Angularアプリケーションプログラミング

- 電子書籍購入
- 説明書

こちらの書籍は 2019/06/19 紙版の3刷に対応するため更新をおこないました。
（概要）
本書は，JavaScriptの定番SPA開発フレームワーク「Angular」の解説書です。データバインディング，コンポーネント，依存性注入といったAngularを理解するための基本要素をはじめ，ディレクティブ／パイプ，フォーム，ルーティング，モジュール／コンポーネントの技術解説，ディレクティブ／パイプ／サービスの自作やテストなどの応用的なテーマまでを網羅。また，Angular 4に対応し，5以降のアップグレードにも留意した解説を行っています。Angularによる動的Webアプリケーション開発に必要となる知識が，この1冊で身に付きます！

（こんな方におすすめ）
・Angularを利用したSPAアプリケーション開発に興味のある人

（目次）
導入編
第1章イントロダクション

第2章Angular の基本

第3章データバインディング

基本編
第4章標準パイプ／ディレクティブ

第5章フォーム開発

第6章コンポーネント開発

第7章サービス開発

応用編
第8章ルーティング

第9章パイプ／ディレクティブの自作

第10章テスト

第11章関連ライブラリ／ツール

AppendixTypeScript簡易リファレンス

フォーマット： Kindle版
ファイルサイズ： 56752 KB
推定ページ数： 838 ページ
出版社: 技術評論社 (2017/8/4)

```
販売： Amazon Services International, Inc.
言語: 日本語
ASIN: B074M54GM5
Text-to-Speech（テキスト読み上げ機能）: 有効
X-Ray:
有効
Word Wise: 有効にされていません
カスタマーレビュー: 5つ星のうち 4.7　14件のカスタマーレビュー
Amazon 売れ筋ランキング: Kindleストア 有料タイトル - 21,411位 (Kindleストア 有料タイトルの売れ筋ランキングを見る)
1165位 – 工学 (Kindleストア)
```

- 書籍情報ページhttps://wings.msn.to/index.php/-/A-03/978-4-7741-9130-0/

第1章イントロダクション

1.1 JavaScriptの歴史

1.1.1 不遇の時代を経てきた JavaScript

- ダイナミックHTMLの話
- ダサいページを作るための低俗な言語

1.1.2 復権のきっかけは Ajax、そして HTML5 の時代へ

- 2005年にAjax(Asynchronous Javascript + XML)が登場
    - ブラウザベンダによる拡張機能合戦も落ち着く
    - ECMAInternatinalによる標準化
    - 大規模な開発にも耐えられるオブジェクト指向的な書き方が求められる
- 2000年代後半HTML5の登場
    - GeolocationAPI
    - CanvasAPI
    - FileAPI
    - WebStrage
    - WebWorkers
    - WebSockets
- TODO: javascriptAPI使う
- TODO: RIA(Rich Internet Applications)覚える

1.1.3 JavaScript ライブラリから JavaScript フレームワークへ

- Javascriptは決して生産性の高い言語ではない
    - 型の制約がゆるい
    - Javascript固有の癖がある

    - ブラウザによって挙動が異なる（クロスブラウザ問題）
- 開発の規模が増大するに伴い、Javascriptの世界でもサーバーサイドでの開発と同じく、本格的なフレームワークが求められるようになってきた

1.2 フレームワークとは？

- アプリケーションフレームワークはどのようなしくみなのでしょうか
    - 一般的なフレームワークの特徴と、導入の利点をまとめます

1.2.1 フレームワークの本質

- ユーザーコードをフレームワークが呼び出す
- フレームワークがアプリのライフサイクル（初期化から実処理、終了までの流れ）を管理
- その要所要所で何をすべきかをユーザーコードに問い合わせる
- このように、プログラム実行の主体が逆転する性質のことを制御の反転（IoC:Inversion of Control）と言います
- 制御の反転こそがフレームワークの本質であるといってもいいでしょう

1.2.2 フレームワーク導入の利点

- 利点
    - 開発生産性の向上
    - メンテナンス性に優れる
    - 先端の技術トレンドにも対応しやすい
        - 最新のセキュリティなどのトレンドにも対応している
    - 一定上の品質が期待できる
        - 信頼性が高い
        - オープンソースとして使われているフレームワークは多くの人の目にさらされ、テストされている
        - 自分や限られた人の目しか通していないコードに比べれば、相対的に信頼性が高い
- NOTE: フレームワーク導入のデメリット
    - ルール（制約）の集合で学習時間が必要

1.2.3 JavaScript で利用可能なフレームワーク

- Javascriptフレームワーク
    - Angular：Googleを中心に開発されているフルスタックフレームワーク
    - AngularJS：Angularの前身。
    - React：Facebook製で、MVCのView相当の機能を提供
    - Vue.js：ビュー層に特化したシンプルかつ高速なフレームワーク。断片的に適用していけることからプログレッシブフレームワークとも
    - Aurelia：元Angularのメンバーが開発↓フレームワーク。次期ECMAScript7の機能も積極的に採用している
- Googleトレンドを見ても、Angularはシェアも大きく注目されている

1.3 Angular の特徴

- Angularの特徴
    - フルスタックのフレームワークである
        - HTMLベースのテンプレートエンジン
        - コンポーネント/テンプレート間のデータバインディング機能
        - コンポーネント/サービス間の依存関係を解決するDIコンテナー
        - 文書ツリーを操作するためのディレクティブ
        - 表示すべき値を加工するためのパイプ
        - ビジネスロジックを実装するためのサービス
        - URLに応じてページを振り分けるルーティング機能
        - 単体テスト/シナリオテストを支援するテストフレームワーク
    - コンポーネント指向である
        - コンポーネントとは、ページを構成するUI部品のこと
        - Angularでは部品化されたコンポーネントを組み合わせることでページを組み立てていくのが基本
    - モダンな技術をふんだんに取り入れている
        - イマドキのJavascript技術が積極的に採用されている
            - TypeScript: C#によくにた構文を持つ、Javascriptの拡張言語
            - SystemJS： モジュールを動的にロードするためのライブラリ
            - RxJS：Javascriptで非同期処理を実装するためのライブラリ
            - Zone.js：非同期処理のコンテキストを管理、操作するためのライブラリ
                - TODO:必要になったらこのライブラリについて学んでいく
    - 開発言語にはTypeScriptがオススメ
        - ドキュメントがほぼほぼTypeScriptを前提で書かれている
        - 新しい仕様も取り入れられている
    - ドキュメント/周辺ライブラリが充実している
        - ngx-bootstrap
        - AngularCLI: Angularアプリの骨格を自動生成する
        - Augury: Chromeプラグイン
        - VSC: Angular+Typescriptで開発に対応

1.3.1 Angular のバージョン

- AngularJS 1.x
- Angular 2
- Angular 4 2017/03 セマンティクスバージョニングで半年に一回メジャーバージョンアップ
- Angular 5 2017/09
- Angular 6 2018/03
- Angular 7 2018/09
- Angular 8 2019/03
- Angular 9 2019/09 現在はここがSTABLE
- next..

第2章Angular の基本

2.1 Angularを利用するための準備

- QuickStartプロジェクトを利用すると便利
- Angularアプリを動作させるための基本的なモジュール/設定ファイルが準備されており、最低限の手続きでアプリ開発を開始できる

2.1.1 Angular アプリの基本構造

- Node.jsインストール
- QuickStartプロジェクトをダウンロードする
- アプリ内容を確認する
- ライブラリをインストールする
- HTTPサーバーを起動する
- 問題なく起動できた
- MEMO: AngularCLIで作成したアプリと何が違うのか、どちらが新しいのか確認したい

2.2 サンプルアプリの内容を確認する

2.2.1 Angular アプリの構成部品を束ねる ー モジュール

- main.ts : アプリを起動するためのスタートアップコードを記載
- app.module.ts : ルートモジュール
- app.component.ts

```
// 利用するオブジェクトをインポート
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './app.component';

// デコレータを定義
@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { } // クラスをエクスポート（外部公開）
```

- MEMO: javaで言うアノテーションをデコレータという
- `@NgModule`デコレータで使用できるパラメータ
    - imports: 現在のモジュールで利用する他のモジュール
    - exports: 現在のモジュールから外部に公開するコンポーネント
    - declarations: 現在のモジュールに属するコンポーネント
    - bootstrap: アプリで最初に起動すべき最上位のコンポーネント（=ルートコンポーネント）

    - id: モジュールのid値

- インポートすべきモジュールは利用する機能によって変化するのでその時々で追加する
    - よく利用するもの
        - FormsMdule: フォーム機能
        - RouterModule: ルーティング機能

