**はじめに**

TypeScriptでシングルページアプリケーションを作成する方法を説明します。対象読者はJavaScriptを使用して開発を行った経験がある方です。

**本書の構成**

第1〜第3章で、開発環境の構築から、代表的なTypeScriptの機能を確認します。そして第4章で、実際にシングルページアプリケーションを作成します。

なお、本文中のソースコードは以下のリポジトリで公開しています。

URL: https://github.com/jsuzuki20120311/start-typescript-note

**検証環境**

本文中のソースコードは以下の2台のマシンで動作検証を行っています。

OS: mac OS Sierra バージョン 10.12.3

MacBook Pro (13-inch, Mid 2010)

プロセッサ: 2.66 GHz Intel Core 2 Duo

メモリ 8 GB

OS: ubuntu 16.04 LTS 64ビット

プロセッサ: Intel Core i5-3230M

メモリ 4 GB

また、Node.jsとnpmは、いずれのマシンでも以下のバージョンを使用しています。

Node.js: v6.10.0

npm: 3.10.0

目次

1章 環境構築 3

1.1 Node.jsのインストール 3

1.2 TypeScriptのインストール 4

1.3 エディタについて 4

2章 Hello World 6

2.1 ワーキングディレクトリの作成 6

2.2 helloworld.tsの作成 6

2.3 トランスパイル 6

2.4 実行 6

3章 TypeScript 8

3.1 インタフェース 8

3.2 型アノテーション 9

3.3 アクセス修飾子 10

3.4構造的部分型 11

4章 シングルページアプリケーションの作成 14

4.1 データベースの準備 17

4.2 サーバーサイドの開発 17

4.3 フロントエンドの開発 39

# 1章 環境構築

環境構築を行います。1章の内容に関しては、自身の端末のインストール済プログラムや環境変数などと比較しながら適宜読み替えてください。

## 1.1 Node.jsのインストール

まずNode.jsをインストールします。Node.jsはサーバーサイドのJavaScript実行環境です。本書ではNode.jsを使用してTypeScriptからJavaScriptへのトランスパイルやサーバーサイドのプログラムの実行を行います。

Node.jsのインストールにはnodebrewの使用をおすすめします。nodebrewはNode.jsのバージョン管理ツールです。Node.jsはバージョンアップの頻度が高く、バージョンの変更の度にアンインストールとインストール作業を繰り返すのは非常に手間がかかります。nodebrewを使用することで、この手間を削減できます。

ターミナルでリスト1.1.1のコマンドを実行しnodebrewをインストールします。

リスト1.1.1

$ curl -L git.io/nodebrew | perl - setup

nodebrewのインストールが完了したら、nodebrewのパスを通します。bashを使用している場合、「.bashrc」にファイルに以下のコードを追記します。

リスト1.1.2

export PATH=$HOME/.nodebrew/current/bin:$PATH

nodebrewのパスを通したら、nodebrewを使用してNode.jsをインストールします。今回は執筆時点でLTSバージョンである6.10.0をインストールします。

リスト1.1.3

$ nodebrew install-binary v6.10.0

正しくNode.jsのv6.10.0がインストールされたかどうか確認します。

リスト1.1.4

$ nodebrew list

これはnodebrewからインストールされたNode.jsのバージョンを一覧表示するコマンドです。インストールが完了していれば、v6.10.0が表示されます。

次に使用するNode.jsのバージョンを指定します。今回は6.10.0を使用するので、以下のコマンドを実行します。

リスト1.1.4

$ nodebrew use v6.10.0

Node.jsのv6.10.0を使用するよう指定されたことを、以下のコマンドで確認します。

リスト1.1.5

$ node --version

正しく指定されていれば『v6.10.0』と表示されます。

## 1.2 TypeScriptのインストール

TypeScriptをインストールします。TypeScriptのインストールにはnpmコマンドを使用します。npmはNode.jsと共にインストールされるモジュール管理ツールです。

リスト1.2.1

$ npm install typescript –global

リスト1.2.2.のコマンドTypeScriptのバージョンを表示し、インストールが完了したか確認します。

リスト1.2.2

$ tsc –version

　なお、本書中のソースコードはTypeScriptのバージョンは2.2.1を使用して動作確認をしています。

## 1.3 エディタについて

エディタについては使い慣れたエディタがあればそれを使用するのが一番ですが、特にこだわりが無い場合、あるいは使い慣れたエディタがTypeScriptに対応していない場合はVisualStudioCodeの使用をおすすめします。

VisualStudioCodeはTypeScript開発元と同じMicrosoft製のエディタです。Microsoft製だけあり、特にプラグインを導入することなくTypeScriptに対応していおり、Windows版だけでなくMac版、Linux版もリリースされていおり、以下のURLからダウンロードできます。

URL: https://code.visualstudio.com/download

# 2章 Hello World

新たにプログラミング言語を学ぶ際や、プログラミングをはじめる環境が整ったことを確認する際には伝統的に『Hello World』という文字列を表示するプログラムを作成します。ここでもそれに習い、ハローワールドプログラムを作ります。

## 2.1 ワーキングディレクトリの作成

適当な場所にワーキングディレクトリを作成します。今回はディレクトリ名を『chapter-2』としす。ディレクトリを作成したら、『helloworld.ts』ファイルを作成します。

## 2.2 helloworld.tsの作成

作成した『helloworld.ts』ファイルにリスト2.2のコードを書き保存します。

リスト2.2

console.log('Hello World');

## 2.3 トランスパイル

コードを書けたら、TypeScriptからJavaScriptへの変換を行います。この変換はコンパイルやトランスパイルあるいはトランスコンパイルなどと呼ばれます。本書ではトランスパイルに統一します。

作成した『chapter-2』ディレクトリに移動しリスト2.3のコマンドを実行します。

リスト2.3

$ tsc helloworld.ts

正常に完了したら、ディレクトリ内に『helloworld.js』ファイルが作成されていることを確認します。この『helloworld.js』がトランスパイルされた結果であり、実際に実行されるコードが書かれたファイルです。

## 2.4 実行

正常にトランスパイルが完了したら、実際に実行します。実行するにはリスト2.4のコマンドを実行します。実行するのはトランスパイル後のファイルなので、指定するのは『helloworld.ts』ではなく『helloworld.js』である点に注意します。

リスト2.4

$ node helloworld.js

ターミナルに『Hello World』が表示されたら成功です。

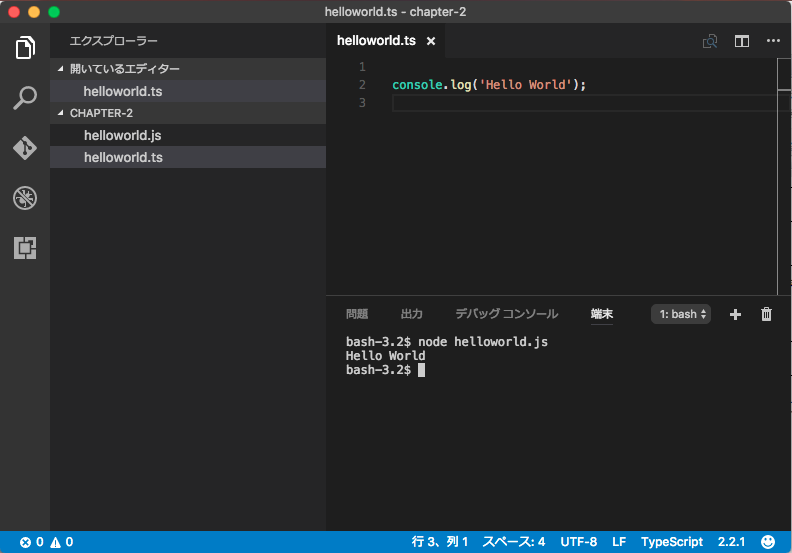


図2.4 VisualStudioCodeのターミナル上での実行例

# 3章 TypeScript

実際にプログラミングを始める段階で、使用する言語のすべてを把握する必要は無いでしょう。しかし、その代表的な機能を知っておくことは、学習を進めるうえで間違いなく助けになります。

TypeScriptの言語仕様は、EcmaScript5の仕様に以下の3つの機能を追加したものです。

* EcmaScript2015由来の機能
  + class 構文
  + let, const
  + ラムダ式

etc…

* EcmaScript2016由来の機能
  + デコレータ
  + async/await構文
* TypeScript独自の機能
  + インタフェース
  + 型アノテーション
  + アクセス修飾子

etc…

ここでは、3番目のTypeScript独自の機能を中心に確認していきます。

## 3.1 インタフェース

EcmaScript5やEcmaScript2015に存在しない機能としてインタフェースがあります。インタフェースにはプロパティとメソッドの定義のみができ、実装することはできません。たとえば、以下のRunnableインタフェースは string型のnameプロパティと、booleanを返すrunメソッドが定義されています。

リスト3.1.1 Runnable.ts

interface Runnable {  
  
 name: string;  
  
 run(): boolean;  
}

上記Runnableインタフェースを実装したTaskクラスを作る場合、以下ようにimplementsキーワードを使用します。

implementsキーワードを使用しインタフェースを指定することで、そのクラスにインタフェースで定義したプロパティとメソッドの実装を強制させることができます。インタフェースで定義したプロパティとメソッドがどれか1つでも実装されていない場合トランスパイルでエラーが発生します。

リスト3.1.2 Task.ts

class Task implements Runnable {  
  
 name: string  
  
 constructor(name: string) {  
 this.name = name;  
 }  
  
 run(): boolean {  
 let result = false;  
 // do something  
 return result;  
 }  
}

## 3.2 型アノテーション

TypeScriptでは、型アノテーションを使用して、変数に静的に型を指定することができます。以下のように型アノテーションは変数名の後に指定し、間に :(コロン) をつけます。

リスト3.2.1

let hoge: number = 1;

また、クラスのプロパティに指定する場合、リスト3.2.2の3行目のようにします。

リスト3.2.2

class Hoge {

**private fuga: string;**

constructor(fuga: string) {

this.fuga = fuga;  
 }  
}

## 3.3 アクセス修飾子

TypeScriptでは、public、protected、privateのアクセス修飾子が使用できます。なお、何も指定しない場合、publicとなります。

* public
  + 外部のクラスからもアクセス可能。
* protected
  + クラス内とそのクラスのサブクラスからのみアクセスが可能。
* private
  + クラス内からのみアクセスが可能。

また、TypeScriptのバージョン2以降では、以下のようにコンストラクタに private や protected が指定できるようになりました。これにより、より安全に Sigleton パターンが使用できるようになりました。

