Web API 講座

48th irom

HTTP の基礎

HTTP とは

- HyperText Transfer Protocol の略
- Web ブラウザと Web サーバーが通信するためのルール
- クライアント（リクエスト）とサーバー（レスポンス）の間でデータのやり取りを行う

HTTP リクエストとレスポンス

- リクエスト: クライアントからサーバーへの要求
  - メソッド、URL、ヘッダー、ボディで構成される

```
POST /api/v1/tasks HTTP/1.1
Host: api.example.com
Content-Type: application/json
{"title": "Buy milk", "description": "2L whole milk" }
```

- レスポンス: サーバーからクライアントへの応答
  - ステータスコード、ヘッダー、ボディで構成される

```
HTTP/1.1 200 OK
Content-Type: application/json
{"title": "Buy milk", "description": "2L whole milk", "done":
false }
```

HTTP ヘッダー

- リクエストやレスポンスに関する付加情報（メタデータ）
- キーと値のペアで記述される
- 例:
  - `Content-Type`: データの形式（`application/json` など）
  - `Authorization`: 認証情報

```
Content-Type: application/json
Authorization: Bearer <token>
```

HTTP ボディと JSON

- ボディ: 実際に送受信されるデータ本体
- JSON (JavaScript Object Notation):
  - Web API で最も一般的に使われるデータ形式
  - 軽量で人間にも読みやすいテキスト形式
  - キーと値のペアでデータを表現する

```
{
  "title": "Buy milk",
  "description": "2L whole milk",
  "done": false
}
```

REST

REST API

- REpresentational State Transfer の略
- Web システムを設計するためのアーキテクチャスタイル
- 分散システムにおいて、効率的でスケーラブルな通信を実現するための指針

RESTful なシステムの制約条件

- クライアントとサーバーの分離
- ステートレス性
- キャッシュ可能性
- 階層化システム
- 統一インターフェース

クライアントとサーバーの分離

- UI（クライアント）とデータ保存（サーバー）の関心を分離する
- 互いに独立して進化・開発できるようになる
- マルチプラットフォーム対応が容易になる
- クライアントはサーバーの内部構造を知らなくてもよい
- API を通じてデータにアクセスする

ステートレス性

- サーバーはクライアントの状態（セッション状態など）を保存しない
- すべてのリクエストは、それだけで処理が完結するよう必要な情報をすべて含める必要がある
- スケーラビリティ（拡張性）が向上する
- クライアント側で状態管理を行う必要がある

キャッシュ可能性

- レスポンスはキャッシュ可能かどうかを明示する
- クライアントがデータを再利用することで、通信量を減らし、レスポンス速度を向上させる
- HTTP ヘッダーの `Cache-Control` や `ETag` を利用する
- 適切なキャッシュ戦略を設計することが重要

階層化システム

- クライアントはサーバーに直接接続しているか、中間のプロキシ等に接続しているか意識しない
- ロードバランサなどを挟むことで負荷分散やセキュリティ向上が可能
- システムの複雑性を隠蔽できる
- クライアントは単一のエンドポイントに対してリクエストを送るだけでよい
- 中間サーバーがレスポンスをキャッシュすることも可能

統一インターフェース

- 構成要素間のインターフェースを統一する
- URI の形式や HTTP メソッドの使い方を標準化することで、システム全体がシンプルになる
- クライアントとサーバーの独立性が高まる

URI 設計とパラメータ

リソース指向アーキテクチャ

- Web 上のすべての情報を「リソース」として扱う
- 各リソースは一意の URI で識別される
- リソースに対する操作は HTTP メソッドで表現する
- 例: `GET /tasks` (タスク一覧の取得), `POST /tasks` (新規タスクの作成)
- リソースの状態は表現（Representation）としてやり取りされる（通常は JSON 形式）

リソース(URI)設計

- リソースは「名詞」で表現する（動詞は使わない）
- 複数形を使うのが一般的
- 階層構造で関係性を表す
- 例: `GET /tasks` (タスク一覧), `GET /tasks/1` (タスク詳細)
- GET /tasks
- POST /tasks
- GET /tasks/1