- 補足：Angularを構成する要素の命名規則
    - クラス名
        - 名前＋種類でUpperCamelCase記法
    - ファイル名
        - 名前.種類.tsでKebabCase記法（すべて小文字で単語の区切りはハイフン）
    - テストスクリプト
        - テスト対象のファイル名に「.spec.ts」を追加

- Note: AngularモジュールとJavascriptモジュール
    - Angularモジュール
        - アプリを構成するコンポーネント/サービス/ディレクティブ/パイプを束ねるための論理的な器
        - 機能単位にグループ化するしくみ
        - @NgModuleデコレータを使って宣言
        - 別のモジュールをインポートする際は、importsパラメータを利用
    - Javascriptモジュール
        - 物理的なファイル１つ１つのこと
        - 別のモジュールからの参照を許可するときは、exportキーワードを指定
    - ※両者を分けて考えること

2.2.2 ページを構成する UI 部品を定義する ― コンポーネント

- Angularアプリのキモとも言うべきコンポーネント（app.component.ts）について見ていく
- シンプルなページの場合、１ページ１コンポーネントもあるが、一般的には組み合わせて構成される
- app.component.tsはアプリで最初に呼び出されるコンポーネント（=ルートコンポーネント、メインコンポーネント）

```
// コンポーネントを定義するために必要となるオブジェクトをインポート
import { Component } from "@angular/core";

// @Componentデコレータでコンポーネントの構成情報を宣言
@Component({
  selector: "my-app",
  template: `
    <h1>Hello {{ name }}</h1>
  `
})
export class AppComponent { // 外部から参照できるようにexport
```

```
  name = "Angular";
}
```

- `@Component`デコレータの主なパラメータ
    - selector: コンポーネントを適用すべき要素を表すセレクター式
    - templete: コンポーネントを適用するビュー（テンプレート）
- {{}} という構文は、Interpolation(補間)という構文
- 予め用意された変数（ビュー変数）を埋め込むためのプレイスホルダーと言い換えても構いません
- MEMO: ここまで理解オッケー

2.2.3 Angular アプリを起動するためのスタートアップコード ― main.ts

- `main.ts` はappフォルダではなく、src直下に配置されている点に注意

```
// ブラウザーでアプリを起動するために用意されたAngular標準の関数
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
// メインモジュールをインポート
import { AppModule } from './app/app.module';
// モジュールを起動するためのメソッド実行
platformBrowserDynamic().bootstrapModule(AppModule);
```

2.2.4 メインページの準備 ― index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Angular QuickStart</title>
    <base href="/" />
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <link rel="stylesheet" href="styles.css" />

    <!-- Polyfill(s) for older browsers -->
    <!-- レガシーなブラウザに対応するためのJS -->
    <script src="node_modules/core-js/client/shim.min.js"></script>

    <!-- Angularの動作に必要なライブラリ -->
    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>

    <!-- SystemJSの設定情報 -->
    <script src="systemjs.config.js"></script>
    <script>
      System.import("main.js").catch(function(err) {
```

```
        console.error(err);
      });
    </script>
  </head>

  <body>
    <!-- コンポーネントがロードされるまでに表示するコンテンツを指定 -->
    <my-app>Loading AppComponent content here ...</my-app>
  </body>
</html>
```

- TODO: 確認> polyfillや必要なライブラリのインポートについて、CLIで作成したプロジェクトはインポートしていないように見えるため確認したい

2.3 基本ライブラリの挙動を宣言する ー 設定ファイル

- アプリコンパイル/実行に必要となる設定ファイル
  - package.json
  - tsconfig.json
  - systemjs.config.js: モジュールローダー（SystemJS）の設定情報
    - TODO: このファイルはAngularCLIで作成したプロジェクトにはない

2.3.1 アプリで利用するライブラリ情報を管理する ー package.json

- package.json
  - npmコマンドで利用できるサブコマンド
    - `"start": "concurrently \"npm run build:watch\" \"npm run serve\""`
      - concurrentlyは指定された命令を並列で実行するコマンド
  - アプリで利用するライブラリ

2.3.2 TypeScript コンパイラーの挙動を設定する ー tsconfig.json

- tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "lib": ["es2015", "dom"],
    "noImplicitAny": true,
    "suppressImplicitAnyIndexErrors": true
  }
}
```

- 公式ページhttps://www.typescriptlang.org/docs/handbook/tsconfig-json.html
- 設定を一覧にしてくれているページhttps://qiita.com/ryokkkke/items/390647a7c26933940470

2.3.3 SystemJS の挙動を宣言する ー systemjs.config.js

- SystemJSは、Javascriptのモジュールを動的にロードするためのライブラリ
- defaultExtensionsサブパラメータ
    - jsを指定している
- TODO: AngularCLIでは、SystemJSがないが、どのようにこれらの値を決めているのか確認する

2.4 学習を進める前に

2.4.1 サンプルファイルの入手

- 入手方法とセットアップ方法

2.4.2 サンプルアプリの利用方法

第3章データバインディング

3.1 データバインディングの基本

3.1.1 4 種類のバインディング構文

3.2 Interpolation（補間）構文

3.2.1 Interpolation の基本ルール

- コンポーネント側にプロパティを定義する
- 関数({{getList()}})でも、計算式({{3*5}})でもよい

3.2.2 {{...}} 式の注意点

- {{}}式で利用できない演算子
    - 代入演算子
    - new演算子
    - [;][,]など連結を表す演算子
    - インクリメント/デクリメント演算子（++、--）
    - ビット演算子（|,&）
- グローバル名前空間のオブジェクトを参照できない
    - window,document,console,Mathなどのようなグローバル名前空間上のオブジェクトにアクセスできない
- {{}}式を利用する上でのガイドライン

    - 式が副作用を伴わないこと
    - 短時間で実行できること
    - シンプルであること
    - 冪等（べきとう）であること（＝同じ操作を何度実行しても同じ結果を返すこと）

3.2.3 安全にプロパティ／メソッドにアクセスする ― 「?.」演算子

- オブジェクトの存在チェックに使用できる
- 普通であればngIfで存在確認する必要があるが、`?.`を使うことでオブジェクトが存在すれば値を出力するといったことが可能
- template: `<div>{{member?.name}}</div>`

3.3 プロパティバインディング

- template: `<img [src]="image" />`

3.3.1 補足：プロパティバインディングの別構文

- bind-xxx
    - `<img bind-src="image">`
- {{}} (Interpolation構文)
    - `<img src="{{image}}" />`

3.3.2 要素に HTML 文字列をバインドする

- HTML文字列はただの文字列として表示される
- `<div [innerHTML]="msg"></div>`
    - として、innerHTMLでバインドすれば、HTMLとして埋め込める
    - ただし、inputやbutton, scriptは除去される
- DomSanitizer#bypassSecurityTrustHtmlメソッドで、信頼済みのHTMLとして出力することもできる

3.3.3 補足：`<iframe>` 要素に外部リソースをバインドする

- iframe要素も、通常、除去されて表示できないが、DomSanitizer#bypassSecurityUrlを使うことで信頼済みのURLとして扱うことができる
- iframeは使わないだろう
    - TODO: 他社連携ボタンなどがiframeの場合があるからそこの実装をAngularでできるか確認

3.4 属性／クラス／スタイルバインディング

3.4.1 HTML 属性に値をバインドする ― 属性バインディング

- 属性とプロパティの違い
    - 属性：デフォルト値

    - プロパティ：現在の値
- 属性と同じプロパティ名を持たない場合もあるので
    - [attr.name]="exp" という属性バインディングを使用する場合もある
- あるべき論としては、プロパティを持つものはプロパティバインディングを使う
- 属性しか無いものは属性バインディングを使う

3.4.2 スタイルクラスを着脱する ― クラスバインディング

- classバインディングを利用せずに、直接class属性に値を入れる例
    - `<div class="line back" [class]="clazz">WINGSプロジェクト</div>`
- classバインディングを利用する場合
    - `<div class="line back" [class.fore]="flg">WINGSプロジェクト</div>`
- TODO: classバインディングで複数指定したい場合は、複数記載もできるが、ngClassディレクティブを利用したほうがスマートらしい

3.4.3 スタイルプロパティを設定する ― スタイルバインディング

- スタイルもクラス同様の構文で記載できる
- スタイルをいじるよるクラスによるスタイルの統一をしたほうがよい

3.5 イベントバインディング

3.5.1 イベントバインディングの基本

- イベントバインディング
    - `<element (event)="exp"></element>`
- イベント
    - click
    - dbclick
    - mousedown
    - mouseup
    - mouseenter
    - mousemove
    - mouseleave
    - focus
    - blur
    - keydown
    - keypress
    - input
    - select
    - reset
    - submit

