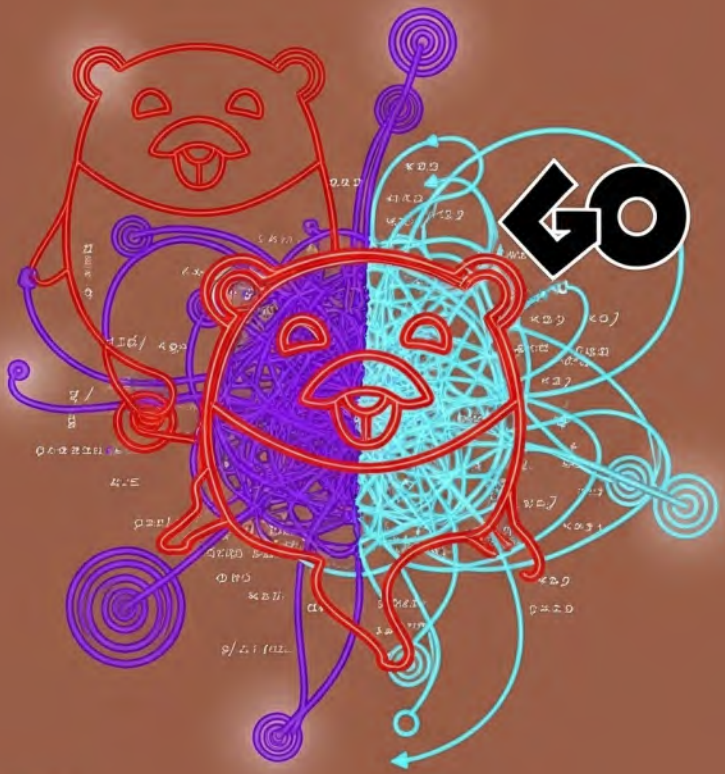
Go Web Programming: Intermediate

(Integrative-Generative AI Edition)



Contents:

Author: Student Price Book Center

คำนำ

การพัฒนาเว็บแอปพลิเคชันด้วยภาษา Go เป็นแนวทางที่ได้รับความนิยมอย่างมากในช่วงหลายปีที่ผ่านมา ด้วยความเรียบง่าย ประสิทธิภาพสูง และรองรับการประมวลผลพร้อมกัน (Concurrency) ทำให้ Go กลายเป็นเครื่องมือสำคัญสำหรับนักพัฒนาที่ต้องการสร้างเว็บแอปพลิเคชันที่มีคุณภาพสูง สำหรับผู้ที่มีพื้นฐานจากการเรียนรู้ Go Web Programming เบื้องต้นแล้ว หนังสือเล่มนี้จะพาผู้อ่านก้าวไปสู่ **ระดับกลาง (Intermediate)** ด้วยการเจาะลึกเรื่องการจัดโครงสร้างโปรเจกต์ การจัดการ middleware, การรับส่งฟอร์มและ validation, การสร้าง JSON API, และการจัดการ error/logging อย่างเป็นระบบ

ใน **บทที่ 6: Structuring a Go Web Project** ผู้อ่านจะได้เรียนรู้วิธีจัดโครงสร้างโปรเจกต์อย่างเป็นระบบ ตั้งแต่การจัดโฟลเดอร์หลัก เช่น handlers, models, routes, และ templates ไปจนถึงการใช้ **Separation of Concerns** ผ่าน **MVC Pattern** และการจัดการ dependency ด้วย go.mod แนวทางเหล่านี้ช่วยให้โค้ดอ่านง่าย ดูแลรักษาได้ง่าย และรองรับการขยายตัวของโปรเจกต์ในอนาคต พร้อมตัวอย่างบูรณาการที่ช่วยให้เข้าใจแนวคิดอย่างเป็นรูปธรรม

บทที่ 7: Middleware จะพาผู้อ่านเจาะลึกแนวคิดของ middleware ใน Go Web Programming ซึ่งเป็นกลไกสำคัญในการจัดการคำร้องขอก่อนหรือหลัง handler เช่น การสร้าง **logging middleware**, **authentication middleware**, และ **CORS middleware** พร้อมการอธิบาย **Chain of Responsibility Pattern** ซึ่งช่วยให้ middleware หลายตัวสามารถทำงานร่วมกันเป็นสายโซ่ การออกแบบ middleware อย่างเป็นระบบช่วยให้เว็บแอปพลิเคชันมีความยืดหยุ่น ปลอดภัย และ maintainable

บทที่ 8: Form Handling and Validation ครอบคลุมการจัดการ **POST forms** และการทำ **server-side validation** อย่างละเอียด ทั้งการตรวจสอบค่าที่จำเป็น, รูปแบบข้อมูล, และการสร้าง **custom validators** สำหรับกรณีเฉพาะ การปฏิบัติจริงและตัวอย่างบูรณาการช่วยให้ผู้อ่านสามารถสร้างฟอร์มที่ปลอดภัยและมีประสิทธิภาพ พร้อม feedback ให้ผู้ใช้เข้าใจข้อผิดพลาดและแก้ไขได้ทันที

