情報工学実験 2

4617043 神保光洋

Web アプリケーションの仕組みについて

Web アプリケーションでは、アプリケーションに関する処理はほぼ全てサーバで行われる。このため、アプリケーションに新たな機能が追加された場合や、新たに別のアプリケーションのサービスを受ける場合や、新たに別のアプリケーションのサービスを受ける場合などにおいても、サービスを提供する側のサーバソフトウェアを変更するだけで良く、利用者側は何も変更することなく新たなサービスが受けられる。

サーバーはクライアントからの要求に従い HTML や画像データなどを送信し、Web ブラウザはそのデータを表示する。サーバは、クライアントからのリクエストを待ち受けており、クライアントからの要求が到着すると処理を行い、レスポンスを返す。また、サーバはクライアントの要求を解釈し、クライアントはサーバの応答を解釈する必要がある。この要求・応答のやり取りの方式や解釈の方式の取り決めを HTTP と呼ぶ。以下が処理の概要である。

1. クライアントからサーバに対して接続を行う。すなわち、TCP によるコネクションの確立が行われ、双方のコンピュータ同士で通信が可能となる。
2. クライアントからサーバに対し、「指定した URL の内容を送れ」という意味の要求が送られる。 URL とは、インターネット上のリソースを示す記述法である。
3. サーバは要求された URL に対するデータや、サーバプログラムによって処理された結果をクライアントに対して送信する。クライアントである Web ブラウザでは、返ってきたデータに表示する。

Web アプリケーションの機能を提供するサーバを Web アプリケーションサーバと呼ぶ。 Web アプリケーションサーバは、

その上で複数のサーバプログラムを動作させ、様々なサービスを提供する。 Web アプリケーションサーバは、通常の Web サーバと同じ HTTP を用いてブラウザと通信を行うため、Web アプリケーションは通常の WWW のコンテンツの閲覧と同様にブラウザを用いて利用することができる。

HTTP リクエスト / レスポンスの内容について

HTTP とは Web サーバと Web クライアントの間でデータの送受信を行うために用いられるプロトコルであり、HTTP でサーバとクライアントがやり取りするメッセージは基本的に以下の形式になっている。なお、記号 “ ” は改行を表している。

メッセージヘッダ

メッセージボディ

まず、リクエストメッセージについて説明する。以下のリクエストは、リクエストメソッドが GET であるので、メッセージボディは存在しない。

```
GET /HTTP/1.1
telnet localhost 80
```

メッセージヘッダの 1 行目では、” リクエストメソッド、リクエスト URL、HTTP のバージョン” が “GET /HTTP/1.1” であることを表している。すなわち、HTTP の 1.1 バージョンの通信で、GET というリクエストの方法でクライアントからサーバに要求していることを意味する。

HTTP のメソッドは、その用途によって GET と POST で使い分ける。

###GET メソッド HTTP 通信でサーバから情報を取得する時に使用する。 GET でのサーバへのデータの送信方法として、リクエスト URL の後に？に続いて情報を送信する方式であるクエリストリングがある。 URL の末尾に「？」マークを付け、続けて「名前＝値」の形式で記述する。値が複数春時は「&」で区切り記述する。

```
htt:// example.com/index.html?name1=value&name2=value2
```

このように送信するデータがアドレスバーに表示されてしまうため、他人に見られる可能性がある。他人に見られたくない情報は GET では送らない。

日本語などの文字は、そのままでは URL で送信することができないので、そのような文字を URL に付与して送信するには、パーセントエンコーディングという技術を使用する。

###POST メソッド HTTP 通信で、サーバへ情報を登録する時に使用する。データ量が多い場合、バイナリデータを送信したい場合、他の人に見られたくない情報を送る場合に使用する。 POST でサーバにデータを送信する際は、メッセージボディにデータが記述される。以下に、POST でのサーバからクライアントへのリクエスト例を示す。

```
POST /test/ HTTP/1.1
Hsot: localhost
Content-Length: 25
Content-Type: application/x-www-form ー urlencoded

name1=value1&name2=value2
```

上記のように、リクエストヘッダの後に一行、空行が入り、その後 POST で送信したクエリストリングが、リクエストボディとしてクライアントからサーバへと送信される。リクエストボディの長さは、「Content-Length:」という項目で表される。次に、レスポンスメッセージについて説明する。いかに具体例を示す。

```
HTTP/1.1/ 200 OK
Content-Type: text/html
Date: Mon, 3 Sep 2017 00:00:00 GMT
Last-Modified: Fri, 1 Sep 2017 00:00:00 GMT
Accept-Ranges: bytes
Etag: "011be787cw2f41"
Server: Apache/2.4.27(Unix)

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Sample</title>
```

```
</head>
<body>
<p>Hello!</p>
</body>
</html>
```

