

実装しながら学ぶ！HTTP サーバー実装の基本

Takuma Kobayashi (@takuma5884rbb)

株式会社 Finatext

目次

- 自己紹介
- 今日学んで欲しいこと
- リクエストを受け取る
- 異常系を考える
- 認証
- まとめ

自己紹介

- 小林拓磨
 - X: [@takuma5884rbb](#)
- 2000 年生まれの 23 卒
- Software Engineer at [Finatext](#)
 - 1 年目からシステムの詳細設計・実装・運用を経験
 - 主な技術スタック
 - Go, AWS, Terraform
- [2024 Japan AWS Jr.Champions](#)
- 趣味は料理・マラソン



アイスブレイク

皆さん、プログラミングするときに意識していることは何ですか？

今日の目的

- HTTP サーバーの実装を通して、実践的な開発手法について学ぶ
 - "ただ動くだけのシステム"を脱却する
- 単なる技術解説ではなく、**なぜ** その技術が必要なのかを理解する
- 実装を通して、実際の開発現場でも役立つ知識とスキルを身につける

今日学んで欲しいこと

- 関数の責務を捉える重要性
 - 一つの関数は一つの責任を持つべき
 - コードの可読性と保守性の向上
- リクエスト処理とバリデーション
 - 不正なデータからシステムを守る方法
 - ユーザー入力を常に疑う姿勢
- 認証の基本と実装方法
 - セキュリティの重要性
 - JWT を使った認証の仕組み

HTTP サーバーの処理を分割してみよう

実際のコード例を見ながら、HTTP サーバーの基本的な処理の流れを理解していきましょう。まずは単純な例から始めて、徐々に実践的な実装へと進めていきます。

Hello, World! を返す HTTP サーバーの実装例

```
package main

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

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello, World!"))
    })
    log.Fatal(http.ListenAndServe(":8080", mux))
}
```

```
$ curl http://localhost:8080/
Hello, World!%
```

クエリパラメータで名前を受け取り、その名前に対応するメールアドレスを返す HTTP サーバーの実装例

```
mux.HandleFunc("/profile", func(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query().Get("name")

    email := fmt.Sprintf("%s@example.com", name) // 実際はデータベースなどから取得する

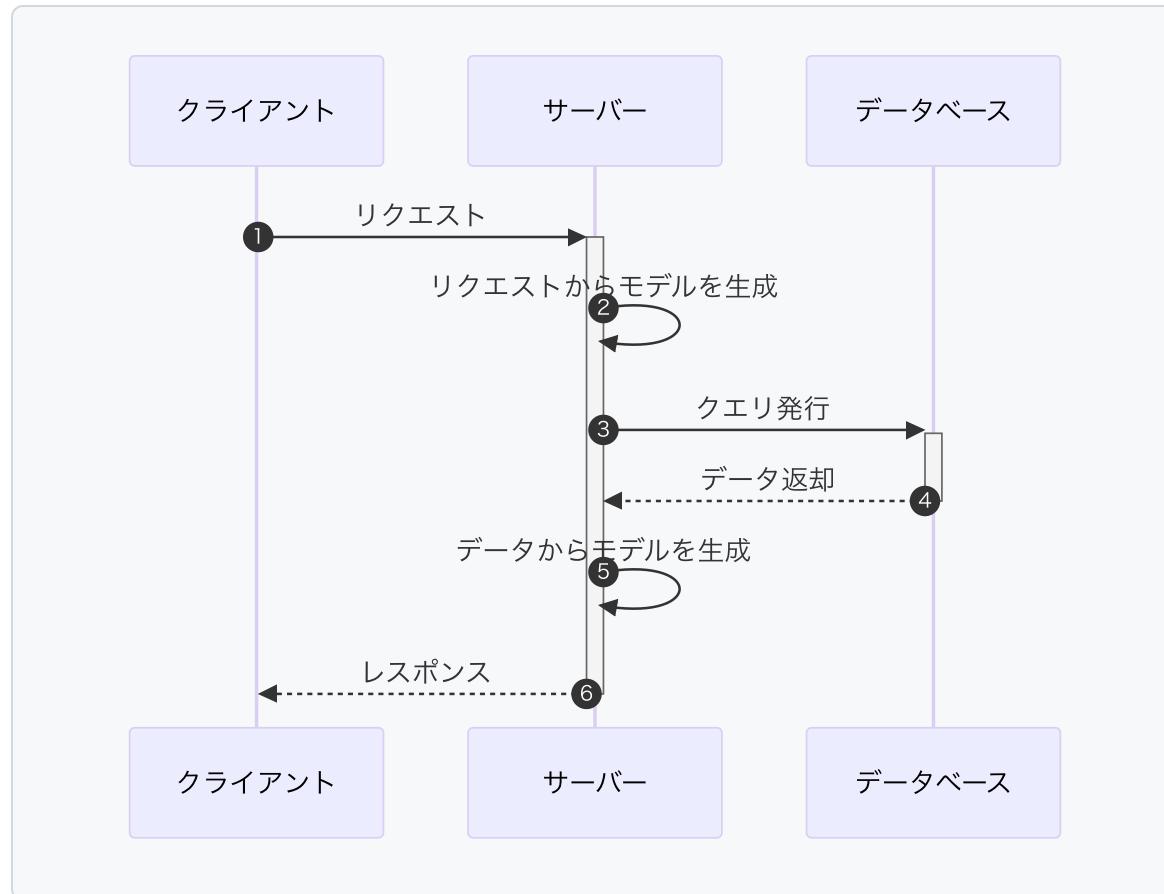
    w.Write([]byte(email))
})
```

```
$ curl "http://localhost:8080/profile?name=hoge"
hoge@example.com%
```

