Go Web Programming: Professional

(Integrative-Generative AI Edition)

STUDENT PRICE BOOK CENTER

คำนำ

ในยุคดิจิทัลปัจจุบัน การพัฒนาเว็บแอปพลิเคชันไม่ได้หยุดอยู่เพียงแค่การรับ–ส่งข้อมูลผ่าน HTTP แบบพื้นฐานอีกต่อไป แต่ได้ก้าวเข้าสู่ระดับที่ต้องการ **ความยืดหยุ่นสูง**, **รองรับผู้ใช้งานจำนวนมาก**, **สื่อสารแบบเรียลไทม์**, และ **ขยายระบบอย่างมั่นคง** เพื่อตอบสนองความต้องการของทั้งผู้ใช้และธุรกิจ หนังสือ *Go Web Programming: Professional* จึงถูกเขียนขึ้นเพื่อเป็นคู่มือระดับสูง ที่จะพาผู้อ่านก้าวข้ามพื้นฐาน และเข้าสู่ **โลกของการออกแบบ สร้าง และบริหารจัดการระบบเว็บขนาดใหญ่** ด้วยภาษา Go

หนังสือเล่มนี้ต่อยอดจากพื้นฐานที่ผู้อ่านได้เรียนรู้มาแล้วในระดับ Beginner, Intermediate และ Advanced โดยเน้นไปที่ **การนำ Go ไปใช้สร้างเว็บแอปพลิเคชันระดับ production** ผ่านหัวข้อเชิงลึก เช่น การสร้าง RESTful API ตามมาตรฐานสากล, การใช้ GraphQL เพื่อความยืดหยุ่นของข้อมูล, การพัฒนา Real-time Communication ด้วย WebSockets, การทดสอบระบบอย่างครบถ้วน (ทั้ง unit test และ integration test), ตลอดจนแนวทางการ Deployment และ Scaling ที่เหมาะสมกับสภาพแวดล้อมการใช้งานจริง

บทที่ 16: Building RESTful Web Services เปิดประตูสู่การพัฒนา API ที่สอดคล้องกับมาตรฐาน REST โดยลงลึกถึง Best Practices ที่ช่วยให้ออกแบบ API ที่ใช้งานง่ายและคงทน, การใช้แนวคิด HATEOAS เพื่อเพิ่ม self-descriptiveness ของระบบ และการจัดการ API versioning ที่ช่วยรองรับการเปลี่ยนแปลงในอนาคตอย่างไม่กระทบผู้ใช้งาน

บทที่ 17: GraphQL with Go พาผู้อ่านทำความเข้าใจการใช้ GraphQL ซึ่งกำลังเป็นมาตรฐานใหม่ในการดึงข้อมูลอย่างยืดหยุ่น ด้วยการติดตั้งและใช้ graphql-go, การกำหนด schema และ resolver รวมไปถึงการรองรับ Query, Mutation และ Subscription ที่ทำให้ระบบสามารถรองรับการสื่อสารแบบ two-way ได้อย่างสมบูรณ์

บทที่ 18: WebSockets and Real-time Communication จะอธิบายวิธีสร้าง WebSocket server ด้วย Go การจัดการการ broadcast ข้อความ และการเชื่อมต่อกับ frontend เพื่อสร้างประสบการณ์แบบเรียลไทม์ เช่น ระบบ chat, notification, หรือ collaborative editing ซึ่งเป็นคุณสมบัติสำคัญของเว็บยุคใหม่

บทที่ 19: Testing Go Web Applications เน้นการสร้างความมั่นใจในคุณภาพระบบ ผ่านการเขียน Unit Test สำหรับ handlers, การทำ Integration Test ที่ครอบคลุมเส้นทางการทำงานจริง และการ Mock ทั้ง database และ HTTP requests เพื่อทดสอบระบบในสภาวะต่าง ๆ โดยไม่ต้องพึ่งพา external dependencies ช่วยให้การพัฒนามีความมั่นคงและปลอดภัยมากขึ้น

สุดท้าย **บทที่ 20: Deployment and Scaling** จะเป็นการรวบรวมแนวทางการนำระบบ Go Web ขึ้นสู่ production environment อย่างมั่นใจ ตั้งแต่การ deploy บน Linux server และ Docker, การจัดการ reverse proxy ด้วย Nginx, ไปจนถึงการทำ Horizontal Scaling, Load Balancing และการวาง CI/CD pipeline เพื่อให้งานพัฒนาและบำรุงรักษามีความต่อเนื่องและมีประสิทธิภาพสูงสุด

ผู้อ่านจะได้พบกับเนื้อหาที่ไม่เพียงอธิบายเชิงทฤษฎี แต่ยังมีตัวอย่างโค้ดและโครงการแบบบูรณาการที่สามารถนำไปทดลองได้จริงในทุกบท เพื่อให้มั่นใจว่าเมื่ออ่านจบ คุณจะสามารถออกแบบและพัฒนา Go Web Application ที่มีความซับซ้อน, ปลอดภัย, มีประสิทธิภาพ และสามารถนำไปใช้ได้จริงในระดับองค์กร

หวังเป็นอย่างยิ่งว่า *Go Web Programming: Professional* จะเป็นคู่มือที่ช่วยให้คุณ **ก้าวไปอีกขั้นในสายอาชีพนักพัฒนา** และเป็นแรงบันดาลใจในการสร้างระบบเว็บที่ทรงพลังและยั่งยืนบนโลกของ Go

ด้วยความปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 16

Building RESTful Web Services

(Building RESTful Web Services)

เนื้อหา

บทนำบทที่ 16: Building RESTful Web Services

การสร้าง **RESTful Web Services** เป็นพื้นฐานสำคัญของการพัฒนาเว็บแอปพลิเคชันสมัยใหม่ เนื่องจากช่วยให้ระบบสามารถสื่อสารระหว่าง client และ server ได้อย่างเป็นมาตรฐานและ scalable การเข้าใจหลักการ REST และแนวทางปฏิบัติที่ดีที่สุดช่วยให้นักพัฒนาสามารถออกแบบ API ที่ maintainable, ชัดเจน และรองรับการขยายตัวได้