บทที่ 9: JSON API Development จะเน้นการสร้าง **RESTful API** ด้วย Go โดยอธิบายการ marshal/unmarshal JSON และการจัดการ **error handling และ status codes** อย่างถูกต้อง การออกแบบ API อย่างเป็นระบบช่วยให้ระบบสื่อสารกับ client หรือบริการอื่น ๆ ได้ชัดเจน ปลอดภัย และ maintainable พร้อมตัวอย่างโปรเจกต์จริงที่ผู้อ่านสามารถทดลองสร้าง CRUD API ได้ทันที

บทที่ 10: Error Handling and Logging ครอบคลุมการจัดการข้อผิดพลาดและการบันทึก log ใน Go Web Programming โดยอธิบายการจัดการ **error ใน handlers**, การใช้ **log และ third-party logger** เช่น logrus, และการสร้าง **centralized error handling** ซึ่งช่วยให้ระบบมีโครงสร้างที่ชัดเจน สามารถตรวจสอบและแก้ไขปัญหาได้รวดเร็ว พร้อมตัวอย่างบูรณาการเพื่อให้ผู้อ่านเข้าใจแนวทางการจัดการ error อย่างครบถ้วน

หนังสือเล่มนี้ออกแบบให้ผู้อ่านสามารถ **ต่อยอดความรู้จากพื้นฐานไปสู่ระดับกลาง** ได้อย่างเป็นระบบ ทั้งในเชิงแนวคิดและการปฏิบัติจริง ผู้อ่านจะได้ฝึกสร้างโปรเจกต์เว็บแอปพลิเคชันที่ **มี**

โครงสร้างชัดเจน, ยืดหยุ่น, ปลอดภัย และ maintainable ตลอดจนสามารถนำความรู้ไปปรับใช้ในโปรเจกต์จริงหรือขยายไปสู่เว็บแอปพลิเคชันขนาดใหญ่ในอนาคต

ด้วยความปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 6

Structuring a Go Web Project

(Structuring a Go Web Project)

เนื้อหา

- Structuring a Go Web Project
- Structuring a Go Web Project (Deep Dive)
- การจัดโฟลเดอร์ใน Go Web Project
- Separation of Concerns (MVC Pattern) ใน Go Web
- การใช้ go.mod และ Dependency Management ใน Go
- ตัวอย่างบูรณาการ

บทนำบทที่ 6: Structuring a Go Web Project

การพัฒนาเว็บแอปพลิเคชันที่ซับซ้อนใน Go ไม่ได้จำกัดเพียงแค่การเขียนโค้ดให้ทำงานได้เท่านั้น แต่ยังต้องให้ความสำคัญกับ **การจัดโครงสร้างโปรเจกต์ (Project Structure)** อย่างเป็นระบบ การจัดโครงสร้างที่ดีช่วยให้นักพัฒนาสามารถดูแลรักษาโค้ด เพิ่มฟีเจอร์ใหม่ได้ง่าย และลดความซับซ้อนของระบบเมื่อโปรเจกต์เติบโตขึ้น

หนึ่งในแนวทางสำคัญคือ **การจัดโฟลเดอร์และไฟล์** ให้เหมาะสม โดยทั่วไปโปรเจกต์เว็บ Go จะมีโฟลเดอร์หลัก เช่น handlers สำหรับจัดการ HTTP handlers, models สำหรับเก็บ struct และการติดต่อกับฐานข้อมูล, routes สำหรับจัดการ routing, และ templates สำหรับไฟล์ HTML หรือ template การจัดโฟลเดอร์อย่างเป็นระบบช่วยให้โค้ดอ่านง่ายและนักพัฒนาสามารถค้นหาไฟล์ที่ต้องการได้อย่างรวดเร็ว

นอกจากนี้ การจัดโครงสร้างโปรเจกต์ต้องอิงกับแนวคิด **Separation of Concerns (SoC)** หรือการแยกความรับผิดชอบของแต่ละส่วนออกจากกัน ซึ่งสอดคล้องกับ **MVC Pattern (Model-View-Controller)** โดย Model จัดการข้อมูลและตรรกะของระบบ, View จัดการการแสดงผล, และ Controller จัดการคำร้องขอและตอบสนองต่อผู้ใช้ การใช้ MVC Pattern ทำให้โค้ดมีความชัดเจน ลดความซับซ้อน และรองรับการขยายตัวของโปรเจกต์ในอนาคต

บทนี้ยังแนะนำการใช้ **go.mod และ Dependency Management** เพื่อควบคุมเวอร์ชันของแพ็กเกจและไลบรารีที่โปรเจกต์ใช้งาน การจัดการ dependency อย่างเป็นระบบช่วยป้องกันปัญหาความ

ไม่เข้ากันของเวอร์ชัน ทำให้โปรเจกต์สามารถติดตั้งและรันได้บนเครื่องอื่นหรือสภาพแวดล้อมต่าง ๆ อย่างมั่นใจ

นอกจากนี้ ผู้อ่านจะได้เรียนรู้แนวทาง **การวางโครงสร้างโปรเจกต์แบบ scalable** สำหรับโปรเจกต์ขนาดใหญ่ เช่น การแยกโฟลเดอร์สำหรับบริการย่อย (microservices) หรือการจัดการหลาย module การทำเช่นนี้ช่วยให้ทีมพัฒนาสามารถทำงานร่วมกันได้อย่างมีประสิทธิภาพ ลดข้อผิดพลาด และเพิ่มความเร็วในการพัฒนา