結局、HTTP サーバーとは何をするシステム？ 🤔

上記の例を見ると、HTTP サーバーは単にリクエストを受け取り、レスポンスを返すだけのシステムのように見えます。しかし、実際にはもっと複雑な処理が必要です。HTTP サーバーの役割を抽象化して考えてみましょう。

抽象化された Web サーバーの処理



1. リクエストを受け取る
2. サーバーのデータモデルに読み替える
3. 出来たモデルに対してクエリを組み立て、発行する
4. クエリの結果を受け取り、モデルに変換する
5. モデルからレスポンスを生成する

分けられた処理をグルーピングしてみる

1. リクエストを受け取る ①
2. サーバーのデータモデルに読み替える ②
3. 出来たモデルに対してクエリを組み立て、発行する ③
4. クエリの結果を受け取り、モデルに変換する ②
5. モデルからレスポンスを生成する ①

グループ

クライアントとの IF 処理

クライアントとの IF とアプリケーション内データモデルの変換

アプリケーション内データモデルと DB とのやりとり

ということで、それぞれの責務を考えてみましょう！

各処理グループの責務を明確にすることで、コードの構造化と保守性の向上につながります。次のセクションでは、関数の責務について詳しく見ていきます。

関数の責務を捉える

関数の責務とは、その関数が担うべき役割や責任のことです。一つの関数は一つの責任を持つべきという「单一責任の原則(SOLID 原則)」は、ソフトウェア設計の基本原則の一つです。

責務...

アーキテクチャの話...? 🤔

The Clean Architecture

クリーンアーキテクチャとは誰々が提唱した概念で～
...という話は今日はしません！

今回は特定のアーキテクチャパターンについて深く掘り下げるのではなく、実際のコードで責務の分離がどのように役立つのかを見ていきます。

画像：<https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

なぜ関数の責務に着目するのか？

- その関数の関心ごとを明確にする
 - 設計を変える場合、依存する部分が最小限になる
 - 同じユースケースにおいて再利用できる
 - テストも簡単

責務を明確にすることで、コードの変更が必要になった時の影響範囲を最小限に抑えることができます。また、機能の再利用性が高まり、テストも書きやすくなります。

関数の関心ごとを明確にする

```
// DBに対してクエリを発行し、プロフィールを取得する
func (r *userRepository) GetProfile(name string) (*Profile, error) {
    return r.db.Query("SELECT * FROM profiles WHERE name = ?", name)
}
```

```
// リクエストからユーザ名を取得し、対応するプロフィールを返す
func (h *handler)GetProfileHandler(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query().Get("name")

    profile, err := h.UserRepository.GetProfile(name)
    if err != nil {
        http.Error(w, "Profile not found", http.StatusNotFound)
        return
    }

    w.Write([]byte(profile))
}
```

設計を変える場合、依存する部分が最小限になる

処理が関数に分かれていない場合

```
mux.HandleFunc("/profile", func(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query().Get("name")

    profile := db.Query("SELECT * FROM profiles WHERE name = ?", name)

    w.Write([]byte(profile))
})
```

```
mux.HandleFunc("/article", func(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query().Get("name")

    profile := db.Query("SELECT * FROM profiles WHERE name = ?", name) // 実装が重複している
    article := db.Query("SELECT * FROM articles WHERE author = ?", name)

    w.Write([]byte(profile + article))
})
```

この例では、プロフィール取得のロジックが複数の場所で重複しています。これは保守性の観点から問題があります。

ユーザー名からプロフィールを取得する処理が重複している
ではプロフィールの項目が増えた場合はどうなる?
テーブルの構造が変わった場合は?
etc.

