AngularWebアプリ開発スタートブック

書籍情報

- 著者
  - 大澤 文孝
- 出版社
  - ソーテック社; B5変形版 (2018/4/7)
- 定価
  - 3,025円
- 発売日
  - 2018/4/7
- ISBN-13
  - 978-4-8026-1185-5
- ISBN-10
  - 4800711975
- 目次
  - Chapter1 Angularって何?
  - Chapter2 開発環境を整えよう
  - Chapter3 Angularプロジェクトを作ろう
  - Chapter4 Angularの基本
  - Chapter5 入力フォームを作ってみよう
  - Chapter6 入力エラーを検知するバリデータ
  - Chapter7 リアクティブフォーム入門
  - Chapter8 さまざまな入力コントロール
  - Chapter9 ページの割り当てと遷移
  - Chapter10 検索機能を実装する
  - Chapter11 Webサーバで動かす
- サンプルプログラム
  - http://www.sotechsha.co.jp/sp/1197/
- こんな方に
  - ○「将来、Webアプリ開発に取り組みたい」
  - ○「Angular特有の機能について学びたい」
  - ○「TypeScriptの作法について学びたい」

    - ○「新しいフレームワークの作法をざっと知りたい」

- はじめに

    - 次の２点を重点的に解説
        - Angularの動作の仕組み
        - 何をどのような書式で記述しなければならないのか

Chapter1 Angularって何?

Angularは平たく言うと、「テンプレート」と「プログラム」を分離し、テンプレートの定められた場所に、プログラムが管理するデータを差し込む仕組みで動くフレームワーク

テンプレートに結び付けられるプログラムのことを「コンポーネント」と呼び、テンプレートとコンポーネントは殆どの場合、１対１で対応する

テンプレートとコンポーネントは双方向で連携される

サービスという概念

コンポーネントから参照されるプログラムの塊

- Angularを構成する３大要素
    - テンプレート
    - コンポーネント（プログラム）
    - サービス

シングルページアプリケーション

Angularはシングルページアプリケーションと呼ばれる形態のアプリケーションを作ることを目的としている

シングルページアプリケーションはクライアント側でページを切り替えて表示を行うため、サーバーへの通信が発生せず高速

Angluarのメリット・デメリット

- Angluarのメリット
    - 効率よくクライアント側のプログラムが作れる様になる
    - 機能ごとに分離ができ、構造がシンプルになり、プログラムしやすい
    - Webシステムとして作りつつも、デスクトップアプリへの転用も可能
- Angluarのデメリット
    - 初期導入への時間
        - プログラムの書き方やファイル名の規則、ファイル同士の連携を記述する設定ファイルの書き方など、決まりごとがあるので作法を習得する
    - フレームワークの制限
        - フレームワークが対応しないことはできない、もしくはやりにくい
        - ★TODO：具体的にどんなこと？？あとで掘り下げる
    - アップデートの多さ

    - 半期ごとにバージョンアップされることにより旧バージョンで構築したシステムが陳腐化し、保守しづらくなる可能性もある

デメリットもあるが、それよりもアプリケーションの作りやすさや開発効率向上など、得られるメリットのほうが大きいはず

Chapter2 開発環境を整えよう

- 環境
    - テキストエディタ（Visual Studio Code）
    - Node.js
    - Typescript
    - Angular CLI

VisualStudioCodeをインストールする

割愛

TypescriptとAngular CLIをインストールする

- Typescript
    - `npm install -g typescript`
- Angular CLI
    - `npm install -g @angular/cli`

Chapter3 Angularプロジェクトを作ろう

プロジェクトの雛形を作る

- 雛形の作成手順
    1. Angularプロジェクトの作成：プロジェクトを保存するためのフォルダを作って雛形作成
    2. ソースファイルの編集：動作をTypescript、表示をHTMLで記述する
    3. 設定ファイルの編集：ソースファイルの関係や役割を設定ファイルに記述
    4. ビルド：Typescriptをビルドし、HTML5＋JavascriptのWebページを作成

Angularプロジェクトを作る

プロジェクトフォルダ作成 `angular_projects` という名前で作成することにする

プロジェクト作成 `ng new simpleform`

CSSとかSCSSとかを選択するプロンプトが表示されたが、一旦CSSで進めることにした

- 起動
    - `ng serve --open`

Chapter4 Angularの基本

プロジェクトフォルダ構成

フォルダ名またはファイル名	編集が必要？	用途
e2e	△	protractorというソフトウェアを使ってE2Eテスト（実際のユーザー操作をエミュレートしたテスト）をするときの構成ファイルを置くフォルダ
node_modules	×	Node.jsで利用するライブラリを格納する
src	◎	プログラムを配置
.angular-cli.json	○	このプロジェクトの設定を記すファイル
.editorconfig	△	エディタの設定ファイル
.gitignore	△	Git除外
karma.conf.js	△	単体テストを実行するkarmaというソフトの設定ファイル
package.json	△	node.jsのパッケージファイル
README.md	○	説明ファイル。概要ドキュメント
tsconfig.json	△	Typescriptの設定ファイル
tslint.json	△	Typescriptの文法チェックを設定するファイル