メッセージヘッダの一行目はステータスラインであり、”HTTP のバージョン、ステータスコード” を表している。ステータスコードを以下の表に示す。

表 1: ステータスコード

ステータスコード	内容
1xx	情報
2xx	成功
3xx	リダイレクション
4xx	クライアントエラー
5xx	サーバエラー

残りのヘッダ部分にはリクエスト先のウェブサーバの情報等が記載されている。メッセージボディ部分が、クライアント側で表示される内容となっている。

MVC モデルの仕組みについて

MVC は、Model, View, Controller の三つに役割を分割させたコーディングモデルでユーザインターフェースをもつアプリケーションを実装するためのデザインパターンである。アプリケーションの内部データをユーザが直接参照、編集する情報から分離する。そのためにアプリケーションを三つの部分に分割する。

主な役割は以下のようになる。

- Model
  - データの改変とそれを View に伝える。アプリケーションデータ、ビジネスロジック、アプリケーションが扱う領域のデータと手続きを表現する要素である。ここで、ビジネスロジックとは、ビジネスオブジェクトをモデル化したデータベース上のデータに対する処理手順を示す。
- View
  - 表示と入出力。グラフや図などの任意の情報表現モデルのデータを取り出してユーザが見るのに適した形で表示する要素である。すなわち、UI への出力を担当する。 Web アプリケーションでは HTML 文章を生成して動的にデータを表示するための部分にあたる。
- Controller
  - ユーザーの入力に基づいた，Model と View の制御。入力を受け取り model と view への命令に変換するユーザからの入力（通常イベントとして通知される。）をモデルへのメッセージへと変換してモデルに伝える要素である。すなわち、UI からの入力を担当する。モデルに変更を引き起こす場合もあるが、直接に描画を行なったり、モデルの内部データを直接操作したりはしない。

Web アプリケーションにおける一連の動作を図 2 に示す。まず、クライアントであるユーザがブラウザを利用してサーバにリクエストを投げる。このリクエストがコントローラに到着すると、モデルと通信し必要とされるデータの取得または保存操作が実行される。次に、この通信が終わると、コントローラはモデ