これらの場合、重複している全ての箇所を修正する必要があります。これは保守性
を低下させ、バグの原因となります。

関数の関心ごとを明確にするで紹介したサンプルコードにおいては、

- API のリクエストパラメータはレスポンス形式を変更したい場合は `GetProfileHandler` を
- DB のクエリを変更したい場合は `GetProfile` を

変更すれば良い！

責務が明確に分離されていれば、変更の影響範囲が限定され、コードの保守性が向上します。これは実際のプロジェクトでは非常に重要な利点です。

レイヤー分けによる責務の分離

このプロジェクトでは、コードを以下のレイヤーに分けています：

1. ドメイン層 (internal/domain)

- データモデルの構造や、その振る舞いを定義
- 例： User 構造体

2. ユースケース層 (internal/usecase)

- データモデルを操作したり、外部とのやり取りを行うロジックを定義
- 例： Register 、 Login 関数

3. インターフェース層 (internal/interface)

- 外部(クライアント、DB)とのやり取りを担当
- 例： UserHandler (HTTP)、 InMemoryUserRepository (DB 操作)

1. ハンドラー (HTTP 処理)

```
// internal/interface/handler/user_handler.go
func (h *UserHandler) RegisterHandler(w http.ResponseWriter, r *http.Request) {
    // HTTPリクエストの解析とレスポンス返却のみを担当
    body, err := io.ReadAll(r.Body)

    var req usecase.RegisterRequest
    if err := json.Unmarshal(body, &req); err != nil {
        http.Error(w, "Invalid request format", http.StatusBadRequest)
        return
    }

    resp, err := h.userUseCase.Register(req)
}
```

ハンドラーの責務は、HTTP リクエストを受け取り、必要なデータを抽出し、ユースケース層に処理を委譲することです。また、ユースケースからの結果を HTTP レスポンスとして返すことも担当します。

2. ユースケース（ビジネスロジック）

```
// internal/usecase/user_usecase.go
func (uc *UserUseCase) Register(req RegisterRequest) (*RegisterResponse, error) {
    if req.Username == "" || req.Password == "" || req.Email == "" {
        return nil, ErrInvalidInput
    }

    _, err := uc.userRepo.FindByUsername(req.Username)
    if err == nil {
        return nil, errors.New("username already taken")
    }

    hashedPassword, err := bcrypt.GenerateFromPassword([]byte(req.Password), bcrypt.DefaultCost)

    if err := uc.userRepo.Store(user); err != nil {
        return nil, err
    }
}
```

ユースケース層は、アプリケーションのビジネスロジックを担当します。入力値の検証、ビジネスルールの適用、データの加工などを行います。データの永続化はリポジトリ層に委譲します。

3. リポジトリ (データアクセス)

```
// internal/interface/repository/user_repository.go
func (r *InMemoryUserRepository) Store(user *domain.User) error {
    r.mutex.Lock()
    defer r.mutex.Unlock()

    if _, exists := r.users[user.ID]; exists {
        return ErrUserExists
    }

    r.users[user.ID] = user
    return nil
}
```

リポジトリ層は、データの永続化と取得を担当します。この例では、インメモリのマップを使用していますが、実際のアプリケーションではデータベースやファイルシステムなどを使用することが多いでしょう。

同じユースケースにおいて再利用できる

- アプリケーションの中で、特定の処理を代表させる
 - その関数に必要なルールを集約できる
 - 例：
 - ユーザー登録の処理を `Register` 関数に集約
 - ユーザー名は 3 文字以上、メールアドレスは正しい形式であることを確認
 - ユーザー登録に必ずこの関数を使うようにすれば、上記のチェックを漏らすことがなくなる

ビジネスロジックをユースケース層に集約することで、同じロジックを複数の場所で再利用できます。例えば、Web API と バッチ処理の両方でユーザー登録機能を提供する場合、同じユースケース関数を使用できます。

テストが簡単

例えば先述のコード

```
mux.HandleFunc("/article", func(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query().Get("name")

    profile := db.Query("SELECT * FROM profiles WHERE name = ?", name) // 実装が重複している
    article := db.Query("SELECT * FROM articles WHERE author = ?", name)

    w.Write([]byte(profile + article))
})
```