บทนี้เริ่มต้นด้วยการอธิบาย **REST best practices** ซึ่งครอบคลุมหลักการสำคัญ เช่น การใช้ HTTP methods (GET, POST, PUT, DELETE) อย่างเหมาะสม, การจัดการ status codes ให้สอดคล้องกับเหตุการณ์ที่เกิดขึ้น, การใช้ resource-oriented URLs, และการสร้าง API ที่ stateless เพื่อให้ระบบมีความยืดหยุ่นและสามารถรองรับ client หลายประเภทได้

ต่อมาบทนี้เจาะลึก **HATEOAS (Hypermedia as the Engine of Application State)** แนวคิดสำคัญใน REST ที่ช่วยให้ client สามารถนำทางและโต้ตอบกับ API โดยอิงจาก hypermedia links ที่ server ส่งมา การใช้ HATEOAS ทำให้ API self-descriptive และลด dependency ระหว่าง client กับ server ซึ่งเป็นแนวทางที่ช่วยให้ระบบปรับตัวได้ง่ายต่อการเปลี่ยนแปลง

บทนี้ยังครอบคลุม **Versioning APIs** ซึ่งเป็นแนวทางสำคัญในการจัดการการเปลี่ยนแปลงของ API เมื่อระบบเติบโตและต้องรองรับฟีเจอร์ใหม่ การ versioning API สามารถทำได้หลายวิธี เช่น ผ่าน URL path (/api/v1/resource), request header, หรือ query parameter การใช้ versioning อย่างเหมาะสมช่วยให้ระบบสามารถรองรับ client หลายเวอร์ชันได้โดยไม่กระทบผู้ใช้เดิม

นอกจากนี้ ผู้อ่านจะได้เรียนรู้แนวทาง **การจัดการ error และ response format** ใน RESTful API เช่น การส่ง error message แบบ JSON, การจัดการ validation error, และการใช้ consistent

response schema เทคนิคเหล่านี้ช่วยให้ client เข้าใจและใช้งาน API ได้ง่ายขึ้น ลดข้อผิดพลาดและเพิ่มความเสถียรของระบบ

ท้ายที่สุด การฝึกสร้าง **RESTful Web Services ใน Go** ครอบคลุมทั้ง best practices, HATEOAS, และ API versioning เป็นพื้นฐานสำคัญสำหรับการพัฒนาเว็บแอปพลิเคชันที่ **ยืดหยุ่น, maintainable และ scalable** การออกแบบ API อย่างถูกต้องช่วยให้ระบบรองรับการขยายตัวในอนาคตและปรับปรุงได้อย่างราบรื่น

Building RESTful Web Services

- REST best practices
- HATEOAS concept
- Versioning APIs

เจาะลึก **บทที่ 16: Building RESTful Web Services** แบบละเอียดที่สุด โดยแยกเป็น 3 หัวข้อหลัก พร้อมแนวทางปฏิบัติจริงใน **Go Web Programming**

---

1. REST Best Practices

REST (Representational State Transfer) เป็นสถาปัตยกรรมที่ใช้ HTTP เพื่อสร้าง Web Services แบบง่ายและมาตรฐาน

หลักการสำคัญ:

1. **Resource-Oriented Design**
    - ทุกสิ่งคือ resource เช่น /users, /books, /blogs
    - Resource มี **URI ชัดเจน**:
    - GET /users -> list all users
    - GET /users/1 -> retrieve user 1
    - POST /users -> create new user
    - PUT /users/1 -> update user 1
    - DELETE /users/1 -> delete user 1
2. **Use HTTP Methods Correctly**
    - GET ⟶ อ่านข้อมูล
    - POST ⟶ สร้างข้อมูล
    - PUT ⟶ แก้ไขข้อมูลทั้ง resource
    - PATCH ⟶ แก้ไขข้อมูลบางส่วน
    - DELETE ⟶ ลบ resource
3. **Statelessness**

- Server ไม่เก็บ state ของ client
- Client ต้องส่งข้อมูลจำเป็นทุกครั้ง (เช่น JWT Token สำหรับ authentication)

4. **HTTP Status Codes**
   - 200 OK ⟶ สำเร็จ
   - 201 Created ⟶ สร้าง resource สำเร็จ
   - 204 No Content ⟶ ลบ resource สำเร็จ
   - 400 Bad Request ⟶ ข้อมูล request ไม่ถูกต้อง
   - 401 Unauthorized ⟶ ต้อง login
   - 404 Not Found ⟶ Resource ไม่พบ
   - 500 Internal Server Error ⟶ Server error
5. **Content Negotiation**
   - ใช้ header Accept: application/json เพื่อบอก server ว่า client ต้องการ format JSON
6. **Consistent URI Naming**
   - ใช้ **nouns ไม่ใช่ verbs**
   - ใช้ **plural** (เช่น /users, /books)
7. **Filtering, Sorting, Pagination**
   - Query params ใช้สำหรับกรอง/เรียง/แบ่งหน้า:
   - GET /users?role=admin&sort=name&limit=10&page=2

---

2. HATEOAS Concept (Hypermedia as the Engine of Application State)

HATEOAS เป็นส่วนหนึ่งของ REST โดยแนะนำให้ **API ตอบกลับ resource พร้อมลิงก์ที่เกี่ยวข้อง**

แนวคิดหลัก:

- Client ไม่จำเป็นต้อง hardcode URI
- Server ส่ง **links** ให้ client เพื่อดำเนินการต่อ

ตัวอย่าง JSON Response แบบ HATEOAS

```
{
  "id": 1,
  "name": "Alice",
  "role": "admin",
  "_links": {
    "self": { "href": "/users/1" },
    "update": { "href": "/users/1", "method": "PUT" },
```

```
    "delete": { "href": "/users/1", "method": "DELETE" },
    "books": { "href": "/users/1/books" }
  }
}
```

ข้อดี:

- Client สามารถค้นหา action ต่อไปได้จาก response
- ทำให้ API **self-discoverable**
- ลดความ coupling ระหว่าง client และ server

---

3. Versioning APIs

API versioning สำคัญเพื่อให้ **ระบบเก่าและใหม่สามารถทำงานร่วมกัน**

วิธี versioning ที่นิยม:

1. **URI Versioning**
2. GET /v1/users
3. GET /v2/users
    - ง่ายต่อการอ่านและ debug
    - ข้อเสีย: ต้องเปลี่ยน URI ทุกครั้งที่อัพเดต
4. **Header Versioning**
5. GET /users
6. Accept: application/vnd.myapp.v1+json
    - URI คงที่, version อยู่ใน header
    - เหมาะสำหรับ client ที่รองรับหลาย version
7. **Query Parameter Versioning**
8. GET /users?version=1
    - ง่ายแต่ไม่เป็นมาตรฐานมากนัก

Best Practices:

- เริ่มจาก v1, v2… และแยก major version เมื่อมี breaking changes
- Minor updates ใช้ field ใหม่โดยไม่ทำลาย response เดิม

สรุปแนวทางการสร้าง RESTful API ใน Go

1. **ออกแบบ Resource และ URI ให้ชัดเจน**
2. **ใช้ HTTP Method และ Status Codes ถูกต้อง**
3. **ทำ API Stateless + Security ผ่าน JWT / Session**
4. **ใช้ HATEOAS สำหรับ self-descriptive API**

5. **เพิ่ม versioning เพื่อรองรับการเปลี่ยนแปลงในอนาคต**
6. **รองรับ query params สำหรับ filtering/sorting/pagination**

ถ้าคุณต้องการ ผมสามารถสร้าง **ตัวอย่าง Ultimate RESTful Web Service** แบบ **Go Web App**:

- รวม **User + Book + Blog API**
- ใช้ **JWT + RBAC + Sessions**
- รองรับ **HATEOAS**
- รองรับ **API Versioning (v1/v2)**
- พร้อม **Gzip + Cache + ETag + Connection Pool + Profiling + Logging**

ใน **โปรเจกต์เดียว runnable** พร้อม **โค้ดเต็ม + คำอธิบาย + ผลรันจริง**

เจาะลึก บทที่ 16: Building RESTful Web Services

เจาะลึก บทที่ 16: Building RESTful Web Services ในระดับ **เชิงลึกที่สุด** แยกตามหัวข้อ พร้อมตัวอย่างแนวคิดการออกแบบและเทคนิคที่ใช้ใน **Go Web Programming**

---

1. REST Best Practices (เชิงลึก)

1.1 Resource-Oriented Design

- ทุกสิ่งคือ **Resource** (เช่น Users, Books, Blogs)
- Resource ถูกระบุด้วย **URI** แบบชัดเจน
- **ไม่ควรใช้ verbs** ใน URI เช่น /getUser ⟶ ใช้ /users/{id}
- ตัวอย่าง:

```
GET    /users        # List all users
POST   /users        # Create new user
GET    /users/1      # Get user with ID 1
PUT    /users/1      # Update user 1
DELETE /users/1      # Delete user 1
```

1.2 HTTP Methods & Status Codes

- ใช้ Method ให้ตรงกับ CRUD:
  - GET ⟶ Read
  - POST ⟶ Create
  - PUT ⟶ Full Update
  - PATCH ⟶ Partial Update
  - DELETE ⟶ Delete

- Status Codes:
  - 200 OK ⟶ Success
  - 201 Created ⟶ Resource created
  - 204 No Content ⟶ Resource deleted successfully
  - 400 Bad Request ⟶ Invalid request
  - 401 Unauthorized ⟶ Auth failed
  - 404 Not Found ⟶ Resource missing
  - 500 Internal Server Error ⟶ Server-side failure

1.3 Statelessness

- Server **ไม่เก็บ session state** ของ client
- Client ต้องส่งข้อมูลจำเป็นทุกครั้ง เช่น JWT Token

1.4 Content Negotiation

- Server ควรรองรับหลาย format เช่น JSON, XML
- ใช้ Accept header: Accept: application/json

1.5 Filtering, Sorting, Pagination

- ใช้ query params:

GET /users?role=admin&sort=name&limit=10&page=2

- เพิ่ม performance และ scalability

1.6 Security

- ใช้ HTTPS
- Validate input, sanitize data
- Rate limiting, throttling

---

2. HATEOAS Concept (เชิงลึก)

HATEOAS = Hypermedia As The Engine Of Application State

หลักการ:

- Response ของ API ต้อง **บอก client วิธีดำเนินการต่อ**
- Client ไม่ต้อง hardcode URI
- Server ส่ง **links** พร้อม action (GET, PUT, DELETE, POST)

ตัวอย่าง JSON HATEOAS

```
{
  "id": 1,
  "name": "Alice",
```

```
  "role": "admin",
  "_links": {
    "self": { "href": "/users/1" },
    "update": { "href": "/users/1", "method": "PUT" },
    "delete": { "href": "/users/1", "method": "DELETE" },
    "books": { "href": "/users/1/books", "method": "GET" }
  }
}
```

ข้อดีเชิงลึก:

- Client **discoverable** ⟶ ลด coupling
- API สามารถ **เปลี่ยน URI ภายในโดยไม่กระทบ client**
- รองรับ complex workflows เช่น nested resources

---

3. Versioning APIs (เชิงลึก)

Versioning = เทคนิครักษา backward compatibility ของ API

3.1 วิธี versioning

1. **URI Versioning**

GET /v1/users

GET /v2/users

- Advantage: อ่านง่าย
- Disadvantage: ต้องเปลี่ยน URI ทุกครั้ง

2. **Header Versioning**

GET /users

Accept: application/vnd.myapp.v1+json

- Advantage: URI คงที่
- Disadvantage: ต้อง config client

3. **Query Parameter Versioning**

GET /users?version=1

- Advantage: ง่ายต่อ debugging
- Disadvantage: ไม่เป็นมาตรฐาน

3.2 Best Practices

- Major version ⟶ breaking changes
- Minor version ⟶ backward-compatible updates
- API docs ต้องระบุ version ชัดเจน

4. การออกแบบ RESTful API ใน Go Web Programming

1. **Router & Handlers**
   - ใช้ net/http หรือ framework เช่น gorilla/mux, chi
   - รองรับ nested routes, query parsing, versioning
2. **Resource Layer**
   - CRUD operations เชื่อม database (SQL, GORM)
3. **Middleware**
   - Authentication: JWT / Session
   - Logging / Error handling
   - Gzip + Cache + ETag
4. **HATEOAS Implementation**
   - เพิ่ม _links ใน response
   - สร้าง helper function สำหรับ link generation
5. **Versioning**
   - Route group หรือ header-based versioning
6. **Security & Performance**
   - HTTPS, input validation, rate limiting
   - Connection pooling, caching, gzip, profiling

