Web API 講座 Part 3

実践的な機能拡張ハンズオン

48th irom

Part 2 の復習

Part 2 の復習

- REST API の設計原則 (リソース, HTTP メソッド)
- Go 言語と Gin フレームワークによる実装
- レイヤードアーキテクチャ (Handler -> Service -> Repository)
- データ構造の分離 (Model vs DTO)
- SQLite を使った Task 管理 API

1. Schedule モデルの追加 (リレーション)

リレーショナルデータベースの基礎

- テーブル間の関係性
  - 1 対 1
  - 1 対多
  - 多対多
- 外部キー (Foreign Key)
  - 子テーブルが親テーブルを参照するための ID
- GORM のリレーション定義
  - 関連データを効率的に取得する

実装: Schedule モデルの作成

- `internal/model/schedule.go` を作成
- TaskID を外部キーとして持つ

```
type Schedule struct {
  ID        uint      `gorm:"primaryKey" json:"id"`
  TaskID    uint      `gorm:"not null;index" json:"task_id"`
  StartAt   time.Time `gorm:"not null" json:"start_at"`
  EndAt     time.Time `gorm:"not null" json:"end_at"`
  CreatedAt time.Time `gorm:"autoCreateTime" json:"created_at"`
  UpdatedAt time.Time `gorm:"autoUpdateTime" json:"updated_at"`
  DeletedAt gorm.DeletedAt `gorm:"index" json:"deleted_at"`
  Task      Task
`gorm:"constraint:OnUpdate:CASCADE,OnDelete:CASCADE;" json:"-"`
}
```

実装: Task モデルの更新

- internal/model/task.go
  - Task から Schedule への参照を追加 (1 対多)

```
type Task struct {
  // ...existing fields...
  ID          uint           `gorm:"primaryKey" json:"id"`
  Title       string         `gorm:"type:varchar(255);not null"
json:"title"`
  Description string         `gorm:"type:text" json:"description"`
  Completed   bool           `gorm:"not null;default:false"
json:"completed"`
  // ...timestamps...

  // 1対多のリレーション
  Schedules   []Schedule     `json:"schedules,omitempty"`
}
```

Schedule 用の DTO 作成

- internal/dto/schedule.go

```
type CreateScheduleRequest struct {
  TaskID  uint      `json:"task_id" binding:"required"`
  StartAt time.Time `json:"start_at" binding:"required"`
  EndAt   time.Time `json:"end_at" binding:"required"`
}

type UpdateScheduleRequest struct {
  StartAt *time.Time `json:"start_at"`
  EndAt   *time.Time `json:"end_at"`
}

type ScheduleResponse struct {
  ID      uint      `json:"id"`
  TaskID  uint      `json:"task_id"`
  StartAt time.Time `json:"start_at"`
```

Schedule 用の DTO 作成

```
  EndAt   time.Time `json:"end_at"`
}
```

Schedule 用の Repository 実装

- internal/repository/schedule.go

```
type ScheduleRepository interface {
  Create(schedule *model.Schedule) error
  FindByID(id uint) (*model.Schedule, error)
  FindByTaskID(taskID uint) ([]model.Schedule, error)
  Update(schedule *model.Schedule) error
  Delete(schedule *model.Schedule) error
  List() ([]model.Schedule, error)
}
```

Schedule 用の Service 実装

- internal/service/schedule.go
  - タスクの存在確認を行う

```
func (s *scheduleService) CreateSchedule(
  req *dto.CreateScheduleRequest) (*dto.ScheduleResponse, error) {
  // タスクの存在確認
  _, err := s.taskRepo.FindByID(req.TaskID)
  if err != nil {
    if errors.Is(err, gorm.ErrRecordNotFound) {
      return nil, ErrTaskNotFound
    }
    return nil, err
  }

  schedule := req.ToModel()
  if err := s.repo.Create(schedule); err != nil {
    return nil, err
  }
```

Schedule 用の Service 実装

```
  return dto.FromScheduleModel(schedule), nil
}
```

Schedule 用の Handler 実装

- internal/handler/schedule.go

```
func (h *ScheduleHandler) CreateSchedule(c *gin.Context) {
  var req dto.CreateScheduleRequest
  if err := c.ShouldBindJSON(&req); err != nil {
    c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
    return
  }

  schedule, err := h.service.CreateSchedule(&req)
  if err != nil {
    if errors.Is(err, service.ErrTaskNotFound) {
      c.JSON(http.StatusNotFound, gin.H{"error": "Task not
found"})
    } else {
      c.JSON(http.StatusInternalServerError, gin.H{"error":
err.Error()})
    }
```

