Day 1 Web開発基礎

赤松 祐希

cookpad

Ruby

ゴール

- 簡単なWebアプリケーションの実装を通してWeb開発の要素技術に触れる
- ちょっとしたWebアプリケーション「フレームワーク」であれば自作できそうと思って欲しい
    - 個々の要素技術の詳細よりは、それぞれの関係性や概念的なところを掴んで欲しい

Ruby基本文法

```ruby
# 変数
str = "hello"

# メソッド
def say(message)
  puts message
end

# ブロック
[1,2,3,4].each do |i|
  puts i
end

```

```ruby
# クラス

class User
  attr_accessor :name, :age

  # 初期化 + キーワード引数
  def initialize(name:, age:)
    @name = name
    @age = age
  end

  def say
    puts "I'm #{@name}"
  end
end
```

rbenv

- Rubyのバージョンを切り替えるツール
  - cpadインストール時に自動で入っています
- 今回はRuby 2.6.3 を使用するので rbenv経由でインストール
  - rbenv install 2.6.3
  - rbenv global 2.6.3 rbenv影響下全体で有効
  - rbenv local 2.6.3 実行したディレクトリ以下で有効(globalより優先される)
    - .ruby-versionというファイルが生成される

bundler

- プロジェクトで使うgem(RubyGems)を管理
- 使用するgemを明示できたり、バージョンを指定できたりする
- rubygems.org以外でホストされているgemをインストールすることも可能
  - 例えばGitHub
- gemの雛形を作る機能などもある

Rackアプリケーション

- RailsやSinatra、多くのRuby製Webフレームワークの理解においてRackというのが非常に重要
- Rackアプリケーションとして実装すると、様々なアプリケーションサーバーに対応することができる
  - 例: Unicorn、Puma

Rackが登場した背景

- 今までに多数のアプリケーションサーバーが登場してきた(恐らくこれからも)
  - Puma、Unicorn、Thin、etc…
- より良いアプリケーションサーバーが登場したときに、アプリケーションの改修コストを払いたくない!
  - お互いにRackに対応することでコストが不要になる
- 元々はPythonで策定されたWSGIというアプリケーションサーバーとアプリケーションの仕様が元になっている

ブラウザからの処理の流れ

- Rackがアプリケーションとアプリケーションサーバーを仲介することで、アプリケーションとアプリケーションサーバーが依存せずにすむ

Rackアプリケーションの仕様

- callメソッドが定義されていること
  - callメソッドは環境変数を引数として受け取る
  - 環境変数にはHTTPリクエストのメソッドやパスが含まれている
- callメソッドの返り値は以下の3つを含むArrayであること
  - HTTPステータスコード
  - HTTPヘッダ
  - HTTPメッセージボディ

HTTPリクエストとレスポンス

- クライアントからサーバーに「何を」「どうして欲しい」か指示するのがHTTPリクエスト
  - 何を = パスで表現
  - どうして欲しい = メソッドで表現
    - GET=取得 POST=作成 PATCH=書き換え DELETE=削除
- それの返事がHTTPレスポンス
  - どういう結果になったか = ステータスコード
    - 200系=成功 300系=移動とか 400系=クライアント起因の失敗 500系=サーバー起因の失敗
  - 内容 = ボディ

サーバーサイドアプリケーションとは

- HTTPリクエストを受け取り、何かしらの処理を行い、HTTPレスポンスを返すアプリケーション
- ブラウザベースであればボディにHTMLを、APIであればJSONを入れて返すことが多い
- 最近はJSによるSPAなどもWebアプリケーションと呼ばれていると思うので、意図的に「サーバーサイド」アプリケーションと表現しています

演習: Rackアプリケーションを書いてみよう

- Rackアプリケーションを実装し、サーバーアプリケーションに「最低限」必要な要素をつかむ
- 演習用資料のSection 1をやってみよう

ルーティングの導入

- サーバーアプリケーションが「何をするか」はHTTPリクエストのパスとメソッドで大体決まる
- 例題ではRack::Requestのpath_infoとrequest_methodで処理を振り分けた
- 今後エンドポイントが触れるにつれ分岐がどんどん増えていくので、コードの見通しをよくするためルーティングを導入する
- ルーティングとはリクエストされたパスやメソッドに対して、それを処理するためのコードを割り当てること
    - 厳密に言うとすでにルーティングしているが、もう少し構造だったものを実装したい

Controller

- 例えばRailsであれば /users にリクエストがあった場合 UsersController で処理するというのが基本
  - RESTfulとかリソースとかの話があるが今日は割愛
- 今回もRailsにならってパスごとに処理するControllerを用意する