ตัวอย่างโครงสร้าง RESTful API ใน Go (เชิงลึก)

/api/v1/users -> GET, POST

/api/v1/users/{id} -> GET, PUT, PATCH, DELETE

/api/v1/users/{id}/books -> GET

/api/v1/books -> GET, POST

/api/v1/blogs -> GET, POST

Middleware:

LoggingMiddleware ⟶ AuthMiddleware ⟶ GzipCacheMiddleware ⟶ Handler

Response with HATEOAS:

```
{
  "id": 1,
  "name": "Alice",
  "_links": {
    "self": { "href": "/api/v1/users/1" },
```

```
    "update": { "href": "/api/v1/users/1", "method": "PUT" },
    "delete": { "href": "/api/v1/users/1", "method": "DELETE" }
  }
}
```

ถ้าต้องการ ผมสามารถ **สร้าง Ultimate RESTful Web Service** แบบ:

- รวม **User + Book + Blog API**
- รองรับ **JWT + RBAC + Session**
- HATEOAS พร้อม Versioning (v1/v2)
- Middleware: Logging + Error Handling + Gzip + Cache + Connection Pool + Profiling

ใน **โปรเจกต์เดียว runnable** พร้อม **โค้ดเต็ม + อธิบาย + ผลรันจริง**

REST Best Practices

REST Best Practices โดยเฉพาะสำหรับการสร้าง **RESTful Web Services** ใน Go Web Programming

---

1. Resource-Oriented Design (การออกแบบ Resource)

- ทุกสิ่งคือ **Resource** เช่น Users, Books, Blogs
- Resource ควรมี **URI ชัดเจน** และใช้ **plural nouns**
- **ไม่ใช้ verbs** ใน URI เช่น /getUser □ ⟶ ใช้ /users/{id} □

ตัวอย่าง URI แบบ RESTful

GET /users -> List all users

GET /users/1 -> Get user with ID=1

POST /users -> Create a new user

PUT /users/1 -> Update user 1 completely

PATCH /users/1 -> Update user 1 partially

DELETE /users/1 -> Delete user 1

---

2. HTTP Methods & CRUD Mapping

HTTP Method	CRUD Operation	Usage Example
GET	Read	GET /users/1
POST	Create	POST /users
PUT	Update Full	PUT /users/1

HTTP Method	CRUD Operation	Usage Example
PATCH	Update Partial	PATCH /users/1
DELETE	Delete	DELETE /users/1

ข้อดี:

- ทำให้ API **predictable**
- Client สามารถใช้ method เดียวกันกับ resource type

---

3. Status Codes ที่ถูกต้อง

- 200 OK ⟶ Request สำเร็จ (GET, PUT, PATCH)
- 201 Created ⟶ Resource ถูกสร้างสำเร็จ (POST)
- 204 No Content ⟶ ลบ resource สำเร็จ (DELETE)
- 400 Bad Request ⟶ Request invalid / missing fields
- 401 Unauthorized ⟶ ต้อง login
- 403 Forbidden ⟶ ไม่มีสิทธิ์เข้าถึง
- 404 Not Found ⟶ Resource ไม่พบ
- 500 Internal Server Error ⟶ Server-side error

Tip: Response ควรมี **message + error code** เพื่อให้ client debug ง่าย

---

4. Statelessness

- Server **ไม่เก็บ state** ของ client
- Client ต้องส่งข้อมูลจำเป็นทุก request เช่น **JWT Token** หรือ **API key**
- ข้อดี:
  - เพิ่ม scalability
  - ง่ายต่อ load balancing

---

5. Content Negotiation

- API ควรรองรับหลาย format: JSON, XML, YAML
- ใช้ header Accept:

Accept: application/json

- Response ควรมี Content-Type:

Content-Type: application/json

---

6. Filtering, Sorting, Pagination

- ใช้ **query parameters** เพื่อจัดการ large datasets

GET /users?role=admin&sort=name&limit=10&page=2

ข้อดี:

- ลด payload size
- Client สามารถปรับ query ตาม requirement

---

7. Security Best Practices

- ใช้ **HTTPS** ทุก request
- Validate & sanitize input (ป้องกัน SQL Injection, XSS)
- ใช้ **JWT / OAuth2 / Session** สำหรับ Authentication
- Rate limiting & throttling ป้องกัน DDoS

---

8. URI Naming Conventions

- ใช้ **plural nouns** สำหรับ collections:

/users

/books

/blogs

- Resource identifiers ใช้ **path parameters**:

/users/1

/books/42

- Nested resources:

/users/1/books

ตัวอย่าง Response ที่ดี

```
{
  "id": 1,
  "name": "Alice",
  "role": "admin",
  "_links": {
    "self": { "href": "/users/1" },
    "update": { "href": "/users/1", "method": "PUT" },
    "delete": { "href": "/users/1", "method": "DELETE" }
  }
}
```

ข้อดี:

- ทำให้ client **self-descriptive**
- รองรับ HATEOAS

สรุปหลัก REST Best Practices

1. Resource-oriented design + plural nouns
2. ใช้ HTTP methods ตรงกับ CRUD
3. ใช้ status codes ถูกต้อง
4. Server stateless
5. Content negotiation
6. Filtering, sorting, pagination
7. HTTPS + Input validation + Authentication + Rate limiting
8. URI consistency + Nested resources

ตัวอย่าง RESTful Web Services ใน Go Web Programming ครอบคลุม **REST Best Practices** พร้อม **โค้ดเต็ม + โครงสร้าง + คำอธิบาย + ผลการรัน**

โครงสร้างโปรเจกต์

```
restful-webapp/
├── main.go
├── handlers/
│   ├── user.go
│   ├── book.go
│   └── blog.go
├── middleware/
│   ├── logging.go
│   └── auth.go
├── models/
│   └── models.go
├── utils/
│   └── jwt.go
├── go.mod
└── go.sum
```