ここで、

- article テーブルの author カラムを削除し、代わりに profile テーブルの profile_id カラムを参照するように変更したい場合
- profile 取得のクエリが他の API でも使われている場合

を考えます

- article テーブルの author カラムを削除し、代わりに profile テーブルの profile_id カラムを参照するように変更したい場合
- profile 取得のクエリが他の API でも使われている場合

テーブル構造を変更した影響が、上記クエリを利用しているすべての API のテストコードに波及してしまう

→ 直すのが大変 😱

責務が分離されていないコードでは、テーブル構造の変更が多くのテストに影響します。一方、責務が適切に分離されていれば、影響範囲は限定され、テストの修正も最小限で済みます。

適切なリクエスト処理とバリデーション

HTTP サーバーを実装する上で、リクエスト処理とバリデーションは非常に重要です。適切なバリデーションがないと、不正なデータがシステムに入り込み、様々な問題を引き起こす可能性があります。

ちょっと脱線

プログラムの特性を一言で表すと何だと思いますか？ 😐

それは「書いた通りにしか動かない」です。

正常に処理できないリクエストを受け取った際に、想定していない動作をしたり、その結果データが壊れたり、不正なリクエストを外部に送ったりしてしまう可能性があります。

プログラムは私たちが書いた通りにしか動きません。そのため、想定外の入力に対しても適切に対応できるようにコードを書く必要があります。これがバリデーションの重要性です。

ハンドリングが必要なデータの例

- 携帯電話番号
 - 日本国内に限って言えば、
 - 10桁の数字
 - 最初の3桁は、060, 070, 080, 090 のいずれか
- マイナンバー
 - 12桁の数字
 - チェックディジットの計算が必要

これらは単なる例ですが、実際のアプリケーションでは様々なデータ形式に対するバリデーションが必要です。適切なバリデーションがないと、不正なデータがシステムに入り込み、様々な問題を引き起こす可能性があります。

バリデーションの重要性

- **バリデーション** : 入力データが期待通りかを確認すること
- なぜ重要な?
 - **セキュリティ** : 悪意ある入力からシステムを守る
 - **データ整合性** : 不正なデータがシステムに入らないようにする
 - **ユーザ一体験** : 早期にエラーを検出し、適切なフィードバックを提供

例えば、

- ユーザーが入力した電話番号を保存 → いざ SMS を送信しようとしたら、番号が不正で送れなかつた 😬 というケース
- 入力された電話番号から人物を検索するシステム(があるとして)に接続したとき、その電話番号が不正だったためにシステムからはエラーが返ってきたが、その想定をしていなかつたため想定外エラーの出方をしてしまい焦る

こういうチェックをしておけば安心！

```
switch input.PhoneNumber[0:3] {
    case "060", "070", "080", "090":
        // 正常な電話番号
    default:
        // 呼び出し元でこのエラーをハンドリングしておいて、400系のエラーにする
        return nil, errors.New("invalid phone number")
}
```

バリデーションの例

1. データ形式が正しいか

```
if err := json.Unmarshal(body, &req); err != nil {
    http.Error(w, "Invalid request format", http.StatusBadRequest)
    return
}
```

2. ビジネスルールに沿っているか

```
if len(req.Password) < 8 {
    return nil, errors.New("password must be at least 8 characters long")
}
```

いずれの場合も、処理の想定外の入力を外接やデータベースに入れないので、データ不整合を未然に防いだり、不正な入力として適切にハンドリングし、エラーメッセージを返すことができます。

エラー処理

- エラーの種類に応じた適切な HTTP ステータスコードを返す
 - 400 番台：クライアントのリクエストに問題がある
 - 基本的にはシステムに問題はなく、クライアント側からの入力がビジネスロジックに即していない → 入力の何が問題なのかをクライアントに提示し、ネクストアクションを促す。特に監視は不要
 - **400 Bad Request** : クライアントのリクエストが不正
 - **401 Unauthorized** : 認証が必要
 - **404 Not Found** : リソースが存在しない
 - 500 番台：サーバー側の問題
 - システム内部で問題が発生しているので、エラーを監視して、開発者が対応する必要がある
 - **500 Internal Server Error** : サーバー内部のエラー

```
// internal/interface/handler/user_handler.go
func (h *UserHandler) LoginHandler(w http.ResponseWriter, r *http.Request) {
    // ...
    resp, err := h.userUseCase.Login(req)
    if err != nil {
        if err == usecase.ErrInvalidCredentials {
            // 認証エラーは401
            http.Error(w, "Invalid username or password", http.StatusUnauthorized)
        } else {
            // その他のエラーは500
            http.Error(w, err.Error(), http.StatusInternalServerError)
        }
        return
    }
    // ...
}
```