บทนี้เน้นให้ผู้อ่าน **ทดลองสร้างโครงสร้างโปรเจกต์จริง** โดยจัดโฟลเดอร์และไฟล์ตามแนวทาง MVC พร้อมใช้งาน go.mod สำหรับจัดการ dependency การปฏิบัติจริงเหล่านี้ทำให้ผู้อ่านเข้าใจแนวคิดการจัดโครงสร้างโปรเจกต์อย่างเป็นระบบและสามารถนำไปใช้ในโปรเจกต์จริงได้ทันที

ท้ายที่สุด การเข้าใจและฝึกฝนการ **Structuring a Go Web Project** เป็นพื้นฐานสำคัญสำหรับการสร้างเว็บแอปพลิเคชันที่มีคุณภาพ มีความยืดหยุ่น และสามารถต่อยอดไปสู่ระบบขนาดใหญ่ที่ซับซ้อน การจัดโครงสร้างที่ดีช่วยให้โค้ดอ่านง่าย ดูแลรักษาได้ง่าย และพร้อมรองรับการขยายตัวในอนาคต

Structuring a Go Web Project

- การจัดโฟลเดอร์ (handlers, models, routes, templates)
- Separation of concerns (MVC pattern)
- การใช้ go.mod และ dependency management

การจัดโครงสร้างโปรเจกต์เว็บใน Go เป็นสิ่งสำคัญมาก เพื่อให้โค้ด **อ่านง่าย, maintainable, scalable** และรองรับทีมงานหลายคนได้

---

1 การจัดโฟลเดอร์

ในโปรเจกต์ Go Web Application แบบมาตรฐาน แนะนำให้แยกโฟลเดอร์ตามหน้าที่ของแต่ละส่วน ดังนี้

1.1 ตัวอย่างโครงสร้างโฟลเดอร์

```
mywebapp/
├── main.go          # entry point ของโปรเจกต์
├── go.mod           # dependency management
├── handlers/        # จัดการ logic ของแต่ละ HTTP handler
│   ├── home.go
│   ├── users.go
│   └── api.go
├── models/          # struct และ logic เกี่ยวกับ data เช่น DB access
```

```
│   └── user.go
├── routes/              # จัดการ router / chi mux / gorilla mux
│   └── routes.go
├── templates/           # HTML templates
│   ├── base.html
│   └── home.html
├── static/              # CSS, JS, images
│   ├── style.css
│   └── app.js
└── utils/               # helper functions / utilities
    └── helpers.go
```

แนวคิด

- แยก **handlers** จาก **models** ⟶ separation of concerns
- **routes** จะรวม router initialization และ subrouters
- **templates** และ **static** อยู่แยกชัดเจนเพื่อง่ายต่อการจัดการ

---

2 Separation of Concerns (MVC pattern)

MVC (Model-View-Controller) เป็นแนวทางยอดนิยมใน Go Web:

Layer	ตัวอย่าง	คำอธิบาย
Model	models/user.go	struct, DB access, business logic
View	templates/*.html	HTML templates, dynamic rendering
Controller / Handler	handlers/home.go, handlers/users.go	รับ request, call model, render view

2.1 ตัวอย่าง flow

1. **Request** ⟶ route (/users) ⟶ **handler**
2. **Handler** ⟶ call **model** (fetch users from DB)
3. **Handler** ⟶ pass data to **template** (render HTML)
4. **Response** ⟶ client

ข้อดี

- เพิ่มความชัดเจน
- แก้ไขส่วนใดส่วนหนึ่งโดยไม่กระทบส่วนอื่น
- รองรับทีมหลายคน

---

3 การใช้ go.mod และ Dependency Management

Go ใช้ **Go Modules** (go.mod) สำหรับจัดการ dependency

3.1 สร้าง go.mod

cd mywebapp

go mod init github.com/username/mywebapp

3.2 ติดตั้ง dependency เช่น Chi router

go get github.com/go-chi/chi/v5

3.3 ตัวอย่าง go.mod

module github.com/username/mywebapp

go 1.21

require github.com/go-chi/chi/v5 v5.8.0

3.4 ข้อดีของ go.mod

- ระบุ **module name**
- บอก Go เวอร์ชันที่ใช้
- เก็บ **dependency version** ⟶ reproducible builds
- ใช้ go get / go mod tidy จัดการ library ที่ต้องการ