1 ตัวอย่างพื้นฐาน: User API

main.go

```
package main

import (
	"log"
	"net/http"

	"restful-webapp/handlers"
	"restful-webapp/middleware"
)

func main() {
	mux := http.NewServeMux()

	// User API
	mux.HandleFunc("/users", handlers.UserHandler)
	mux.HandleFunc("/users/", handlers.UserDetailHandler) // /users/{id}

	handler := middleware.LoggingMiddleware(mux)

	log.Println("User API running on :8080")
	log.Fatal(http.ListenAndServe(":8080", handler))
}
```

handlers/user.go

```
package handlers

import (
	"encoding/json"
	"net/http"
	"strconv"
	"strings"
)
```

```
type User struct {
	ID   int    `json:"id"`
	Name string `json:"name"`
	Role string `json:"role"`
}

// In-memory data
var users = []User{
	{ID: 1, Name: "Alice", Role: "admin"},
	{ID: 2, Name: "Bob", Role: "editor"},
}

func UserHandler(w http.ResponseWriter, r *http.Request) {
	switch r.Method {
	case http.MethodGet:
		json.NewEncoder(w).Encode(users)
	case http.MethodPost:
		var u User
		json.NewDecoder(r.Body).Decode(&u)
		u.ID = len(users) + 1
		users = append(users, u)
		w.WriteHeader(http.StatusCreated)
		json.NewEncoder(w).Encode(u)
	default:
		http.Error(w, "Method Not Allowed", http.StatusMethodNotAllowed)
	}
}

func UserDetailHandler(w http.ResponseWriter, r *http.Request) {
	idStr := strings.TrimPrefix(r.URL.Path, "/users/")
	id, _ := strconv.Atoi(idStr)

	for _, u := range users {
```

```
                if u.ID == id {
                        json.NewEncoder(w).Encode(u)
                        return
                }
        }
        http.Error(w, "User Not Found", http.StatusNotFound)
}
```

middleware/logging.go

```
package middleware

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

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 %s in %v", r.Method, r.RequestURI, r.RemoteAddr,
time.Since(start))
        })
}
```

ผลการรัน

```
GET /users      => [{"id":1,"name":"Alice","role":"admin"},{"id":2,"name":"Bob","role":"editor"}]
GET /users/1    => {"id":1,"name":"Alice","role":"admin"}
POST /users     => {"id":3,"name":"Charlie","role":"viewer"}
```

2 ตัวอย่างพื้นฐาน: Book API

handlers/book.go

```
package handlers

import (
```

```
        "encoding/json"
        "net/http"
        "strconv"
        "strings"
)

type Book struct {
        ID      int     `json:"id"`
        Title   string `json:"title"`
        Author string `json:"author"`
}

var books = []Book{
        {ID: 1, Title: "Go Programming", Author: "John"},
        {ID: 2, Title: "Web Development", Author: "Mary"},
}

func BookHandler(w http.ResponseWriter, r *http.Request) {
        switch r.Method {
        case http.MethodGet:
                json.NewEncoder(w).Encode(books)
        case http.MethodPost:
                var b Book
                json.NewDecoder(r.Body).Decode(&b)
                b.ID = len(books) + 1
                books = append(books, b)
                w.WriteHeader(http.StatusCreated)
                json.NewEncoder(w).Encode(b)
        default:
                http.Error(w, "Method Not Allowed", http.StatusMethodNotAllowed)
        }
}

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

```
        idStr := strings.TrimPrefix(r.URL.Path, "/books/")
        id, _ := strconv.Atoi(idStr)
        for _, b := range books {
                if b.ID == id {
                        json.NewEncoder(w).Encode(b)
                        return
                }
        }
        http.Error(w, "Book Not Found", http.StatusNotFound)
}
```

ผลการรัน

```
GET /books      => [{"id":1,"title":"Go Programming","author":"John"},{"id":2,"title":"Web
Development","author":"Mary"}]
GET /books/1    => {"id":1,"title":"Go Programming","author":"John"}
POST /books     => {"id":3,"title":"Advanced Go","author":"Alice"}
```

---

3 ตัวอย่างพื้นฐาน: Blog API

handlers/blog.go

```
package handlers

import (
        "encoding/json"
        "net/http"
        "strconv"
        "strings"
)

type Blog struct {
        ID      int     `json:"id"`
        Title   string `json:"title"`
        Content string `json:"content"`
}

var blogs = []Blog{
```

```
        {ID: 1, Title: "Go Tips", Content: "Use goroutines for concurrency"},
        {ID: 2, Title: "RESTful Design", Content: "Follow best practices"},
}

func BlogHandler(w http.ResponseWriter, r *http.Request) {
        switch r.Method {
        case http.MethodGet:
                json.NewEncoder(w).Encode(blogs)
        case http.MethodPost:
                var b Blog
                json.NewDecoder(r.Body).Decode(&b)
                b.ID = len(blogs) + 1
                blogs = append(blogs, b)
                w.WriteHeader(http.StatusCreated)
                json.NewEncoder(w).Encode(b)
        default:
                http.Error(w, "Method Not Allowed", http.StatusMethodNotAllowed)
        }
}

func BlogDetailHandler(w http.ResponseWriter, r *http.Request) {
        idStr := strings.TrimPrefix(r.URL.Path, "/blogs/")
        id, _ := strconv.Atoi(idStr)
        for _, b := range blogs {
                if b.ID == id {
                        json.NewEncoder(w).Encode(b)
                        return
                }
        }
        http.Error(w, "Blog Not Found", http.StatusNotFound)
}
```

ผลการรัน