演習: Controllerの実装

- Section 2に従いControllerの実装を行う

View

- レスポンスに含めるHTMLをもう少し複雑なものを扱えるようにしたい
- Rubyスクリプト内に文字列として埋め込むことも可能だが可読性に問題がでることが予想できる
- そこでViewのファイルをapp.rbとは別の場所に切り出して記述できるようにする

テンプレートエンジン

- 今回はERBを使用する
  - Embedded RuBy の略
  - テキストファイルにRubyを埋め込むことができる
- 他にもHamlやSlimなどがある
  - こちらの方が記述量が少ない、HTMLの閉じタグ忘れを防げるなどの理由により一般的
- サーバーサイドアプリケーションを作る場合、Viewを動的に生成したいことがほとんどなので何かしらのテンプレートエンジンを使うことが多い

演習: Viewの実装

- Section 3に従いViewを実装する

Model

- ControllerからViewに値を渡して表示できるようになった
- そこで今度はデータベースから取得した値を表示するよう対応したい

リレーショナルデータベース(RDB)

- Webフレームワークでは一般的なデータストア
  - MySQL、PostgreSQLなど

SQL

- RDBにおいてデータの操作や定義を行うためのクエリ言語
- ORM(後述)の登場によって、アプリケーション上に直接記述する機会は減ってきているが、SQLを理解しておくことは重要
- ORMがどういうSQLを発行しているのか理解しておく
    - ここを理解しておかないと非効率なクエリを発行してしまう、またそれの解決手段がわからないなどが起きる
- 複雑な集計はまだまだORMではカバーしきれないこともある

ORM

- RDBとオブジェクト指向におけるインピーダンス・ミスマッチを解消する
  - インピーダンス・ミスマッチ = データモデルの違いによるギャップ
- ActiveRecordパターン
  - RailsのActiveRecordは名前の通りActiveRecordパターンが元になっている
  - RDBのテーブルと1つのモデルを関連づける考え方
  - 例: Userクラスとusersテーブル
- 注意: 1つのテーブルと1つのモデルを関連づけるのがORMではない
  - 複数のテーブルを1つのモデルに関連づけるものもある
- オブジェクトを通してSQLの実行やオブジェクトへのマッピング、DBのコネクション管理なども行う

演習: データベースのデータを画面に表示

- データベースはSQLiteを使用
  - データベースサーバーが不要な軽量のDBMS
  - セットアップが簡単
  - PostgreSQLやMySQLと比較すると機能が貧弱
- ORMライブラリは使用しない
  - 規模感的に不要
  - SQLの結果とオブジェクトのマッピングを感じて欲しい

発展課題

- 個別のTodoを表示するページの実装
  - /todos/:id のルーティングをどうするか
- Todoを保存するページの実装
  - フォームページの実装やPOST先のエンポイント
- app.rbのリファクタリング
  - 例えばControllerやModelのクラスを外に切り出す
- フレームワークとしてもっと汎用的に作る

TypeScript

ゴール

- TypeScriptに親しむ
  - 特に普段動的言語で静的型付けの言語を触ったことない人達
  - とはいえ、今回は型に関しては大分ゆるっといきます
- 非同期処理について理解する
  - APIの呼び方
  - 非同期処理の実装方法
    - Day 3でも登場するので、今日頑張って理解してください

フロントとサーバーの分離

- Webフロントエンドが複雑になるにつれ、フロントとサーバーが分離することが増えてきた
  - APIサーバーとそれを呼ぶクライアントアプリケーション
  - MVCのVが複雑化・独立してきたとも言える
- jQueryだけでは戦えない世界

JavaScriptとTypeScript

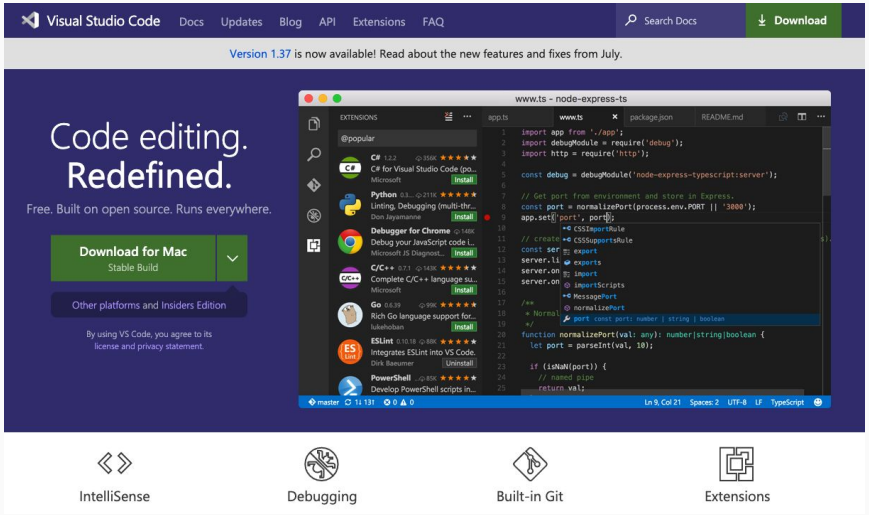
- 以前よりマシとは言え、複雑なクライアントサイドアプリケーションを実装するにはJavaScriptの言語機能では厳しいことも多い
- TypeScriptはJavaScriptに静的な型機能を提供するトランスパイラ
  - 個人的には原則TypeScriptを導入するのがオススメ
  - 型機能をフル活用しないにしてもTypeScriptの方がJavaScriptより厳格