---

4 ตัวอย่างรวมทั้งหมด (Mini Structure)

```
mywebapp/
├── main.go
├── go.mod
├── handlers/
│   └── home.go
├── models/
│   └── user.go
├── routes/
│   └── routes.go
├── templates/
│   ├── base.html
│   └── home.html
└── static/
```

```
    ├── style.css
    └── app.js
```

main.go

```
package main

import (
        "log"
        "net/http"

        "github.com/username/mywebapp/routes"
)

func main() {
        r := routes.InitRoutes()
        log.Println("Server running at :8080")
        http.ListenAndServe(":8080", r)
}
```

routes/routes.go

```
package routes

import (
        "net/http"
        "github.com/go-chi/chi/v5"
        "github.com/username/mywebapp/handlers"
)

func InitRoutes() *chi.Mux {
        r := chi.NewRouter()
        r.Get("/", handlers.HomeHandler)
        r.Get("/users", handlers.UsersHandler)
        return r
}
```

handlers/home.go

```
package handlers
```

```
import (
        "html/template"
        "net/http"
)

func HomeHandler(w http.ResponseWriter, r *http.Request) {
        tmpl := template.Must(template.ParseFiles("templates/base.html",
"templates/home.html"))
        data := map[string]interface{}{"Title": "Home Page", "User": "Guest"}
        tmpl.ExecuteTemplate(w, "base.html", data)
}
```

models/user.go

```
package models

type User struct {
        Name   string
        Age    int
        Active bool
}
```

□ สรุปแนวทางสำคัญ

1. แยก **handlers / models / routes / templates / static**
2. ใช้ **MVC pattern** ⟶ separation of concerns
3. ใช้ **go.mod** และ Go Modules สำหรับ dependency management
4. ง่ายต่อการ **maintain, test, scale, และทำงานเป็นทีม**

Structuring a Go Web Project (Deep Dive)

การจัดโครงสร้างโปรเจกต์เว็บใน Go สำคัญเพราะ Go เป็นภาษาที่ **เน้นความชัดเจนและง่ายต่อ maintain** การวางโครงสร้างที่ดีช่วยให้ทีมทำงานร่วมกันได้สะดวก และลดความซับซ้อนเมื่อโปรเจกต์โตขึ้น

---

1 □ การจัดโฟลเดอร์ (Folder Organization)

การจัดโฟลเดอร์ใน Go Web App ควรแยกตามหน้าที่การทำงานหลักของโปรเจกต์

1.1 โฟลเดอร์หลัก

โฟลเดอร์	เนื้อหา	คำอธิบาย
main.go	Entry point	จุดเริ่มต้นของโปรเจกต์
handlers/	HTTP handlers	Logic ที่รับ request, call model, render response
models/	Data models	Struct, validation, DB access, business logic
routes/	Router setup	รวม router, subrouter, middleware
templates/	HTML templates	Layout, content blocks, template functions
static/	Static files	CSS, JS, images, font
utils/	Utilities / helpers	ฟังก์ชันซ้ำที่ใช้หลายที่ เช่น validation, helpers

1.2 ตัวอย่างโครงสร้างโปรเจกต์

```
mywebapp/
├── main.go
├── go.mod
├── handlers/
│   ├── home.go
│   ├── users.go
│   └── api.go
├── models/
│   └── user.go
├── routes/
│   └── routes.go
├── templates/
│   ├── base.html
│   └── home.html
├── static/
│   ├── style.css
│   └── app.js
└── utils/
    └── helpers.go
```

เหตุผล: แยกแต่ละหน้าที่ชัดเจน ⟶ maintain ง่าย, test ง่าย, ทีมงานสามารถทำงานพร้อมกันได้

2 Separation of Concerns (MVC Pattern)

MVC (Model-View-Controller) เป็นแนวทางมาตรฐานของการพัฒนาเว็บ

Layer	ตัวอย่าง	หน้าที่
Model	models/user.go	จัดการข้อมูล, struct, validation, DB access
View	templates/*.html	แสดงผล HTML, ใช้ template functions, layout
Controller/Handler	handlers/*.go	รับ request, call model, render view หรือ return JSON

2.1 Workflow ของ MVC

1. **Request** ⟶ router ⟶ handler (Controller)
2. **Handler** ⟶ call **model** (query DB หรือประมวลผล business logic)
3. **Handler** ⟶ pass ข้อมูลไป **template** (View) หรือ return JSON
4. **Response** ⟶ client

2.2 ข้อดี

- โค้ดชัดเจนแต่ละส่วนทำหน้าที่เฉพาะ
- สามารถแก้ไข template โดยไม่กระทบ handler
- สามารถแก้ไข model หรือ DB logic โดยไม่กระทบ view
- รองรับการทำงานเป็นทีม: Frontend ทำ template, Backend ทำ handlers/models

3 การใช้ go.mod และ Dependency Management

Go ใช้ **Go Modules** (go.mod) สำหรับจัดการ dependency และ version ของ library

3.1 เริ่มต้น Go Module

```
cd mywebapp
go mod init github.com/username/mywebapp
```

3.2 ติดตั้ง Dependency

```
go get github.com/go-chi/chi/v5
```

3.3 ตัวอย่าง go.mod

```
module github.com/username/mywebapp

go 1.21

require github.com/go-chi/chi/v5 v5.8.0
```

3.4 คำอธิบาย

- module ⟶ ชื่อโมดูลโปรเจกต์
- go ⟶ ระบุเวอร์ชัน Go
- require ⟶ library ที่โปรเจกต์ใช้ + version
- go mod tidy ⟶ ลบ dependency ที่ไม่ได้ใช้และดาวน์โหลด dependency ที่ขาด

ข้อดี

- Build reproducible: ทุกคนใช้ dependency version เดียวกัน
- ง่ายต่อการจัดการ package/library
- สนับสนุนการแชร์และ deploy โปรเจกต์