パスパラメータ

- リソースを一意に特定するために URI の一部として埋め込む値
- 例: `/tasks/1` の `1`
- 特定のデータを取得・更新・削除する場合に使用する

```
// Go (routing)
r := gin.Default()
v1 := r.Group("/api/v1")
v1.GET("/tasks/:id", getTaskHandler)
```

クエリパラメータ

- リソースに対する操作の条件を指定するために使う
- URI の末尾に `?key=value` の形式で付与
- フィルタリング、ソート、ページネーションなどに使用
- 例: `/tasks?status=done&limit=10`

```
func listTasksHandler(c *gin.Context) {
  status := c.Query("status") // "done"
  limit := c.DefaultQuery("limit", "20")
  // ...処理...
}
```

HTTP メソッドと CRUD 操作

HTTP メソッド

- クライアントがサーバーに対して行う操作の種類を指定する
- 主なメソッド:
  - GET: データの取得
  - POST: データの新規作成
  - PUT: データの更新（全体置換）
  - PATCH: データの部分更新
  - DELETE: データの削除

GET

- リソースの 取得 (Read)
- サーバーのデータを変更しない
- 例: タスク一覧の取得、タスク詳細の取得

```
curl -s https://api.example.com/api/v1/tasks/1
```

POST

- リソースの 新規作成 (Create)
- ボディに作成するデータを含めて送信する
- 例: 新しいタスクの登録

```
curl -X POST https://api.example.com/api/v1/tasks \
  -H "Content-Type: application/json" \
  -d '{"title":"New task"}'
```

PUT / PATCH

- リソースの 更新 (Update)
- PUT: リソース全体を置き換える
- PATCH: リソースの一部を変更する
- 例: タスクの内容変更、完了状態の更新

```
curl -X PATCH https://api.example.com/api/v1/tasks/1 \
  -H "Content-Type: application/json" \
  -d '{"done":true}'
```

DELETE

- リソースの 削除 (Delete)
- 指定されたリソースを削除する
- 例: タスクの削除

```
curl -X DELETE https://api.example.com/api/v1/tasks/1
```

CRUD 操作と HTTP メソッドの対応

- CRUD 操作とは、データの基本的な操作を指す
- Create (作成): POST
- Read (読み取り): GET
- Update (更新): PUT / PATCH
- Delete (削除): DELETE

ステータスコード

- 処理結果を表す 3 桁の数字
- 2xx (成功): `200 OK`, `201 Created`
- 4xx (クライアントエラー): `400 Bad Request`, `401 Unauthorized`, `404 Not Found`
- 5xx (サーバーエラー): `500 Internal Server Error`
- POST -> 201 Created + Location ヘッダー

アーキテクチャと設計

責務の分離

- コードを役割ごとに分割して管理する
- メリット:
  - 可読性の向上
  - テストの容易性
  - 変更の影響範囲を限定できる

レイヤードアーキテクチャ

- レイヤードアーキテクチャは、ソフトウェアを**複数の層（レイヤー）**に**分割する設計手法**
- 各レイヤーは**特定の責務**を持ち、他のレイヤーと**明確に分離されている**
- **主**なレイヤー:
  - Handler (Controller): HTTP リクエストの受付
  - Service (UseCase): ビジネスロジック
  - Repository: データアクセス

Handler

- 役割: HTTP 通信の窓口
- リクエストパラメータの受け取りとバリデーション
- Service 層の呼び出し
- レスポンス（JSON など）の返却
- ビジネスロジックは書かない

Service

- 役割: ビジネスロジックの実装
- 「アプリケーションが何をするか」を記述
- データの加工、計算、複数リポジトリの操作など
- 特定の Web フレームワークに依存させないのが理想

Repository

- 役割: データの永続化
- データベース(DB)へのアクセスを担当
- SQL の実行や ORM の操作
- 保存先が DB でもファイルでも、Service 層への影響を最小限にする