環境構築

- Visual Studio Codeのインストール
  - フォーマッターのPrettierも入れる
- Node.jsのインストール
  - マシンにnodebrewが入っているはずなのでそれを使う

Visual Studio Code

- TypeScriptを書くにあたってコード補完や型定義の参照などで便利
- とりあえず覚えてほしいショートカット
  - ファイル検索 cmd+p
  - コマンドパレット cmd+shift+p
  - ターミナル ctrl+shift+`
  - 定義へジャンプ F12



TypeScirptのコンパイル

- tsc --init
  - TypeScriptのコンパイル設定が生成される
  - デフォルトから以下の3点を変更します
  - noImplicitAnyをfalse
  - outDirをdist
  - includeに[“src/**/*”]
- tsc
  - TypeScriptをJavaScriptへコンパイルします

noImplicitAnyについて

- 型定義がない変数はany型(つまり何でもいい)になる
- noImplicitAnyがtrueだと暗黙のany型が許容されなくなる
- 今回falseにした理由
  - 型をつけるということになれていないと結構しんどい
  - falseでも十分にTypeScriptの恩恵を得られる

演習: TypeScriptをコンパイルする

- 演習資料 Section 1を参考に環境構築、TypeScriptのコンパイルまでやろう

基本文法: 変数

```
1 // 再代入不可な変数。原則constを使う。
2 const str: string = "hello";
3 const array: number[] = [1, 2, 3, 4, 5];
4
5 // 再代入が必要になる場合letを使う
6 let myName: string = "Yuki";
7 myName += " Akamatsu";
```

基本文法: 関数

```
1 // 関数
2
3 function say(message: string) {
4   console.log(message);
5 }
6
7 const twice = function(x: number): number {
8   return x * 2;
9 };
10
11 const sum = (x: number, y: number): number => {
12   return x + y;
13 };
```

基本文法: クラス

```
1 // class
2
3 class User {
4   name: string;
5   constructor(name: string) {
6     this.name = name;
7   }
8
9   say(): string {
10    return `My name is ${this.name}`;
11  }
12 }
13
14 new User("ukstudio").say()
```

基本的な型

- boolean
- number
- string
- array
- tple
- any
- void

型推論

- 変数などの型を明示的に指定しなくても型システムの方で推測し、型を決定してくれる仕組み
  - 推測できない場合、anyやneverなどになる

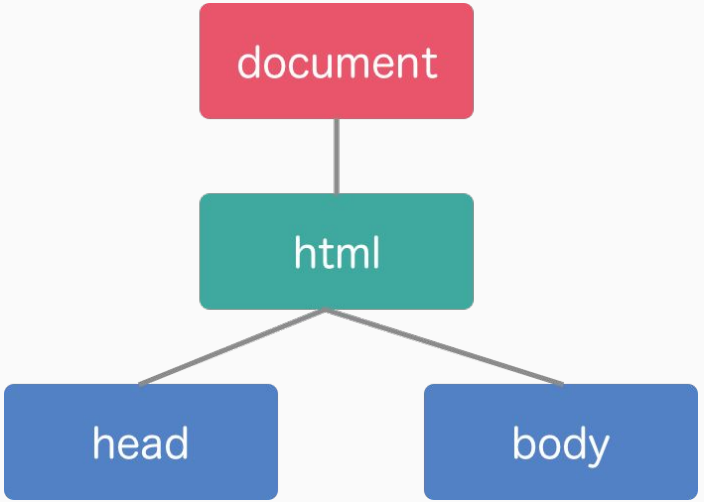
```
let str = "abc"; // string
let num = [1, 2, 3, 4, 5]; // number[]

let foo; // any
let bar = []; // never[]
//bar.push(1); 型エラー

let hoge: number[] = [];
hoge.push(1);