- 利用できない構文
    - new演算子
    - インクリメント/デクリメント演算子
    - 複合代入演算子（+=,-=など）
    - ビット演算子
    - テンプレート演算子
- 使える構文
    - 代入演算子（=）
    - 連結演算子（;、,）
- グローバル名前空間のオブジェクトを参照できない
- テンプレートステートメントのガイドライン
- イベントバインディングの別構文
    - `<input type="button" on-click="show()" value="現在時刻">`

3.5.2 イベント情報を取得する ー $event

- 標準のjavascriptでは、イベントハンドラーの第一引数にeventを与えることで、イベントオブジェクトを参照できる
- 渡す際に引数に `$event` と明示的に記載してあげる必要があるよという説明
- eventのデフォルト動作をキャンセル
    - `e.preventDefault();`
- イベントのバブリングをキャンセルする
    - `e.stopPropagation();`

3.5.3 テンプレート参照変数による入力値の取得

- e.target.value でイベントが保持している値を取得
- イベントハンドラー（コンポーネント）がテンプレートの構造を把握していなければならないため、
- 関心の分離という観点からも望ましい状態ではない
- テンプレート参照変数の基本
    - `<element #variable ... />`
- 要素オブジェクト、要素値の受け渡しには、まず、($eventではなく)テンプレート参照変数を利用するとおぼえておく
- 補足：テンプレート参照変数をテンプレートの別の場所から参照する
    - テンプレート内の別の場所でも参照できるという説明

    - この際に、変数を定義した要素に `(change)="0"` などの動作はしない定義を入れておく必要がある
    - これがあることで、データバインディングをし直してくれるため
    - changeじゃなくても、focusを外したらとかでもOKだと

3.5.4 キーイベントのフィルタリング ー keyup.enter イベント

- Enterキーによって何らかの処理をしたい場面は多いハズ
- `(keyup.enter)=show($event)` などとすることで、エンターキー押下時に処理を実行できる

3.6 双方向バインディング

3.6.1 双方向バインディングの基本

- 今までのは片方向バインディング
    - プロパティバインディング、属性バインディング
        - コンポーネント→テンプレート（ビュー）
    - イベントバインディング
        - テンプレート（ビュー）→コンポーネント
- ルートモジュールを編集する
    - FormsModuleを追加する
- ルートコンポーネントを編集する
    - `ngModel`ディレクティブを`[()]`で囲む点に注目

3.6.2 双方向バインディングのしくみ

- プロパティバインディングとイベントバインディングを組み合わせることで実現している

3.6.3 補足：テキストボックスのデフォルト値

- MEMO: ディレクティブとは（２種類ある）
    - 構造ディレクティブ
    - 属性ディレクティブ

第4章標準パイプ／ディレクティブ

4.1 パイプ

- パイプとは：テンプレート上に埋め込まれたデータを加工/整形するための仕組み

4.1.1 パイプの基本

- `{{price | currency:&'JPY'}}`
    - JPY付きの文字列に加工される

- Angularの標準パイプ
    - lowercase: 大文字から小文字に変換
    - uppercase: 小文字から大文字に変換
    - titlecase: 単語の先頭文字を大文字に変換
    - slice: 文字列から部分文字列を切り出し
    - date: 日付/時刻を整形
    - number: 数値を桁区切り文字で整形
    - percent: 数値を%形式に整形
    - json: オブジェクトをJSON形式に変換　（JSON.stringifyに相当するパイプ）
    - i18nPlural: 数値によって表示文字列を変化
    - i18nSelect: 文字列に応じて出力を切り替え
    - async:　Observable/Promiseによる非同期処理の結果を取得
- TODO: Observableについて理解していないので確認すること

4.1.2 文字列を大文字／小文字に整形する ― lowercase／uppercase／titlecase

4.1.3 オブジェクトを JSON 形式に変換する ― json

- function型、undefinedである場合、出力対象とならないことに注意

4.1.4 文字列から特定範囲の部分文字列を切り出す ― slice（1）

4.1.5 配列から特定範囲の要素を取り出す ― slice（2）

4.1.6 数値を桁区切り文字で整形して出力する ― number

- ロケールを指定する
- app.module.ts
    - providers: [{provide: LOCALE_ID, useValue: 'de-DE'}]

4.1.7 数値を通貨形式に整形する ― currency

4.1.8 数値をパーセント形式に整形する ― percent

4.1.9 日付／時刻を整形する ― date

- 日付書式にフォーマットできる
- 詳細な書式設定は都度調べよう

4.1.10 数値によって表示文字列を変化させる ― i18nPlural

- TODO: 使いどころ調べる

4.1.11 文字列に応じて出力を切り替える ― i18nSelect

- TODO: 使いどころ調べる

4.2 ディレクティブ

- ディレクティブ
    - HTMLに対して、ngFor、ngStyleのような独自の要素/属性を追加することで、ページに機能を付与している要素/属性のこと
- ディレクティブの分類
    - コンポーネント
        - テンプレートを伴ったディレクティブ
    - 構造ディレクティブ
        - 要素を追加/削除することで、文書ツリーを操作
    - 属性ディレクティブ
        - 属性の形式で、要素/コンポーネントの見た目や動作を変更
- 構造ディレクティブ
    - ngIf: 式の真偽によって表示/非表示を切り替え
    - ngSwitch: 式の値によって表示を切り替え
    - ngFor: 配列をループ処理
    - ngTemplateOutlet: 用意されたテンプレートの内容をインポート
    - ngComponentOutlet: 用意されたコンポーネントをインポート
- 属性ディレクティブ
    - ngStyle: 要素にスタイルプロパティを付与
    - ngClass: 要素にスタイルクラスを着脱
    - ngPlural: 数値に応じて出力を切り替え

4.2.1 式の真偽によって表示／非表示を切り替える ― ngIf

- NOTE: ngIfの頭の「*」
    - ngIfディレクティブの頭の「*」は、配下の要素を、あとで再利用可能なテンプレートとして扱うことを意味します。
    - ここでは、大雑把に、「*」とは、構造ディレクティブであることのしるしとだけ理解しておきましょう
- 補足：ngIfディレクティブの注意点
    - 表示/非表示で、要素が挿入/破棄される
- 頻繁に表示非表示を切り替えるような要素は、スタイルバインディングでdisplayスタイルプロパティを設定する方が望ましいケースもあることを意識する

```
<div [style.display]="show ? 'inline':'none'">
<p>表示するテキスト</p>
</div>
```

4.2.2 条件式を満たさない場合の出力を指定する ― ngIf（else）

```
let x = {
  template: `
    <form>
      <label for="show">表示／非表示：</label>
      <input id="show" name="show" type="checkbox" [(ngModel)]="show" />
    </form>
    <div *ngIf="show; then trueContent; else elseContent">
      この部分は無視される！
    </div>
    <ng-template #trueContent>
      <p>WINGSプロジェクトは、当初、ライター山田祥寛のサポート（検証・査読・校正作業）集団という位置づけで開始されたコミュニティでしたが、2002年12月にメンバを大幅に増強し、本格的な執筆者プロジェクトとして生まれ変わりました。</p>
      <p>その後、「基礎PHP」インプレス）の執筆を皮切りに、「Java PRESS」「Web+DB PRESS」（技術評論社）、「@IT」（IT Media）、「CodeZine」（翔泳社）のような紙／ネットワークを問わず、広い媒体で実績を積んで、現在に至ります。</p>
      <p>2005年5月には「有限会社 WINGSプロジェクト」として法人化を果たし、ますます質の高い情報を読者の方々にお届けしてまいります。</p>
    </ng-template>
    <ng-template #elseContent>
      <h3 style="color:Red">非表示中です。</h3>
    </ng-template>
  `
}
```

- trueの場合表示されるコンテンツと、
- falseの場合表示されるコンテンツを分けて書くことができる

4.2.3 式の値によって表示を切り替える ― ngSwitch

- 多岐分岐するのであればこちらを使う

4.2.4 配列をループ処理する ― ngFor

- ngForディレクティブ
    - `<element *ngFor="let tmp of list">`
- 特殊変数
    - index
    - first
    - last
    - even
    - odd
- `<ng-container>`というダミーコンテナ要素を使うことで余計な要素を使うことなくすむ
- トラッキング式（trackBy関数）の説明
    - TODO: 必要になった場合、処理に組み込む必要がある