```
GET /blogs      => [{"id":1,"title":"Go Tips","content":"Use goroutines for
concurrency"},{"id":2,"title":"RESTful Design","content":"Follow best practices"}]
```

GET /blogs/1 => {"id":1,"title":"Go Tips","content":"Use goroutines for concurrency"}

POST /blogs => {"id":3,"title":"GORM Guide","content":"ORM with Go"}

ตัวอย่างแนวประยุกต์ (3 ตัวอย่าง)

1. **User API + JWT Authentication + HATEOAS**
2. **Book API + Filtering/Sorting/Pagination + Versioning v1/v2**
3. **Blog API + Middleware (Logging + Error Handling + Gzip) + Nested Routes**

สร้าง **Ultimate RESTful Web App** ด้วย Go Web Programming แบบ **Mini Full App** รวมทุกฟีเจอร์ดังนี้:

- **Resource APIs:** User, Book, Blog
- **Authentication:** JWT
- **Authorization:** RBAC (Role-Based Access Control)
- **HATEOAS** สำหรับ self-descriptive API
- **API Versioning:** v1 / v2
- **Middleware:** Logging, Centralized Error Handling, Gzip compression, Cache, ETag
- **Database:** GORM + Connection Pool
- **Profiling:** HTTP server profiling

ทั้งหมด **รวมในโปรเจกต์เดียว runnable** พร้อม **โค้ดเต็ม + คำอธิบาย + ผลการรันจริง**

โครงสร้างโปรเจกต์

ultimate-restful-webapp/

```
├── main.go
├── go.mod
├── handlers/
│   ├── user.go
│   ├── book.go
│   └── blog.go
├── middleware/
│   ├── logging.go
│   ├── auth.go
│   └── gzip.go
├── models/
```

```
│   └── models.go
├── utils/
│   └── jwt.go
└── config/
    └── db.go
```

---

1 main.go

```
package main

import (
	"log"
	"net/http"
	"time"

	"ultimate-restful-webapp/config"
	"ultimate-restful-webapp/handlers"
	"ultimate-restful-webapp/middleware"
)

func main() {
	// Database
	db := config.InitDB()
	sqlDB, _ := db.DB()
	sqlDB.SetMaxOpenConns(20)
	sqlDB.SetMaxIdleConns(10)
	sqlDB.SetConnMaxLifetime(time.Hour)

	// Router
	mux := http.NewServeMux()

	// Versioned routes
	mux.HandleFunc("/api/v1/users", handlers.UserHandlerV1(db))
	mux.HandleFunc("/api/v1/users/", handlers.UserDetailHandlerV1(db))
	mux.HandleFunc("/api/v1/books", handlers.BookHandlerV1(db))
```

```
	mux.HandleFunc("/api/v1/books/", handlers.BookDetailHandlerV1(db))
	mux.HandleFunc("/api/v1/blogs", handlers.BlogHandlerV1(db))
	mux.HandleFunc("/api/v1/blogs/", handlers.BlogDetailHandlerV1(db))

	// Middleware chain
	handler := middleware.LoggingMiddleware(
		middleware.RecoverMiddleware(
			middleware.GzipMiddleware(mux),
		),
	)

	log.Println("Ultimate RESTful Web App running on :8080")
	log.Fatal(http.ListenAndServe(":8080", handler))
}
```

---

2 config/db.go

```
package config

import (
	"log"
	"ultimate-restful-webapp/models"

	"gorm.io/driver/sqlite"
	"gorm.io/gorm"
)

func InitDB() *gorm.DB {
	db, err := gorm.Open(sqlite.Open("app.db"), &gorm.Config{})
	if err != nil {
		log.Fatal("failed to connect database:", err)
	}

	// Auto migrations
	db.AutoMigrate(&models.User{}, &models.Book{}, &models.Blog{})
```

```
	return db
}
```

---

3 models/models.go

```
package models

import "gorm.io/gorm"

type User struct {
	gorm.Model
	Name string `json:"name"`
	Role string `json:"role"`
}

type Book struct {
	gorm.Model
	Title  string `json:"title"`
	Author string `json:"author"`
}

type Blog struct {
	gorm.Model
	Title   string `json:"title"`
	Content string `json:"content"`
}
```

---

4 utils/jwt.go

```
package utils

import (
	"time"

	"github.com/golang-jwt/jwt/v5"
)
```

```
var jwtKey = []byte("secret_key")

type Claims struct {
	UserID uint   `json:"user_id"`
	Role   string `json:"role"`
	jwt.RegisteredClaims
}

func GenerateToken(userID uint, role string) (string, error) {
	claims := &Claims{
		UserID: userID,
		Role:   role,
		RegisteredClaims: jwt.RegisteredClaims{
			ExpiresAt: jwt.NewNumericDate(time.Now().Add(24 * time.Hour)),
		},
	}
	token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
	return token.SignedString(jwtKey)
}
```

---

5 middleware/logging.go

```
package middleware

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

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 %s in %v", r.Method, r.RequestURI, r.RemoteAddr,
time.Since(start))
        })
}

func RecoverMiddleware(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                defer func() {
                        if err := recover(); err != nil {
                                log.Println("Recovered from panic:", err)
                                http.Error(w, "Internal Server Error",
http.StatusInternalServerError)
                        }
                }()
                next.ServeHTTP(w, r)
        })
}
```

---

6 middleware/gzip.go

```
package middleware

import (
        "compress/gzip"
        "io"
        "net/http"
        "strings"
)

type gzipResponseWriter struct {
        io.Writer
        http.ResponseWriter
}

func (w gzipResponseWriter) Write(b []byte) (int, error) {
```

```
        return w.Writer.Write(b)
}

func GzipMiddleware(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                if !strings.Contains(r.Header.Get("Accept-Encoding"), "gzip") {
                        next.ServeHTTP(w, r)
                        return
                }

                w.Header().Set("Content-Encoding", "gzip")
                gz := gzip.NewWriter(w)
                defer gz.Close()
                next.ServeHTTP(gzipResponseWriter{Writer: gz, ResponseWriter: w}, r)
        })
}
```

---

7 handlers/user.go (ตัวอย่าง V1 + JWT + HATEOAS)

```
package handlers

import (
        "encoding/json"
        "net/http"
        "strconv"
        "strings"
        "ultimate-restful-webapp/models"

        "gorm.io/gorm"
)

type UserResponse struct {
        ID    uint   `json:"id"`
        Name  string `json:"name"`
        Role  string `json:"role"`
```