リスト3.3 コンストラクタにprivate修飾子を指定したSingletonクラス

class Singleton {  
  
 private static instance: Singleton;  
  
 private constructor() {  
 // do nothing  
 }  
  
 public static getInstance(): Singleton {  
 if (!Singleton.instance) {  
 Singleton.instance = new Singleton();  
 }  
 return Singleton.instance;  
 }  
  
}

なお、インタフェースで定義したプロパティとメソッドは自動的にpublicとなります。実装しているクラスでprotectedやprivateを指定するとトランスパイルエラーが発生します。

## 3.4構造的部分型

TypeScriptには、JavaやC#のようなクラスベースのオブジェクト指向プログラミング言語とは異なる考え方に、構造的部分型があります。

構造的部分型とは、型の派生関係をextendsやimplementsからではなく、オブジェクトの構造から判断する仕組みのことです。

例えばJavaにおいて以下のような異なる2つの、但し構造は同じインタフェースがあり、それぞれに実装したクラスがあったとします。

リスト3.4.1 Flyable.java

public interface Flyable {  
 void fly();  
}

リスト3.4.1 Dragon.java

public interface Dragon {  
 void fly();  
}

リスト3.4.2 Raven.javaa

public class Raven implements Flyable {  
 public void fly() {  
 System.out.println("fly");  
 }  
}

リスト3.4.3 BlueDragon.java

public class BlueDragon implements Dragon {  
 public void fly() {  
 System.out.println("fly");  
 }  
}

その上で、リスト3.4.4の処理を書いた場合、9行目でコンパイルエラーが発生します。

リスト3.4.4 Main.java

public class Main {  
  
 public static void main(String[] args) {  
 Main main = new Main();  
  
 Flyable raven = new Raven();  
 Dragon blueDragon = new BlueDragon();  
 main.register(raven);  
 main.register(blueDragon);  
 }  
  
 private void register(Flyable flyable) {  
 // do something  
 }  
}

それはBlueDragonクラスがFlyableインタフェースを実装しておらず、Javaが構造ではなくextends や implements キーワードを元に型の派生関係を判断しているためです。

一方で、TypeScriptで似たようなコードを書いた場合トランスパイルエラーが発生しません。

リスト3.4.5 Flyable.ts

interface Flyable {  
  
 fly(): void;  
}  
  
export default Flyable;

リスト3.4.6 Dragon.ts

interface Dragon {

fly(): void;  
}  
  
export default Dragon;

リスト3.4.7 Raven.ts

import Flyable from './Flyable';  
  
class Raven implements Flyable {  
  
 fly(): void {  
 console.log('fly');   
 }  
}

export default Raven;

リスト3.4.8 BlueDragon.ts

import Dragon from './Dragon';  
  
class BlueDragon implements Dragon {  
  
 fly(): void {  
 console.log('fly');   
 }  
}  
  
export default BlueDragon;

Javaのケースでコンパイルエラーが起きた行に相当する17行目があっても、TypeScriptではトランスパイルエラーは発生しません。これは構造が同じためです。

リスト3.4.9 Main.ts

import BlueDragon from './models/BlueDragon';  
import Raven from './models/Raven';  
import Flyable from './models/Flyable';  
  
class Main {  
  
 register(flyable: Flyable): void {  
 flyable.fly();  
 }  
}  
  
let main = new Main();  
  
let raven = new Raven();  
let blueDragon = new BlueDragon();  
main.register(raven);  
main.register(blueDragon);

# 4章 シングルページアプリケーションの作成

2章でごく簡単なプログラムをTypeScriptを用いて作成し、3章でTypeScriptの代表的な機能を確認しました。4章では以下のような機能を持った記事管理シングルページアプリケーションを作成します。

* 記事の**投稿**
* 記事の**閲覧**
* 記事の**更新**
* 記事の**削除**

4章ではサーバーサイド、フロントエンドそれぞれに以下のフレームワークを使用します。

* Express
  + Node.js用Webアプリケーションフレームワークです。
* Angular2
  + Googleが開発を主導しているフロントエンドWebアプリケーションフレームワークです。

なお、本書ではワーキングディレクトリとして『chapter-4』という名前のディレクトリ下で作業する前提で進めます。　さらに、『chapter-4』ディレクトリの下に『client』ディレクトリと『server』ディレクトリの、2つのディレクトリを作成します。これはそれぞれ、フロント側のプロジェクト、サーバー側のプロジェクト用のディレクトリとなります。

chapter-4/

　├ client/ フロント側プロジェクト用ディレクトリ

　└ server/ サーバー側プロジェクト用ディレクトリ

図4.1〜4.4は完成形の画面イメージです。

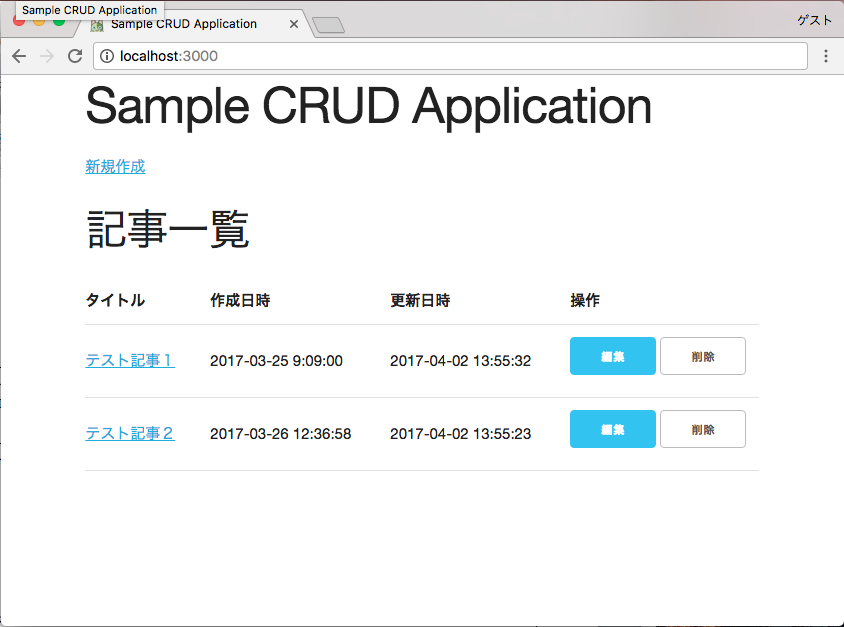


図4.1 画面イメージ 記事一覧画面



図4.2画面イメージ記事 内容確認画面



図4.3画面イメージ記事 記事更新画面



図4.4画面イメージ記事　記事更新画面

## 4.1 データベースの準備

本書ではMySql5.7の使用を前提に進めます。また、MySQLのインストール手順や設定手順は割愛します。MySQLのインストールが完了したら、リスト4.1のSQL文を実行しarticleテーブルを作成します。

リスト4.1

create table article

(

id int not null auto\_increment,

title varchar(2000) not null,

body longtext null,

created\_at datetime null,

updated\_at datetime null,

constraint `PRIMARY`

primary key (id),

constraint article\_id\_uindex

unique (id)

);

## 4.2 サーバーサイドの開発

サーバーサイドのアプリケーションは以下の3階層の構成で、APIを作成します。

* コントローラクラス
  + 個々のリクエストに対してレスポンスを返すクラスです。今回の場合内部でサービスクラスのインスタンスを保持し、そのインスタンスの処理結果でもってレスポンスの内容を操作します。
* サービスクラス
  + 実処理を行います。内部でDaoのインスタンスを作り、Daoインスタンスを通してDBとのやりとりや行います。
* Daoクラス
  + データベースへのアクセスを行います。

サーバーサイドの開発を進めるにあたって、『server』ディレクトリ直下の構成は以下のようにします。ディレクトリに関しては今の段階ですべて作ってしまうことをおすすめします。

chapter-4/

　├ client/ フロント側プロジェクト用ディレクトリ

　└ server/ サーバー側プロジェクト用ディレクトリ

　　　　├ config/ 設定保持系のクラス

　　　　├ controllers/ コントローラ用ディレクトリ

　　　　├ dao/ Daoクラス用ディレクトリ

　　　　├ json/ jsonファイル置き場

　　　　├ models/ Modelクラス用ディレクトリ

　　　　├ node\_modules/ npmモジュール用ディレクトリ

　　　　├ public/ 静的フファイル置き場

　　　　├ routes/ ルーティング用ディレクトリ

　　　　├ services/ サービスクラス用ディレクトリ

　　　　├ views/ テンプレート用ディレクトリ

　　　　├ www/ Webサーバー起動処理用ディレクトリ

　　　　├ app.ts Expressアプリケーション設定処理

　　　　├ tsconfig.json

　　　　└ package.json

### 4.2.1 tsconfig.jsonの作成

まず『tsconfig.json』という名前のファイルをserverディレクトリに作成します。これはTypeScriptで記述したファイルをトランスパイルする際のオプションなどを設定するファイルです。ファイルに記述する内容ですが、今回が以下の内容で作成します。

リスト4.2.1

{  
 "compilerOptions": {  
 "target": "ES5",  
 "module": "commonjs",  
 "moduleResolution": "node",  
 "sourceMap": true,  
 "emitDecoratorMetadata": true,  
 "experimentalDecorators": true,  
 "lib": ["es2015"],  
 "noImplicitAny": true,  
 "suppressImplicitAnyIndexErrors": true,  
 "typeRoots": [  
 "./node\_modules/@types"  
 ],  
 "alwaysStrict": true  
 },  
 "compileOnSave": true,  
 "exclude": [  
 "node\_modules"  
 ]  
}

### 4.2.2 package.jsonの作成とモジュールのインストール

次にpackage.jsonを設定します。全くの新規で作成する場合、リスト4.2.2. 1のコマンドを実行しますが、今回は予め必要なモジュールを指定したファイルを作成します。リスト4.2.2.1は実行せず、リスト4.2.2.2の内容でpackage.jsonという名前でファイルを新規作成します。

リスト4.2.2.1

$ npm init -y

リスト4.2.2.2 package.json

{

"name": "sample-crud-application-server",

"version": "1.0.0",

"main": "./www/index.js",

"scripts": {

"start": "node ./www/index.js NODE\_ENV=production",

"dev": "tsc --watch & node-dev ./www/index.js",

"build": "tsc",

"watch": "tsc --watch"

},

"license": "MIT",

"dependencies": {

"body-parser": "^1.15.2",

"ejs": "^2.5.6",

"express": "^4.14.0",

"mysql": "^2.11.1",

"serve-favicon": "^2.4.0"