        - ただ単に、配列にpushするだけでも、全体を再生成するかどうか挙動の確認をする必要あり

- sliceパイプと連携することで、ページング機能付きのテーブルを簡単に作成できる
    - TODO: ページング実装時に参考にする

4.2.5 要素にスタイルプロパティを付与する ― ngStyle

- TODO: 使い所がいまいち

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    <input type="button" (click)="back=!back" value="背景色" />
    <input type="button" (click)="fore=!fore" value="前景色" />
    <input type="button" (click)="space=!space" value="余白" />
    <div [ngStyle]="styles">
      <p>WINGSプロジェクトは、当初、ライター山田祥寛のサポート（検証・査読・校正作業）集団という位置づけで開始されたコミュニティでしたが、2002年12月にメンバを大幅に増強し、本格的な執筆者プロジェクトとして生まれ変わりました。</p>
      <p>その後、「基礎PHP」インプレス）の執筆を皮切りに、「Java PRESS」「Web+DB PRESS」（技術評論社）、「@IT」（IT Media）、「CodeZine」（翔泳社）のような紙／ネットワークを問わず、広い媒体で実績を積んで、現在に至ります。</p>
      <p>2005年5月には「有限会社 WINGSプロジェクト」として法人化を果たし、ますます質の高い情報を読者の方々にお届けしてまいります。</p>
    </div>`
})
export class AppComponent  {
  back = false;
  fore = false;
  space = false;

  get styles() {
    return {
      'background-color': this.back  ? '#f00' : '',
      'color'           : this.fore  ? '#fff' : '#000',
      'padding.px'      : this.space ? 15 : 5
    };
  }
}
```

4.2.6 スタイルクラスを着脱する ― ngClass

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
```

```
  template: `
    <form>
      <input type="button" (click)="styles.back=!styles.back"
        value="背景色" />
      <input type="button" (click)="styles.fore=!styles.fore"
        value="前景色" />
      <input type="button" (click)="styles.space=!styles.space"
        value="余白" />
    </form>
    <div [ngClass]="styles">
      <p>WINGSプロジェクトは、当初、ライター山田祥寛のサポート（検証・査読・校正作業）集団という位置づけで開始されたコミュニティでしたが、2002年12月にメンバを大幅に増強し、本格的な執筆者プロジェクトとして生まれ変わりました。</p>
      <p>その後、「基礎PHP」インプレス）の執筆を皮切りに、「Java PRESS」「Web+DB PRESS」（技術評論社）、「@IT」（IT Media）、「CodeZine」（翔泳社）のような紙／ネットワークを問わず、広い媒体で実績を積んで、現在に至ります。</p>
      <p>2005年5月には「有限会社 WINGSプロジェクト」として法人化を果たし、ますます質の高い情報を読者の方々にお届けしてまいります。</p>
    </div>`,

  /*
  //複数のスタイルをまとめてオンオフ
  template: `
    <form>
      <input type="button" (click)="flag=!flag" value="有効／無効" >
    </form>
    <div [ngClass]="{'back fore space': flag}">
      <p>WINGSプロジェクトは、当初、ライター山田祥寛のサポート（検証・査読・校正作業）集団という位置づけで開始されたコミュニティでしたが、2002年12月にメンバを大幅に増強し、本格的な執筆者プロジェクトとして生まれ変わりました。</p>
      <p>その後、「基礎PHP」インプレス）の執筆を皮切りに、「Java PRESS」「Web+DB PRESS」（技術評論社）、「@IT」（IT Media）、「CodeZine」（翔泳社）のような紙／ネットワークを問わず、広い媒体で実績を積んで、現在に至ります。</p>
      <p>2005年5月には「有限会社 WINGSプロジェクト」として法人化を果たし、ますます質の高い情報を読者の方々にお届けしてまいります。</p>
    </div>`,
    */
      styles: [`
    .back  { background-color: #f00 }
    .fore  { color: #fff }
    .space { padding: 15px }
  `]
})
export class AppComponent  {
  styles = {
    back  : false,
    fore  : false,
    space : false

    //flag = false;
  };
}
```

- NOTE: スタイルクラスをまとめてオンオフしたい
    - オブジェクトのキーとして複数のスタイル（空白区切り）を渡すことで、複数のスタイルをまとめてオンオフできる

4.2.7 数値に応じて出力を切り替える ー ngPlural

- i18nPluralパイプのディレクティブ版
- メッセージをテンプレートに記載して表示をk梨花エル
- メッセージリストをテンプレートにまとめられるので、ある程度メッセージの分量が多いときはディレクティブ表記のほうがすっきり
- 逆にメッセージをコンポーネント側で管理したい場合には、パイプ表現の方が便利

4.2.8 用意されたテンプレートの内容をインポートする ー ngTemplateOutlet

- ntTemplateOutletディレクティブを利用することで、あらかじめ用意したテンプレートをコンポーネント内の任意の場所に挿入できる

- 利用シーン

    - 繰り返し表示では無いコンポーネント
    - フォーマットが決まっている
    - プルダウンなどで表示するコンポーネントを切り替える
        - などが想定されるのではと考える

- NOTE: $implicitキー

    - コンテキストオブジェクトには、$implicitという名前でデフォルトキーを指定することもできる
        - TODO: あとで再読

4.2.9 コンポーネントを動的にインポートする ー ngComponentOutlet

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

import { EventComponent } from './event.component';
import { BookComponent } from './book.component';
import { WingsComponent } from './wings.component';

@Component({
  selector: 'my-app',
  template: `
    <div>
      広告バナー：<br />
      <ng-container *ngComponentOutlet="banner"></ng-container>
    </div>
  `
})
export class AppComponent implements OnInit, OnDestroy {
  interval: any;
  comps = [ EventComponent, BookComponent, WingsComponent ];
```

```
  current = 0;
  banner: any = EventComponent;

  ngOnInit() {
    this.interval = setInterval(() => {
      this.current = (this.current + 1) % this.comps.length;
      this.banner = this.comps[this.current];
    }, 3000);
  }

  ngOnDestroy() {
    clearInterval(this.interval);
  }
}
```

- 動的に切り替えするバナーなど
- TODO: 利用シーンについて確認

第5章フォーム開発

5.1 フォーム開発の基本

5.1.1 基本的なフォーム

- 設定
    1. FormsModuleモジュールをインポートする
    2. 入力フォームを準備する`<form>`要素
    3. 入力ボックスを配置する`<input>`要素
    4. 検証結果を参照する`errors`オブジェクト
    5. 検証エラー時にサブミットボタンを無効にする
- Column: 開発モードと本番モード
    - 開発モードは低速
    - 本番モードを有効にするには、`main.ts`に対して、以下のコードを追加
        - `enableProdMode();`
- TODO: 実際に登録フォームを作る過程で吸収してく

5.2 フォームの構成要素

フォームの基本を理解できたところで、個別の要素/機能の詳細を確認しながら、より理解を深めていく

5.2.1 フォームの状態を検知する

- Angularでは、フォームの状態を監視する手段が複数用意されている
- 検証項目単位でのエラーの有無をチェックする

    - 入力要素名.errors.検証型
    - 検証型として標準で利用できるのは
        - required
        - minlength
        - maxlength
        - pattern
        - email
        - max
        - min
            - 自作の検証型を利用する方法は9.2.4項で説明

- フォーム/入力項目の単位でエラーの有無をチェック

    - 入力要素名.valid - 入力値が正しいか
    - 入力要素名.invalid - エラーがあるか
    - フォーム全体で検証エラーの有無を確認したいとき
        - フォーム名.valid
        - フォーム名.invalid
    - この属性を使って、１つでも入力エラーがあれば、submitできないように制御

- 入力の有無を判定

    - 特定の入力項目に対して入力が行われたか（＝入力によって値が変更されたか）を判定
        - name.pristine：フォーム/入力要素が変更されていない
        - name.dirty：フォーム/入力要素が更新された
        - name.touched：フォーム/入力要素に一度でもフォーカスがあたった
        - name.untouched：フォーム/入力要素に一度でもフォーカスがあたっていない
    - これらのプロパティを使えば、「なにかしらフォームの内容が変更された場合だけ、リセットボタンを有効にする」といった動作も簡単に実現できる

- サブミット済みかどうかを判定する

    - submittedプロパティを利用することで、フォームがサブミット済みかどうか判定できる
    - 一度フォームを送信したら、再度クリックできないサブミットボタンを作成できる
    - `<input type="submit" value="送信" [disabled]="myForm.invalid || myForm.submitted" />`

- 検証エラー時に入力ボックスのスタイルを変更する

    - Angularではフォームの状態に応じて、以下のようなスタイルクラスを付与する
        - ng-valid
        - ng-invalid
        - ng-pristine
        - ng-dirty
        - ng-touched
        - ng-untouched
        - ng-submitted
    - ng-dirty + ng-invalid でエラーとすべき

5.2.2 ラジオボタンを設置する

- ラジオボタン実装について

5.2.3 チェックボックスを設置する

- チェックボックス実装について

5.2.4 選択ボックスを設置する

- select要素の選択ボックス実装
- optgroupを用いて選択要素のグループ化

5.3 フォーム開発に役立つミニサンプル集

5.3.1 文字数カウント機能付きのテキストエリアを設置する

- テキストエリアの入力文字数を監視
- Twitterクライアントなどでよく見かけるしかけ

5.3.2 テキストボックスの内容を区切り文字で分割する

- 必要があれば確認する

5.3.3 ファイルをアップロードする

- TODO: あとで再確認

5.4 モデル駆動型のフォーム

- ここまでは、より手軽に実装可能なテンプレート駆動形のフォームについて学んだ
- テンプレート駆動形のフォームでは、検証ルールをテンプレートに記述していたが、
- コンポーネントにも記述できる
    - これがモデル駆動形のフォーム
- モデル駆動形（Reactive駆動形）のフォームはテンプレート駆動型よりもコードは冗長になりがちですが、
- より柔軟に、複雑な要件を表現できる

5.4.1 ReactiveFormsModule モジュールの有効化

- app.module.tsでimportするモジュールを `ReactiveFormsModule` に変更する

5.4.2 モデル駆動型フォームの基本

- TODO: モデル駆動型

5.4.3 補足：group メソッドとプロパティ定義は両方必要？

第6章コンポーネント開発

6.1 複数コンポーネントの連携

6.1.1 コンポーネントを入れ子に配置する ― @Input デコレーター

- 書籍情報を一覧/詳細表示するページを作成

1. 書籍情報を表すBookクラスを準備
2. 書籍を一覧表示するAppComponentコンポーネントを準備
3. DetailsComponentコンポーネントで詳細情報を表示する
4. AppModuleモジュールにDetailsComponentコンポーネントを登録する
    1. `app.module.ts`に使用するコンポーネントの情報はすべて登録しておく必要がある

- `@Input`デコレーター
    - プロパティ名と属性名が異なる場合
        - `@Input('data') item:Book;`でdataという属性と紐付けることができる