---

4 ตัวอย่างโครงสร้าง Mini Web Project (เชิงปฏิบัติ)

```
mywebapp/
├── main.go
├── go.mod
├── handlers/
│   ├── home.go
│   └── users.go
├── models/
│   └── user.go
├── routes/
│   └── routes.go
├── templates/
│   ├── base.html
│   └── home.html
└── static/
    ├── style.css
    └── app.js
```

main.go

```
package main

import (
        "log"
        "net/http"
        "github.com/username/mywebapp/routes"
```

```
)

func main() {
	r := routes.InitRoutes()
	log.Println("Server running at :8080")
	http.ListenAndServe(":8080", r)
}
```

routes/routes.go

```
package routes

import (
	"net/http"
	"github.com/go-chi/chi/v5"
	"github.com/username/mywebapp/handlers"
)

func InitRoutes() *chi.Mux {
	r := chi.NewRouter()
	r.Get("/", handlers.HomeHandler)
	r.Get("/users", handlers.UsersHandler)
	return r
}
```

handlers/home.go

```
package handlers

import (
	"html/template"
	"net/http"
)

func HomeHandler(w http.ResponseWriter, r *http.Request) {
	tmpl := template.Must(template.ParseFiles("templates/base.html",
"templates/home.html"))
	data := map[string]interface{}{"Title": "Home Page", "User": "Guest"}
```

```
        tmpl.ExecuteTemplate(w, "base.html", data)
}
```

models/user.go

```
package models

type User struct {
        Name   string
        Age    int
        Active bool
}
```

---

5 Best Practices

1. **Keep it simple** ⟶ อย่าใส่โค้ดทุกอย่างใน main.go
2. **Use packages** ⟶ แยก handler, model, routes
3. **Use template inheritance** ⟶ base.html + content block
4. **Use Go Modules** ⟶ จัดการ dependency, version control
5. **Middleware separation** ⟶ Logging, Recovery, Authentication
6. **Static files** ⟶ แยกชัดเจน /static

สรุป:

- การจัดโครงสร้างโปรเจกต์ใน Go Web ⟶ **สำคัญต่อ scalability และ maintainability**
- แนะนำใช้ **MVC + handlers/models/routes + templates/static + utils**
- **go.mod** ⟶ จัดการ dependency ให้ reproducible และ deploy ง่าย

การจัดโฟลเดอร์ใน Go Web Project

การจัดโฟลเดอร์ใน Go Web Project แบบเจาะลึก โดยเฉพาะโฟลเดอร์หลัก 4 ส่วน: handlers, models, routes, templates

---

1 handlers

หน้าที่

- จัดการ **HTTP requests** และ **HTTP responses**
- ประมวลผลข้อมูลเบื้องต้น (เช่น validation, session check)
- Call **model** เพื่อดึงข้อมูลหรือบันทึกข้อมูล

- Render **view** (HTML template) หรือ return **JSON / XML**

ตัวอย่างโฟลเดอร์และไฟล์

handlers/

├── home.go

├── users.go

└── api.go

home.go (ตัวอย่าง handler พื้นฐาน)

```
package handlers

import (
	"html/template"
	"net/http"
)

func HomeHandler(w http.ResponseWriter, r *http.Request) {
	tmpl := template.Must(template.ParseFiles("templates/base.html",
"templates/home.html"))
	data := map[string]interface{}{
		"Title": "Home Page",
		"User":  "Guest",
	}
	tmpl.ExecuteTemplate(w, "base.html", data)
}
```

ข้อดี

- แยก logic ของ HTTP ออกจาก business logic
- ทำให้ handler แต่ละไฟล์รับผิดชอบเฉพาะ route

---

2 models

หน้าที่

- เก็บ **structs / data types** ของโปรเจกต์
- ประมวลผล **business logic / DB operations**
- แยกข้อมูลออกจาก handler ⟶ reuse ได้ง่าย

ตัวอย่างโฟลเดอร์และไฟล์

models/
└── user.go

user.go

```
package models

type User struct {
	Name   string
	Age    int
	Active bool
}

// ฟังก์ชัน utility สำหรับ model
func NewUser(name string, age int) User {
	return User{Name: name, Age: age, Active: true}
}
```

ข้อดี

- จัดการข้อมูลให้ชัดเจน
- Business logic อยู่แยกจาก HTTP handler
- ง่ายต่อ unit testing

---

3 routes

หน้าที่

- รวม router และ subrouter
- กำหนด **URL paths ⟶ handlers**
- เพิ่ม middleware ให้ router เช่น logging, auth

ตัวอย่างโฟลเดอร์และไฟล์

routes/
└── routes.go

routes.go

```
package routes

import (
	"github.com/go-chi/chi/v5"
	"github.com/username/mywebapp/handlers"
```

```
)

func InitRoutes() *chi.Mux {
        r := chi.NewRouter()
        r.Get("/", handlers.HomeHandler)
        r.Get("/users", handlers.UsersHandler)
        return r
}
```

ข้อดี

- Router logic อยู่รวมที่เดียว ⟶ maintain ง่าย
- รองรับ **subrouters** และ middleware chain

---

4 templates

หน้าที่

- เก็บ **HTML templates** และ layout
- แยก template functions, partials, inheritance