プログラムを格納するsrcフォルダ

フォルダ名またはファイル名	編集が必要？	用途
app	◎	アプリケーションを構成するフォルダ
assets	○	画像や動画など、参照させたいファイル群
environments	△	実行環境を設定するファイル
favicon.ico	○	このアプリケーションのアイコンファイル
index.html	◎	ブラウザが最初にアクセスするときに表示されるテンプレートファイル
main.ts	◎	ブラウザが最初にアクセスするときに表示されるTypeScriptファイル
polyfils.ts	△	ブラウザによる違いを吸収するためのファイル
style.css	◎	index.htmlに適用されるスタイルシート
tsconfig.app.json	△	TypeScriptの動作を設定するファイル
tsconfig.spec.json	△	karmaによる単体テストで使われるときのTypeScriptの動作を設定するファイル
typing.d.ts	△	TypeScriptの型情報を記述する

ページを構成する３つのファイル

- テンプレートファイル(*.html)
- TypeScriptファイル(*.ts)
- CSSファイル(*.css)

フォルダまたはファイル名	用途
app.module.ts	モジュールを構成するファイル
app.component.css	ルートコンポーネントのレイアウトを構成するCSSファイル
app.component.html	ルートコンポーネントを構成するHTMLファイル
app.component.ts	ルートコンポーネントを構成するTypeScriptファイル
app.component.spec.ts	ルートコンポーネントを構成する単体テスト用のTypeScriptファイル

```
// app.component.ts
import { Component } from '@angular/core';

// @ で始まる設定はTypeScriptに対して、動作の設定値を与える命令であり「デコレータ」と呼ばれる
@Component({
  selector: 'app-root', // HTMLファイルから参照するときの名称
  templateUrl: './app.component.html', // テンプレートファイル名
  styleUrls: ['./app.component.css'] // CSSファイル名
})
export class AppComponent {
  title = 'simpleform';
}
```

```
    <span>{{ title }} app is running!</span>
    <!-- {{ xxx }} という形で変数を注入する -->
```

```
// app.module.ts
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
```

```
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

新しいコンポーネントを追加する

- コマンドで追加したほうが、設定ファイルも書き換わるので安心安全
  - `ng g component simple-form`

引数	意味
g	generateの略
component	コンポーネントを構成するファイルの作成や登録をする
simple-form	コンポーネント名

`app`ディレクトリ直下に、フォルダが作成されることを確認

- 追加したコンポーネントを表示してみる
  - `app.component.html`に`<app-simple-form></app-simple-form>`を追加することで埋め込める

Chapter4のまとめ

- ページはテンプレート、TypeScript、CSSで構成される
- ページではコンポーネントを利用する
- ページにはコンポーネントのデータを`{{プロパティ}}`という表記で差し込める
- コンポーネントを追加するには、「`ng g component コンポーネント名`」を実行する

Chapter5 入力フォームを作ってみよう

足し算アプリを作る

```
<p>simple-form works!</p>

<input />
+
<input />
<button>CALC</button>
<div>{{ result }}</div>
```

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

@Component({
  selector: "app-simple-form",
  templateUrl: "./simple-form.component.html",
  styleUrls: ["./simple-form.component.css"]
})
export class SimpleFormComponent implements OnInit {
  result: string = "足し算しましょう";
  // コンポーネントが作られるときに実行したいプログラム
  constructor() {}
  // Angularによってコンポーネントが初期化されるときに実行したいプログラムを書く場所
  ngOnInit() {}
}
```

ボタンがクリックされたときの処理を作る

```
<p>simple-form works!</p>

<input />
+
<input />
<button (click)="addAndShow()">CALC</button>
<div>{{ result }}</div>
```

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

@Component({
  selector: "app-simple-form",
  templateUrl: "./simple-form.component.html",
  styleUrls: ["./simple-form.component.css"]
})
export class SimpleFormComponent implements OnInit {
  result: string = "足し算しましょう";
  constructor() {}

  ngOnInit() {}
  addAndShow(): void {
    // something
    this.result = "これはテスト";
  }
}
```

テキストボックスから値を読み込む

- テキストボックスを操作する仕組み
    - FormsModuleを使えるようにする
        - ngコマンドで自動生成したプロジェクトでは、最初FormsModuleが使えないので使えるようにする
            - `app.module.ts`を編集
    - プロパティを用意する
        - コンポーネントに対して、テキストボックスに入力された値を受け取るプロパティを実装する
    - テキストボックスとプロパティを関連付ける

FormsModuleを使えるようにするための設定