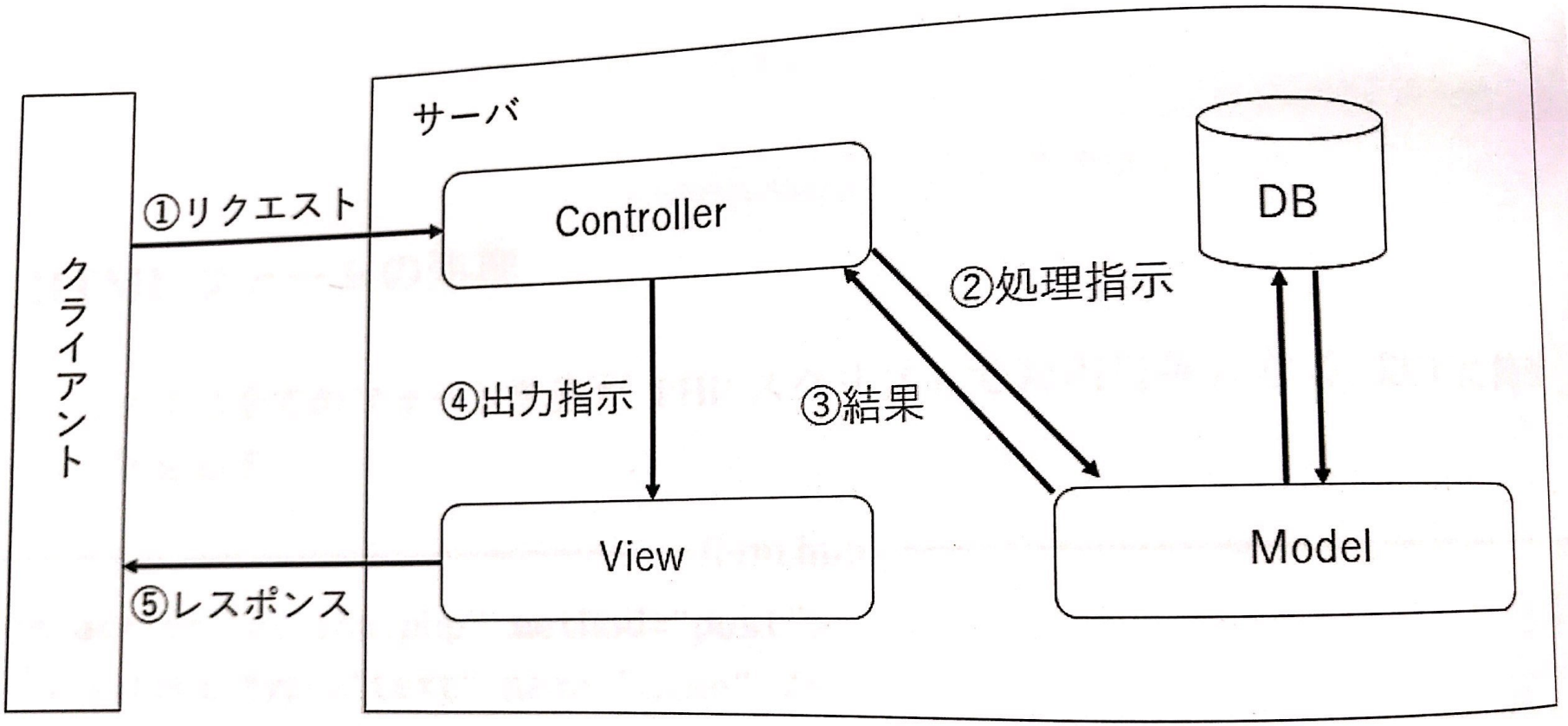

図 2: Web アプリケーションの概要

図 1: すごい図

ルから提供されたデータに基づく結果を生成するタスクを適切なビューに渡す。最後にビューで生成された出力がクライアントに返されて、その内容がユーザのブラウザに対して描画される。

開発において、機能ごとの分離が明確になることによって、それぞれの独立性が保たれ、分業がしやすくなり、各分野の実装に集中することができる。また、各要素間の依存性が最小限に抑えられるため、ほかの部分の変更による影響を受けにくく実装にすることが可能となる。これにより各要素の再利用性が高まる。また、変更を加えた際、複数の担当者が同一のソースに対してメンテナンスを行うことが減り、保守性も確保されるなどのメリットがある。

フレームワークのメリット、デメリットについて

Web フレームワークとは、Web アプリケーションの開発をサポートするために設計されたクラスやライブラリの集まりである。フレームワークの目的は、Web 開発で用いられる共通した作業に伴う労力を軽減することである。主な機能としては、データベースへのアクセスのためのライブラリ、ルーティング、テンプレートエンジン、セッション管理を提供している。 Web フレームワークを用いることにより、本格的な Web アプリケーション開発が可能になる。なお、ルーティングとは、リクエストされた URL をそれに対応した HTML を生成するコードに割り当てるプロセスのことである。

フレームワークのメリット

早く作ることができる。

あらかじめよく使う機能が提供されているため一度フレームワークの仕様自体を把握してしまえば開発効率を上げることができる。

コーディングスタイルの統一化

フレームワークはコーデイングスタイルをある程度統一することによって複数人の開発において保守性を高めることができる。

フレームワークのデメリット

フレームワークへの依存

フレームワークにより処理の詳細がラップされてしまうため。深い知識がなくてもプロダクトができてしまう。そのため、バグの特定に時間がかかってしまうなど、保守・運用などで支障をきたすことがある。

フレームワークでサポートされていない機能を追加しづらい

フレームワークでサポートされていない機能を追加する際、一から全て実装しなければならない時があるが、フレームワークを使う必要性がなくなってしまう。

実験中に説明・演習したことについて

telenet の接続

Telnet は、ネットワークに接続された機器を遠隔操作するために使用するアプリケーション層プロトコル。　マシンルームにあるサーバ、ルータ等の機器をパソコン上で操作することができる。パスワード情報を含め全てのデータが暗号化されずに送信される ### 接続方法 cygwin にて

```
GET telenet localhost 80
Host: local host
```

とコマンドを打つと localhost 80 にあるサイトに接続し、情報を標準出力に出すことができる。 ## URL の構成 URL は以下の 3 つないしつの構成に分けられる。

スキーム

インターネット上にあるそのリソースへのアクセスに使用されるプロトコルを識別する。すなわち、HTTP (SSL なし) または HTTPS (SSL あり) のいずれである。