},

"devDependencies": {

"@types/body-parser": "0.0.34",

"@types/ejs": "^2.3.33",

"@types/express": "^4.0.33",

"@types/mysql": "0.0.31",

"@types/serve-favicon": "^2.2.28",

"node-dev": "^3.1.3",

"typescript": "^2.2.1"

}

}

package.jsonを作成したら、リスト4.2.2.3のコマンドを実行し、必要なnpmモジュールをインストールします。

リスト4.2.2.3

$ npm install

リスト4.2.2.2のコマンドを実行すると、package.jsonのdependenciesとdevDependenciesで管理されているnpmモジュールがインストールされます。

### 4.2.3 実装

まずMySQLに接続するための設定を保持するためのファイルを作ります。今回は設定をjsonディレクトリ内にdb\_config.jsonというファイル名でJSONファイルを作成します。ファイルを作成したら以下のように、host、port、user、password、databaseに、自身のデータベースの接続情報を記述し保存します。

chapter-4/

　├ client/ フロント側プロジェクト用ディレクトリ

　└ server/ サーバー側プロジェクト用ディレクトリ

　　　　├ config/

│ └ DbConfigManager.ts //

// 省略

│

　　　　├ json/

│ └ db\_config.json // データベースへの接続設定ファイル

// 省略

リスト4.2.3.1 server/json/db\_confoig.json

{

"host": "localhost",

"port": 3306,

"user": "root",

"password": "hogehoge",

"database": "sample\_crud\_app\_db"

}

データベース接続情報のファイルを作成したら、次に接続情報の設定を保持するためのクラスを作成します。

リスト4.2.3.2 server/config/DbConfigManager.ts

import \* as mysql from 'mysql';

import \* as fs from 'fs';

export class DbConfigManager {

private static dbConfig: mysql.IConnectionConfig;

static initialize(): void {

const data = fs.readFileSync('./json/db\_config.json', 'utf-8');

DbConfigManager.dbConfig = JSON.parse(data);

}

static getConfig(): mysql.IConnectionConfig {

if (!DbConfigManager.dbConfig) {

throw new Error('Did not initialized!');

}

return DbConfigManager.dbConfig;

}

}

次にWebサーバー起動処理を書いていきます。『www』ディレクトリ内に、リスト4.1.3の内容でindex.tsファイルを作成します。このindex.ts内では、4行目でデータベース接続設インスタンスの初期化も行っています。

chapter-4/

　├ client/ フロント側プロジェクト用ディレクトリ

　└ server/ サーバー側プロジェクト用ディレクトリ

// 省略

　　　　├ www/

│ └ index.ts // Webサーバー起動処理

// 省略

リスト4.2.3.3 server/www/index.ts

import \* as http from 'http';

import app from '../app';

import { DbConfigManager } from '../config/DbConfigManager';

DbConfigManager.initialize();

const server = http.createServer(app);

const port = normalizePort(process.env.PORT || '3000');

server.listen(port);

server.on('error', onError);

server.on('listening', onListening);

/\*\*

\* Normalize a port into a number, string, or false.

\*/

function normalizePort(val: string) {

const port = parseInt(val, 10);

if (isNaN(port)) {

// named pipe

return val;

}

if (port >= 0) {

// port number

return port;

}

return false;

}

/\*\*

\* Event listener for HTTP server "error" event.

\*/

function onError(error: any) {

if (error.syscall !== 'listen') {

throw error;

}

const bind = typeof port === 'string'

? 'Pipe ' + port

: 'Port ' + port;

// handle specific listen errors with friendly messages

switch (error.code) {

case 'EACCES':

console.error(bind + ' requires elevated privileges');

process.exit(1);

break;

case 'EADDRINUSE':

console.error(bind + ' is already in use');

process.exit(1);

break;

default:

throw error;

}

}

/\*\*

\* Event listener for HTTP server "listening" event.

\*/

function onListening() {

const addr = server.address();

const bind = typeof addr === 'string'

? 'pipe ' + addr

: 'port ' + addr.port;

console.log(addr + bind);

}

次に、リスト4.2.3.3の2行目で参照しているapp.tsをリスト4.1.3.4の内容で作成します。

chapter-4/

　├ client/ フロント側プロジェクト用ディレクトリ

　└ server/ サーバー側プロジェクト用ディレクトリ

// 省略

　　　　└ app.ts

リスト4.2.3.4 server/app.ts

import \* as bodyParser from 'body-parser';

import 'ejs';

import \* as express from 'express';

import \* as path from 'path';

import \* as favicon from 'serve-favicon';

import api from './routes/api';

import index from './routes/index';

/\*\*

\* Expressアプリケーションオブジェクト

\* @type {Express}

\*/

const app = express();

// ファビコンを設定

app.use(favicon(\_\_dirname + '/public/favicon.png'));

// テンプレートエンジンに ejs を使用するための設定

app.set('views', \_\_dirname + '/views');

app.set('view engine', 'ejs');

app.use(bodyParser.urlencoded({ extended: true }));

app.use(bodyParser.json());

// 静的ファイルのルーティング

app.use(express.static(path.join(\_\_dirname, 'public')));

app.use('/', index);

app.use('/api', api);

// 404のルーティング

app.use((req, res, next) => {

const err = {

status: 404,

message: 'Not Found.'

};

next(err);

});

// エラーハンドラ

app.use((err: any, req: express.Request, res: express.Response, next: express.NextFunction) => {

err.status = err.status || 500;

res.status(err.status);

if (req.xhr) {

res.send({ status: err.status, message: err.message });

} else {

res.render('error', { status: err.status, message: err.message });

}

});

export default app;

### 4.2.2 ルーティング

app.tsの作成が完了したら次にルーティングの設定を行います。ここでは以下の機能を実装します。

* 以下のURLでは常にpublicディレクトリ直下の『index.html』を返す。  
  この設定はフロント側でのルーティングと対応させるための設定です。
  + http://ドメイン:ポート番号/
  + http://ドメイン:ポート番号/createArticle/
  + http://ドメイン:ポート番号/updateArticle/ + 記事データのID
  + http://ドメイン:ポート番号/viewArticle/ + 記事データのID
* 以下のURLで各APIを呼び出します。  
  ソースコード上は、それぞれコントローラの対応するメソッドを呼び出しています。
  + http://ドメイン:ポート番号/api/v1/article/
    - PPOSTメソッド

記事データ一覧を返す。

* + http://ドメイン:ポート番号/api/v1/article/ + 記事データのID + .json
    - GETメソッド

記事データを返す

* + - PUTメソッド

記事データを更新する。

* + - DELTEメソッド

記事データを削除する。

以上の処理を、URLの階層構造と同じように以下の構成で、ディレクトリとTypeScriptファイルを作成します。ルーティングの設定にはExpress.Routerのget, post, put, delete などのメソッドとuseメソッドを使用します。

chapter-4/

　├ client/ フロント側プロジェクト用ディレクトリ

　└ server/ サーバー側プロジェクト用ディレクトリ

// 省略

　　　　├ routes/

│ ├ api/

│ │ ├ v1/

│ │ │ ├ article.ts

│ │ │ └ index.ts

│ │ └ index.ts

│ └ index.ts

// 省略

リスト4.2.2.1 server/routes/index.ts

import \* as express from 'express';

import \* as path from 'path';

const index = express.Router();

const sendIndexHtml = (req: express.Request, res: express.Response) => {

res.sendFile(path.join(\_\_dirname, '../', 'public', 'index.html'));

};

index.get('/', sendIndexHtml);

index.get('/createArticle', sendIndexHtml);

index.get('/updateArticle/:id', sendIndexHtml);

index.get('/viewArticle/:id', sendIndexHtml);

export default index;

リスト4.2.2.2 server/routes/index.ts

import \* as express from 'express';

import v1 from './v1';

const api = express.Router();

api.use('/v1', v1);

export default api;

リスト4.1.2.3 server/routes/index.ts

import \* as express from 'express';

import article from './article';

const v1 = express.Router();

v1.use('/article', article);

export default v1;

リスト4.2.2.4 server/routes/index.ts

import \* as express from 'express';

import { ArticleController } from '../../../controllers/ArticleController';

const article = express.Router();

article.get('/', (req, res, next) => {

const articleController = new ArticleController();

articleController.index(req, res, next);

});

article.post('/', (req, res, next) => {

const articleController = new ArticleController();

articleController.create(req, res, next);

});

article.get('/all.json', (req, res, next) => {

const articleController = new ArticleController();

articleController.all(req, res, next);

});

article.get('/count.json', (req, res, next) => {

const articleController = new ArticleController();

articleController.count(req, res, next);

});

article.get('/:id\.json', (req, res, next) => {

const articleController = new ArticleController();

articleController.read(req, res, next);

});

article.put('/:id\.json', (req, res, next) => {

const articleController = new ArticleController();

articleController.update(req, res, next);

});

article.delete('/:id\.json', (req, res, next) => {

const articleController = new ArticleController();

articleController.delete(req, res, next);

});

export default article;

### 4.2.3モデルの作成

次はモデルを作成します。ここで言うモデルとは、デーテベースに格納される1つ１つのレコードの各カラムの属性を持つクラスまたはインタフェースのことを指します。所謂ビジネスロジックのことではなく、また、データベースへアクセスするための機能も持ちません。

また、今回はサンプル・アプリケーション全体で使用するAppErrorクラスもmodelsディレクトリに置いてしまいます。他にもエラー系のクラスが増えるようであればerrorsやexceptionsなどの名前でディレクトリを作成し、そこに置くべきでしょう。

リスト4.2.3.1 server/models/Article.ts

/\*\*

\* 記事インタフェース

\*/

export interface Article {

title: string;

body: string;

}

リスト4.2.3.2 server/models/RegisteredArticle.ts

import { Article } from "./Article";

import { RegisteredItem } from "./RegisteredItem";

/\*\*

\* 登録済記事インタフェース

\*/

export interface RegisteredArticle extends Article, RegisteredItem {

}

リスト4.2.3.3 server/models/RegisteredArticle.ts

/\*\*

\* 登録済情報インタフェース

\*/

export interface RegisteredItem {

id: number,

createdAt: string,

updatedAt: string

}

### 4.2.4 エラーページ用テンプレートの作成

エラーページ用のテンプレートを作成します。今回はテンプレートエンジンにEJSを使用します。サンプルアプリのエラーページなので、単純にステータスコードとエラーメッセージを表示するだけの簡素なものを作ります。

リスト4.2.5 server/views/error.ejs

<!doctype html>

<html lang="ja">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>ERROR | <%-status%></title>