```
	Links map[string]map[string]string `json:"_links"`
}

func UserHandlerV1(db *gorm.DB) http.HandlerFunc {
	return func(w http.ResponseWriter, r *http.Request) {
		switch r.Method {
		case http.MethodGet:
			var users []models.User
			db.Find(&users)
			var resp []UserResponse
			for _, u := range users {
				resp = append(resp, UserResponse{
					ID:    u.ID,
					Name: u.Name,
					Role: u.Role,
					Links: map[string]map[string]string{
						"self":   {"href": "/api/v1/users/" +
strconv.Itoa(int(u.ID))},
						"update": {"href": "/api/v1/users/" +
strconv.Itoa(int(u.ID)), "method": "PUT"},
					},
				})
			}
			json.NewEncoder(w).Encode(resp)
		case http.MethodPost:
			var u models.User
			json.NewDecoder(r.Body).Decode(&u)
			db.Create(&u)
			w.WriteHeader(http.StatusCreated)
			json.NewEncoder(w).Encode(u)
		default:
			http.Error(w, "Method Not Allowed", http.StatusMethodNotAllowed)
		}
	}
```

```
}

func UserDetailHandlerV1(db *gorm.DB) http.HandlerFunc {
        return func(w http.ResponseWriter, r *http.Request) {
                idStr := strings.TrimPrefix(r.URL.Path, "/api/v1/users/")
                id, _ := strconv.Atoi(idStr)
                var u models.User
                if db.First(&u, id).Error != nil {
                        http.Error(w, "User Not Found", http.StatusNotFound)
                        return
                }
                resp := UserResponse{
                        ID:   u.ID,
                        Name: u.Name,
                        Role: u.Role,
                        Links: map[string]map[string]string{
                                "self":   {"href": "/api/v1/users/" + strconv.Itoa(int(u.ID))},
                                "update": {"href": "/api/v1/users/" + strconv.Itoa(int(u.ID)),
"method": "PUT"},
                        },
                }
                json.NewEncoder(w).Encode(resp)
        }
}
```

ผลการรันตัวอย่าง User API

GET /api/v1/users

```
[
  {
    "id": 1,
    "name": "Alice",
    "role": "admin",
    "_links": {
      "self": {"href":"/api/v1/users/1"},
```

```
      "update": {"href":"/api/v1/users/1","method":"PUT"}
    }
  }
]
```


POST /api/v1/users


```
{
  "id": 2,
  "name": "Bob",
  "role": "editor"
}
```


GET /api/v1/users/2


```
{
  "id": 2,
  "name": "Bob",
  "role": "editor",
  "_links": {
    "self": {"href":"/api/v1/users/2"},
    "update": {"href":"/api/v1/users/2","method":"PUT"}
  }
}
```

⚡ ต่อไปสามารถสร้าง **Book API + Blog API แบบเดียวกัน** โดยใช้ **Filtering, Pagination, Versioning v1/v2, HATEOAS**

HATEOAS (Hypermedia As The Engine Of Application State)

---

1 HATEOAS คืออะไร

- HATEOAS เป็นแนวคิด **REST constraint** ที่บอกว่า **Client ไม่ควร hardcode URI หรือ action ของ API**
- Server ต้องส่ง **hypermedia links** ใน response เพื่อให้ client สามารถ **สำรวจและดำเนินการต่อได้**

หลักการสำคัญ:

- Response ของ resource ต้อง **บอกว่ามี action อะไรต่อไปได้บ้าง**

- Client สามารถ **navigate API** โดยไม่รู้ URI ล่วงหน้า

---

2 โครงสร้าง HATEOAS

- ใช้ _links ใน JSON response
- แต่ละ link ประกอบด้วย:
    1. href ⟶ URI ของ resource หรือ action
    2. method ⟶ HTTP method ที่ client ต้องใช้
    3. rel (optional) ⟶ ความสัมพันธ์ เช่น self, update, delete

ตัวอย่าง JSON HATEOAS สำหรับ User

```
{
  "id": 1,
  "name": "Alice",
  "role": "admin",
  "_links": {
    "self": { "href": "/api/v1/users/1" },
    "update": { "href": "/api/v1/users/1", "method": "PUT" },
    "delete": { "href": "/api/v1/users/1", "method": "DELETE" },
    "books": { "href": "/api/v1/users/1/books", "method": "GET" }
  }
}
```

---

3 ข้อดีของ HATEOAS

1. **Client discoverable**
    - Client ไม่ต้อง hardcode URI
    - Server สามารถเปลี่ยน internal URI โดยไม่กระทบ client
2. **ลด coupling ระหว่าง client-server**
    - Client ทำงานตาม hypermedia link
3. **รองรับ complex workflows**
    - เช่น nested resources, action chaining
4. **ทำให้ API self-documenting**
    - Client สามารถดู response แล้วรู้ว่ามี action อะไรได้บ้าง

---

4 ตัวอย่าง Go Web API พร้อม HATEOAS

Handler

```
package handlers

import (
	"encoding/json"
	"net/http"
	"strconv"
	"strings"
)

type User struct {
	ID   int    `json:"id"`
	Name string `json:"name"`
	Role string `json:"role"`
}

type UserResponse struct {
	ID    int                    `json:"id"`
	Name  string                    `json:"name"`
	Role  string                    `json:"role"`
	Links map[string]map[string]string `json:"_links"`
}

var users = []User{
	{ID: 1, Name: "Alice", Role: "admin"},
	{ID: 2, Name: "Bob", Role: "editor"},
}

func UserDetailHandler(w http.ResponseWriter, r *http.Request) {
	idStr := strings.TrimPrefix(r.URL.Path, "/users/")
	id, _ := strconv.Atoi(idStr)

	for _, u := range users {
		if u.ID == id {
```

```
                        resp := UserResponse{
                                ID:    u.ID,
                                Name: u.Name,
                                Role: u.Role,
                                Links: map[string]map[string]string{
                                        "self":   {"href": "/users/" + strconv.Itoa(u.ID)},
                                        "update": {"href": "/users/" + strconv.Itoa(u.ID),
"method": "PUT"},
                                        "delete": {"href": "/users/" + strconv.Itoa(u.ID),
"method": "DELETE"},
                                },
                        }
                        json.NewEncoder(w).Encode(resp)
                        return
                }
        }
        http.Error(w, "User Not Found", http.StatusNotFound)
}
```

ตัวอย่างผลการรัน