データ構造

- アプリケーション内で使用するデータの形を定義する
- Model と DTO の 2 種類が一般的
  - Model (Entity): データベースの構造を表す
  - DTO (Data Transfer Object): API の入出力専用のデータ構造
- 内部のデータ構造と API のインターフェースを分離するために使い分ける

Model

- DB のテーブル定義と 1 対 1 で対応する構造体
- Gorm などの ORM で使用するタグを含むことが多い
- データベースの都合（外部キーなど）が含まれる

DTO

- クライアントからのリクエスト受け取りや、レスポンス返却に使う
- 必要なデータのみを定義する
- バリデーションタグなどを付与する
- クライアントに見せたくない情報（パスワードハッシュなど）を除外できる

Go と Gin による実装

Go 言語

- Google が開発したプログラミング言語
- 静的型付け、コンパイル言語、高速な実行速度
- 並行処理（Goroutine）が強力
- シンプルな文法

Gin フレームワーク

- Go 言語用の高速な HTTP Web フレームワーク
- ルーティング、ミドルウェア、JSON バリデーションなどの機能を提供
- パフォーマンスが高く、API 開発によく使われる

Gin の基本フロー

1. ルーターの作成 (`gin.Default()`)
2. ミドルウェアの設定(CORS など)
3. ルーティングの定義 (URL とハンドラの紐付け)
4. サーバーの起動 (`r.Run()`)

```
r := gin.Default()
r.GET("/ping", func(c *gin.Context) {
  c.JSON(200, gin.H{
    "message": "pong",
  })
})
r.Run() // デフォルトで:8080で起動
```

ルーティングとグループ化

- HTTP メソッドと URL パスに対して、実行する関数を指定する

```
r.GET("/tasks", listTasksHandler)
r.POST("/tasks", createTaskHandler)
```

- グループ化: 共通のパスプレフィックスやミドルウェアをまとめる

```
v1 := r.Group("/api/v1")
v1.GET("/tasks", listTasksHandler)
v1.POST("/tasks", createTaskHandler)
```

リクエストのバインディングとバリデーション

- `ShouldBindJSON`: リクエストボディの JSON を Go の構造体に変換

```
type CreateTaskRequest struct {
  Title       string `json:"title" binding:"required"`
  Description string `json:"description"`
}
```

- 構造体のタグ (`binding:"required"` 等) で入力チェックを自動化
- 型が違う場合や必須項目がない場合はエラーになる

```
var req CreateTaskRequest
if err := c.ShouldBindJSON(&req); err != nil {
  c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
  return
}
```

レスポンスのエラーハンドリング

- エラーが発生した場合、適切なステータスコードを返す

```
if err != nil {
  c.JSON(http.StatusInternalServerError, gin.H{"error": "Internal
Server Error"})
  return
}
```

- エラーレスポンスの形式を統一しておくと、クライアント側が扱いやすい

Task 管理 API の実装例

プロジェクト構成

- 役割ごとにディレクトリ分割
- テストや差し替えが容易に

```
cmd/api/main.go
internal/handler
internal/service
internal/repository
internal/model
internal/dto
```

model (DB の単位)

- DB テーブルと 1 対 1 で対応する構造体
- GORM タグでマイグレーション可能

```
package model

// Task: DBのエンティティ
type Task struct {
  // ...existing fields...
  ID        uint   `gorm:"primaryKey" json:"id"`
  Title     string `gorm:"type:varchar(255);not null"
json:"title"`
  Completed bool   `gorm:"not null;default:false"
json:"completed"`
  // ...existing fields...
}
```

repository (データ永続化)

- DB アクセスを集約し、interface で抽象化
- テスト時はモック実装と差し替え可能

```
package repository

import "part2/internal/model"

type TaskRepository interface {
  Create(task *model.Task) error
  FindByID(id uint) (*model.Task, error)
  // ...other methods...
}

// NewTaskRepository は gorm.DB を受け取り実装を返す
```

dto (API 入出力)