</head>

<body>

<div>

<%-status%> <%-message%>

</div>

</body>

</html>

### 4.2.5 Daoクラス

次はDaoクラスを作成します。今回使用するテーブルは一つだけなので、1ファイルで済ませてしまいます。基底クラスやインタフェースも作成しません。

リスト4.2.5 server/dao/ArticleDao.ts

import \* as mysql from 'mysql';

import { Article } from '../models/Article';

import { RegisteredArticle } from '../models/RegisteredArticle';

import { AppError } from "../models/AppError";

/\*\*

\* 記事データ用Daoクラス

\*/

export class ArticleDao {

/\*\*

\* データベースコネクション

\*/

private connection: mysql.IConnection;

/\*\*

\* コンストラクタ

\* @param {mysql.IConnection} connection データベースコネクション

\*/

constructor(connection: mysql.IConnection) {

this.connection = connection;

}

insertArticle(article: Article): Promise<number> {

const query = 'insert into article (title, body, created\_at, updated\_at) values (?, ?, now(), now())';

const param = [

article.title,

article.body

];

return new Promise<number>((resolve, reject) => {

this.connection.query(query, param, (error, result) => {

if (error) {

reject(error);

return;

}

resolve(result.insertId);

});

});

}

selectAllArticles(): Promise<RegisteredArticle[]> {

return new Promise<Article[]>((resolve, reject) => {

const query = 'select' +

' id' +

' ,title' +

' ,body' +

' ,DATE\_FORMAT(created\_at, \'%Y-%m-%d %k:%i:%s\') as createdAt' +

' ,DATE\_FORMAT(updated\_at, \'%Y-%m-%d %k:%i:%s\') as updatedAt' +

' from article' +

' order by id';

this.connection.query(query, [], (error, results) => {

if (error) {

reject(error);

return;

}

resolve(results);

});

});

}

selectCount(): Promise<number> {

return new Promise<number>((resolve, reject) => {

const query = 'select count(id) as count from article';

this.connection.query(query, [], (error, results) => {

if (error) {

reject(error);

return;

}

if (typeof results[0].count !== 'number') {

throw new TypeError();

}

resolve(results[0].count);

});

});

}

selectArticles(offset: number = 0, limit: number = 0): Promise<RegisteredArticle[]> {

return new Promise<Article[]>((resolve, reject) => {

const query = 'select' +

' id' +

' ,title' +

' ,body' +

' ,DATE\_FORMAT(created\_at, \'%Y-%m-%d %k:%i:%s\') as createdAt' +

' ,DATE\_FORMAT(updated\_at, \'%Y-%m-%d %k:%i:%s\') as updatedAt' +

' from article' +

' order by id' +

' limit ?, ?';

this.connection.query(query, [offset, limit], (error, results) => {

if (error) {

reject(error);

return;

}

resolve(results);

});

});

}

selectArticleById(id: number): Promise<RegisteredArticle[]> {

return new Promise<RegisteredArticle[]>((resolve, reject) => {

const query = 'select ' +

' id' +

' ,title' +

' ,body' +

' ,DATE\_FORMAT(created\_at, \'%Y-%m-%d %k:%i:%s\') as createdAt' +

' ,DATE\_FORMAT(updated\_at, \'%Y-%m-%d %k:%i:%s\') as updatedAt' +

' from article' +

' where id = ?';

this.connection.query(query, [id], (error, results) => {

if (error) {

reject(error);

return;

}

if (!Array.isArray(results) || results.length === 0) {

const appError = new AppError('Article data is not found.');

appError.status = 404;

reject(appError);

return;

}

resolve(results);

});

});

}

updateArticle(id: number, article: Article): Promise<any> {

return new Promise<any>((resolve, reject) => {

const query = 'update article ' +

'set title = ? ' +

', body = ? ' +

', updated\_at = now() ' +

'where id = ?';

const params = [

article.title,

article.body,

id

];

this.connection.query(query, params, (error, results) => {

if (error) {

reject(error);

return;

}

resolve(results);

});

});

}

deleteArticle(id: number): Promise<any> {

return new Promise<void>((resolve, reject) => {

const query = 'DELETE FROM article WHERE ID = ?';

this.connection.query(query, [id], (error, result) => {

if (error) {

reject(error);

return;

}

resolve(result);

});

});

}

lock(id: number): Promise<RegisteredArticle> {

return new Promise<RegisteredArticle>((resolve, reject) => {

const query = 'select ' +

' id ' +

' ,title' +

' ,body' +

' ,DATE\_FORMAT(created\_at, \'%Y-%m-%d %k:%i:%s\') as createdAt' +

' ,DATE\_FORMAT(updated\_at, \'%Y-%m-%d %k:%i:%s\') as updatedAt' +

' from article where id = ? for update';

this.connection.query(query, [id], (error, results) => {

if (error) {

reject(error);

return;

}

resolve(results);

});

});

}

}

### 4.2.6 サービスクラス

次にサービスクラスを作成します。これもDaoクラス同様1クラスのみ作成します。

リスト4.2.6 server/dao/ArticleService.ts

import \* as mysql from 'mysql';

import { DbConfigManager } from '../config/DbConfigManager';

import { ArticleDao } from '../dao/ArticleDao';

import { RegisteredArticle } from '../models/RegisteredArticle';

import { Article } from "../models/Article";

import { AppError } from "../models/AppError";

/\*\*

\* 記事データに関するロジック

\* @export

\* @class ArticleService

\*/

export class ArticleService {

/\*\*

\* 登録済みの全ての記事データを取得します。

\* @returns {Promise<RegisteredArticle[]>}

\*/

async findAllArticles(): Promise<RegisteredArticle[]> {

const connection = mysql.createConnection(DbConfigManager.getConfig());

connection.connect();

const articleDao = new ArticleDao(connection);

try {

return await articleDao.selectAllArticles();

} catch(error) {

throw error;

} finally {

connection.destroy();

}

}

/\*\*

\* 第一引数と第二引数で指定された登録済記事データを取得します。

\* @param {number} offset

\* @param {number} limit

\* @returns {Promise<RegisteredArticle[]>}

\*/

async findArticles(offset: number, limit: number): Promise<RegisteredArticle[]> {

const connection = mysql.createConnection(DbConfigManager.getConfig());

connection.connect();

const articleDao = new ArticleDao(connection);

try {

return await articleDao.selectArticles(offset, limit);

} catch(error) {

throw error;

} finally {

connection.destroy();

}

}

/\*\*

\* 登録されている記事の件数を取得します。

\* @returns {Promise<number>}

\*/

async findArticleCount(): Promise<number> {

const connection = mysql.createConnection(DbConfigManager.getConfig());

connection.connect();

const articleDao = new ArticleDao(connection);

try {

return await articleDao.selectCount();

} catch(error) {

throw error;

} finally {

connection.destroy();

}

}

/\*\*

\* 記事を新規登録します。登録完了時、登録した記事のidを返します。

\* @param article

\* @returns {Promise<number>}

\*/

async createArticle(article: Article): Promise<number> {

const connection = mysql.createConnection(DbConfigManager.getConfig());

connection.connect();

const articleDao = new ArticleDao(connection);

try {

return await articleDao.insertArticle(article);

} catch(error) {

throw error;

} finally {

connection.destroy();

}

}

/\*\*

\* 引数で指定されたidの記事を取得します。

\* @param id

\* @returns {Promise<RegisteredArticle>}

\*/

async findArticle(id: number): Promise<RegisteredArticle[]> {

const connection = mysql.createConnection(DbConfigManager.getConfig());

connection.connect();

const articleDao = new ArticleDao(connection);

try {

return await articleDao.selectArticleById(id);

} catch(error) {

throw error;

} finally {

connection.destroy();

}

}

async modifyArticle(id: number, article: RegisteredArticle): Promise<RegisteredArticle> {

const connection = mysql.createConnection(DbConfigManager.getConfig());

connection.connect();

const articleDao = new ArticleDao(connection);

try {

const results = await articleDao.lock(id);

if (!Array.isArray(results) || results.length === 0) {

const error = new AppError();

error.status = 404;

throw error;

}

if (results[0].updatedAt !== article.updatedAt) {

const error = new AppError();

error.status = 500;

throw error;

}

return await articleDao.updateArticle(id, article);

} catch(error) {

throw error;

} finally {

connection.destroy();

}

}

async removeArticle(id: number): Promise<void> {

const connection = mysql.createConnection(DbConfigManager.getConfig());

connection.connect();

const articleDao = new ArticleDao(connection);

try {

const results = await articleDao.lock(id);

if (!Array.isArray(results) || results.length === 0) {

const error = new AppError();

error.status = 404;

throw error;

}

return await articleDao.deleteArticle(id);

} catch(error) {

throw error;

} finally {

connection.destroy();

}

}

}

### 4.2.7 コントローラクラス

最後にコントローラクラスを作成します。

リスト4.2.7 server/controllers/ArticleController.ts

import \* as express from 'express';

import { ArticleService } from '../services/ArticleService';

import { AppError } from "../models/AppError";

import { Article } from "../models/Article";

import { RegisteredArticle } from "../models/RegisteredArticle";

/\*\*

\* 記事API用コントローラ

\*/

export class ArticleController {

private articleService: ArticleService;

/\*\*

\* コンストラクタ

\*/

constructor() {

this.articleService = new ArticleService();

}

all(req: express.Request, res: express.Response, next: express.NextFunction): void {

this.articleService.findAllArticles()

.then((articles) => {

res.send({data: articles});

})

.catch((error) => {

error.status = error.status || 500;

next(error);

});

}

index(req: express.Request, res: express.Response, next: express.NextFunction): void {

const offset = Number.parseInt(req.query.offset, 10);

const limit = Number.parseInt(req.query.limit, 10);

if (Number.isNaN(offset) || Number.isNaN(limit)) {

const error = new AppError('Not found.');

error.status = 400;

next(error);

return;

}

this.articleService.findArticles(offset, limit)

.then((articles) => {

res.send({ data: articles })

})

.catch((error) => {

error.status = error.status || 500;

next(error);

});

}

count(req: express.Request, res: express.Response, next: express.NextFunction): void {

this.articleService.findArticleCount()

.then((count) => {

res.send({ data: count });

})

.catch((error) => {

error.status = error.status || 500;

next(error);

});

}

create(req: express.Request, res: express.Response, next: express.NextFunction): void {

const article: Article = req.body;

if (!article.title) {

const error = new AppError('タイトルが空です。');

error.status = 400;

next(error);

return;

}

this.articleService.createArticle(article)