ホスト

そのリソースを保持するホストを識別する。サーバーはホストの名前でサービスを提供するが。 ホストとサーバーの間で 1 対 1 のマッピングは行われていない。ホスト名の後にポート番号が付く場合もある。サービスの予約済みポート番号は、通常 URL から省略されている。ほとんどのサーバーは、HTTP および HTTPS 用に予約済みのポート番号を使用するため、 HTTP URL からポート番号は除外される。

パス。

Web クライアントがアクセスする、ホスト内の特定のリソースを識別する。

照会ストリング。

照会ストリングが使用される場合はパス構成要素 の後に付加され、そのリソースの使用目的に関する情報ストリングを提供します。照会ストリングは通常、名前と値がペアになったストリングである。名前と値のペアは、(&) で互いに分離されます。

CRUD について

RUD（クラッド）とは、データを参照する際に必要な最小限の機能のことで以下 4 つのイニシャルの略である。

Create（生成)、Read（読み取り)、Update（更新)、Delete（削除）

ユーザインタフェースが備えるべき機能（情報の参照/検索/更新）を指す用語としても使われる。

データベースシステムの完全性を備えるためには、CRUD のそれぞれの機能を取り入れる必要がある。

課題１、２、３の説明と考察

課題 1

説明

Slim framework と Twing を利用して、消費税計算機 Web アプリケーションを作成する。 index.php のコードは以下のようになった

```php
<?php
require 'vendor/autoload.php';

$app = new \Slim\App();

$loader = new Twig_Loader_Filesystem('templates');
$twig = new Twig_Environment($loader);

$app->get('/', function ($request, $response, $args) use ($twig) {
    $data = [];
    $data['price'] = 100;
    $html = $twig->render('tax.html', $data);
    return $response->write($html);
});

$app->post('/', function($request, $response, $args) use ($twig) {
    $data = $request->getParsedBody();

    if ($data['field'] == "include") {
      $data['rate'] = 8;
      $data['taxinclude'] = $data['price'];
      $data['tax'] = $data['price'] / 0.08;
      $data['taxexclude'] = $data['price'] / 1.08;
    } else {
      $data['rate'] = 8;
      $data['taxinclude'] = $data['price'] * 1.08;
      $data['tax'] = $data['price'] * 0.08;
      $data['taxexclude'] = $data['price'];
    }

    $html = $twig->render('tax.html', $data);
    return $response->write($html);
});

$app->run();
```

また tax.html の中身は以下のようになった

まず post 関数の方の最後にも

```php
$html = $twig->render('tax.html', $data);
return $response->write($html);
```

を入れるこれにより post 関数の操作がレンダリングされるようになる。次に post 関数内の

```php
if ($data['field'] == "include") {
  $data['rate'] = 8;
  $data['taxinclude'] = $data['price'];
  $data['tax'] = $data['price'] / 0.08;
```

```
  $data['taxexclude'] = $data['price'] / 1.08;
} else {
  $data['rate'] = 8;
  $data['taxinclude'] = $data['price'] * 1.08;
  $data['tax'] = $data['price'] * 0.08;
  $data['taxexclude'] = $data['price'];
}
```

の部分について説明する。まず if での評価であるが後述する tax.html にて税込のラジオボタンには

`name="field" value="include"`

が、 税抜きのラジオボタンには

`name="field" value="exclude"`

と記述したので

`$data['field']` には税込みの場合は `include` が税抜きの場合には `exclude` が入っているはずなのでそれにより処理を変える。税込みの場合には 0.08 で割ってやり税抜きの場合には 0.08 で掛けてやれば良い

また tax.html は以下のようになる。

```
<!DOCTYPE html>
<html lang="ja">
<head>
    <meta charset="utf-8">
    <title>消費税計算機</title>
</head>
<body>
<h1>消費税計算機</h1>
<form action="" method="POST">
    <p>
      <label>
        金額     <input type="number" name="price" value={{price}}>
      </label>
    </p>

    <p>
    <input type="radio" name="field" value="include" {% if not rate or mode == 'include' %}checked {
    <input type="radio" name="field" value="exclude" {% if not rate or mode == 'exclude' %}checked {
    </p>

    <p><input type="submit" value="送信"></p>
</form>
<table border="1">
  <tr>

<label>
    <td>消費税率</td>
    <td>税抜金額</td>
```

```
    <td>消費税</td>
    <td>税込金額</td>
  </tr>
  <tr>
    <td>{{rate}}%</td>
    <td>{{taxexclude}}円</td>
    <td>{{tax}}円</td>
    <td>{{taxinclude}}円</td>
  </tr>
</table>
</body>
</html>
```