```
// app.module.tsファイル
import { BrowserModule } from "@angular/platform-browser";
import { NgModule } from "@angular/core";
import { FormsModule } from "@angular/forms"; // これを追加する！！★

import { AppRoutingModule } from "./app-routing.module";
import { AppComponent } from "./app.component";
import { SimpleFormComponent } from "./simple-form/simple-form.component";

@NgModule({
  declarations: [AppComponent, SimpleFormComponent],
  imports: [BrowserModule, AppRoutingModule, FormsModule], // 追加する！！★
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Chapter5のまとめ

- ボタンがクリックされたときの処理
- テキストボックスに入力された値の読み取り(FormsModuleを使う)
- 数値に変換
- 文字列に値を埋め込む（テンプレート記法）

Chapter6 入力エラーを検知するバリデータ

バリデータの基礎

入力状況によって表示を変更するためのCSSを作っていく

クラス名	意味
ng-touched	クリック（タッチ）されていて、編集中の項目である

クラス名	意味
ng-untouched	クリック（タッチ）されていない状態を表す
ng-valid	値が有効である
ng-invalid	値が無効である
ng-pending	バリデータの検証作業まち
ng-pristine	pristineとは「きれいな」という意味。入力内容が保存されていて、今ページを閉じても問題ないことを指す
ng-dirty	dirtyとは「きたない」という意味。ユーザーがデータを編集などして保存されていない状態

実際にフォームの値を変更しながら開発者コンソールでクラス名が変更されることを確認した

バリデータの設定

バリデータ	意味
min	最小値
max	最大値
required	空欄でない必須
requiredTrue	チェックが付いているなど選択されている
email	メールアドレスとして有効な書式
minLength	最小文字数
maxLength	最大文字数
pattern	指定した正規表現に合致する書式

```
<p>simple-form works!</p>

<input [(ngModel)]="text1" required />
+
<input [(ngModel)]="text2" />
<button (click)="addAndShow()">CALC</button>
<div>{{ result }}</div>
```

エラーメッセージを表示する

- Angularには、条件によって表示・非表示を切り替える仕組みがある
- 「*nglf」という属性を指定します。

    - `<span *ngIf="条件">左側のフィールドが空白です</span>`

```
<p>simple-form works!</p>

<input [(ngModel)]="text1" required #fieldOne="ngModel" />
+
<input [(ngModel)]="text2" required #fieldTwo="ngModel" />
<button (click)="addAndShow()">CALC</button>
<div>{{ result }}</div>
<div>
  <span *ngIf="fieldOne.invalid">左側のフィールドが空白です。</span><br />
  <span *ngIf="fieldTwo.invalid">右側のフィールドが空白です。</span>
</div>
```

未入力のときはボタンがクリックできないようにする

- 条件が成り立たないときは無効にする
    - `[disabled]="条件式"`
    - `<button [disabled="条件式"]>ラベル</button>`

```
<p>simple-form works!</p>

<input [(ngModel)]="text1" required #fieldOne="ngModel" />
+
<input [(ngModel)]="text2" required #fieldTwo="ngModel" />
<button
  (click)="addAndShow()"
  [disabled]="fieldOne.invalid || fieldTwo.invalid"
>
  CALC
</button>
<div>{{ result }}</div>
<div>
  <span *ngIf="fieldOne.invalid">左側のフィールドが空白です。</span><br />
  <span *ngIf="fieldTwo.invalid">右側のフィールドが空白です。</span>
</div>
```

まとめてひとつのフォームとして管理する

```
<p>simple-form works!</p>

<form #calcForm="ngForm">
  <input [(ngModel)]="text1" name="fieldOne" required #fieldOne="ngModel" />
  +
  <input [(ngModel)]="text2" name="fieldTwo" required #fieldTwo="ngModel" />
```

```
    <button
      (click)="addAndShow()"
      [disabled]="calcForm.invalid"
    >
      CALC
    </button>
    <div>{{ result }}</div>
    <div>
      <span *ngIf="fieldOne.invalid">左側のフィールドが空白です。</span><br />
      <span *ngIf="fieldTwo.invalid">右側のフィールドが空白です。</span>
    </div>
  </form>
```

- ポイント
  - formでくくる
  - input要素にはname属性を付与する
- 表示崩れ対応
  - form要素にもng-invalid属性とかが付与されるので除外する

```
.ng-valid:not(form) {
  border-bottom: 2px solid green;
}

.ng-invalid:not(form) {
  border-bottom: 2px solid red;
}
```