.then((insertId) => {

res.send({ data: insertId });

})

.catch((error) => {

error.status = error.status || 500;

next(error);

});

}

read(req: express.Request, res: express.Response, next: express.NextFunction): void {

const id = Number.parseInt(req.params.id, 10);

if (Number.isNaN(id)) {

const error = new AppError();

error.status = 400;

next(error);

return;

}

this.articleService.findArticle(id)

.then((result) => {

res.send({ data: result });

})

.catch((error) => {

error.status = error.status || 500;

next(error);

});

}

update(req: express.Request, res: express.Response, next: express.NextFunction): void {

const id = Number.parseInt(req.params.id, 10);

const article: RegisteredArticle = req.body;

if (!article.title) {

const error = new AppError('タイトルが空です。');

error.status = 400;

next(error);

return;

}

if (Number.isNaN(id)) {

const error = new AppError();

error.status = 400;

next(error);

return;

}

this.articleService.modifyArticle(id, article)

.then((result) => {

res.send({ data: result });

})

.catch((error) => {

error.status = error.status || 500;

next(error);

});

}

delete(req: express.Request, res: express.Response, next: express.NextFunction): void {

const id = Number.parseInt(req.params.id, 10);

if (Number.isNaN(id)) {

const error = new AppError();

error.status = 400;

next(error);

return;

}

this.articleService.removeArticle(id)

.then((result) => {

res.send({ data: result });

})

.catch((error) => {

error.status = error.status || 500;

next(error);

});

}

}

## 4.3 フロントエンドの開発

サーバーサイドの開発が一通り完了したので、次はフロントエンドの開発を進めていきます。フロントエンドのソースコード類は、『chapter-4/client』ディレクトリ内に作成していきます。

### 4.3.1 package.jsonの作成

サーバーサイドの開発時の同様package.jsonを作成します。

リスト4.3.1 client/package.json

{

"name": "sample-crud-application-client",

"version": "1.0.0",

"scripts": {

"build": "rimraf ../server/public && webpack --progress --colors",

"build:prod": "NODE\_ENV=production rimraf ../server/public && webpack --progress --colors",

"build-aot": "rimraf ../server/public && webpack --progress --colors --config webpack.config.aot.js",

"build-aot:prod": "NODE\_ENV=production rimraf ../server/public && webpack --progress --colors --config webpack.config.aot.js",

"dev": "webpack --progress --colors --watch"

},

"license": "MIT",

"dependencies": {

"@angular/common": "~2.4.8",

"@angular/compiler": "~2.4.8",

"@angular/core": "~2.4.8",

"@angular/forms": "~2.4.8",

"@angular/http": "~2.4.8",

"@angular/platform-browser": "~2.4.8",

"@angular/platform-browser-dynamic": "~2.4.8",

"@angular/router": "~3.4.0",

"@angular/upgrade": "~2.4.8",

"angular-in-memory-web-api": "~0.2.4",

"core-js": "^2.4.1",

"jquery": "^3.1.1",

"reflect-metadata": "^0.1.8",

"rxjs": "^5.2.0",

"toastr": "^2.1.2",

"zone.js": "^0.7.7"

},

"devDependencies": {

"@angular/compiler-cli": "^2.4.8",

"@ngtools/webpack": "^1.2.10",

"@types/jquery": "^2.0.40",

"@types/reflect-metadata": "^0.0.5",

"@types/rx": "^4.1.1",

"@types/toastr": "^2.1.32",

"@types/zone.js": "^0.0.27",

"angular2-template-loader": "^0.6.2",

"copy-webpack-plugin": "^4.0.1",

"raw-loader": "^0.5.1",

"rimraf": "^2.6.1",

"ts-loader": "^2.0.1",

"typescript": "^2.2.1",

"webpack": "^2.2.1"

}

}

### 4.3.2 tsconfig.jsonの作成

tsconfig.jsonについてもサーバーサイド開発時同様ファイルを作成します。リスト4.3.2の内容でフロントエンド用のtsconfig.jsonを作成します。

リスト4.3.2 client/tsconfig.json

{  
 "compilerOptions": {  
 "target": "ES5",  
 "module": "commonjs",  
 "moduleResolution": "node",  
 "sourceMap": true,  
 "emitDecoratorMetadata": true,  
 "experimentalDecorators": true,  
 "noImplicitAny": true,  
 "suppressImplicitAnyIndexErrors": true,  
 "lib": ["es2015", "dom"],  
 "typeRoots": [  
 "./node\_modules/@types"  
 ],  
 "alwaysStrict": true  
 },  
 "exclude": [  
 "node\_modules",  
 "\*\*/\*-aot.ts"  
 ]  
}

### 4.3.3 モジュールのインストール

サーバーサイドの開発時と同様必要なモジュールをインストールします。

リスト4.2.1

$ npm install

また、見た目をそれなりに整えるために、今回はSkeletonというCSSフレームワークを使用します。Skeletonはグリッドシステムやボタンなど、使用頻度の高いスタイルのみを定義している非常に学習コストの低いフレームワークです。以下のWebサイトからZIPファイルをダウンロードします。

URL: <http://getskeleton.com/>

ZIPファイルを解凍したら、normalize.cssとskeleton.cssをstylesディレクトリ内に配置します。これで一通り必要なモジュールのインストールが完了しました。

### 4.3.4 ビルドの設定

必要なモジュールの準備ができたら、ビルドの設定を行います。フロントエンドのJavaScriptを実行するのはブラウザですが、筆者が知るかぎり、現状ブラウザは以下のような構文に対応していません。そのため、ブラウザが実行できる形にJavaScriptファイルを出力する必要があります。

リスト4.3.4.1

import Hoge from './Hoge';

またTypeScriptからJavaScriptへのトランスパイルと、フロントエンドのプロジェクトで管理しているファイル（skeleton.css、normalize.css、ファビコンなど）をpublicディレクトリへのコピーを行わなければいけません。

今回はこれらをwebpackで行います。webpackは、JavaScriptやCSSなどの依存関係を解決し配布用ファイルを作成するためのツールです。webpackの設定ファイルは『webpack.config.js』という名前で作成します。

リスト4.3.4.2 client/webpack.config.js

var CopyWebpackPlugin = require('copy-webpack-plugin');  
var webpack = require('webpack');  
var helpers = require('./helpers');  
  
module.exports = {  
 devtool: 'source-map',  
 entry: {  
 app: './app/app.ts',   
 vendor: './app/vendor.ts'  
 },  
 module: {  
 rules: [  
 {  
 test: /\.ts$/,  
 loader: 'ts-loader'  
 }  
 ]  
 },  
 output: {  
 path: '../server/public/',  
 filename: '[name].js'  
 },  
 plugins: [  
 new CopyWebpackPlugin([  
 {  
 from: './index.html',  
 to: './'  
 },  
 {  
 from: './styles/\*.css',  
 to: './'  
 },  
 {  
 from: './favicon.png',  
 to: './'  
 },  
 {  
 from: './app/\*\*/\*.html',  
 to: './',  
 flatten: true  
 },  
 {  
 from: './app/\*\*/\*.css',  
 to: './',  
 flatten: true  
 }  
 ],  
 {  
 ignore: [ '.DS\_Store', '.gitkeep' ]  
 }  
 ),  
 new webpack.ContextReplacementPlugin(  
 // The (\\|\/) piece accounts for path separators in \*nix and Windows  
 /angular(\\|\/)core(\\|\/)(esm(\\|\/)src|src)(\\|\/)linker/,  
 helpers.root('./src'), // location of your src  
 {} // a map of your routes  
 ),  
 new webpack.optimize.CommonsChunkPlugin({  
 name: ['app', 'vendor']  
 })  
 ],  
 resolve: {  
 extensions: ['.ts', '.js', '.html', '.css']  
 }  
};

さらに、もう1ファイル、『webpack.config-aot.js』というファイルも作成します。このファイルについては、4.3.14で解説します。

リスト4.3.4.3 client/webpack.config.js

var CopyWebpackPlugin = require('copy-webpack-plugin');

var webpack = require('webpack');

var AotPlugin = require('@ngtools/webpack').AotPlugin;

var helpers = require('./helpers');

module.exports = {

devtool: 'source-map',

entry: {

app: process.env.NODE\_ENV === 'production' ? './app/product-app.ts' : './app/app.ts',

vendor: './app/vendor.ts'

},

module: {

rules: [

{

test: /\.html$/,

loader: 'raw-loader'

},

{

test: /\.css$/,

loader: 'raw-loader'

},

{

test: /\.ts$/,

loader: '@ngtools/webpack'

}

]

},

output: {

path: '../server/public/',

filename: '[name].js'

},

plugins: [

new AotPlugin({

tsConfigPath: './tsconfig.json',

entryModule: helpers.root('app/app.module#AppModule')

}),

new webpack.optimize.CommonsChunkPlugin({

name: ['app', 'vendor']

}),

new CopyWebpackPlugin([

{

from: './index.html',

to: './'

},

{

from: './styles/\*.css',

to: './'

},

{

from: './favicon.png',

to: './'

}

]),

new webpack.optimize.UglifyJsPlugin({

beautify: false,

mangle: {

screw\_ie8: true,

keep\_fnames: true

},

compress: {

warnings: false,

screw\_ie8: true

},

comments: false

})

],

resolve: {

extensions: ['.ts', '.js', '.html', '.css']

}

};

最後に、webpack.config.js及びwebpack.config-aot.jsから使用するヘルパーモジュールを作成します。

リスト4.3.4.4 client/herper.js

var path = require('path');

var \_root = path.resolve(\_\_dirname);

function root(args) {

args = Array.prototype.slice.call(arguments, 0);

return path.join.apply(path, [\_root].concat(args));

}

exports.root = root;

### 4.3.5 index.htmlの作成

エントリポイントとなるHTMLファイルを作成します。以下の内容でindex.htmlという名前で新規ファイルを作成します。bodyタグ最後で読み込んでいるvendor.jsとapp.jsが、webpackでビルドされたJavaScriptファイルです。

リスト4.2.3 client/index.html

<!DOCTYPE html>  
<html lang="ja">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1">  
 <!-- skeleton css -->  
 <link rel="stylesheet" href="/styles/normalize.css">  
 <link rel="stylesheet" href="/styles/skeleton.css">  
 <!-- toastr 用css -->  
 <link rel='stylesheet' href='https://cdnjs.cloudflare.com/ajax/libs/toastr.js/latest/css/toastr.css'>  
 <title>Sample CRUD Application</title>  
 <base href="/">  
</head>  
<body>  
 <app>  
 Now loading...  
 </app>  
 <script src="/vendor.js"></script>  
 <script src="/app.js"></script>  
</body>  
</html>