適切なエラーハンドリングは、クライアントに対して明確なフィードバックを提供し、デバッグを容易にします。

認証

適切なリクエスト処理とバリデーションについて学んだところで、次は Web アプリケーションにおける重要なセキュリティ機能の一つである認証について見てきましょう。認証は、ユーザーが本人であることを確認するプロセスであり、保護されたリソースへのアクセスを制御するために不可欠です。

認証とは

- **認証 (Authentication)** : ユーザーを識別するためのプロセス
- **認可 (Authorization)** : ユーザーが特定のリソースにアクセスする権限があるかを確認するプロセス

例えば、前半で紹介した以下の実装

```
mux.HandleFunc("/profile", func(w http.ResponseWriter, r *http.Request) {  
    name := r.URL.Query().Get("name")  
  
    email := getEmailByNameFromDB(name)  
  
    w.Write([]byte(email))  
})
```

ユーザー名さえわかれば、クエリに載せれば他人のメールアドレスでも取得できてしまう！

→ ユーザーがアクセスできるリソース(情報)を制限する必要がある

今回は、認証方式の代表例として、Web アプリケーションでよく使われる **JSON Web Token(JWT)** を題材に解説します！

JSON Web Token(JWT)とは

- JSON 形式のデータを安全に転送するための標準規格
- 構成：
 - i. ヘッダー：署名に使用しているアルゴリズム
 - ii. ペイロード：トークンに記載されている内容(クレーム)
 - iii. 署名：トークンが改ざんされていないことを確認するための署名