Schedule 用の Handler 実装

```
    return
  }
  c.JSON(http.StatusCreated, schedule)
}
```

2. テストコードの実装

テストの種類

- ユニットテスト (単体テスト)
  - 関数やメソッド単位での検証
  - 外部依存 (DB など) はモック化する
- 統合テスト (結合テスト)
  - 複数のモジュールを連携させた検証

モックと DI

- 依存性注入 (DI) の恩恵
- Repository の実装をモックに差し替えることで、DB なしで Service 層のロジックをテスト可能
- GoMock (go.uber.org/mock) を使用

```
go install go.uber.org/mock/mockgen@latest
```

モックの生成

- Repository 層のモック

```
mockgen -source=internal/repository/task.go \
  -destination=internal/repository/mock_task.go \
  -package=repository
```

- Service 層のモック

```
mockgen -source=internal/service/task.go \
  -destination=internal/service/mock_task.go \
  -package=service
```

Service 層のユニットテスト例

- internal/service/task_test.go

```
func TestCreateTask(t *testing.T) {
  ctrl := gomock.NewController(t)
  mockRepo := repository.NewMockTaskRepository(ctrl)

  // Createが呼ばれたらnilを返す
  mockRepo.EXPECT().
    Create(gomock.Any()).
    Return(nil)

  service := NewTaskService(mockRepo)
  req := &dto.CreateTaskRequest{
    Title:       "Test Task",
    Description: "This is a test task",
  }

  res, err := service.CreateTask(req)
```

Service 層のユニットテスト例

```
  assert.NoError(t, err)
  assert.Equal(t, req.Title, res.Title)
}
```

Handler 層の統合テスト例

- internal/handler/task_test.go

```
func TestCreateTask(t *testing.T) {
  ctrl := gomock.NewController(t)
  defer ctrl.Finish()

  mockService := service.NewMockTaskService(ctrl)
  h := NewTaskHandler(mockService)

  expectedResponse := &dto.TaskResponse{
    ID: 1, Title: "New Task",
  }
  mockService.EXPECT().
    CreateTask(gomock.Any()).
    Return(expectedResponse, nil)

  gin.SetMode(gin.TestMode)
  r := gin.Default()
```

Handler 層の統合テスト例

```
  r.POST("/tasks", h.CreateTask)

  // ...リクエスト実行と検証...
}
```

テストの実行

```
go test -v ./...
go test -v ./internal/service
go test -v ./internal/handler
```

3. 認証・認可の実装

ステートレス認証と JWT

- ステートレス認証
  - サーバー側でセッション情報を保持しない
- JSON Web Token (JWT)
  - クライアントにトークンを発行し、各リクエストで送信
  - トークンにユーザー情報や権限を含める
  - 構成: ヘッダー、ペイロード、署名

User モデルの作成

- internal/model/user.go

```
type User struct {
  ID        uint           `gorm:"primaryKey" json:"id"`
  Username  string         `gorm:"unique;not null"
json:"username"`
  Password  string         `gorm:"not null" json:"-"` // JSONには含
めない
  CreatedAt time.Time      `json:"created_at"`
  UpdatedAt time.Time      `json:"updated_at"`
  DeletedAt gorm.DeletedAt `gorm:"index" json:"-"`
}
```

AuthService の実装

- internal/service/auth.go

```
func (s *authService) Register(username, password string) error {
  hashedPassword, err := bcrypt.GenerateFromPassword(
    []byte(password), bcrypt.DefaultCost)
  if err != nil { return err }

  user := model.User{
    Username: username,
    Password: string(hashedPassword),
  }
  return s.db.Create(&user).Error
}

func (s *authService) Login(username, password string) (string,
error) {
  // ...ユーザー検証とJWTトークン生成...
}
```

JWT トークンの生成

- internal/service/auth.go

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

func (s *authService) Login(username, password string) (string,
error) {
  // ...パスワード検証...

  token := jwt.NewWithClaims(jwt.SigningMethodHS256,
jwt.MapClaims{
    "sub": user.ID,
    "exp": time.Now().Add(time.Hour * 24).Unix(),
  })

  tokenString, err := token.SignedString(jwtSecretKey)
  if err != nil { return "", err }
  return tokenString, nil
}
```

Middleware の実装

- internal/middleware/auth.go