Chapter6のまとめ

- 状態に応じてCSSが適用される
- バリデータを設定する
  - requiredなどビルドインバリデータがあるので、それを利用することで正しく値が入力されたかどうか判定できる
- 条件によって表示・非表示を変更する
  - `*ngIf="条件式"`という属性を使う
- 条件によってクリックできないようにする
  - `[disabled]="条件"`

個人的に気になることリスト

- ビルドインバリデータ以外の実装方法について確認する
- CSSフレームワークの導入方法について確認する どのようにBootstrapとかを当てていくのがベストなのか

Chapter7 リアクティブフォーム入門

テンプレート駆動フォームとリアクティブフォーム

- Angularでフォームを構成する方法
    - テンプレート駆動フォーム
    - リアクティブフォーム
        - の２種類が存在する
- リアクティブフォーム
    - コンポーネントにあらかじめFormControlオブジェクトやFormGroupオブジェクト、バリデータオブジェクトなどを作っておく
    - テンプレートの入力コントロールからそれらのオブジェクトを参照して利用する

リアクティブフォームを作る

- 前章で作成したフォームをリアクティブフォームに作り直していく
- 新しいコンポーネントを作成する `ng g c better-form`

ReactiveFormsModuleを使えるようにする

```
// app-module.ts
import { BrowserModule } from "@angular/platform-browser";
import { NgModule } from "@angular/core";
import {
  FormsModule,
  FormGroup, // ★追加
  FormControl, // ★追加
  ReactiveFormsModule // ★追加
} from "@angular/forms";

import { AppRoutingModule } from "./app-routing.module";
import { AppComponent } from "./app.component";
import { SimpleFormComponent } from "./simple-form/simple-form.component";
import { BetterFormComponent } from "./better-form/better-form.component";

@NgModule({
  declarations: [AppComponent, SimpleFormComponent, BetterFormComponent],
  imports: [BrowserModule, AppRoutingModule, FormsModule, ReactiveFormsModule], //
★追加
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

FormGroupやFormControlを作って連結する

```
import { Component, OnInit } from "@angular/core";
import { FormControl, FormGroup } from "@angular/forms";

@Component({
  selector: "app-better-form",
  templateUrl: "./better-form.component.html",
  styleUrls: ["./better-form.component.css"]
})
export class BetterFormComponent implements OnInit {
  calcForm: FormGroup;
  result: string = "足し算しましょう";
  constructor() {}

  ngOnInit() {
    this.calcForm = new FormGroup({
      fieldOne: new FormControl("0"),
      fieldTwo: new FormControl("0")
    });
  }
  get fieldOne() {
    return this.calcForm.get("fieldOne");
  }
  get fieldTwo() {
    return this.calcForm.get("fieldTwo");
  }
  addAnyway() {
    let text1: string = this.fieldOne.value;
    let text2: string = this.fieldTwo.value;
    let resultStr: string = "";
    if (Number.isNaN(Number(text1)) || Number.isNaN(Number(text2))) {
      resultStr = text1 + text2;
    } else {
      resultStr = `${text1}+${text2}=${Number(text1) + Number(text2)}`;
    }
    this.result = resultStr;
  }
}
```

```
<p>better-form works!</p>

<form [formGroup]="calcForm">
  <input formControlName="fieldOne" />
  +
  <input formControlName="fieldTwo" />
  <button (click)="addAnyway()" [disabled]="calcForm.invalid">
    CALC
  </button>
  <div>{{ result }}</div>
  <div>
```

```
    <span *ngIf="fieldOne.invalid">左側のフィールドが空白です。</span><br />
    <span *ngIf="fieldTwo.invalid">右側のフィールドが空白です。</span>
  </div>
</form>
```

Validatorを追加する

```
<p>better-form works!</p>

<form [formGroup]="calcForm">
  <input
    formControlName="fieldOne"
    [class.input-invalid]="fieldOne.invalid"
    [class.input-valid]="fieldOne.valid"
  />
  +
  <input
    formControlName="fieldTwo"
    [class.input-invalid]="fieldTwo.invalid"
    [class.input-valid]="fieldTwo.valid"
  />
  <button (click)="addAnyway()" [disabled]="calcForm.invalid">
    CALC
  </button>
  <div>{{ result }}</div>
  <div *ngIf="fieldOne.invalid && fieldOne.errors.required">
    左側のフィールドが空白です。
  </div>
  <div *ngIf="fieldTwo.invalid && fieldTwo.errors.required">
    右側のフィールドが空白です。
  </div>
  <div *ngIf="fieldTwo.invalid && fieldTwo.errors.maxlength">
    右側のフィールドの最大は５文字
  </div>
</form>
```

```
// better-form.component.ts
import { Component, OnInit } from "@angular/core";
import { FormControl, FormGroup, Validators } from "@angular/forms";