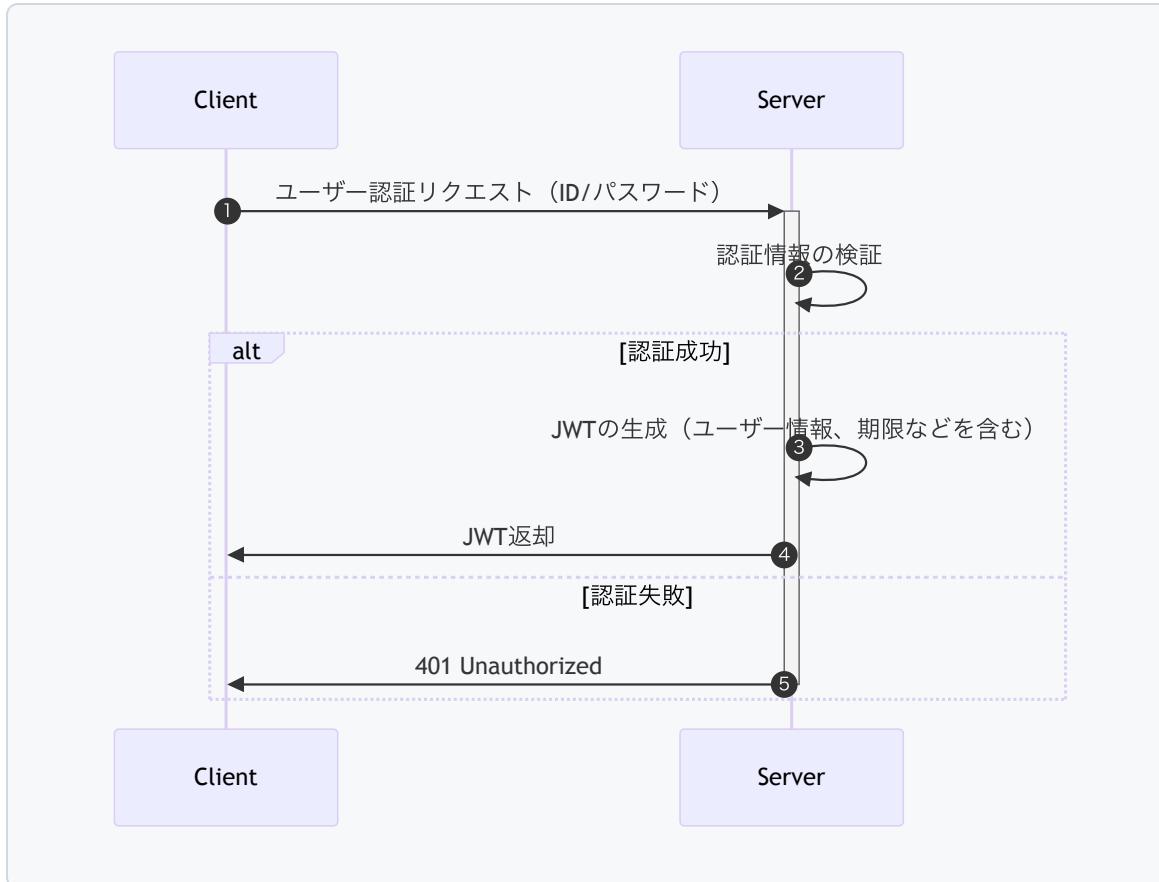
ヘッダー.ペイロード.署名 の形で表現される。

例

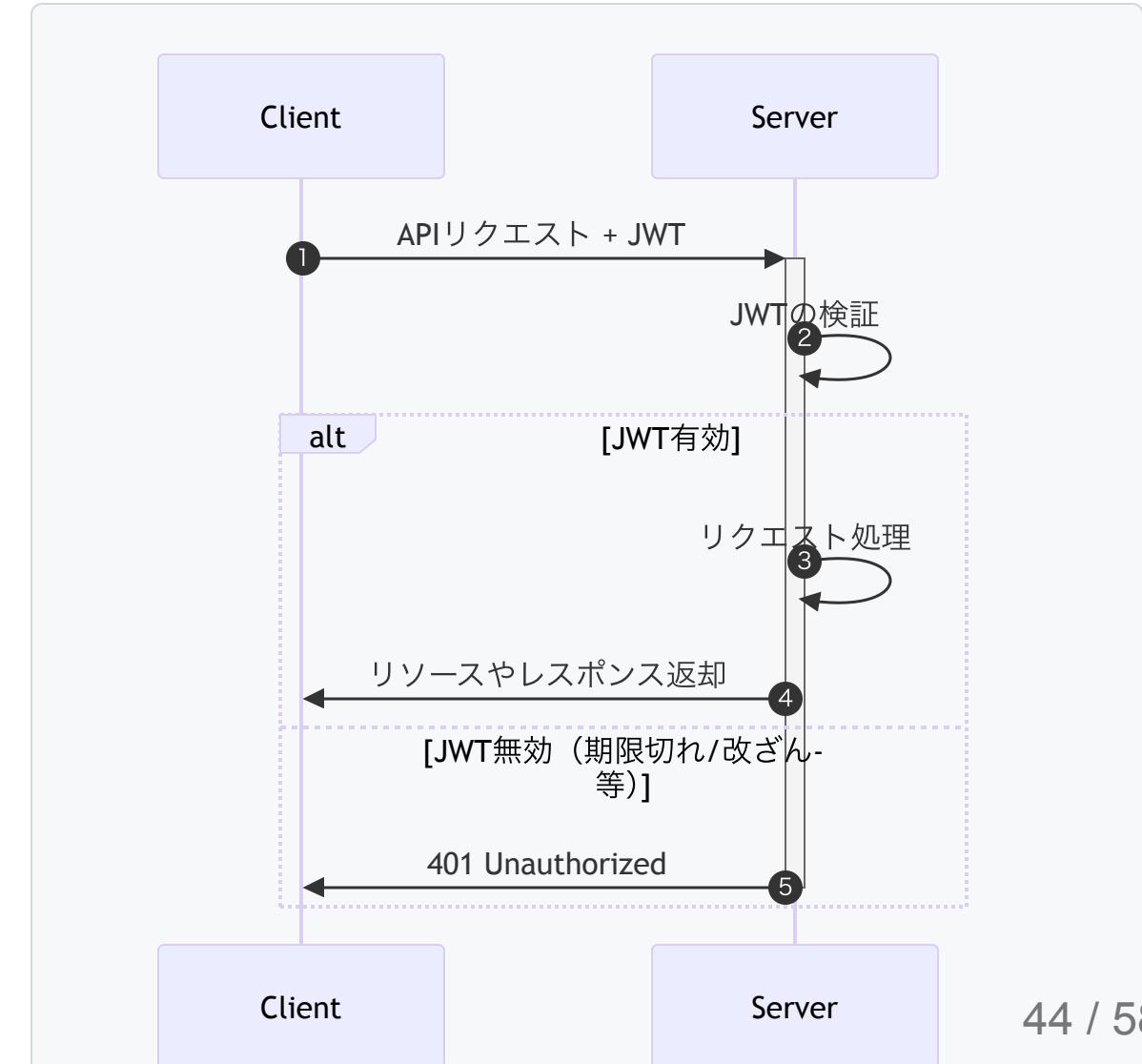
```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpv  
aG4gRG9lIiwiYWRtaW4iOnRydWUsImhlhdCI6MTUxNjIzOTAyMn0.  
KMUFsIDTnFmyG3nMiGM6H9FNFUR0f3wh7SmqJp-QV30
```