はじめにラジオボタンを書いた。その後 index.php にて内部処理が計算されその結果が tax.html に渡ってくる。それらを Twing により表示する。

考察

今回は MVC におけるコントローラとして Slim、ビューとして Twig を使っている index.html において

```
$html = $twig->render('tax.html', $data);
```

の一行が見られるがこれより twig で tax.html と`$data` が使えるようになっていることが確認できる。 以前 flask, Django を勉強した経験があったが MVC モデル（Django は MVC ではないけども) について理解しておらず。うまく使いこなすことができなかった。

今回 Slim, Twing というフルスタックでないフレームワークを通してコントローラ、ビューは MVC の中でそれぞれの役割が明確になる。

課題 2

説明

index.php は以下のようになった。

```
<?php
require 'vendor/autoload.php';

// データベース接続
try {
    $pdo = new PDO('sqlite:vending.db');
} catch (PDOException $e) {
    die("データベース接続失敗" . $e->getMessage());
}
// テーブル作成
$pdo->exec("create table if not exists items (
    id    integer primary key autoincrement,
    name  varchar,
    price integer,
    num   integer)"
```

```
);

// SLIM framework の準備
$app = new \Slim\App();

// twig の準備
$loader = new Twig_Loader_Filesystem('templates');
$twig = new Twig_Environment($loader);

//---------------------------------
// ルーティング
//---------------------------------
$app->get('/', function ($request, $response, $args) use ($twig, $pdo) {
    $data = [];
    $data['pay'] = 0;

    $stmt = $pdo->prepare("SELECT * FROM items");
    $stmt->execute();
    $data['items'] = $stmt->fetchALL();

    $html = $twig->render('machine.html', $data);
    return $response->write($html);
});

$app->post('/', function ($request, $response, $args) use ($twig, $pdo) {
    $data = $request->getParsedBody();

    // データ表示テスト
    //echo "<pre>";
    //var_dump($data);
    //echo "</pre>";
    // データ表示テスト

    $stmt = $pdo->prepare("SELECT * FROM items");
    $stmt->execute();
    $data['items'] = $stmt->fetchALL();


    $data['buyed'] = array();
    $sum = 0;

    for ($i = 0; $i < count($data['buyitem']); $i++) {
        $sum = $sum + $data['buyitem'][$i + 1] * $data['items'][$i]['price'];
    }

    for ($i = 0; $i < count($data['buyitem']); $i++) {
        if ($data['buyitem'][$i + 1] > 0) {
```

```
            array_push($data['buyed'], $data['items'][$i]['name']);
        }
    }

    $data['result'] = True;
    $data['sum'] =  $sum;
    $turi = $data['pay'] - $sum;
    $data['turi'] = $turi;

    if ($sum > $data['pay']) {
      $data['lessmoney'] = "投入金額不足";
    }
    else {
      $data['lessmoney'] = "";
      for ($i = 0; $i < count($data['buyitem']); $i++) {
        $lest = $data['items'][$i]['num'] - $data['buyitem'][$i + 1];
        if ($lest > 0) {
          $stmt = $pdo->prepare("update items set num = ? where id = ?");
          $stmt -> execute([$lest, $data['items'][$i]['id']]);
        }
      }
    }

    $html = $twig->render('machine.html', $data);
    return $response->write($html);
});

$app->run();
```

上から順に説明していく。合計額の計算をまず行うこれはベクトルの内積となるので、for 分で記述する際は二変数が必要となるこれらを先にマップして zip をしてその後、foreach 文で計算させる方法も考えられたが処理数が増え、map はもともと命令型のパラダイムでないため、可読性を下げる可能性があり、foreach で少し可読性は上がるが総括するとシンプルさが下がり命令型のパターンとは外れるので for 文を採用するに至った。

次に購入した商品リストを作成するためもう一度 for 文を行なっている繰り返し回数は上と変わらず上の for ぶんの処理の内部にかけるが今回はあくまで課題であるため速さよりも可読性を重視した。配列の追加にはもう一つ技巧的な関数を使わない方法があるがこれも可読性の向上のため関数を呼び出し push させている。

