r.HandleFunc("/", homeHandler).Methods("GET")
    r.HandleFunc("/health", healthHandler).Methods("GET")
    r.HandleFunc("/tasks", getTasksHandler).Methods("GET")

    // Subrutas protegidas
    protected := r.PathPrefix("/api").Subrouter()
    protected.Use(AuthMiddleware)

    // Rutas CRUD con parámetros
    protected.HandleFunc("/tasks", createTaskHandler).Methods("POST")
    protected.HandleFunc("/tasks/{id:[0-9]+}",
getTaskByIDHandler).Methods("GET")
    protected.HandleFunc("/tasks/{id:[0-9]+}",
updateTaskHandler).Methods("PUT")
    protected.HandleFunc("/tasks/{id:[0-9]+}",
deleteTaskHandler).Methods("DELETE")

    // Rutas con query parameters
```

```
    r.HandleFunc("/tasks/search", searchTasksHandler).Methods("GET")

    fmt.Println("🚀 Servidor con Gorilla Mux iniciado en puerto 8080")
    fmt.Println("📚 Rutas disponibles:")
    fmt.Println("  GET    /tasks              - Listar tareas (público)")
    fmt.Println("  GET    /tasks/search       - Buscar tareas (público)")
    fmt.Println("  POST   /api/tasks          - Crear tarea (protegido)")
    fmt.Println("  GET    /api/tasks/{id}     - Obtener tarea (protegido)")
    fmt.Println("  PUT    /api/tasks/{id}     - Actualizar tarea (protegido)")
    fmt.Println("  DELETE /api/tasks/{id}     - Eliminar tarea (protegido)")

    log.Fatal(http.ListenAndServe(":8080", r))
}

// Handlers específicos con Gorilla Mux
func getTaskByIDHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    id, err := strconv.Atoi(vars["id"])

    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        json.NewEncoder(w).Encode(map[string]string{
            "error": "ID inválido",
        })
        return
```

```
    }

    // Buscar tarea
    for _, task := range tasks {
        if task.ID == id {
            w.Header().Set("Content-Type", "application/json")
            w.WriteHeader(http.StatusOK)
            json.NewEncoder(w).Encode(task)
            return
        }
    }

    w.WriteHeader(http.StatusNotFound)
    json.NewEncoder(w).Encode(map[string]string{
        "error": "Tarea no encontrada",
    })
}

func updateTaskHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    id, _ := strconv.Atoi(vars["id"])

    // Encontrar tarea y actualizar
    for i := range tasks {
        if tasks[i].ID == id {
```

```
var updates Task
if err := json.NewDecoder(r.Body).Decode(&updates); err != nil {
    w.WriteHeader(http.StatusBadRequest)
    json.NewEncoder(w).Encode(map[string]string{
        "error": "JSON inválido",
    })
    return
}

// Actualizar campos
if updates.Title != "" {
    tasks[i].Title = updates.Title
}
if updates.Description != "" {
    tasks[i].Description = updates.Description
}
tasks[i].Completed = updates.Completed

w.Header().Set("Content-Type", "application/json")
w.WriteHeader(http.StatusOK)
json.NewEncoder(w).Encode(map[string]interface{}{
    "message": "Tarea actualizada",
    "task":    tasks[i],
})
return
```

```
        }
    }

    w.WriteHeader(http.StatusNotFound)
    json.NewEncoder(w).Encode(map[string]string{
        "error": "Tarea no encontrada",
    })
}

func deleteTaskHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    id, _ := strconv.Atoi(vars["id"])

    for i, task := range tasks {
        if task.ID == id {
            tasks = append(tasks[:i], tasks[i+1:]...)
            w.Header().Set("Content-Type", "application/json")
            w.WriteHeader(http.StatusOK)
            json.NewEncoder(w).Encode(map[string]string{
                "message": "Tarea eliminada exitosamente",
            })
            return
        }
    }
```