### 4.3.6 外部ライブラリ用アセットのエントリポイント

次に外部ライブラリ用アセットのエントリポイントを作成します。これを作成することで、外部ライブラリのソースコードをアプリケーション本体のアセットであるapp.jsに含めなくなります。そうすると、開発中のビルド時間の短縮の削減と、ブラウザでJavaScriptファイルを読み込む際に、アプリケーション側のコードに変更があっても外部ライブラリ分についてはキャッシュを見るようにすることができます。

リスト4.3.6 client/vendor.ts

// polyfills

import 'core-js/shim';

import 'reflect-metadata';

import 'zone.js/dist/zone';

// Angular

import '@angular/platform-browser';

import '@angular/platform-browser-dynamic';

import '@angular/core';

import '@angular/common';

import '@angular/http';

import '@angular/router';

// RxJS

import 'rxjs';

// others

import 'jquery';

import 'toastr';

### 4.3.7 設定値保持クラスの作成

更に設定値保持クラスを作成します。また、本番環境と開発環境でAPIのURLが異なることを想定して、本番環境と開発環境とを切り替えて初期化ができるようにしておきます。

リスト4.3.7

/\*\*

\* 設定値保持クラス

\* @export

\* @class Config

\*/

export class Config {

/\*\*

\* 本番環境用APIのURL

\*/

private static readonly PRODUCT\_API\_ROOT\_URL = 'http://localhost:3000/api/v1/';

/\*\*

\* 開発環境用APIのURL

\*/

private static readonly DEV\_API\_ROOT\_URL = 'http://localhost:3000/api/v1/';

private static instance: Config;

private apiRoot: string;

/\*\*

\* コンストラクタ

\* @param {string} apiRoot

\*/

private constructor(apiRoot: string) {

this.apiRoot = apiRoot;

}

/\*\*

\* 設定値の初期化を行います。

\* @param {boolean} isProduct

\*/

static initialize(isProduct: boolean) {

if (isProduct) {

Config.instance = new Config(Config.PRODUCT\_API\_ROOT\_URL);

} else {

Config.instance = new Config(Config.DEV\_API\_ROOT\_URL);

}

}

/\*\*

\* 設定値保持クラスのインスタンスを取得します。

\* @returns {Config}

\*/

static getInstance(): Config {

if (!Config.instance) {

throw new Error('Not initialized!');

}

return Config.instance;

}

/\*\*

\* APIのURLを取得します。

\* @returns {string}

\*/

getApiRoot(): string {

return this.apiRoot;

}

}

### 4.3.8 エントリポイントの作成

下準備は整いました。いよいよアプリケーション本体のプログラミングです。

まずは、エントリポイントから作成します。リスト4.3.8.1のエントリポイントではモジュールの起動の他に設定クラスの初期化を行っています。リスト4.3.8.3では、アプリケーションで使用するモジュールを定義しています。

特に注目したいのは27以降の『RouterModule.forRoo…』です。ここでフロントエンドでのルーティングの設定を行っています。これをサーバーサイドのルーティングと一致させることで、URL直打ちから遷移した場合も、意図した画面が表示されるようになります。

リスト4.3.8.1 client/app/app.ts

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app.module';

import { Config } from './Config';

/\*\*

\* 開発環境用エントリポイント

\*/

Config.initialize(false);

platformBrowserDynamic().bootstrapModule(AppModule);

リスト4.3.8.2 client/app/product-app.ts

import { enableProdMode } from '@angular/core';

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app.module';

import { Config } from './Config';

/\*\*

\* 本番環境用エントリポイント

\*/

Config.initialize(true);

enableProdMode();

platformBrowserDynamic().bootstrapModule(AppModule);

リスト4.2.8.3 client/app/app.module.ts

import { NgModule } from '@angular/core';

import { FormsModule } from '@angular/forms';

import { HttpModule } from '@angular/http';

import { BrowserModule } from '@angular/platform-browser';

import { RouterModule } from '@angular/router';

import { AppComponent } from './components/app.component';

import { ArticleListComponent } from './components/article-list.component';

import { CreateArticleComponent } from './components/create-article.component';

import { UpdateArticleComponent } from './components/update-article.component';

import { ViewArticleComponent } from './components/view-article.component';

@NgModule({

bootstrap: [AppComponent],

declarations: [

AppComponent,

ArticleListComponent,

CreateArticleComponent,

UpdateArticleComponent,

ViewArticleComponent

],

imports: [

BrowserModule,

FormsModule,

HttpModule,

RouterModule.forRoot([

{

path: '',

component: ArticleListComponent

},

{

path: 'createArticle',

component: CreateArticleComponent

},

{

path: 'viewArticle/:id',

component: ViewArticleComponent

},

{

path: 'updateArticle/:id',

component: UpdateArticleComponent

}

])

]

})

export class AppModule { }

### 4.3.9 コンポーネントの作成

次にコンポーネントクラスを作成します。コンポーネントとは画面を構成する部品のことを指します。VelocityやSmartyといったテンプレートと異なる点として、テンプレートはあくまで静的な見た目を定義するものですが、コンポーネントは見た目に加えボタンを押した時の処理などの振る舞いも定義します。

Angular2でのコンポーネントは、TypeScript、HTML、CSSの3つから構成されます。これらは1つのファイルにすべて記述することもできますし、それぞれ別のファイルにすることもできます。今回はすべて別のファイルに記述していきます。HTML、CSSへの参照方法はTypeScriptファイルの『@Component』デコレータ内の『templateUrl』、『styleUrls』属性で指定します。

今回は以下の5つのコンポーネントを作成します。

* app.component
  + アプリケーション全体を包括するコンポーネント
* article-list.component
  + 記事一覧表示部分のコンポーネント
* create-article.component
  + 記事作成フォームのコンポーネント
* update-article.component
  + 記事更新フォームのコンポーネント
* view-article.component
  + 記事内容表示領域のコンポーネント

#### 4.3.9.1 app.component

リスト4.3.9.1.1 client/app/components/app.component.ts

import { Component, OnInit, OnDestroy, ElementRef } from '@angular/core';

import { ActivatedRoute, Router } from '@angular/router';

import { AppStore } from '../common/AppStore';

import { AppState } from '../models/AppState';

@Component({

selector: 'app',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit, OnDestroy {

isProcessing: boolean;

private htmlElement: HTMLElement;

/\*\*

\* コンストラクタ

\*/

constructor(

private router: Router,

private route: ActivatedRoute,

private elementRef: ElementRef

) {

this.isProcessing = false;

this.htmlElement = this.elementRef.nativeElement;

}

ngOnInit(): void {

this.onChangeAppState = this.onChangeAppState.bind(this);

AppStore.getInstance().registerHandler('CHANGE', this.onChangeAppState);

}

ngOnDestroy(): void {

AppStore.getInstance().removeHandler('CHANGE', this.onChangeAppState);

}

private onChangeAppState(eventName: string, beforeAppState: AppState, currentAppState: AppState) {

this.isProcessing = currentAppState.isProcessing;

}

}

リスト4.3.9.1.2 client/app/components/app.component.html

<div class="container">

<h1>Sample CRUD Application</h1>

<router-outlet></router-outlet>

<div \*ngIf="isProcessing" class="loading-container">

<div class="loading"></div>

</div>

</div>

リスト4.3.9.1.3 client/app/components/app.component.css

.loading-container {

position: fixed;

top: 0px;

left: 0px;

opacity: 0.5;

height: 100vh;

width: 100vw;

}

.loading {

border-radius: 50%;

border: 8px solid #2c2c2c;

border-right-color: transparent;

height: 50px;

margin: 0 auto;

text-align:center;

top: calc(50vh - 15px);

position: relative;

pointer-events:none;

width: 50px;

-webkit-animation: spin 1s infinite linear;

-moz-animation: spin 1s infinite linear;

-ms-animation: spin 1s infinite linear;

}

#### 4.3.9.2 article-list.component

リスト4.3.9.2.1 client/app/components/article-list.component.ts

import { Component, OnInit, OnDestroy, ElementRef } from '@angular/core';

import \* as toastr from 'toastr';

import { AppStore } from '../common/AppStore';

import { ArticleService } from '../services/ArticleService';

import { AppState } from '../models/AppState';

import { RegisteredArticle } from '../models/RegisteredArticle';

import { ArticleAction } from '../actions/ArticleAction';

import { ProcessingModalAction } from '../actions/ProcessingModalAction';

@Component({

selector: 'article-list',

providers: [ArticleService],

templateUrl: './article-list.component.html'

})

export class ArticleListComponent implements OnInit, OnDestroy {

private htmlElement: HTMLElement;

articles: RegisteredArticle[];

isProcessing: boolean;

constructor(

private elementRef: ElementRef,

private articleService: ArticleService,

) {

this.isProcessing = false;

this.htmlElement = this.elementRef.nativeElement;

}

ngOnInit(): void {

this.onChangeAppState = this.onChangeAppState.bind(this);

AppStore.getInstance().registerHandler('CHANGE', this.onChangeAppState);

this.updateArticleList();

}

ngOnDestroy(): void {

AppStore.getInstance().removeHandler('CHANGE', this.onChangeAppState);

}

deleteButtonClicked(articleId: number): void {

ProcessingModalAction.setProcessingFlag(true);

this.articleService.delete(articleId).subscribe(

this.onDeleteArticle.bind(this),

this.onError.bind(this));

}

private updateArticleList(): void {

ProcessingModalAction.setProcessingFlag(true);

this.articleService

.findAllArticles()

.subscribe(

this.onFindArticles.bind(this),

this.onError.bind(this));

}

private onChangeAppState(eventName: string, beforeAppState: AppState, currentAppState: AppState) {

this.articles = currentAppState.articles;

this.isProcessing = currentAppState.isProcessing;

}

private onFindArticles(registerdArticles: RegisteredArticle[]): void {

ArticleAction.change(registerdArticles);

ProcessingModalAction.setProcessingFlag(false);

}

private onDeleteArticle(): void {

toastr.success('削除完了');

this.updateArticleList();

}

private onError(errorMessage: string): void {

toastr.error(errorMessage);

ProcessingModalAction.setProcessingFlag(false);

}

}

.article\_\_body {

height: calc(100vh - 500px);