@Component({
  selector: "app-better-form",
  templateUrl: "./better-form.component.html",
  styleUrls: ["./better-form.component.css"]
})
export class BetterFormComponent implements OnInit {
  calcForm: FormGroup;
```

```
  result: string = "足し算しましょう";
  constructor() {}

  ngOnInit() {
    this.calcForm = new FormGroup({
      fieldOne: new FormControl("0", [Validators.required]),
      fieldTwo: new FormControl("0", [
        Validators.required,
        Validators.maxLength(5)
      ])
    });
  }
  get fieldOne() {
    return this.calcForm.get("fieldOne");
  }
  get fieldTwo() {
    return this.calcForm.get("fieldTwo");
  }
  addAnyway() {
    let text1: string = this.fieldOne.value;
    let text2: string = this.fieldTwo.value;
    let resultStr: string = "";
    if (Number.isNaN(Number(text1)) || Number.isNaN(Number(text2))) {
      resultStr = text1 + text2;
    } else {
      resultStr = `${text1}+${text2}=${Number(text1) + Number(text2)}`;
    }
    this.result = resultStr;
  }
}
```

入力項目にキー入力されたタイミングで計算結果を消去する

- inputタグに `(keyup)="clearResult()"` で関数を設定する

```
<input (keyup)="clearResult()"/>
```

```
  clearResult() {
    this.result = "";
  }
```

Chapter7のまとめ

- リアクティブフォームとは（⇔テンプレート駆動フォーム）
  - FormGroupやFormControlなどをコンポーネント側に持つ仕組みのこと
  - これらのオブジェクトはプロパティとして公開しておく

- テンプレートとリアクティブフォームとの連結
  - `<form [formGroup]="プロパティ名">`
  - `<input formControlName="プロパティ名">`
- 入力された値の取得
  - `let text1:string = this.fieldOne.value;`
- バリデータの設定
  - `new FormControl("0", Validators.required)`
  - `new FormControl("0", [Validators.required, Validators.maxLength(5)])` 複数ある場合は配列で指定する
- 条件付きのCSSクラスを指定する
  - `[class.クラス名="条件式"]`という書式を使う
- キーボードイベント
  - `(keyup)="メソッド名()"`のように記述すると指定されたメソッドを実行できる

Chapter8 さまざまな入力コントロール

新しいコンポーネントを作成する

- `ng g c controls`

FormBuilderを使った入力フォームの作成

```
<p>controls works!</p>
<h2>コーヒー品目リスト作成</h2>
<form [formGroup]="coffeeForm" novalidate>
  <div>
    <label>品名: <input formControlName="name"/></label>
  </div>
  <div>
    <label>テイスト：<input formControlName="taste"/></label>
  </div>
</form>
<p>フォーム入力値: {{ coffeeForm.value | json }}</p>
```

- Angularで提供されている標準フィルタ

フィルタ	意味
filter	条件に合致するものだけに絞り込む
currency	金銭表示に整形する
number	数値表示に整形する
date	日付表示に整形する
json	JSON形式に整形する
lowercase	小文字に変換する

フィルタ	意味
uppercase	大文字に変換する
limitTo	取り出す最大数を制限する
orderBy	並べ替える

```
import { Component, OnInit } from "@angular/core";
import { FormControl, FormGroup, FormBuilder } from "@angular/forms";

@Component({
  selector: "app-controls",
  templateUrl: "./controls.component.html",
  styleUrls: ["./controls.component.css"]
})
export class ControlsComponent implements OnInit {
  coffeeForm: FormGroup;
  // FormBuilderの注入
  constructor(private fb: FormBuilder) {
    this.coffeeForm = this.fb.group({
      name: "ブレンド",
      taste: "バランスの良い口当たり"
    });
  }

  ngOnInit() {}
}
```

```
<p>controls works!</p>
<h2>コーヒー品目リスト作成</h2>
<form [formGroup]="coffeeForm" novalidate>
  <div>
    <label>品名: <input formControlName="name"/></label>
  </div>
  <div>
    <label>テイスト：<input formControlName="taste"/></label>
  </div>
</form>
<p>フォーム入力値: {{ coffeeForm.value | json }}</p>
```

ラジオボタンの実装

```
import { Component, OnInit } from "@angular/core";
import { FormControl, FormGroup, FormBuilder } from "@angular/forms";
```