```
// Bookクラス
export class Book {
  isbn: string;
  title: string;
  price: number;
  publisher: string;
}

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

@Component({
  selector: 'my-app',
  template: `
    <div>
      <span *ngFor="let b of books">
         [<a href="#" (click)="onclick(b)">{{b.title}}</a>]
      </span>
    </div>
    <hr />
    <detail-book [item]="selected"></detail-book>
    <!--<detail-book [data]="selected"></detail-book>-->
  `,
})
export class AppComponent {
  selected: Book;

  books = [
    {
      isbn: '978-4-7741-8411-1',
```

```
      title: '改訂新版JavaScript本格入門',
      price: 2980,
      publisher: '技術評論社',
    },
    {
      isbn: '978-4-7980-4853-6',
      title: 'はじめてのAndroidアプリ開発 第2版',
      price: 3200,
      publisher: '秀和システム',
    },
    {
      isbn: '978-4-7741-8030-4',
      title: '［改訂新版］Javaポケットリファレンス',
      price: 2680,
      publisher: '技術評論社',
    },
    {
      isbn: '978-4-7981-3547-2',
      title: '独習PHP 第3版',
      price: 3200,
      publisher: '翔泳社',
    },
    {
      isbn: '978-4-8222-9893-7',
      title: '基礎からしっかり学ぶC++の教科書',
      price: 2800,
      publisher: '日経BP社',
    }
  ];

  onclick(book: Book) {
    this.selected = book;
  }
}

// details.component.ts
import { Component, Input } from '@angular/core';
import { Book } from './book';

@Component({
  selector: 'detail-book',
  template: `
    <ul *ngIf="item">
      <li>ISBNコード：{{item.isbn}}</li>
      <li>書名：{{item.title}}</li>
      <li>価格：{{item.price | number}}円</li>
      <li>出版社：{{item.publisher}}</li>
    </ul>
  `,
})
export class DetailsComponent {
  // @Inputをつけたプロパティには外から値を渡せる
  @Input() item: Book;
  //@Input('data') item: Book;
```

```
    //@Input('item') item: Book;