}

リスト4.3.9.2.2 client/app/components/article-list.component.html

<p>

<a [routerLink]="['/createArticle']">新規作成</a>

</p>

<h2>記事一覧</h2>

<div \*ngIf="articles.length === 0" class="u-full-width">

登録された記事がありません。

</div>

<table \*ngIf="articles.length > 0" class="u-full-width">

<tr>

<th>タイトル</th>

<th>作成日時</th>

<th>更新日時</th>

<th>操作</th>

</tr>

<tr \*ngFor="let article of articles">

<td><a [routerLink]="['/viewArticle', article.id]">{{ article.title }}</a></td>

<td>{{ article.createdAt }}</td>

<td>{{ article.updatedAt }}</td>

<td>

<a class="button button-primary" [routerLink]="['/updateArticle', article.id]">

編集

</a>

<button (click)="deleteButtonClicked(article.id)" [attr.disabled]="isProcessing ? true : null">

削除

</button>

</td>

</tr>

</table>

#### 4.3.9.3 create-article.component

リスト4.3.9.3.1 client/app/components/create-article.component.ts

import { Component, OnInit, OnDestroy } from '@angular/core';

import \* as toastr from 'toastr';

import { ProcessingModalAction } from '../actions/ProcessingModalAction';

import { AppState } from '../models/AppState';

import { Article } from '../models/Article';

import { AppStore } from '../common/AppStore';

import { ArticleService } from '../services/ArticleService';

@Component({

selector: 'article-editor',

providers: [ArticleService],

templateUrl: './create-article.component.html',

styleUrls: ['./create-article.component.css']

})

export class CreateArticleComponent implements OnInit, OnDestroy {

article: Article;

isProcessing: boolean;

isCompleted: boolean;

constructor(

private articleService: ArticleService

) {

}

ngOnInit(): void {

this.article = {

title: '',

body: ''

};

this.isProcessing = false;

this.isCompleted = false;

this.onChangeAppState = this.onChangeAppState.bind(this);

AppStore.getInstance().registerHandler('CHANGE', this.onChangeAppState);

}

ngOnDestroy(): void {

AppStore.getInstance().removeHandler('CHANGE', this.onChangeAppState);

}

titleChanged(event: Event): void {

this.article.title = (event.target as HTMLInputElement).value;

}

bodyChanged(event: Event): void {

this.article.body = (event.target as HTMLInputElement).value;

}

registerButtonClicked(): void {

ProcessingModalAction.setProcessingFlag(true);

this.articleService.create(this.article).subscribe(

this.onCreateArticle.bind(this),

this.onCreatedArticleError.bind(this));

}

private onCreateArticle(): void {

toastr.success('登録完了');

this.isCompleted = true;

ProcessingModalAction.setProcessingFlag(false);

}

private onCreatedArticleError(errorMessage: string): void {

toastr.error(errorMessage);

ProcessingModalAction.setProcessingFlag(false);

}

private onChangeAppState(eventName: string, beforeAppState: AppState, currentAppState: AppState): void {

this.isProcessing = currentAppState.isProcessing;

}

}

リスト4.3.9.3.2 client/app/components/create-article.component.html

<p>

<a [routerLink]="['/']">もどる</a>

</p>

<h2>記事の作成</h2>

<div \*ngIf="!isCompleted" class="u-full-width">

<div>

<label>記事タイトル</label>

<input type="text"

placeholder="タイトル"

class="u-full-width"

value="{{ article.title }}"

(change)="titleChanged($event)"

[attr.disabled]="isProcessing ? true : null">

</div>

<div>

<label>本文</label>

<textarea

class="u-full-width article\_\_body"

placeholder="本文"

rows="10"

value="{{ article.body }}"

(change)="bodyChanged($event)"

[attr.disabled]="isProcessing ? true : null">

</textarea>

</div>

<button class="button-primary"

(click)="registerButtonClicked()"

[attr.disabled]="isProcessing ? true : null">

登録

</button>

</div>

<div \*ngIf="isCompleted">

<a [routerLink]="['/']">一覧にもどる</a>

</div>

リスト4.3.9.3.3 client/app/components/create-article.component.css

.article\_\_body {

height: calc(100vh - 500px);

}

#### 4.3.9.4 update-article.component

リスト4.3.9.4.1 client/app/components/update-article.component.ts

import {Component, OnInit, OnDestroy} from '@angular/core';

import { ActivatedRoute } from '@angular/router';

import \* as toastr from 'toastr';

import { ProcessingModalAction } from '../actions/ProcessingModalAction';

import { AppStore } from '../common/AppStore';

import { AppState } from '../models/AppState';

import { RegisteredArticle } from '../models/RegisteredArticle';

import { ArticleService } from '../services/ArticleService';

@Component({

selector: 'update-article-page',

providers: [ ArticleService ],

templateUrl: './update-article.component.html',

styleUrls: ['./update-article.component.css']

})

export class UpdateArticleComponent implements OnInit, OnDestroy {

isCompleted: boolean;

isProcessing: boolean;

article: RegisteredArticle;

constructor(

private activeRoute: ActivatedRoute,

private articleService: ArticleService

) {

}

ngOnInit(): void {

this.isProcessing = false;

this.onChangeAppState = this.onChangeAppState.bind(this);

AppStore.getInstance().registerHandler('CHANGE', this.onChangeAppState);

this.readArticle();

}

ngOnDestroy(): void {

AppStore.getInstance().removeHandler('CHANGE', this.onChangeAppState);

}

private readArticle(): void {

ProcessingModalAction.setProcessingFlag(true);

this.activeRoute.params

.subscribe((param) => {

this.articleService.findArticleById(Number.parseInt(param.id, 10))

.subscribe(

this.onReadArticles.bind(this),

this.onError.bind(this)

);

});

}

private onReadArticles(articles: RegisteredArticle[]): void {

if (articles.length === 0) {

throw new RangeError('指定された記事が見つかりません');

}

this.article = articles[0];

ProcessingModalAction.setProcessingFlag(false);

}

titleChanged(event: Event): void {

this.article.title = (event.target as HTMLInputElement).value;

}

bodyChanged(event: Event): void {

this.article.body = (event.target as HTMLInputElement).value;

}

updateButtonClicked(): void {

ProcessingModalAction.setProcessingFlag(true);

this.articleService.update(this.article.id, this.article).subscribe(

this.onUpdateArticle.bind(this),

this.onError.bind(this));

}

private onUpdateArticle(): void {

toastr.success('更新完了');

this.isCompleted = true;

ProcessingModalAction.setProcessingFlag(false);

}

private onError(errorMessage: string): void {

toastr.error(errorMessage);

ProcessingModalAction.setProcessingFlag(false);

}

private onChangeAppState(eventName: string, beforeAppState: AppState, currentAppState: AppState): void {

this.isProcessing = currentAppState.isProcessing;

}

}

リスト4.3.9.4.2 client/app/components/update-article.component.html

<p>

<a [routerLink]="['/']">もどる</a>

</p>

<h2>記事の更新</h2>

<div \*ngIf="!isCompleted" class="u-full-width">

<div \*ngIf="article">

<div>

<label>記事タイトル</label>

<input type="text"

placeholder="タイトル"

class="u-full-width"

value="{{ article.title }}"

(change)="titleChanged($event)"

[attr.disabled]="isProcessing ? true : null">

</div>

<div>

<label>本文</label>

<textarea

class="u-full-width article\_\_body"

placeholder="本文"

rows="10"

value="{{ article.body }}"

(change)="bodyChanged($event)"

[attr.disabled]="isProcessing ? true : null">

</textarea>

</div>

<div>

<button class="button-primary"

[attr.disabled]="isProcessing ? true : null"

(click)="updateButtonClicked()">

更新

</button>

</div>

</div>

</div>

<div \*ngIf="isCompleted">

<a [routerLink]="['/']">一覧にもどる</a>

</div>

リスト4.3.9.4.3 client/app/components/update-article.component.css

.article\_\_body {

height: calc(100vh - 500px);

}

#### 4.3.9.5 view-article.component

リスト4.3.9.5.1 client/app/components/view-article.component.ts

import { Component, OnInit } from '@angular/core';

import { ActivatedRoute } from '@angular/router';

import \* as toastr from 'toastr';

import { ProcessingModalAction } from '../actions/ProcessingModalAction';

import { RegisteredArticle } from '../models/RegisteredArticle';

import { ArticleService } from '../services/ArticleService';

@Component({

selector: 'view-article-page',

providers: [ArticleService],

templateUrl: './view-article.component.html',

styleUrls: ['./view-article.component.css']

})

export class ViewArticleComponent implements OnInit {

article: RegisteredArticle;

constructor(private activeRoute: ActivatedRoute, private articleService: ArticleService) {

}

ngOnInit(): void {

this.readArticle();

}

private readArticle(): void {

ProcessingModalAction.setProcessingFlag(true);

this.activeRoute.params

.subscribe((param) => {

this.articleService.findArticleById(Number.parseInt(param.id, 10))

.subscribe(

this.onReadArticles.bind(this),

this.onError.bind(this)

);

});

}

private onReadArticles(articles: RegisteredArticle[]): void {

if (articles.length === 0) {

throw new RangeError('指定された記事が見つかりません');

}

this.article = articles[0];

ProcessingModalAction.setProcessingFlag(false);

}

private onError(errorMessage: string): void {

toastr.error(errorMessage);

ProcessingModalAction.setProcessingFlag(false);

}

}

リスト4.3.9.5.2 client/app/components/view-article.component.html

<p>

<a [routerLink]="['/']">もどる</a>

</p>

<div \*ngIf="article" class="u-full-width">

<h2>{{ article.title }}</h2>

<div>

<p class="u-full-width article\_\_body">{{ article.body }}</p>

</div>

</div>

リスト4.3.9.5.3 client/app/components/view-article.component.css

.article\_\_body {

white-space: pre-wrap;

}

### 4.3.10 サービスクラスの作成

コンポーネントを作成したら、次にサービスクラスを作成します。フロントエンドのサービスクラスの役割は、主にサーバーサイドとの通信です。

リスト 4.3.10.1 client/app/services/ArticleService.ts

import { Injectable } from '@angular/core';

import { Headers, Http, RequestOptions, Response } from '@angular/http';

import \* as Rx from 'rxjs';

import { Config } from '../Config';

import { Article } from "../models/Article";

@Injectable()

export class ArticleService {

private static readonly API = 'article';

/\*\*

\* コンストラクタ

\* @param {Http} http

\*/

constructor(private http: Http) {

}

findArticles(offset: number, limit: number): Rx.Observable<any> {

const url = `${Config.getInstance().getApiRoot()}${ArticleService.API}?offset=${offset}&limit=${limit}`;

return this.http.get(encodeURI(url))

.map(this.extractData)