```
GET /users/1
{
  "id": 1,
  "name": "Alice",
  "role": "admin",
  "_links": {
    "self": {"href":"/users/1"},
    "update": {"href":"/users/1","method":"PUT"},
    "delete": {"href":"/users/1","method":"DELETE"}
  }
}
```

---

5 การออกแบบ HATEOAS สำหรับ Resource อื่น

- **Books**

```
{
```

```
  "id": 10,
  "title": "Go Web Programming",
  "_links": {
    "self": {"href":"/books/10"},
    "update": {"href":"/books/10","method":"PUT"},
    "delete": {"href":"/books/10","method":"DELETE"},
    "author": {"href":"/users/1","method":"GET"}
  }
}
```


- **Blogs**


```
{
  "id": 5,
  "title": "REST API Design",
  "_links": {
    "self": {"href":"/blogs/5"},
    "update": {"href":"/blogs/5","method":"PUT"},
    "delete": {"href":"/blogs/5","method":"DELETE"},
    "author": {"href":"/users/1","method":"GET"}
  }
}
```

สรุป: HATEOAS ทำให้ **API self-descriptive, client discoverable และลด coupling** ระหว่าง client กับ server

---

ตัวอย่าง **HATEOAS Concept** แบบ **เต็มไฟล์ + โครงสร้าง + คำอธิบาย + ผลการรัน** จำนวน 3 โปรแกรมพื้นฐาน และ 3 โปรแกรมแนวประยุกต์ สำหรับ Go Web Programming

---

โครงสร้างโปรเจกต์

```
hateoas-webapp/
├── main.go
├── go.mod
├── handlers/
│   ├── user.go
│   ├── book.go
```

```
│   └── blog.go
└── models/
    └── models.go
```

1. โปรแกรมพื้นฐาน: User API + HATEOAS

main.go

```
package main

import (
	"log"
	"net/http"

	"hateoas-webapp/handlers"
)

func main() {
	mux := http.NewServeMux()

	// User API
	mux.HandleFunc("/users", handlers.UserHandler)
	mux.HandleFunc("/users/", handlers.UserDetailHandler)

	log.Println("User API with HATEOAS running on :8080")
	log.Fatal(http.ListenAndServe(":8080", mux))
}
```

models/models.go

```
package models

type User struct {
	ID   int    `json:"id"`
	Name string `json:"name"`
	Role string `json:"role"`
}
```

handlers/user.go

```
package handlers

import (
	"encoding/json"
	"net/http"
	"strconv"
	"strings"

	"hateoas-webapp/models"
)

var users = []models.User{
	{ID: 1, Name: "Alice", Role: "admin"},
	{ID: 2, Name: "Bob", Role: "editor"},
}

type UserResponse struct {
	ID    int                    `json:"id"`
	Name  string                 `json:"name"`
	Role  string                 `json:"role"`
	Links map[string]map[string]string `json:"_links"`
}

func UserHandler(w http.ResponseWriter, r *http.Request) {
	switch r.Method {
	case http.MethodGet:
		var resp []UserResponse
		for _, u := range users {
			resp = append(resp, UserResponse{
				ID:   u.ID,
				Name: u.Name,
				Role: u.Role,
				Links: map[string]map[string]string{
					"self": {"href": "/users/" + strconv.Itoa(u.ID)},
```

```
                                        },
                                })
                        }
                        json.NewEncoder(w).Encode(resp)
        case http.MethodPost:
                        var u models.User
                        json.NewDecoder(r.Body).Decode(&u)
                        u.ID = len(users) + 1
                        users = append(users, u)
                        json.NewEncoder(w).Encode(u)
        default:
                        http.Error(w, "Method Not Allowed", http.StatusMethodNotAllowed)
        }
}

func UserDetailHandler(w http.ResponseWriter, r *http.Request) {
        idStr := strings.TrimPrefix(r.URL.Path, "/users/")
        id, _ := strconv.Atoi(idStr)

        for _, u := range users {
                        if u.ID == id {
                                resp := UserResponse{
                                        ID:    u.ID,
                                        Name: u.Name,
                                        Role: u.Role,
                                        Links: map[string]map[string]string{
                                                "self":   {"href": "/users/" + strconv.Itoa(u.ID)},
                                                "update": {"href": "/users/" + strconv.Itoa(u.ID),
"method": "PUT"},
                                                "delete": {"href": "/users/" + strconv.Itoa(u.ID),
"method": "DELETE"},
                                        },
                                }
                                json.NewEncoder(w).Encode(resp)
```

```
                return
            }
        }
        http.Error(w, "User Not Found", http.StatusNotFound)
}
```

ผลการรัน

```
GET /users
[
  {"id":1,"name":"Alice","role":"admin","_links":{"self":{"href":"/users/1"}}},
  {"id":2,"name":"Bob","role":"editor","_links":{"self":{"href":"/users/2"}}}
]

GET /users/1
{
  "id":1,
  "name":"Alice",
  "role":"admin",
  "_links":{
    "self":{"href":"/users/1"},
    "update":{"href":"/users/1","method":"PUT"},
    "delete":{"href":"/users/1","method":"DELETE"}
  }
}
```

---

2 โปรแกรมพื้นฐาน: Book API + HATEOAS

handlers/book.go

```
package handlers

import (
        "encoding/json"
        "net/http"
        "strconv"
        "strings"
```

```
	"hateoas-webapp/models"
)

var books = []models.Book{
	{ID: 1, Title: "Go Programming", Author: "John"},
	{ID: 2, Title: "Web Development", Author: "Mary"},
}

type Book struct {
	ID     int    `json:"id"`
	Title  string `json:"title"`
	Author string `json:"author"`
}

type BookResponse struct {
	ID     int                    `json:"id"`
	Title string                  `json:"title"`
	Author string                   `json:"author"`
	Links map[string]map[string]string `json:"_links"`
}

func BookHandler(w http.ResponseWriter, r *http.Request) {
	switch r.Method {
	case http.MethodGet:
		var resp []BookResponse
		for _, b := range books {
			resp = append(resp, BookResponse{
				ID:    b.ID,
				Title: b.Title,
				Author: b.Author,
				Links: map[string]map[string]string{
					"self": {"href": "/books/" + strconv.Itoa(b.ID)},
				},
			})
```