```
@Component({
  selector: "app-controls",
  templateUrl: "./controls.component.html",
  styleUrls: ["./controls.component.css"]
})
export class ControlsComponent implements OnInit {
  coffeeForm: FormGroup;
  // FormBuilderの注入
  constructor(private fb: FormBuilder) {
    this.coffeeForm = this.fb.group({
      name: "ブレンド",
      taste: "バランスの良い口当たり",
      hotcold: "Hot"
    });
  }

  ngOnInit() {}
}
```

```
<p>controls works!</p>
<h2>コーヒー品目リスト作成</h2>
<form [formGroup]="coffeeForm" novalidate>
  <div>
    <label>品名：<input formControlName="name"/></label>
  </div>
  <div>
    <label>テイスト：<input formControlName="taste"/></label>
  </div>
  <div>
    <span><input type="radio" formControlName="hotcold" value="Hot" />Hot</span>
    <span
      ><input type="radio" formControlName="hotcold" value="Cold" />Cold</span
    >
  </div>
</form>
<p>フォーム入力値：{{ coffeeForm.value | json }}</p>
```

***ngForで繰り返し処理を書く**

```
import { Component, OnInit } from "@angular/core";
import { FormControl, FormGroup, FormBuilder } from "@angular/forms";

@Component({
  selector: "app-controls",
  templateUrl: "./controls.component.html",
  styleUrls: ["./controls.component.css"]
})
```

```
export class ControlsComponent implements OnInit {
  coffeeForm: FormGroup;
  hotcoldsel = ["Hot", "Cold", "VeryHot", "VeryCold"];
  // FormBuilderの注入
  constructor(private fb: FormBuilder) {
    this.coffeeForm = this.fb.group({
      name: "ブレンド",
      taste: "バランスの良い口当たり",
      hotcold: this.hotcoldsel[0]
    });
  }

  ngOnInit() {}
}
```

```
<p>controls works!</p>
<h2>コーヒー品目リスト作成</h2>
<form [formGroup]="coffeeForm" novalidate>
  <div>
    <label>品名: <input formControlName="name"/></label>
  </div>
  <div>
    <label>テイスト：<input formControlName="taste"/></label>
  </div>
  <div>
    <span *ngFor="let state of hotcoldsel"
      ><input type="radio" formControlName="hotcold" [value]="state" />{{
        state
      }}</span
    >
  </div>
</form>
<p>フォーム入力値: {{ coffeeForm.value | json }}</p>
```

チェックボックスを追加する

```
import { Component, OnInit } from "@angular/core";
import { FormControl, FormGroup, FormBuilder, FormArray } from "@angular/forms";
@Component({
  selector: "app-controls",
  templateUrl: "./controls.component.html",
  styleUrls: ["./controls.component.css"]
})
export class ControlsComponent implements OnInit {
  coffeeForm: FormGroup;
  hotcoldsel = ["Hot", "Cold", "VeryHot", "VeryCold"];
  addssel = ["Milk", "Sugar"];
```

```
  // FormBuilderの注入
  constructor(private fb: FormBuilder) {
    this.coffeeForm = this.fb.group({
      name: "ブレンド",
      taste: "バランスの良い口当たり",
      hotcold: this.hotcoldsel[0],
      adds: this.fb.array([])
    });
  }
  onCheckChanged(item: string, isChecked: boolean) {
    let formArray = <FormArray>this.coffeeForm.controls.adds;
    if (isChecked) {
      formArray.push(new FormControl(item));
    } else {
      let index = formArray.controls.findIndex(elm => elm.value == item);
      formArray.removeAt(index);
    }
  }

  ngOnInit() {}
}
```

```
<p>controls works!</p>
<h2>コーヒー品目リスト作成</h2>
<form [formGroup]="coffeeForm" novalidate>
  <div>
    <label>品名: <input formControlName="name"/></label>
  </div>
  <div>
    <label>テイスト：<input formControlName="taste"/></label>
  </div>
  <div>
    <span *ngFor="let state of hotcoldsel"
      ><input type="radio" formControlName="hotcold" [value]="state" />{{
        state
      }}</span
    >
  </div>
  <div>
    <span *ngFor="let item of addssel"
      ><input
        type="checkbox"
        (change)="onCheckChanged(item, $event.target.checked)"
      />{{ item }}</span
    >
  </div>
</form>
<p>フォーム入力値: {{ coffeeForm.value | json }}</p>
```

ドロップダウンリストを追加する

```
import { Component, OnInit } from "@angular/core";
import { FormControl, FormGroup, FormBuilder, FormArray } from "@angular/forms";
@Component({
  selector: "app-controls",
  templateUrl: "./controls.component.html",
  styleUrls: ["./controls.component.css"]
})
export class ControlsComponent implements OnInit {
  coffeeForm: FormGroup;
  hotcoldsel = ["Hot", "Cold", "VeryHot", "VeryCold"];
  addssel = ["Milk", "Sugar"];
  nutsel = ["ピーナッツ", "アーモンド", "くるみ"];
  // FormBuilderの注入
  constructor(private fb: FormBuilder) {
    this.coffeeForm = this.fb.group({
      name: "ブレンド",
      taste: "バランスの良い口当たり",
      hotcold: this.hotcoldsel[0],
      adds: this.fb.array([]),
      nut: this.nutsel[0]
    });
  }
  onCheckChanged(item: string, isChecked: boolean) {
    let formArray = <FormArray>this.coffeeForm.controls.adds;
    if (isChecked) {
      formArray.push(new FormControl(item));
    } else {
      let index = formArray.controls.findIndex(elm => elm.value == item);
      formArray.removeAt(index);
    }
  }