.catch(this.handleError);

}

findArticleById(id: number): Rx.Observable<any> {

const headers = new Headers({ 'x-requested-with': 'XMLHttpRequest' });

const requestOptions = new RequestOptions({ headers: headers });

const url = `${Config.getInstance().getApiRoot()}${ArticleService.API}/${id}.json`;

return this.http.get(encodeURI(url), requestOptions)

.map(this.extractData)

.catch(this.handleError);

}

findAllArticles(): Rx.Observable<any> {

const headers = new Headers({ 'x-requested-with': 'XMLHttpRequest' });

const requestOptions = new RequestOptions({ headers: headers });

const url = `${Config.getInstance().getApiRoot()}${ArticleService.API}/all.json`;

return this.http.get(encodeURI(url), requestOptions)

.map(this.extractData)

.catch(this.handleError);

}

create(article: Article): Rx.Observable<any> {

const sendData = JSON.stringify(article);

const headers = new Headers({

'x-requested-with': 'XMLHttpRequest',

'Content-Type': 'application/json'

});

const requestOptions = new RequestOptions({ headers: headers });

return this.http.post(encodeURI(`${Config.getInstance().getApiRoot()}${ArticleService.API}`), sendData, requestOptions)

.map(this.extractData)

.catch(this.handleError);

}

update(id: number, article: Article): Rx.Observable<any> {

const url = `${Config.getInstance().getApiRoot()}${ArticleService.API}/${id}.json`;

const sendData = JSON.stringify(article);

const headers = new Headers({

'x-requested-with': 'XMLHttpRequest',

'Content-Type': 'application/json'

});

const requestOptions = new RequestOptions({ headers: headers });

return this.http.put(encodeURI(url), sendData, requestOptions)

.map(this.extractData)

.catch(this.handleError);

}

delete(id: number): Rx.Observable<any> {

const headers = new Headers({ 'x-requested-with': 'XMLHttpRequest' });

const requestOptions = new RequestOptions({ headers: headers });

const url = `${Config.getInstance().getApiRoot()}${ArticleService.API}/${id}.json`;

return this.http.delete(encodeURI(url), requestOptions)

.map(this.extractData)

.catch(this.handleError);

}

private extractData(res: Response): any {

if (res.status < 200 || res.status >= 300) {

throw new Error('Bad response status: ' + res.status);

}

return res.json().data;

}

private handleError (error: any): any {

const errorMessage = error.message || error.json().message || 'Server error';

return Rx.Observable.throw(errorMessage);

}

}

### 4.3.11 Storeの作成

フロント側のアプリケーションの状態を管理するStoreクラスを作成します。このStoreはアプリケーションの状態に変更があった場合に、予めハンドラーとして登録されているコンポーネント側の処理を呼び出す役割も持っています。これにより、常にアプリケーションの状態とコンポーネント側の表示を常に一致させることができます。

リスト4.3.11 client/app/common/AppStore.ts

import { AppState } from '../models/AppState';

interface Handler {

(eventName: string, beforeState: AppState, currentState: AppState) :void

}

interface EmitInfo {

eventName: string;

handlers: Handler[];

}

/\*\*

\* アプリケーションの状態を保持します。

\* @export

\* @class AppStore

\*/

export class AppStore {

private static instance: AppStore;

private appState: AppState;

private emitInfoList: EmitInfo[];

static getInstance(): AppStore {

if (!AppStore.instance) {

AppStore.instance = new AppStore();

}

return AppStore.instance;

}

private constructor() {

this.appState = {

articles: [],

isProcessing: false

};

this.emitInfoList = [];

}

getAppState(): AppState {

return this.appState;

}

applyAppState(eventName: string, chagedState: any): void {

const emitInfo = this.emitInfoList.find((emitInfo: EmitInfo) => {

return emitInfo.eventName === eventName

});

if (!emitInfo) {

return;

}

const beforeAppState = Object.assign({}, this.appState);

this.appState = Object.assign(this.appState, chagedState);

emitInfo.handlers.forEach((handler: Handler) => {

handler(eventName, beforeAppState, this.appState);

});

}

registerHandler(eventName: string, handler: Handler): void {

const emitInfo = this.emitInfoList.find((emitInfo: EmitInfo) => {

return emitInfo.eventName === eventName

});

if (emitInfo) {

emitInfo.handlers.push(handler);

} else {

this.emitInfoList.push({

eventName: eventName,

handlers: [handler]

});

}

}

removeHandler(eventName: string, handler: Handler): void {

const emitInfo = this.emitInfoList.find((emitInfo: EmitInfo) => {

return emitInfo.eventName === eventName;

});

if (!emitInfo) {

return;

}

const handlerIndex = emitInfo.handlers.indexOf(handler);

if (handlerIndex === -1) {

return;

}

emitInfo.handlers.splice(handlerIndex, 1);

}

}

### 4.3.12 Actionの作成

Storeを作成したら、Storeにアプリケーションの状態の変更を通知するためのアクションを作成します。アプリケーション全体の状態を変更する場合、必ずActionを通して行うようにします。また、状態を複雑に変更する必要がある場合もAction内に処理を記述することで、コンポーネントにView以外の処理が増えることを防げます。

リスト4.3.12.1 client/app/actions/ArticleActions.ts

import { AppStore } from '../common/AppStore';

import { RegisteredArticle } from '../models/RegisteredArticle';

export class ArticleAction {

static change(articles: RegisteredArticle[]) {

AppStore.getInstance().applyAppState('CHANGE', {

articles: articles

});

}

}

リスト4.3.12.2 client/app/actions/ProcessingModalAction.ts

import { AppStore } from "../common/AppStore";

export class ProcessingModalAction {

static setProcessingFlag(isProcessing: boolean): void {

AppStore.getInstance().applyAppState('CHANGE', {

isProcessing: isProcessing

});

}

}

### 4.3.13モデルの作成

最後にモデルを作成します。ただし、以下の3ファイルに関しては server/modelsディレクトリ内のファイルを client/app/models ディレクトリ内にコピーするだけです。

* Article.ts
* RegisteredArticle.ts
* RegisteredItem.ts

さらに、新たにAppState.tsを作成します。

リスト 4.3.13 client/app/models/AppState.ts

import { RegisteredArticle } from "./RegisteredArticle";

/\*\*

\* フロントエンドアプリケーション全体の状態

\*/

export interface AppState {

articles: RegisteredArticle[];

isProcessing: boolean;

}

### 4.3.14 タスクの実行

サーバーサイド、フロントエンドの開発手順の中で、それぞれのpackage.jsonの設定を行いました。このpackage.json内のscripts属性に設定したスクリプトを実行することで、作成したプログラムの実行やデバッグを行うことができます。ここでは、開発手順中に作成したpackage.jsonに設定したスクリプトの使用方法を説明します。

なお、とりあえず実行して動作を確認する場合、リスト4.3.14のコマンドを実行するのが簡単です。

リスト 4.3.14

$ cd [ワーキングディレクトリ『chapter-4』のパス]

$ cd ./client

$ npm run build

$ cd ../server

$ npm start

それぞれタスクが完了したら以下のURLをブラウザで開きます。

URL: http://localhost:3000/

#### 4.3.14.1 サーバーサイドのビルド

サーバーサイドのアプリケーションをビルドします。単にTypeScriptファイルをトランスパイルしているだけです。

リスト 4.3.14.1

$ npm run build

#### 4.3.14.2 サーバーサイドのアプリケーションの実行

サーバーサイドのアプリケーションの実行は以下のコマンドで行います。

リスト 4.3.14.2.1

$ npm run start

なお、testや、startなど一部のコマンドは以下のようにrunを省略できます。

リスト 4.3.14.2.2

$ npm start

#### 4.3.14.3 サーバーサイドの開発用スクリプト

開発をしていると、ソースコードの修正後すぐに動作を確認したいことがあるでしょう。そういう場合は以下のコマンドを使用するとスムーズに開発できます。これはTypeScriptファイルを監視して変更があった場合トランスパイルを行い、自動でアプリケーションの再実行まで行うスクリプトを定義しています。

リスト4.3.14.3

$ npm run dev

#### 4.3.14.4 フロントエンドの開発環境用ビルド

フロントエンドのアプリケーションをビルドします。ビルドした結果は、サーバーサイドのpublicディレクトリ内に出力されます。

リスト4.3.14.4

$ npm run build

#### 4.3.14.5 フロントエンドの本番環境用ビルド

フロントエンドのアプリケーションの本番環境用にビルドします。

リスト4.3.14.5

$ npm run build:prod

#### 4.3.14.6 フロントエンドの本番環境用AoTビルド

フロントエンドのアプリケーションの本番環境用にAoTビルドします。AoTとはAhead of Timeの略です。通常Angular2が内部でJavaScript、HTML、CSSからコンポーネントのコンパイルを実行時に行っています。AoTビルドでは、コンポーネントのコンパイルを、事前に行います。これにより、実行時のコンポーネントのコンパイルを行う必要がなくなるため起動時間が早くなります。

AoTビルドはコンポーネントのコンパイルも行う分ビルドに時間がかかります。そのため、開発中は通常のビルドで確認を行い、本番用AoTビルドを行うのが良いでしょう。

リスト4.3.14.6

$ npm run build:prod-aot

#### 4.3.14.6 フロントエンドの開発用スクリプト

こちらはフロント側のTypeScriptファイルの監視を行い、変更があった場合ビルドを行うスクリプトを定義しています。開発中はこちらを使用するのが便利です。

リスト4.3.14.6

$ npm run dev

おわりに

本書の執筆にあたり私を支えてくれた全ての人に感謝します。それから、私がTypeScriptを使ってプログラミングをするきっかけを与えてくれた、今はクローズしてしまった、とあるプロジェクトに感謝します。

**著者紹介**

**鈴木 潤（すずきじゅん）**

1989年6月27日生まれ。福島県いわき市出身。2013年茨城大学工学部情報工学科を卒業。システムインテグレータに入社。2016年にHR企業にエンジニアとして転職し今日に至る。好きな漫画はポプテピピック。

■Webサイト: https://www.belltree.tokyo/

■ブログ: ゆとり世代プログラマの備忘録<http://jbelltree.hatenablog.com/>

■GitHub: https://github.com/jsuzuki20120311

**TypeScriptでつくるシングルページアプリケーション**

2017年４月9日（技術書典2） 初版第一刷発行

著者: 鈴木潤

発行者: 鈴木潤

連絡先: shukatu.2012.5.25@gmail.com

印刷・製本: 株式会社ちょこっと

©2017 Jun Suzuki