ตัวอย่างโฟลเดอร์และไฟล์

```
templates/
├── base.html
├── home.html
└── users.html
```

base.html

```
<html>
<head><title>{{.Title}}</title></head>
<body>
<header><h1>{{.Title}}</h1></header>
<main>{{template "content" .}}</main>
<footer>© 2025 My Web App</footer>
</body>
</html>
```

home.html

```
{{define "content"}}
<p>Welcome, {{.User}}!</p>
{{end}}
```

ข้อดี

- แยก **presentation** ออกจาก **logic**
- Template inheritance ⟶ reuse layout ได้ง่าย
- รองรับ dynamic content ผ่าน Go templates

☐ **สรุปการจัดโฟลเดอร์ 4 หลัก**

Folder	หน้าที่หลัก	ตัวอย่างไฟล์
handlers	รับ request / call model / render response	home.go, users.go, api.go
models	Struct, DB logic, business logic	user.go, post.go
routes	Router initialization, subrouters, middleware	routes.go
templates	HTML templates, layout, partials	base.html, home.html, users.html

ข้อดีของโครงสร้างนี้

- Maintainable & Readable
- Separation of Concerns ชัดเจน
- รองรับทีมหลายคนทำงานพร้อมกัน
- ขยายฟีเจอร์ได้ง่าย

ตัวอย่างโปรเจกต์ Go Web แบบเต็มไฟล์ 3 ตัวอย่างพื้นฐาน + 3 แนวประยุกต์ โดยใช้ **โฟลเดอร์ handlers/models/routes/templates** ครบตามโครงสร้างมาตรฐาน พร้อมคำอธิบายโค้ดและผลการรัน

☐ **ตัวอย่างพื้นฐาน 3 โปรแกรม**

Project Structure

```
basic-webapp-1/
├── main.go
├── go.mod
├── handlers/
│   └── home.go
├── models/
│   └── user.go
├── routes/
│   └── routes.go
```

```
└── templates/
    ├── base.html
    └── home.html
```

---

1 Hello World Page

main.go

```
package main

import (
	"log"
	"net/http"
	"basic-webapp-1/routes"
)

func main() {
	r := routes.InitRoutes()
	log.Println("Server running at :8080")
	http.ListenAndServe(":8080", r)
}
```

routes/routes.go

```
package routes

import (
	"net/http"
	"basic-webapp-1/handlers"
	"github.com/go-chi/chi/v5"
)

func InitRoutes() *chi.Mux {
	r := chi.NewRouter()
	r.Get("/", handlers.HomeHandler)
	return r
}
```

handlers/home.go

```
package handlers

import (
	"html/template"
	"net/http"
)

func HomeHandler(w http.ResponseWriter, r *http.Request) {
	tmpl :=
template.Must(template.ParseFiles("templates/base.html","templates/home.html"))
	data := map[string]interface{}{"Title":"Hello World","Message":"Welcome to Basic Web
App!"}
	tmpl.ExecuteTemplate(w,"base.html",data)
}
```

templates/base.html

```
<html>
<head><title>{{.Title}}</title></head>
<body>
<header><h1>{{.Title}}</h1></header>
<main>{{template "content" .}}</main>
<footer>© 2025 Basic Web App</footer>
</body>
</html>
```

templates/home.html

```
{{define "content"}}
<p>{{.Message}}</p>
{{end}}
```

ผลการรัน

- เปิด browser ⟶ http://localhost:8080/
- แสดง **Hello World Page** พร้อมข้อความ: Welcome to Basic Web App!

---

2 Display User List

models/user.go

```
package models
```

```
type User struct {
	Name string
	Age  int
}

var Users = []User{
	{"Alice",25},
	{"Bob",30},
}
```

handlers/home.go

```
package handlers

import (
	"html/template"
	"net/http"
	"basic-webapp-1/models"
)

func HomeHandler(w http.ResponseWriter, r *http.Request) {
	tmpl :=
template.Must(template.ParseFiles("templates/base.html","templates/home.html"))
	data := map[string]interface{}{
		"Title":"User List",
		"Users": models.Users,
	}
	tmpl.ExecuteTemplate(w,"base.html",data)
}
```

templates/home.html

```
{{define "content"}}
<ul>
{{range .Users}}
<li>{{.Name}} - {{.Age}} years old</li>
{{end}}
```

```
</ul>
{{end}}
```

ผลการรัน

- Browser ⟶ แสดงรายการผู้ใช้ Alice, Bob

---

3 Form Submission (GET/POST)

handlers/home.go

```
package handlers

import (
	"html/template"
	"net/http"
	"basic-webapp-1/models"
	"strconv"
)

func HomeHandler(w http.ResponseWriter,r *http.Request){
	tmpl :=
template.Must(template.ParseFiles("templates/base.html","templates/home.html"))
	if r.Method=="POST"{
		name := r.FormValue("name")
		age,_ := strconv.Atoi(r.FormValue("age"))
		models.Users = append(models.Users,models.User{Name:name,Age:age})
		http.Redirect(w,r,"/",http.StatusSeeOther)
		return
	}
	data := map[string]interface{}{"Title":"Add User","Users":models.Users}
	tmpl.ExecuteTemplate(w,"base.html",data)
}
```

templates/home.html

```
{{define "content"}}
<h3>Add User</h3>
<form method="POST" action="/">
Name: <input type="text" name="name">
```