  ngOnInit() {}
}
```

```
<p>controls works!</p>
<h2>コーヒー品目リスト作成</h2>
<form [formGroup]="coffeeForm" novalidate>
  <div>
    <label>品名: <input formControlName="name"/></label>
  </div>
  <div>
    <label>テイスト：<input formControlName="taste"/></label>
  </div>
  <div>
    <span *ngFor="let state of hotcoldsel"
      ><input type="radio" formControlName="hotcold" [value]="state" />{{
```

```
          state
        }}</span
      >
    </div>
    <div>
      <span *ngFor="let item of addssel"
        ><input
          type="checkbox"
          (change)="onCheckChanged(item, $event.target.checked)"
        />{{ item }}</span
      >
    </div>
    <div>
      <label
        >おつまみ：
        <select formControlName="nut">
          <option *ngFor="let nut of nutsel" [value]="nut">{{ nut }}</option>
        </select>
      </label>
    </div>
  </form>
  <p>フォーム入力値: {{ coffeeForm.value | json }}</p>
```

Chapter8のまとめ

- 各種入力コントロール
  - テキストボックスと同じようにformControlNameで、FormControlと結び付ける
- FormBuilder
  - FormBuilderを使うと、FormGroupやFormControlの作成が簡単になります。
- *ngFor
  - 繰り返し処理が出来る
  - `*ngFor="let 変数名 of 配列などの値"`
- チェックボックス
  - 値が配列になるので処理が特殊
  - `(change)="メソッド名"`でチェックの状態が変わったときには、その状態によってオブジェクトを追加したり削除したりすることでどの選択肢が選択されているかを設定するようにプログラミングします。

Chapter9 ページの割り当てと遷移

- URLと表示するコンポーネントを関連付ける方法「ルーティング」
- `Routermodule`というモジュールを使ってプログラミングする

```
ng g module app-routing --flat --module=app
```

- app-routing.module.tsを編集
  - 上記のコマンドで生成されると思っていたが、最近のAngularだと初期で生成しているみたいで、実行するとエラーになった。

- 以下のファイルにルーティング情報を追加

```
import { NgModule } from "@angular/core";
import { Routes, RouterModule } from "@angular/router";
import { SimpleFormComponent } from "./simple-form/simple-form.component"; // ★追記
import { BetterFormComponent } from "./better-form/better-form.component"; // ★追記
import { ControlsComponent } from "./controls/controls.component"; // ★追記

const routes: Routes = [];

@NgModule({
  imports: [
    RouterModule.forRoot([
      { path: "simple-form", component: SimpleFormComponent }, // ★追記
      { path: "better-form", component: BetterFormComponent }, // ★追記
      { path: "controls", component: ControlsComponent } // ★追記
    ])
  ],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

- app.component.htmlを編集

```
<nav>
  <a routerLink="./simple-form">simple-form</a> |
  <a routerLink="./better-form">better-form</a> |
  <a routerLink="./controls">controls</a>
</nav>
<div><router-outlet></router-outlet></div>
<!-- <app-simple-form></app-simple-form> -->
<!-- <app-better-form></app-better-form> -->
<!-- <app-controls></app-controls> -->
```

タブらしい表示にする

- CSSを定義する

```
<nav>
  <a routerLink="./simple-form" routerLinkActive="selected-item">simple-form</a>
  |
  <a routerLink="./better-form" routerLinkActive="selected-item">better-form</a>
  |
  <a routerLink="./controls" routerLinkActive="selected-item">controls</a>
```

```
</nav>
<div><router-outlet></router-outlet></div>
<!-- <app-simple-form></app-simple-form> -->
<!-- <app-better-form></app-better-form> -->
<!-- <app-controls></app-controls> -->
```

```
a {
  padding: 10px;
  margin-right: 4px;
  background-color: darkolivegreen;
  color: khaki;
  text-decoration: none;
}
a:link,
a:visited,
a:hover {
  color: khaki;
}
a.selected-item {
  color: darkolivegreen;
  background-color: khaki;
}
div {
  padding: 50px;
  background-color: khaki;
  color: darkolivegreen;
}
```

ループでリンクを構成する

- 重要
  - 要素の中に変数を用いるときは、属性を角括弧で囲み、変数は文字列に入れるルールがある
  - テキストとして出力する場合は{{}}二重の波括弧で囲む