- リクエスト/レスポンス専用構造体で内部 Model と分離
- バリデーションタグを持たせる

```
package dto

type CreateTaskRequest struct {
  Title string `json:"title" binding:"required"`
  // ...other fields...
}

func (r *CreateTaskRequest) ToModel() *model.Task {
  // ...map to model...
}
```

service (ビジネスロジック)

- リポジトリを使ってアプリケーションの振る舞いを実装
- エラー定義やトランザクション制御をここに置く

```
package service

// TaskService: UseCaseを表すインターフェース
type TaskService interface {
  CreateTask(req *dto.CreateTaskRequest) (*dto.TaskResponse,
error)
  // ...other methods...
}
```

handler (HTTP 層)

- リクエストのバインドとレスポンス整形を担当
- Service を呼び出すだけに留める

```
package handler

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

func (h *TaskHandler) CreateTask(c *gin.Context) {
  var req dto.CreateTaskRequest
  if err := c.ShouldBindJSON(&req); err != nil {
    c.JSON(400, gin.H{"error": err.Error()})
    return
  }
  // ...call service and respond...
}
```

main.go (起動・ルーティング)

- 依存関係の組み立てとルート定義

```
package main

func main() {
  // ...Open DB, AutoMigrate...
  taskRepo := repository.NewTaskRepository(db)
  taskService := service.NewTaskService(taskRepo)
  taskHandler := handler.NewTaskHandler(taskService)

  r := gin.Default()
  r.POST("/tasks", taskHandler.CreateTask)
  // ...other routes...
  r.Run(":8080")
}
```

依存性注入 (DI)

- コンストラクションで依存を注入することでテスト容易性を確保
- メリット:
  - モジュールの独立性向上
  - テスト時にモックと差し替え可能
- 例: main.go で repository→service→handler の順に組み立てる

```
taskRepo := repository.NewTaskRepository(db)
taskService := service.NewTaskService(taskRepo)
taskHandler := handler.NewTaskHandler(taskService)
```

動作確認

go init と依存関係のインストール

- Go モジュールの初期化
- 依存関係のインストール
- go.mod と go.sum が生成される

```
cd part2
go mod init part2
go mod tidy
```

Docker での起動

- Docker イメージのビルド

```
docker build -t go-app .
```

- Docker コンテナの実行 / 停止

```
docker run -p 8080:8080 go-app
docker ps
docker stop <コンテナID>
```

動作確認: 新しいタスクの作成

- 新しいタスクの作成例（複数を連続で作成）

```
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"}'

curl -X POST http://localhost:8080/tasks \
  -H "Content-Type: application/json" \
  -d '{"title": "スライド作成3", "description": "API講座③のスライドを作成する"}'
```

動作確認: タスク一覧の取得

- タスク一覧を取得する

```
curl http://localhost:8080/tasks
```

- ID 指定で詳細取得

```
curl http://localhost:8080/tasks/1
```

動作確認: タスクの更新・削除

- 更新

```
curl -X PUT http://localhost:8080/tasks/1 \
  -H "Content-Type: application/json" \
  -d '{"title": "スライド作成1 (完了)", "completed": true}'
```

- 削除

```
curl -X DELETE http://localhost:8080/tasks/3
```

- 最終的な一覧を再取得して変更を確認

```
curl http://localhost:8080/tasks
```

まとめ

まとめ

- HTTP と REST の基本概念
- URI 設計と HTTP メソッドの使い分け
- レイヤードアーキテクチャで責務の分離
- Go 言語と Gin フレームワークで API 実装

次回予告

- Schedule モデルの追加
  - Task との関連付け
- テストコードの実装
  - ユニットテストと統合テスト
- 認証・認可の実装
  - JWT トークンの利用
- データベースの永続化
  - Docker Compose で PostgreSQL を利用
- Swagger ドキュメントの生成
  - API 仕様の自動生成