Age: <input type="number" name="age">

<button type="submit">Add</button>

</form>

<h3>User List</h3>

{{range .Users}}

{{.Name}} - {{.Age}} years old

{{end}}

{{end}}

ผลการรัน

- หน้าเว็บแสดง form + user list
- กรอกชื่อและอายุ ⟶ submit ⟶ เพิ่มผู้ใช้ใหม่ใน list

☐ ตัวอย่างแนวประยุกต์ 3 โปรแกรม

Project Structure

app-webapp/

```
├── main.go
├── go.mod
├── handlers/
│   ├── home.go
│   └── api.go
├── models/
│   └── user.go
├── routes/
│   └── routes.go
├── templates/
│   ├── base.html
│   └── home.html
└── static/
    ├── style.css
```

└── app.js

---

1 JSON API + HTML List

handlers/api.go

```
package handlers

import (
        "encoding/json"
        "net/http"
        "app-webapp/models"
)

func UsersAPI(w http.ResponseWriter,r *http.Request){
        w.Header().Set("Content-Type","application/json")
        json.NewEncoder(w).Encode(models.Users)
}
```

routes/routes.go

```
r.Get("/api/users", handlers.UsersAPI)
```

ผลการรัน

- Browser ⟶ /api/users
- JSON output:

```
[{"Name":"Alice","Age":25},{"Name":"Bob","Age":30}]
```

---

2 Middleware Logging

routes/routes.go

```
r.Use(func(next http.Handler) http.Handler{
        return http.HandlerFunc(func(w http.ResponseWriter,r *http.Request){
                log.Printf("Request: %s %s",r.Method,r.URL.Path)
                next.ServeHTTP(w,r)
        })
})
```

ผลการรัน

- ทุก request ⟶ log แสดงใน console เช่น:

```
Request: GET /
```

Request: GET /api/users

---

3 Subrouter / Admin Section

routes/routes.go

```
adminRouter := chi.NewRouter()
adminRouter.Get("/dashboard", handlers.AdminDashboard)
r.Mount("/admin", adminRouter)
```

handlers/home.go

```
func AdminDashboard(w http.ResponseWriter,r *http.Request){
        w.Write([]byte("Welcome to Admin Dashboard"))
}
```

ผลการรัน

- Browser ⟶ /admin/dashboard ⟶ แสดงข้อความ Welcome to Admin Dashboard
- แยก route admin ออกจาก public routes

 ทั้ง 6 ตัวอย่างนี้ ครอบคลุม:

Feature	ตัวอย่าง
Template rendering	1,2,3
Form POST	3
JSON API	1 ของแนวประยุกต์
Middleware	2 ของแนวประยุกต์
Subrouter	3 ของแนวประยุกต์
Handler/Model separation	ทุกตัวอย่าง
Static / CSS / JS	แนวประยุกต์สามารถต่อเติม

Separation of Concerns (MVC Pattern) ใน Go Web

Separation of Concerns (SoC) คือหลักการออกแบบซอฟต์แวร์ที่เน้นให้แต่ละส่วนของโปรแกรมทำงานเฉพาะหน้าที่ของตัวเอง เพื่อให้โค้ด **อ่านง่าย, maintainable, scalable**

ใน Go Web Project นิยมใช้ **MVC Pattern** ซึ่งแบ่งเป็น **3 ส่วนหลัก**

---

1 Model (Data Layer)

หน้าที่

- จัดการ **ข้อมูลและ business logic**
- ติดต่อกับ **database** (CRUD operations)
- จัดการ struct และ validation

ตัวอย่าง

models/user.go

```
package models

type User struct {
        Name   string
        Age    int
        Active bool
}

var Users = []User{
        {"Alice",25,true},
        {"Bob",30,true},
}
```

ข้อดี

- แยกข้อมูลออกจาก handler
- สามารถทดสอบ business logic แยกจาก HTTP request

---

2 View (Presentation Layer)

หน้าที่

- แสดง **HTML templates / JSON / XML**
- ใช้ template functions, inheritance, partials
- ไม่ควรมี business logic

ตัวอย่าง

templates/base.html

```
<html>
<head><title>{{.Title}}</title></head>
<body>
<header>{{.Title}}</header>
<main>{{template "content" .}}</main>
```

```
<footer>© 2025 Web App</footer>
</body>
</html>
```

templates/home.html

```
{{define "content"}}
<ul>
{{range .Users}}
<li>{{.Name}} - {{.Age}} years old</li>
{{end}}
</ul>
{{end}}
```

ข้อดี

- แยก **presentation** จาก logic
- ปรับแต่ง layout / UI โดยไม่กระทบ business logic

---

3 Controller / Handler (Application Layer)

หน้าที่

- รับ **HTTP request** ⟶ route ⟶ handler
- Call **model** เพื่อดึง/บันทึกข้อมูล
- Pass data ไป **view** (HTML template) หรือ return JSON
- ตรวจสอบ request validation, session, auth