合計金額を計算したらそれが支払額より小さいことを確認して支払額よりも大きかったらユーザに支払額が不足していることを伝えそうでなければデータベースに変更を加える。

```
<!DOCTYPE html>
<html lang="ja">
<head>
    <meta charset="utf-8">
    <title>自動販売機</title>
</head>
```

```
<body>
<form action="" method="POST">
<table>
    <caption>商品を選んでください</caption>
    <tr>
        <th>商品名</th>
        <th>価格</th>
        <th>在庫</th>
        <th>個数</th>
    </tr>
    {% for item in items %}
    <tr>
        <td>{{item.name}}</td>
        <td>{{item.price}}</td>
        <td>{{item.num}}</td>
        <td><input type="number" name="buyitem[{{item.id}}]" value="0"></td>
    </tr>
    {% endfor %}
</table>
<p>
    <label>
        投入額
        <input type="number" name="pay" value="{{pay}}">
        <input type="submit" value="購入">
    </label>
</p>
</form>

{% if result %}
  <p>購入商品: </p>
  {% for b in buyed %}
      <p> {{b}} </p>
  {% endfor %}

  <p>おつり: {{turi}}</p>

  <p>合計金額: {{sum}}</p>

  <p>{{lessmoney}}</p>
{% endif %}

</body>
</html>
```

MVC よりビューとコントローラを分離させるため、index.php の処理のあと結果をそのまま index.php 内でそのまま `echo` や `print` を使わずに一度 twig にデータを渡し、twig により結果を表示させる。

考察

今回は twig にて if 文や for 文の使用も行なった。なぜ slim だけでなく twig にも if 文 for 文があるのかこれはまさに MVC における機能の独立化のためであることがわかる。

課題 3

index.php のソースコードは以下のようになった

```
//新商品追加
$app->get('/insert/{name}/{num}', function ($request, $response, $args) use ($twig, $pdo) {
    $stmt = $pdo->prepare("insert into items (name, price, num) values (?,?,?)");
    $stmt->execute([$args['name'], 0, $args['num']]);

    return $response->write('新商品' . $args['name'] . 'を' . $args['num'] . '個追加しました');
});

//商品削除
$app->get('/delete/{name}', function ($request, $response, $args) use ($twig, $pdo) {
    $stmt = $pdo->prepare("delete from items where name = ?");
    $stmt->execute([$args['name']]);

    return $response->write('商品' . $args['name'] . 'を削除しました');
});

//価格設定
$app->get('/set/{name}/{price}', function ($request, $response, $args) use ($twig, $pdo) {
    $stmt = $pdo->prepare("update items set price = ? where name = ?");
    $stmt->execute([$args['price'], $args['name']]);
    //$stmt->execute([$args['price']], [$args['name']]);

    return $response->write('商品' . $args['name'] . 'の価格を' . $args['price'] . 'に変更しました');
});

//数量追加
$app->get('/add/{name}/{num}', function ($request, $response, $args) use ($twig, $pdo) {
    $stmt = $pdo->prepare("SELECT * FROM items");
    $stmt->execute();
    $data['items'] = $stmt->fetchALL();

    $doflag = 0;
    for ($i = 0; $i < count($data['items']); $i++) {
        if ($data['items'][$i]['name'] == $args['name']) {
            $doflag = 1;
            $newnum = $data['items'][$i]['num'] + $args['num'];
        }
    }
```

```
if ($doflag == 1) {
    $stmt = $pdo->prepare("update items set num = ? where name = ?");
    $stmt->execute([$newnum, $args['name']]);
    return $response->write(' 商品' . $args['name'] . ' の個数を' . $newnum . ' に変更しました');
}
else {
    return $response->write(' 商品' . $args['name'] . "は見つかりませんでした");
}
```

説明

商品削除、価格設定についてはただ sql のコマンドを送るのみである。ただし、価格設定において execute にて変数を送る際、sql コマンドに変数が遍在していても c 言語の print 文のようにコンマで区切っておけば変数は自動的に各？に渡される。

数量追加においては sql から現在の商品数を取ってこなければならない。これは購入した際に商品の数をデータベースから差し引く場合と同様の操作を行えば良い。ただしデータベースに変更する商品がない場合が考えられるのでそこについて例外処理を付け足しておく。

考察

execute の際、C 言語などと違って型指定を行わないことによりコードがすっきりとしている型指定を行わないことは別に悪いことでないことがわかる。

参考文献

slim ホームページ https://www.slimframework.com/

twig ホームページ https://twig.symfony.com/