ドキュメントルートをリダイレクトする

- app-routing.module.tsを編集して、リダイレクトするようにする

```
import { NgModule } from "@angular/core";
import { Routes, RouterModule } from "@angular/router";
import { SimpleFormComponent } from "./simple-form/simple-form.component";
import { BetterFormComponent } from "./better-form/better-form.component";
import { ControlsComponent } from "./controls/controls.component";

const routes: Routes = [];
```

```
@NgModule({
  imports: [
    RouterModule.forRoot([
      { path: "", redirectTo: "/simple-form", pathMatch: "full" }, // ★追加
      { path: "simple-form", component: SimpleFormComponent },
      { path: "better-form", component: BetterFormComponent },
      { path: "controls", component: ControlsComponent }
    ])
  ],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

マスター/ディテイルアプリを構成する

「一覧をクリックすると、その詳細情報が表示される」というように、「主たるデータのページ」から「その詳細ページ」に遷移する方式のものもある

こうした構成を取るものをマスター/ディテイルアプリ(master-detail application)といいます

- このSectionで作るアプリケーション「レシピ一覧メニュー」

COLUMN もっとたくさんのパラメータを渡したいとき

- さらにパラメータをつける
- パラメータを「;」で区切る
- 「?」を使う

一覧ページと詳細ページをリンクする

- `import {Location} from "@angular/common"`をインポートして
- コンストラクタに追加
- `this.location.back()`をメソッドに定義して直前の画面に戻るリンクを追加する

詳細ページに画像などを表示する

- assetsフォルダに画像ファイル配置
- 対象のデータをfindで見つけて、コンポーネント側で保持する

Chapter9のまとめ

- RoutingModuleを使ったURLパスとコンポーネントのマッピング
  - ページ遷移するには、RoutingModuleを使って、URLパスとコンポーネントをマッピング
- コンポーネントの表示
  - コンポーネントを表示したいところには、「`<router-outlet></router-outlet>`」と記述
- URLでパラメータを渡す
  - マスターディテイルアプリケーションを構成する場合など、詳細ページにパラメータを渡したいときは、RouterModuleで構成するパス内に「`:パラメータ名`」例えば:id

    - と記述します。その値は、AcrivatedRouteオブジェクトの「`.snapshot.paramMap.get(パラメータ名)`」として取得します。

Chapter10 検索機能を実装する

データ操作するためのサービス

- データ操作するサービスを作る
    - `ng g s recipe`
        - 本書では、`ng g s recipe --module=app`となっていたが、オプションをつけると、
        - `Unknown option: '--module'`と表示されたので削除した

一覧ページ修正

- `ng-container`タグを表示したくないが表示非表示の条件などを入れたい場合に使用するコンテナ

詳細ページを修正する

- 詳細ページに「何人分」「材料のリスト」を表示できるようにHTMLを修正する

検索機能を作る

- サービスに実装して呼び出す

COLUMN リンク先から戻ったときにテキストボックスの内容が消えないようにする

- 方法はいくつかあるらしいが、サービス側に変数として保持しておいて、戻ったときに再度その値を取得する方法を記載してくれている

Chapter10のまとめ

- サービスオブジェクトを導入する
- `<ng-container>`で要素なしの出力になる
- ループのインデックス
    - `*ngFor`を使ってループ処理をするときにindexを指定すると、ループ回数を取得できます。

Chapter11 Webサーバで動かす

Webサーバで動かすには

- ビルド種類
    - developmentモード
        - 開発に使うモード。もとのtypescriptファイルを維持するように変k何します。
        - sourcemapファイルと呼ばれるデバッグ用のファイルを作成し、Webサーバーに配置した状態でソースを確認しながらデバッグできます。
    - productionモード
        - 本番稼働に使うモード。不要なコードを排除し、ファイルサイズが小さく、また実行効率がよくなるように調整されます。

- ビルドコマンド
    - `ng build --prod`
        - このオプションをつけると、puroductionモードでビルドされる
        - つけないとdevelopmentモードになる

Webサーバ経由で実行してみる

http-serverで動かしてみる例

```
npm install -g http-server
// 生成されたdistディレクトリをカレントにして以下を実行
http-server
```

COLUMN サブディレクトリに公開する

- index.htmlのbaseを修正する
    - `<base href="/cookbook">`
- bhオプションをつけてビルドする
    - `ng build --prod --bh /cookbook/`

Chapter11のまとめ

- ビルドする
    - `ng build --prod`
- distフォルダのコピー

個人的アウトプット

Overview

if you are looking for property that is perfect for you, then this site may be useful for you.

Architecture

used the following services.

- AWS S3
- AWS API Gateway
- AWS Lambda

URL

http://contents.ang-fnt-suumo.t-tsutsui.s3-website-ap-northeast-1.amazonaws.com/property-list

VSCODEの拡張機能

- **Angular Essentials**をインストールする