JWT を用いた際のアクセスフロー

トークン取得時



APIアクセス時



JWT の作成フロー

1. ヘッダーとペイロードを JSON 形式で作成し、それぞれを Base64URL でエンコード
2. エンコードされたヘッダーとペイロードを連結し、秘密鍵を使って署名を生成
3. ヘッダー・ペイロード・署名 の形式で 3 つの部分を連結

JWT の検証フロー

検証時には、受け取ったトークンを分解し、ヘッダーとペイロードから同じ方法で署名を再計算して、トークンに含まれる署名と一致するか確認します。これにより、トークンが改ざんされていないことを確認できます。

JWT の生成

```
func (m *JWTManager) Generate(userID, username string) (string, error) {
    now := time.Now()
    claims := JWTClaims{
        UserID:     userID,
        Username:   username,
        ExpiresAt:  now.Add(m.expiry).Unix(), // トークンの有効期限
        IssuedAt:   now.Unix(),              // トークンの発行時刻
    }

    header := map[string]string{
        "alg": "HS256", // 署名アルゴリズム
        "typ": "JWT",   // トークンタイプ
    }

    // ヘッダーとペイロードをBase64エンコード
    headerEncoded := base64.RawURLEncoding.EncodeToString(headerJSON)
    payloadEncoded := base64.RawURLEncoding.EncodeToString(payloadJSON)

    // 署名の作成
    signatureInput := headerEncoded + "." + payloadEncoded
    h := hmac.New(sha256.New, m.secretKey)
    h.Write([]byte(signatureInput))
    signature := h.Sum(nil)
    signatureEncoded := base64.RawURLEncoding.EncodeToString(signature)

    // JWTの組み立て
    token := headerEncoded + "." + payloadEncoded + "." + signatureEncoded
    return token, nil
}
```

JWT の検証

```
func (m *JWTManager) Verify(token string) (*JWTClaims, error) {
    // トークンを3つの部分に分割
    parts := strings.Split(token, ".")
    if len(parts) != 3 {
        return nil, ErrInvalidToken
    }

    // 署名の検証
    signatureInput := parts[0] + "." + parts[1]
    h := hmac.New(sha256.New, m.secretKey)
    h.Write([]byte(signatureInput))
    signature := h.Sum(nil)
    expectedSignature := base64.RawURLEncoding.EncodeToString(signature)

    if parts[2] != expectedSignature {
        return nil, ErrInvalidToken // 署名が一致しない場合はエラー
    }

    // ペイロードのデコードとパース
    payloadJSON, err := base64.RawURLEncoding.DecodeString(parts[1])
    // ...

    // 有効期限のチェック
    if time.Now().Unix() > claims.ExpiresAt {
        return nil, ErrExpiredToken // 有効期限切れの場合はエラー
    }

    return &claims, nil // 検証成功
}
```

実際のプロダクトには、上手にライブラリを使いましょう(学習目的以外では、車輪の再発明をしない)！

Go: <https://github.com/golang-jwt/jwt>

Java: <https://github.com/jwtk/jjwt>

Ruby: <https://github.com/jwt/ruby-jwt>

...

他にもいろいろ！ <https://jwt.io/libraries>

これらのライブラリを使うことで、先ほど説明した JWT の生成や検証のロジックを安全かつ簡単に実装できます。

認証方式の比較

認証方式	メリット	デメリット	適した用途
セッション	<ul style="list-style-type: none"> ・サーバー側で完全に制御可能 ・トークンの即時無効化が容易 	<ul style="list-style-type: none"> ・サーバーにセッション状態を保存 ・スケーリングが難しい 	<ul style="list-style-type: none"> ・単一サーバーのアプリケーション ・高セキュリティが必要な場合
JWT	<ul style="list-style-type: none"> ・ステートレス ・スケーラビリティが高い ・クロスドメイン対応 	<ul style="list-style-type: none"> ・トークンの即時無効化が難しい ・ペイロードサイズの制限 	<ul style="list-style-type: none"> ・マイクロサービス ・分散システム ・SPA と API の連携
OAuth	<ul style="list-style-type: none"> ・サードパーティ認証が可能 ・権限の細かい制御 	<ul style="list-style-type: none"> ・実装が複雑 ・フロー管理が必要 	<ul style="list-style-type: none"> ・サードパーティ連携 ・API プラットフォーム