}
```

- 補足：セッター／ゲッターによる属性の定義
  - メリット
    - 不正な値が設定された場合や、値が設定されなかった場合にデフォルト値を自動的に設定
    - 値を出し入れする際に、データ型を変更、または演算
    - 値を設定する際に、その妥当性を検証
  - TODO: セッター/ゲッターのメリットや、使い所をいまいちわかっていないので再度確認する

6.1.2 子コンポーネントからイベントを受け取る ― @Output デコレーター

- `@Output`デコレーターを利用することで、子コンポーネントで発生したイベントを親コンポーネントに通知できる

6.1.3 補足：テンプレート参照変数による子コンポーネントの参照

- TODO: テンプレート参照変数（#xxx）経由のほうが良いのか、一般的にはどっちか確認する

6.1.4 モジュールの分離

- MEMO: モジュールを分割する方法はスタートアップブックには紹介されてなかったのでためになった

6.2 コンポーネントのライフサイクル

- コンポーネントのライフサイクル
- ライフサイクルメソッド（ライフサイクルフック）
  - ライフサイクルの変化に応じて呼び出される様々なメソッドのこと
- コンポーネントのライフサイクル一覧
  - コンポーネント生成
  - コンストラクター
  - ngOnChanges：`@input`経由で入力値が（再）設定された時
  - ngOnInit：入力値（`@ionput`）が処理されたあと、コンポーネントの初期化時（最初のngOnChangesメソッドの後で一度だけ）
  - ngDoCheck：状態の変更（9.3.2項 Cloumn）を検出した時
  - ngAfterContentInit：外部コンテンツを初期化した時（最初のngDoCheckメソッドで一度だけ）
  - ngAfterContentChecked：外部コンテンツの変更をチェックした時
  - ngAfterViewInit：現在のコンポーネントと子コンポーネントのビューを生成した時（最初のngAfterContentCheckedメソッドのあとで一度だけ）
  - ngAfterViewChecked：現在のコンポーネントと子コンポーネントのビューが変更された時
  - ngOnDestroyed：コンポーネントが破棄される時

    - コンポーネントの破棄

- 外部コンテンツとは
    - コンポーネント要素の配下（呼び出し側）で指定されたコンテンツのこと
- ビューとは
    - コンポーネントで定義されたテンプレートそのもの

6.2.1 ライフサイクルメソッドの確認

- 実際にログ仕込んで確認

6.2.2 ページの初期化／終了処理を実施する ― ngOnInit／ngOnDestroy

- ページの初期化
    - constructor
    - ngOnInit
    - どちらを使えばよいか
    - 違いは、`ngOnChanges`が呼ばれる前か後か
    - つまり、入力プロパティ（`@Input`デコレータ）が処理される前か後か
    - 一般的には、コンポーネントの初期化処理は、ngOnInitメソッドに集約するとおぼえておいた方がシンプル
- 終了処理
    - ngOnDestroyメソッドは主にコンポーネントで利用したリソースの後始末に利用
        - 具体的には
            - タイマーのクリア
            - Observableの購読解除

6.2.3 入力プロパティの変更を検知する ― ngOnChanges

- `SimpleChanges`クラスを引数に取る
    - `previousValue`
    - `currentValue`
    - `isFirstChange()`のメンバーを持つ

6.2.4 ビューの初期化／変更時の処理を実装する ― ngAfterViewInit／ngAfterViewChecked

- 子コンポーネントを参照する`@ViewChildren`デコレータ
- 単一コンポーネントを取得する`@ViewCHild`デコレータ

6.2.5 コンポーネント配下のコンテンツをテンプレートに反映させる ―

- 親コンポーネントから子コンポーネントへ値を埋め込む例

6.2.6 外部コンテンツの初期化／変更時の処理を実装する ― ngAfterContentInit／ngAfterContentChecked

- TODO: 再読

6.3 コンポーネントのスタイル定義

- `@Component`デコレータのstyles/styleUrlsパラメータで指定
- コンポーネントで定義されたスタイルはコンポーネントローカルであり、他のコンポーネントで定義されたスタイルと衝突することはない

6.3.1 コンポーネントスタイルの定義

- コンポーネント外にスタイルが影響しないことを確認

6.3.2 補足：スタイルカプセル化のしくみ

- カプセル化の仕組みは、`_ngcontent-pum-0`などの属性が該当のコンポーネントに付与されるため

6.3.3 コンポーネントスタイルを定義する方法

- styles/styleUrlsパラメータの他にもスタイルを宣言する方法があるのでその紹介
- テンプレートにインラインスタイルとして定義
- テンプレートからスタイルシートをインポート(`<link>`要素)
    - `index.html`からの相対パスで指定
- `@import`ディレクティブによるインポート
    - 現在のスタイルシートからの相対パスで指定
- あるべき論
    - 基本的に style/styleUrlsパラメータで指定
    - まかないない部分は、`@import`ディレクティブでインポート

6.3.4 コンポーネントスタイルで利用できる特殊セレクター

- コンポーネント本体にスタイルを適用する　:host疑似セレクター
- あくまでコンポーネントの中にテンプレートがある
    - コンポーネント自身にスタイルを適用するために使用するのが、`:host`疑似セレクター
- コンポーネントの外部の状態に応じてスタイルを適用する `:host-context`疑似セレクター
- 子コンポーネントにも適用するスタイルを定義する `/deep/`セレクター
    - `:host/deep/ p` とかで使う
    - エイリアスを使って
        - `:host>>>p`でもよい

6.3.5 カプセル化の挙動を変更する ― encapsulation パラメーター

- `@Component`デコレータの`encapsulation`パラメータを利用することで、コンポーネントスタイルの処理方法を変更できる
- `encapsulation`
    - Native
    - Emulated（デフォルト）
    - None
- 特別理由がない限りデフォルトで実装すること

6.4 アニメーション機能

6.4.1 アニメーション機能を利用するための準備

- `@angular/animations`パッケージをインストール
- `systemjs.config.js`を編集する
- ルートモジュールにアニメーションモジュールを追加する
- メインページにポリフィルを追加する

6.4.2 アニメーションの基本

- アニメーションの定義
    - 状態（state）
    - 遷移（transition）
- アニメーション可能なスタイルプロパティ
    - background-color
    - background-position
    - border-bottom-color
    - border-bottom-width
    - border-left-color
    - border-left-width
    - border-right-color
    - border-right-width
    - border-spacing
    - border-top-color
    - border-top-width
    - bottom
    - clip
    - color
    - font-size
    - font-weight
    - height
    - left
    - letter-spacing
    - line-height

    - margin-bottom
    - margin-left
    - margin-right
    - margin-top
    - max-height
    - max-width
    - min-height
    - min-width
    - opacity
    - outline-color
    - outline-width
    - padding-bottom
    - padding-left
    - padding-right
    - padding-top
    - right
    - text-indent
    - text-shadow
    - top
    - vertical-align
    - visibility
    - width
    - word-spacing
    - z-index

- イージング：変化の具合を表す情報

    - ease: 最初と最後をゆっくり
    - linear: 一定の速度で変化
    - ease-in: ゆっくり開始
    - ease-out: ゆっくり終了
    - ease-in-out:最初と最後をゆっくり（easeとほぼ同義）
    - cubic-bezier(x1,y1,x2,y2):制御点(x1,y1)/(x2,y2)から成るベジェ曲線

6.4.3 状態／遷移のさまざまな設定方法

- transition関数でアニメーション前後のスタイルを宣言する
- 任意の状態を表す「*」
- ビューにアタッチされていない状態を表す「void」
- NOTE: `:enter`/`:leave`キーワード

    - `void => *`, `* => void`はいずれも要素の表示/非表示を表す要素としてよく利用することから、特別なキーワードで代替できる

- アニメーションの途中経過を制御するキーフレーム

- 複数のアニメーションを並列実行する

6.4.4 アニメーションの前後で任意の処理を実行する ― アニメーションコールバック

- アニメーションの開始/終了時に関数を実行することができる
- MEMO: あまり使用用途が無いだろうと割愛

6.5 コンポーネントのその他の話題

6.5.1 テンプレートの外部ファイル化 ― templateUrl パラメーター

- HTMLをtemplateで記載するのではなく、templateUrlでパス指定する説明

6.5.2 テンプレート構文の制約

1. `<script>`要素は削除される
2. 一部の要素は意味をなさない
    1. `<html>`/`<head>`/`<title>`/`<body>`など、ページの外枠/ヘッダー情報を表す要素も意味を持たない
    2. タイトルを変更する場合は、Titleサービスを利用すること
3. `{`は利用できない
    1. `{`を文字列として出力したい場合は、`{{'{'}}`とすること

6.5.3 コンポーネント要素の操作 ― host パラメーター

- hostパラメータ(`@Componentデコレータ`})を利用することで、コンポーネント要素に対して、イベント/属性をバインドできる