```
    w.WriteHeader(http.StatusNotFound)
    json.NewEncoder(w).Encode(map[string]string{
        "error": "Tarea no encontrada",
    })
}

func searchTasksHandler(w http.ResponseWriter, r *http.Request) {
    query := r.URL.Query().Get("q")

    var results []Task
    for _, task := range tasks {
        if query == "" ||
           strings.Contains(strings.ToLower(task.Title),
strings.ToLower(query)) ||
           strings.Contains(strings.ToLower(task.Description),
strings.ToLower(query)) {
            results = append(results, task)
        }
    }

    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(map[string]interface{}{
        "query":   query,
        "results": results,
        "total":   len(results),
    })
```

```
}


// Middlewares adaptados para Gorilla Mux
func CORSMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Access-Control-Allow-Origin", "*")
        w.Header().Set("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE,
OPTIONS")
        w.Header().Set("Access-Control-Allow-Headers", "Content-Type,
Authorization")

        if r.Method == "OPTIONS" {
            w.WriteHeader(http.StatusOK)
            return
        }

        next.ServeHTTP(w, r)
    })
}

func LoggingMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        start := time.Now()
        next.ServeHTTP(w, r)
        log.Printf("%s %s %v", r.Method, r.URL.Path, time.Since(start))
    })
```

```
}


func AuthMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        authHeader := r.Header.Get("Authorization")

        if authHeader == "" || len(authHeader) < 7 || authHeader[:7] != "Bearer
" {
            w.Header().Set("Content-Type", "application/json")
            w.WriteHeader(http.StatusUnauthorized)
            json.NewEncoder(w).Encode(map[string]string{
                "error": "Token requerido",
            })
            return
        }

        token := authHeader[7:]
        if token != "secret123" {
            w.Header().Set("Content-Type", "application/json")
            w.WriteHeader(http.StatusUnauthorized)
            json.NewEncoder(w).Encode(map[string]string{
                "error": "Token inválido",
            })
            return
        }
```

```
        next.ServeHTTP(w, r)
    })
}
```

Implementación con Chi Router

Para poder usar Chi Router primero debemos descargarlo e instalarlo en nuestro proyecto para ello ejecutamos lo siguiente:

```shell
# Instalar Chi
go get github.com/go-chi/chi/v5
```

```go
// main_chi.go
package main

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

```
    "strconv"

    "github.com/go-chi/chi/v5"
    "github.com/go-chi/chi/v5/middleware"
)

func main() {
    r := chi.NewRouter()

    // Middlewares incorporados de Chi
    r.Use(middleware.Logger)
    r.Use(middleware.Recoverer)
    r.Use(CORSMiddleware)

    // Rutas públicas
    r.Get("/", homeHandler)
    r.Get("/health", healthHandler)
    r.Get("/tasks", getTasksHandler)
    r.Get("/tasks/search", searchTasksHandler)

    // Grupo de rutas protegidas
    r.Route("/api", func(r chi.Router) {
        r.Use(AuthMiddleware)

        r.Route("/tasks", func(r chi.Router) {
```

```
            r.Post("/", createTaskHandler)

            r.Route("/{id}", func(r chi.Router) {
                r.Get("/", getTaskByIDHandler)
                r.Put("/", updateTaskHandler)
                r.Delete("/", deleteTaskHandler)
            })
        })
    })

    fmt.Println("🚀 Servidor con Chi Router iniciado en puerto 8080")
    log.Fatal(http.ListenAndServe(":8080", r))
}

// Handler para obtener tarea por ID con Chi
func getTaskByIDHandler(w http.ResponseWriter, r *http.Request) {
    idStr := chi.URLParam(r, "id")
    id, err := strconv.Atoi(idStr)

    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        json.NewEncoder(w).Encode(map[string]string{
            "error": "ID inválido",
        })
        return
```

```
    }

    for _, task := range tasks {
        if task.ID == id {
            w.Header().Set("Content-Type", "application/json")
            json.NewEncoder(w).Encode(task)
            return
        }
    }

    w.WriteHeader(http.StatusNotFound)
    json.NewEncoder(w).Encode(map[string]string{
        "error": "Tarea no encontrada",
    })
}
```