```
func AuthMiddleware() gin.HandlerFunc {
  return func(c *gin.Context) {
    authHeader := c.GetHeader("Authorization")
    if authHeader == "" {
      c.AbortWithStatusJSON(401, gin.H{"error": "Authorization
header required"})
      return
    }

    tokenString := strings.TrimPrefix(authHeader, "Bearer ")
    token, err := jwt.Parse(tokenString, func(token *jwt.Token)
(interface{}, error) {
      return jwtSecretKey, nil
    })

    if err != nil || !token.Valid {
```

Middleware の実装

```
      c.AbortWithStatusJSON(401, gin.H{"error": "Invalid token"})
      return
    }

    c.Next()
  }
}
```

ルーティングへの適用

- cmd/api/main.go

```
r := gin.Default()

// 公開API
r.POST("/register", authHandler.Register)
r.POST("/login", authHandler.Login)
r.POST("/tasks", taskHandler.CreateTask)
// ...その他のTaskエンドポイント...

// 認証必須API (Schedule)
authGroup := r.Group("/schedules")
authGroup.Use(middleware.AuthMiddleware())
{
  authGroup.POST("/", scheduleHandler.CreateSchedule)
  authGroup.GET("/:id", scheduleHandler.GetSchedule)
  authGroup.GET("/tasks/:taskId/schedules",
scheduleHandler.GetSchedulesByTask)
```

ルーティングへの適用

```
  authGroup.PUT("/:id", scheduleHandler.UpdateSchedule)
  authGroup.DELETE("/:id", scheduleHandler.DeleteSchedule)
  authGroup.GET("/", scheduleHandler.ListSchedules)
}
```

ユーザー登録とログインの流れ

1. ユーザー登録

```
curl -X POST http://localhost:8080/register \
  -H "Content-Type: application/json" \
  -d '{"username":"testuser","password":"password123"}'
```

2. ログイン（トークン取得）

```
curl -X POST http://localhost:8080/login \
  -H "Content-Type: application/json" \
  -d '{"username":"testuser","password":"password123"}'
# レスポンス: {"token":"eyJhbGci..."}
```

3. 認証が必要な API へのアクセス

```
curl http://localhost:8080/schedules/ \
  -H "Authorization: Bearer eyJhbGci..."
```

4. DB 永続化(Docker Compose)

SQLite から PostgreSQL への移行

Part 2 では SQLite を使用していたが、Part 3 では PostgreSQL に変更

理由:

- 本番環境での利用を想定
- 複数コンテナからの同時アクセスに対応
- より高度な機能(外部キー制約など)

Docker Compose の設定

- docker-compose.yml

```yaml
services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    ports: ["8080:8080"]
    environment:
      - DB_HOST=db
      - DB_PORT=5432
      - DB_USER=user
      - DB_PASSWORD=password
      - DB_NAME=app_db
    depends_on:
      db:
        condition: service_healthy
    restart: on-failure
```

PostgreSQL コンテナの設定

```yaml
  db:
    image: postgres:15
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: app_db
    ports: ["5432:5432"]
    volumes:
      - db_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U user -d app_db"]
      interval: 5s
      timeout: 5s
      retries: 5

volumes:
  db_data:
```

DB 接続の変更

- cmd/api/main.go

```
import (
  "gorm.io/driver/postgres"  // SQLiteから変更
)

func main() {
  dbHost := getEnv("DB_HOST", "localhost")
  dbPort := getEnv("DB_PORT", "5432")
  dbUser := getEnv("DB_USER", "user")
  dbPassword := getEnv("DB_PASSWORD", "password")
  dbName := getEnv("DB_NAME", "app_db")

  dsn := fmt.Sprintf("host=%s port=%s user=%s password=%s
dbname=%s sslmode=disable",
    dbHost, dbPort, dbUser, dbPassword, dbName)

  db, err := gorm.Open(postgres.Open(dsn), &gorm.Config{})
```

DB 接続の変更

```
  // ...
}
```

リトライロジックの追加

- データベース接続の堅牢性向上

```
var db *gorm.DB
var err error
maxRetries := 5
for i := 0; i < maxRetries; i++ {
  db, err = gorm.Open(postgres.Open(dsn), &gorm.Config{})
  if err == nil {
    log.Println("Successfully connected to database")
    break
  }
  log.Printf("Failed to connect (attempt %d/%d): %v", i+1,
maxRetries, err)
  time.Sleep(time.Second * 2)
}
if err != nil {
  log.Fatal("failed to connect database after retries:", err)
}
```

マイグレーションの更新