```
@Component({
  selector: "my-app",
  template:``,
  host:{
    '(click)':'onclick($event)',
    'role':'banner',
    '[class.disabled]':'true'
  }
})
```

第7章サービス開発

7.1 サービスの基本

- コンポーネントの役割は、ビュー（テンプレート）との値の受け渡し、サービスの呼び出しにとどめて、
- アプリ固有のビジネスロジックについてはサービスに委ねるべき
- TODO: それぞれサービス作って書き直す、APIアクセッサもサービスとして分断すべき？また別？

7.1.1 コンポーネント／サービスの連携

- サービスクラスを準備する
    - `@Injectable`デコレータ
- NOTE: `@Injectable`デコレータは省略可能
    - 特定の条件化で、省略することができる
        - →
        - サービスクラスで、「他のサービスを利用していないこと」
        - 一般的に、サービスクラスでは、Ajax通信のために、Httpなど他のサービスに依存する状況がほとんど
        - 今利用していなくても将来的に利用する可能性を考えると、`@Injectable`を記載しておくことをおすすめする
- モジュールにサービスを登録する
    - app.module.tsなどにサービスを登録しておく必要がある
    - TODO: この作業は、新しいバージョンだと不要かも？確認する
- コンポーネントを修正する

7.1.2 依存性注入

- 呼び出しのコードをその場に書いてしまうことのデメリット
    - 実装の影響がそのままお互いのコードに影響する
        - メンテナンスがしにくい
    - オブジェクト単位の単体テストを実行しにくい
- AngularにはDIコンテナが標準で備えられており、フレームワークの至るところでこれを活用している

7.1.3 依存性注入の基本

- コンストラクターの引数型と、登録済みのサービスを照合して、注入すべきオブジェクト（サービス）を決定する

7.2 依存性注入のしくみ

7.2.1 providers パラメーターの記法

- Providerオブジェクトを生成
    - provide: サービスを注入する際に利用するDIトークン
    - useXxxxx: サービス（インスタンス）の生成方法
    - multi: 同一のDIトークンに対して複数のProviderを追加するか
- トークンは一般的には、サービス本来の型と等しくなる
- useXxxxプロパティはサービスをどのように生成するかを表す情報

7.2.2 サービス生成のアプローチ ― useXxxxx プロパティ

- `useXxxxx`プロパティ
    - useClass: 指定されたクラスを、注入のたびにインスタンス化
    - useValue: 指定されたオブジェクトを常に引き渡す
    - useExisting: 指定されたトークンのエイリアスを生成
        - 別名でエイリアスを使える。用途としては、定義を消すことできないlegacyクラスを新しいものに置換したいなど
    - useFactory: 指定されたファクトリー関数で、注入の際にオブジェクトを生成
        - 関数をプロパティに設定し、returnでオブジェクトを返す
- それぞれのプロパティの使い方、用途を解説している

7.2.3 依存性注入のためのトークンを宣言する ― provide プロパティ

- 非クラス型の値を注入したい場合
    - provideプロパティを使う
- MEMO: この使い方だと型定義してないので、設定値などを読むには使いづらいのでは
    - あくまで、起動時に固定で出力したいログ情報など？
- NOTE: 型情報を付与したInjectionToken
    - ジェネリック構文を利用して注入すべき型を宣言することも可能
    - TODO: あとで実際に使ってみる

7.2.4 単一のトークンに複数のサービスを紐付ける ― multi プロパティ

- trueにすると、provideがかぶっても両方配列で登録できる
- MEMO: かぶるならはじめから配列値を定義すればいいと思うので使用する機会はないのではと思う

7.3 依存性注入の高度な話題

7.3.1 インジェクターの階層構造

- コンポーネントツリーに沿って、インジェクターもツリー構造で存在する
- 自コンポーネントのインジェクターに存在しなければ、親のインジェクターに問い合わせ、
- 定義があれば生成する
- インジェクターの最上位は、モジュール呼び出し元である`bootstrapModule`メソッド
- アプリ全体で単一のサービスを共有したい場合は`bootstrapModule`メソッドで`Provider`を宣言
- TODO: アプリのルールとして盛り込む
- 子コンポーネントで親コンポーネントで定義したprovideを書き換えることができる

7.3.2 任意のサービスを宣言する ― @Optional デコレーター

- `@Optional`デコレータ
    - サービスが任意であることを宣言する

7.3.3 外部コンテンツからアクセスできないサービスを登録する ― viewProviders パラメーター

- TODO: そもそも外部コンテンツを使わないような...
    - 外部コンテンツを定義するメリットを考える

7.3.4 インジェクターを手動で呼び出す ― inject メソッド

- Injectオブジェクトを注入して、getメソッドでオブジェクトを取得する例
- あまり使う機会は無いだろうとおもう

7.4 非同期通信の実行 ― Http／Jsonpサービス

- Angularのサービス
    - Title
    - Meta
    - DOMSanitizer
    - ActivatedRoute
- Angular標準サービスの中でも大粒で、更に実践的なアプリ開発に欠かせないHttp/Jsonpサービスについて解説

7.4.1 SPA とXMLHttpRequest オブジェクト

- 昨今のJavascriptアプリに欠かせないのが、XMLHttpRequest(XHR)オブジェクトによるサーバとの非同期通信
- XHRオブジェクトのAngularによるラッパーがHttpサービス

7.4.2 Http サービスの基本

- HttpModuleモジュールをインポートする
- コンポーネントからHttpサービスを呼び出す
- NOTE:　URLSearchParamsクラス
    - paramsパラメータには、クエリ情報のセットをURLSearchParamsオブジェクトで渡すことも可能
    - TODO: この記載方法のメリットがあまりわからない
- 非同期通信を処理するためのコードを準備する
- NOTE: Reactive Extensions (Rx)
    - Angularでは、内部的にReactive Extensionsというライブラリに依存している
    - 大雑把に言うと、Rxは非同期処理を実装する際に役立つライブラリ

7.4.3 HTTP POST による非同期通信

7.4.4 JSON 形式の Web API にアクセスする

7.4.5 例：非同期通信処理をサービスに分離する

7.4.6 ローカル環境で Web API をエミュレートする ― angular-in-memory-web-api

7.4.7 Observable／Promise 経由で渡された値を取得する ― async パイプ

7.4.8 標準のリクエストオプションを上書きする

7.4.9 Http サービスでのセキュリティ対策

第8章ルーティング

8.1 ルーティングとは？

8.2 ルーティングの基本

8.2.1 基底パスの設定 ― メインページ

8.2.2 ルートの定義

- ルーティングを記載する順番が大事
- `**`ですべてのパスにマッチする記載ができる

8.2.3 ルート対応のリンク／表示領域の準備 ― ルートコンポーネント

- aタグを使用
  - `routerLink`ディレクティブを利用する
  - 先頭文字は`/`でなければならない点に注意
  - `touterLinkActive`ディレクティブには、リンク先が現在と同じである場合に適用されるスタイルクラスを指定
    - `routerLinkActive="current"`など
    - RouterModuleモジュール経由で呼び出されたコンポーネントは、`<router-outlet>`要素で確保された領域に反映

8.2.4 ルーター対応コンポーネントの作成

- 呼び出されるコンポーネント側に特別な設定はいらないという話

8.2.5 別のルートにリダイレクトする

- 存在しないパスに遷移された場合にエラーページを出すのは不親切
- トップページに移動させて上げる方がよい
  - `{path:'**', redirectTo: '/'}`
  - そんなときにリダイレクト設定
  - TODO: リダイレクト設定入れる
  - 処理エラー時のみエラーページに移す

- Column: 関連サイト「Angular TIPS」
    - Angularの機能をTIPS形式で解説する連載
    - データバインディング/ディレクティブ/パイプを始め、目的別に主要な機能が紹介されている
    - https://www.atmarkit.co.jp/ait/series/9383/

8.3 ルーター経由で情報を渡す手法

- ルーター経由では以下のコンポーネントに値を引き渡す方法
    - ルートパラメータ
    - ルートパラメータ（可変長）
    - クエリ情報/フラグメント
    - ルートデータ（dataプロパティ）
    - リゾルバー

8.3.1 パスの一部としてパラメーターを引き渡す ― ルートパラメーター

- ルーティング情報を準備する
- ルートコンポーネントにリンクを追加する
- コンポーネントでパラメータ値を取得する
    - MEMO: パラメータの取得方法がAngularスタートブックと異なるので、どちらがいいか確認する必要がある
- NOTE: Javascript経由でページを遷移する
    - `Router#navigate`メソッドを使えば、コード側からページ遷移できる
    - TODO: ページ遷移の管理方法も、テンプレート側にするべきか、コンポーネント側の処理に記載するべきか、考える必要あり

8.3.2 ハイパーリンクにクエリ情報／フラグメントを引き渡す ― queryParams／fragment 属性

- ルーティング情報を追加する
- ルートコンポーネントにリンクを追加する

```
<li><a routerLink="param" routerLinkActive="current" [queryParams]="
{category:'Angular', keyword:'Routing'}" fragment="hoge">クエリ情報/フラグメント</a>
</li>
```

- コンポーネントでパラメータ値を取得する
- 補足：現在のクエリ情報/フラグメントを引き継ぐ
    - `queryParamHandling="preserve" preserveFragment`
- NOTE: queryPramsHandling属性の設定値
    - `preserve`ではなく、`merge`を指定すると、既存のクエリ情報と統合される

    - TODO: 動作ついて理解していないので実際に書いてみる

8.3.3 ルーティング情報に任意のデータを指定する - dataプロパティ

ルート定義の際にdataパラメータを指定すると、ルーティング情報にコンポーネントに引き渡すデータを含ませることもできる

- MEMO: このような使い方はあまり考えられないのでは無いかと感じた。メリットがあれば後で調べる

8.3.4 可変長のパラメータを引き渡す

- `/search/Angular/Karma/Rx`のようなパス表現を使って、search以下の値を渡す方法
- ルーター定義にchildrenと`**`を指定している

8.3.5 ルーティング情報に任意のデータを指定する（非同期データ）

- Resolverの使い方の説明
- TODO: メリットがいまいちわからないので再確認する

8.4 マルチビュー／入れ子のビュー／ガード

- 1つのページに複数のビュー領域を設置（マルチビュー）
- ビューを入れ子構造に配置
- ルーティングに際して任意の処理を実行（ガード）

8.4.1 テンプレートに複数のビュー領域を設置する

- `<router-outlet>要素`を配置することで、複数のビュー領域を表現できる
- この際に、ここのビュー領域を区別するために`<router-outlet>要素`には任意の名前（name属性）を付与しなければならない

8.4.2 入れ子のビューを設置する

- childrenパラメータを利用すれば、ビューを入れ子にすることもできる

- TODO: 用途があまり無いような気がするためスキップ。後でメリット確認

8.4.3 補足：ルーターのマッチング戦略（prefix とfull）

- `pathMatch: 'full'`を指定すると、パスが完全一致したときにコンポーネントが選択される
- `{path: '', redirectTo:'/error', pathMatch: 'full'}`とすることで、ルートが指定された時のみエラーページに遷移させることができる
    - MEMO: このような使い方ないのでは？とも思う

8.4.4 ルート遷移の可否を判定する ー ガード

- canXxxxを実装して実装する

第9章パイプ／ディレクティブの自作

9.1 パイプの自作

- パイプの自作はごく単純、与えられたスカラー値/配列を加工するメソッドを準備するだけ

9.1.1 パイプの基本

- パイプ本体を定義
  - PipeTransformインターフェースを実装する
  - パイプの実態を実装するのは、transformメソッド
  - パイプの情報を定義するのは、`@Pipe`デコレータ
- TrimPipeクラスをモジュールに登録する
- trimパイプを呼び出す

9.1.2 例：改行文字を `<br>` 要素に変換する nl2br パイプ

- 使い方の例示

9.1.3 パラメーター付きのパイプを定義する

- `truncate`パイプの実装例
  - 長い文字列を区切って、...表示みたいなことができる

9.1.4 例：配列の内容を任意の条件でフィルターする

- `grep`パイプの実装例
  - 引数にコールバック関数を渡している

9.1.5 pure なパイプとimpure なパイプ

- pureなパイプ
  - パイプ処理を高速化するために、以下のタイミングで評価される
    - プリミティブ型（String,Number,Boolean,Symbol）の値が変更された時
    - オブジェクト型（Array,Function,Objectなど）の参照が変更された時
  - 配列に値が追加されただけでは、pipeが再評価してくれない

この問題を解決するのが、「impureなパイプ」