```

演習: TypeScriptに慣れる

- 演習資料 Section2を写経してみよう
- 他にも以下のことを試してみよう
  - 型と違う値を入れようとしたらどうなるか
  - 複数の型の要素を1つの配列にいれたらどういう型に推論されるか
  - dist以下の生成されたJSとsrcのTSを比較してみよう

DOMとDOM API

- HTMLをツリー構造として表現、操作することができるAPIとその仕様
- HTMLを変更して画面を書き換えることが多かったJSとは関係が深い
- DOM APIの例
  - getElementById
  - createElement
  - addEventListener



イベント

- Webフロントエンドでは画面上やプログラム内で起きたイベントをトリガーに処理を実行したいことが多い
- イベントの例
  - HTMLの読み込みが終わった
  - ボタンがクリックされた
  - ユーザーがキーボードのキーを押した

addEventListener

```
1 element.addEventListiner("click", () => {
2   alert("Clickされたよ!");
3 })
```

- ある要素に対して特定のイベントが起きたときの処理をコールバック関数で指定する
- ちなみにaddEventListenerに限らず、関数に別のコールバック関数を渡すのは頻出

演習: DOM APIを使ってみる

- HTMLファイルを用意し、コンパイルして生成されたJSを読み込む
- HTML上のボタンがクリックされたらbodyタグに新しくdivタグが追加されるようにする

非同期処理

- 応答に時間がかかる処理などを行う際には非同期処理をすることが多い
  - Web APIの呼び出し、ファイルアップロード、ファイルシステムへのアクセス
- 同期的に処理をすると応答を待つ間プログラムの実行がブロックされるなど、ユーザーの体験を損う

```
1  console.log("start");
2  setTimeout(() => {
3   console.log("in setTimeout");
4  }, 1000);
5  console.log("finish");
6
7  // start
8  // finish
9  // in setTimeout
```

非同期処理の方法

- コールバック
- Promise
- Async Function(async/await)
- 上記3つそれぞれについてWeb APIを叩くコードを書きながら学習していきます

httpモジュール

- Node.jsの標準モジュールの1つ
  - 若干低レベルなインターフェースなのであまり使い勝手はよくない
- HTTPサーバーやクライアントとしての機能を提供

APIサーバー

- Sinatraで簡単なAPIサーバーを用意
- エンドポイントは以下の2つのみ
  - /users
  - /users/:id/todos

演習: httpモジュールでAPIを呼ぶ

- Section 4、5の2つをやってください
- 細かい構文のことよりhttp.getの第2引数のコールバック関数に着目してください

非同期処理のネストは辛い

- ある非同期処理を完了したら更に別の非同期処理を行いたい時はままある
- 今回は2段階のネストだけどこれが3段階、4段階だったら…?

Promise

- Promiseとは非同期処理を抽象化しオブジェクトとして扱えるようにしたもの
  - 厳密に言うとPromiseという非同期処理のデザインパターンをJSに組み込んだもの
- Promiseの導入により、非同期処理のインターフェースを統一できる
  - コールバック関数の場合、コールバック関数の引数が使用などで決まっているわけではない

Promise(2)

```
new Promise((resolve, reject) => {
    // 非同期処理を行う
    resolve("success")
}).then((value) => {
    // resolveが呼ばれた場合
    // valueはresolveに与えらた値
    console.log(value)
}).catch((value) => {
    // rejectが呼ばれた場合
    // valueはrejectに与えられた値
    console.log(value)
})

//=> success
```

演習: Promiseでコードを書き直してみよう

- Section 6のコードを書いて実行してみよう
- 今回はAPIがエラーを吐いたときの対応も入れてあります
  - rejectの感じを掴んで欲しい

Fetch API

- XMLHTtpRequestの代わりとなる非同期通信のためのAPI
  - IE対応が必要の場合はPolyfillが必要
- Node.jsも対応していないのでNode.js環境ではnode-fetchモジュールが必要
  - 今回はnode-fetchモジュールが必要
- fetchはPromiseのインスタンスを返すので、そのままthen/catchなどが使える
  - httpの場合、Promiseインスタンスを自分で作る必要があった

演習: Fetch APIを使う

- Section 7のコードを書いて実行してみよう
  - httpをPromiseでラップしてた時にくらべて大分ラクなはず

Async Function(async/await)

- Async Functionは非同期処理を行うための関数を定義する構文
- 現在でもっともモダンな書き方
- Async Functionとして定義された関数は必ずPromiseのインスタンスを返す
    - つまり構文が違うだけで中身的にはPromiseなのでPromiseの理解が重要

演習: Async Functionで書いてみる

- Section 8をやってみる
- asyncと関数定義に書かれているのがPromiseを返す関数
  - つまり非同期に実行される
- awaitを付けて呼ぶとそこで非同期処理の完了を待つ