ตัวอย่าง

handlers/home.go

```
package handlers

import (
        "html/template"
        "net/http"
        "mywebapp/models"
)

func HomeHandler(w http.ResponseWriter,r *http.Request){
        tmpl :=
template.Must(template.ParseFiles("templates/base.html","templates/home.html"))
```

```
        data := map[string]interface{}{"Title":"User List","Users":models.Users}
        tmpl.ExecuteTemplate(w,"base.html",data)
}
```

ข้อดี

- แยก **HTTP logic** จาก **business logic** และ **presentation**
- รองรับ unit testing ของ handler / model แยกกัน

---

4 Workflow ของ MVC

1. Client ส่ง **request** ⟶ Router
2. Router ⟶ Controller / Handler
3. Handler ⟶ Call Model (ดึงข้อมูล / business logic)
4. Handler ⟶ Pass data ⟶ Template (View)
5. Response ⟶ Client

Request ⟶ Router ⟶ Handler ⟶ Model ⟶ Handler ⟶ Template ⟶ Response

---

5 ข้อดีของ Separation of Concerns ใน Go Web

ข้อดี	รายละเอียด
Maintainability	แก้ไขส่วนใดส่วนหนึ่งโดยไม่กระทบส่วนอื่น
Scalability	รองรับโปรเจกต์ใหญ่, ทีมหลายคน
Reusability	Model และ template ใช้ซ้ำได้หลาย handler
Testability	สามารถ unit test model และ handler แยกกัน

---

6 ตัวอย่าง Integration

```
mywebapp/
├── main.go          # Entry point
├── models/user.go   # Model
├── handlers/home.go # Controller / Handler
├── routes/routes.go # Router
└── templates/       # View
    ├── base.html
    └── home.html
```

- **Controller** ⟶ handlers/home.go

- **Model** ⟶ models/user.go
- **View** ⟶ templates/*.html
- **Router** ⟶ routes/routes.go

ทุกส่วนมีหน้าที่ชัดเจน **Separation of Concerns** ทำให้ระบบอ่านง่ายและ maintain ได้สะดวก

ตัวอย่างโปรเจกต์ Go Web แบบ MVC ครบทั้ง **พื้นฐาน 3 โปรแกรม** และ **แนวประยุกต์ 3 โปรแกรม** โดยใช้โฟลเดอร์ handlers, models, routes, templates ครบ พร้อมคำอธิบายโค้ดและผลการรัน

□ ตัวอย่างพื้นฐาน 3 โปรแกรม

Project Structure (พื้นฐาน)

```
basic-mvc-web/
├── main.go
├── go.mod
├── handlers/
│   └── home.go
├── models/
│   └── user.go
├── routes/
│   └── routes.go
└── templates/
    ├── base.html
    └── home.html
```

1□ Hello World Page

main.go

```
package main

import (
	"log"
	"net/http"
	"basic-mvc-web/routes"
)
```

```
func main() {
	r := routes.InitRoutes()
	log.Println("Server running at :8080")
	http.ListenAndServe(":8080", r)
}
```

routes/routes.go

```
package routes

import (
	"basic-mvc-web/handlers"
	"github.com/go-chi/chi/v5"
)

func InitRoutes() *chi.Mux {
	r := chi.NewRouter()
	r.Get("/", handlers.HomeHandler)
	return r
}
```

handlers/home.go

```
package handlers

import (
	"html/template"
	"net/http"
)

func HomeHandler(w http.ResponseWriter, r *http.Request) {
	tmpl := template.Must(template.ParseFiles("templates/base.html",
"templates/home.html"))
	data := map[string]interface{}{"Title": "Hello World", "Message": "Welcome to MVC
Web App!"}
	tmpl.ExecuteTemplate(w, "base.html", data)
}
```

templates/base.html

```
<html>
<head><title>{{.Title}}</title></head>
<body>
<header><h1>{{.Title}}</h1></header>
<main>{{template "content" .}}</main>
<footer>© 2025 MVC Web</footer>
</body>
</html>
```

templates/home.html

```
{{define "content"}}
<p>{{.Message}}</p>
{{end}}
```

ผลการรัน:

- เปิด browser ⟶ http://localhost:8080/
- แสดงข้อความ **Welcome to MVC Web App!**

---

2 User List Page

models/user.go

```
package models

type User struct {
        Name string
        Age  int
}

var Users = []User{
        {"Alice", 25},
        {"Bob", 30},
}
```

handlers/home.go

```
package handlers

import (
```

```
        "html/template"
        "net/http"
        "basic-mvc-web/models"
)

func HomeHandler(w http.ResponseWriter, r *http.Request) {
        tmpl := template.Must(template.ParseFiles("templates/base.html",
"templates/home.html"))
        data := map[string]interface{}{
                "Title": "User List",
                "Users": models.Users,
        }
        tmpl.ExecuteTemplate(w, "base.html", data)
}
```

templates/home.html

```
{{define "content"}}
<h3>Users</h3>
<ul>
{{range .Users}}
<li>{{.Name}} - {{.Age}} years old</li>
{{end}}
</ul>
{{end}}
```

ผลการรัน:

- Browser ⟶ แสดงรายการผู้ใช้ Alice, Bob

---

3 Add User Form (GET/POST)

handlers/home.go

```
package handlers

import (
        "basic-mvc-web/models"
        "html/template"
        "net/http"
```