JWT の使い分け

- JWT を使うべき場合
 - マイクロサービスアーキテクチャなど、複数サーバーで認証情報をやりとりする必要がある場合
 - サーバーがセッション状態を保持せず、水平スケーリングが必要な場合
- JWT を避けるべき場合
 - トークン自体に機密データを含める必要がある場合(情報自体はデコードすれば見れてしまうため)

適切な認証を用いた API アクセスの例

```
// internal/interface/handler/user_handler.go
func (h *UserHandler) GetUserHandler(w http.ResponseWriter, r *http.Request) {
    // ...
    // トークンの抽出と検証
    claims, err := h.jwtManager.Verify(token)

    // 認証情報から得られたユーザーIDを用いることで、第三者の情報を取得できないようにする
    resp, err := h.userUseCase.GetUser(usecase.GetUserRequest{ID: claims.UserID})
    if err != nil {
        http.Error(w, err.Error(), http.StatusNotFound)
        return
    }
    // ...
}
```

このように、認証情報から得られたユーザー ID を使用することで、ユーザーは自分自身の情報のみにアクセスできるようになります。

まとめ

1. 関数の責務を捉える

- 単一責任の原則に基づいたコード設計
- レイヤー分けによる関心事の分離

2. 適切なリクエスト処理とバリデーション

- 多層的なバリデーションの重要性
- セキュリティを考慮したエラーハンドリング

3. 認証の基本と実装方法

- JWT を使ったステートレス認証
- セキュアな認証フローの実装

次のステップ

- 学んだ概念を自分のプロダクトに適用してみる
 - 「こここの処理は単一の責務として切り出せないだろうか？」
 - 「もしこういうデータパターンが来たら、想定外のこととは起きないだろうか？」
 - 「見えてはいけないデータが見れてしまうつくりになっていないだろうか？」

「想定外」を漏れなく潰せるエンジニアは、「システムに強い」エンジニアの一人です！

今日学んだことを実践に活かし、より"実践的"なシステムを構築できるエンジニアを目指しましょう！



宣伝

ご清聴ありがとうございました！

質問やフィードバックがあればお気軽にどうぞ！

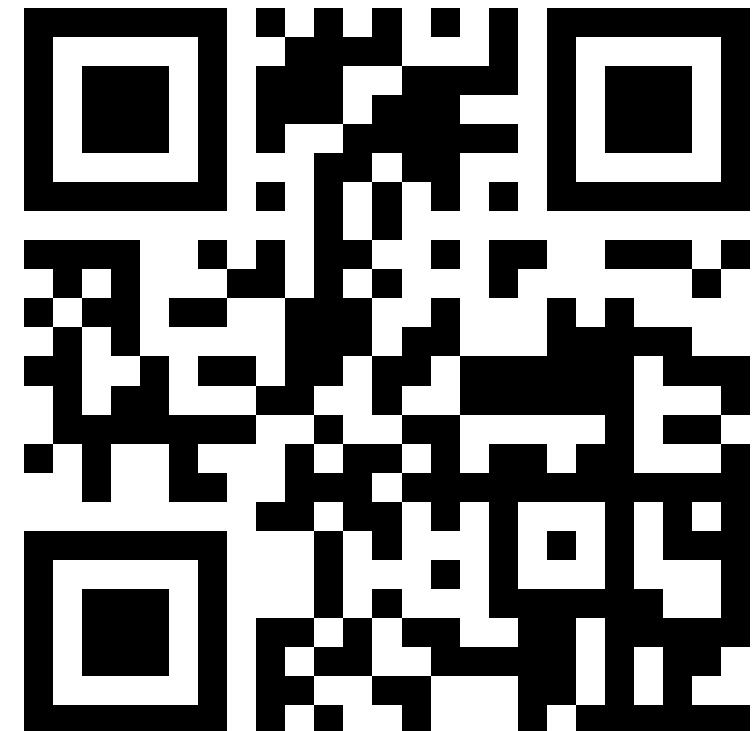
例)

- ・「自分なりにコード書いてみたのでレビューしてください！」
- ・「この部分の実装が難しいのでアドバイスください！」

X の DM でも受け付けています！→

教材のリポジトリ：

<https://github.com/noritama73/basic-http-server>



Q&A