```
// Task, Schedule, Userの3モデルをマイグレーション
if err := db.AutoMigrate(&model.Task{}, &model.Schedule{},
&model.User{}); err != nil {
  log.Fatal("failed to migrate database:", err)
}
```

Docker Compose の実行

```
# ビルドして起動
docker-compose up --build

# バックグラウンド実行
docker-compose up -d

# ログ確認
docker-compose logs -f app

# 停止・削除
docker-compose down

# ボリュームも含めて削除
docker-compose down -v
```

5. 動作確認

全体の流れ

1. Docker Compose で起動
2. ユーザー登録
3. ログイン（トークン取得）
4. Task CRUD 操作（認証不要）
5. Schedule CRUD 操作（認証必須）

Step 1: 起動

```
docker-compose up --build
```

Step 2: ユーザー登録

```
curl -X POST http://localhost:8080/register \
  -H "Content-Type: application/json" \
  -d '{"username":"testuser","password":"password123"}'
```

Step 3: ログイン

```
curl -X POST http://localhost:8080/login \
  -H "Content-Type: application/json" \
  -d '{"username":"testuser","password":"password123"}'

# レスポンス例
{"token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."}

# トークンを環境変数に保存
export TOKEN="eyJhbGci..."
```

Step 4: Task 作成

```
curl -X POST http://localhost:8080/tasks \
  -H "Content-Type: application/json" \
  -d '{"title":"スライド作成1","description":"API講座①のスライドを作成
する"}'

curl -X POST http://localhost:8080/tasks \
  -H "Content-Type: application/json" \
  -d '{"title":"スライド作成2","description":"API講座②のスライドを作成
する"}'
```

Step 5: Schedule 作成（認証必須）

```
curl -X POST http://localhost:8080/schedules/ \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $TOKEN" \
  -d '{
    "task_id": 1,
    "start_at": "2025-01-20T10:00:00Z",
    "end_at": "2025-01-20T12:00:00Z"
  }'
```

Schedule 一覧取得

```
curl http://localhost:8080/schedules/ \
  -H "Authorization: Bearer $TOKEN"
```

特定 Task の Schedule 取得

```
curl http://localhost:8080/schedules/tasks/1/schedules \
  -H "Authorization: Bearer $TOKEN"
```

認証エラーのテスト

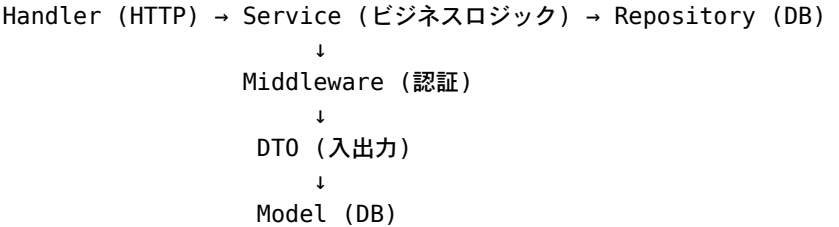
```
# トークンなし（401エラー）
curl http://localhost:8080/schedules/

# 無効なトークン（401エラー）
curl http://localhost:8080/schedules/ \
  -H "Authorization: Bearer invalid_token"
```

まとめ

Part 3 で実装した機能

1. Schedule モデルの追加とリレーション
   - Task (1) - Schedule (多) の関係
2. テストコードの実装
   - GoMock を使ったユニットテスト
3. 認証・認可の実装
   - JWT + Middleware
4. PostgreSQL への移行
   - Docker Compose による環境構築

アーキテクチャの完成形



学んだこと

- RESTful API の実践的な設計
- レイヤードアーキテクチャによる責務の分離
- テスタビリティを考慮した実装(DI、モック)
- 認証・認可の実装パターン
- Docker を使った開発環境の構築
- リレーショナルデータベースの扱い方

発展的な学習

- OpenAPI/Swagger による API ドキュメント生成
- CI/CD パイプラインの構築
- ロギング・モニタリング
- エラーハンドリングの統一
- バリデーションの強化
- ページネーション・フィルタリング
- キャッシュ戦略
- レート制限

参考資料

- Go 公式ドキュメント: https://go.dev/doc/
- Gin フレームワーク: https://gin-gonic.com/
- GORM: https://gorm.io/
- JWT: https://jwt.io/
- Docker Compose: https://docs.docker.com/compose/