```
@Pipe({
  name: 'grep',
  pure: false
})
```

- impureなパイプは濫用しないこと
- MEMO: filterやorderbyなどの処理は、オブジェクト監視で実行すべきでなく、ボタン押下などのイベントで実行すべきなのではと思った

9.2 属性ディレクティブの自作

- 実際のアプリ開発では、標準のディレクティブだけでは十分に要件を満たせない状況が発生
- 安易にコンポーネント/サービスから文書ツリーを操作するのは避けるべき

9.2.1 属性ディレクティブの基本

シンプルな形で属性ディレクティブを作成

現在の要素に対して、背景色を付与するmyColoredディレクティブの例

- ディレクティブ本体を定義する
    - ElementRefクラスをゆうk上にする
    - ネイティブの要素オブジェクトを取得する
        - ElementRefオブジェクトによるネイティブ要素の操作は時として脆弱性の原因となる場合がある
            - TODO: どのような脆弱性が考えられるのか確認
        - 代替手段として、Renderer2クラスを使用する方法が紹介されている
    - ディレクティブの情報を定義するのは`@Directive`デコレータ
        - selectorパラメータは必須で、ここで属性を指定する
        - 接頭辞は任意で決めていいが、Angularが標準で利用している`ng`は利用するべきではない
- ColoredDirectiveクラスをモジュールに登録する
    - app.moudle.tsに登録
- myColoredディレクティブを呼び出す

9.2.2 パラメーター付きのディレクティブを定義する

- パラメータ付きのディレクティブを定義することもできる

9.2.3 イベント処理を伴うディレクティブ

- 属性ディレクティブに対してイベントハンドラーを紐付ける事ができる
    - マウスオーバーで背景色を変更するなど

9.2.4 例：検証ディレクティブを準備する

- TODO: 検証用ディレクティブは自作するべきなのか考える

9.3 構造ディレクティブの自作

9.3.1 構造ディレクティブとは？

- Angularの「*」構文は、内部的には
    - ターゲット要素を`<ng-template>要素`に展開するためのシンタックスシュガー
        - だったのです。
- 構造ディレクティブとは
    - テンプレート化された要素をもとにコンテンツを生成＆ページに反映させる
        - 機能を持ったディレクティブとも言えるでしょう

9.3.2 構造ディレクティブの実装

- 実際の実装例を紹介（割愛）
    - TODO: アプリを作成する際にもう一度確認する

第10章テスト

- 昨今のアプリ開発では、テストのためのコードを用意し、テストを自動化するのが一般的
    - メリット
        - 人間の目と手を介さなければならない作業を最小限に抑える
        - 繰り返しテストがしやすい
- Angularでも初期のバージョンから、テスト自動化を重視
    - ユニットテスト
    - E2Eテスト（シナリオテスト）

10.1 テストの基本

Angularが対応しているテストの種類

テスト名	概要
ユニットテスト	単体テストとも言う。サービス/コンポーネント/ディレクティブ/パイプなど、ここの構成要素を単体でチェック
E2E(End to End)テスト	インテグレーションテスト、シナリオテストとも言う。複数のコンポーネント/ビューにまたがる、ユーザーの実際の操作に沿った挙動の成否をチェック

10.2 ユニットテスト（基本）

10.2.1 ユニットテストのためのツール

- Javascriptで利用できるテスティングフレームワーク
    - スクリプトで表現されたテストを実施し、テスト対象のコードが期待された値を返すかを検証するためのツール
    - 一覧
        - QUnit
        - Mocha
        - Nodeunit
        - Jasmine(Angular標準)
            - RSpecに似たBDD(Behavior Driven Development)形式の構文を採用している
            - テストコードを英文に近い構文で、アプリの振る舞いとして表現できるため、とても読みやすいという特徴がある
- テストランナー
    - テストを実行するためのツール
    - Angularのために開発されたツールであるKarmaを採用
    - Karmaの特徴
        - コマンドラインから簡単に起動でき、実行スピードに優れる

        - 複数のブラウザー環境で同時にテストが可能
        - ファイルの変更を検知して、自動的にテストを実行
        - Jasmine、Mocha、QUnit、Nodeunitなど主要なテスティングフレームワークに対応
        - CI（継続的インテグレーション）ツールであるJenkins/Travisなどとも連携が可能
    - Angularでは、Karma+Jasmineの組み合わせでユニットテストを実施するのが一般的であり、関連する資料も豊富

10.2.2 ユニットテストの準備

- Karma.conf.jsについて説明
    - テスト対象のブラウザー
    - テスト実行前に実施すべき処理
    - ログレベルなどの設定が記載されている

10.2.3 テストの基本

- describeで囲ってitでテストを記載する
- beforeEachで前処理
- afterEachで後処理を記載する
- toBeなどのmatcherと呼ばれるメソッドもJasmineは持っていて、
- それぞれ値の検証ができる
- notというメソッドを間に挟めば、否定することもでき、英文に近い書き方ができる
- テストを起動するとChromeが立ち上がり検証結果が表示される

10.3 ユニットテスト（Angularアプリ）

10.3.1 パイプのテスト

10.3.2 サービスのテスト

10.3.3 コンポーネントのテスト（基本）

- コンポーネントの値は自動的にテンプレートに反映されるわけではない点に注意

- NOTE: 分離単体テスト

    - Angularでは、単体テストのためにさまざまな支援クラスが提供されている
    - TestBed、ComponentFixtureなどのクラスがこれに当たる
    - これらのクラスを総称して、`Angular testing utilities API(TestAPI)`と呼びます
    - コンポーネントはAngularに強く依存していることから、テストに際してはTestAPIを使うのが一般的
    - 一方で、TestAPIを利用しない（＝標準的なJasmineのAPIでのみ表された）テストのことを、`Isolated Unit Tests(分離単体テスト)`とも言います。
        - パイプやサービスのテストがこれに当たる
        - コードをシンプルに纏めやすい
        - TestAPIの機能を要しない状況では、積極的に分離単体テストを併用することをオススメする

- 補足：変更検知の自動化

    - ComponentFixtureAutoDetectサービスにtrueを設定するだけで変更検知を自動化できる
    - ただし、コンポーネントへの直接の変更は検知できない
    - タイマー、イベント、Promiseの解決など非同期の動作は検知する

10.3.4 外部テンプレート付きコンポーネントのテスト

10.3.5 入出力を伴うコンポーネントのテスト

10.3.6 サービスに依存したコンポーネントのテスト

10.3.7 サービスに依存したコンポーネントのテスト（スパイ）

10.3.8 サービスに依存したコンポーネントのテスト（非同期版）

10.3.9 「浅い」コンポーネントのテスト ― NO_ERRORS_SCHEMA

- 認識できない要素/属性を無視する（＝エラーを出さない）ことが可能になります。

10.3.10 ディレクティブのテスト ― ホストコンポーネント

- テストのためのホストコンポーネントを用意してテストすることをオススメする
- MEMO: ディレクティブのテストは他のコンポーネントにまとめるのも手なのかなと感じた
    - 独立させる意味でも分けたほうがいいか...

10.3.11 Http／Jsonp を利用したサービスのテスト

- テストの際は、HttpやJsonpのモックを利用することが一般的
- Jsonpサービスのモックを取得する
- サービス/バックエンドを取得する
- ダミーのレスポンスを設定する
- HatenaServiceサービスを実行＆テストする
- TODO: 詳細について再確認必要

10.4 E2E（End to End）テスト

10.4.1 E2E テストの準備

- ユニットテストで利用していたKarmaの代わりに、**Protractor**というテストランナー（テスティングフレームワーク）を利用するのが一般的
- Protractorは内部的にSelenium WebDriverというライブラリを利用しており、ブラウザーに文字を入力したり、ボタンをクリックしたり、ページを遷移したりといった操作の仕組みを標準で備えている
- Selenium Serverをインストールする

    - Protractorは単体では直接ブラウザーを操作できないため、内部的には中継サーバーであるSelenium Serverを介して対象にアクセス
    - `npm run preprotractor`

- Protractorの設定ファイルを準備する
    - 詳細はGithubドキュメントを参照

10.4.2 E2E テストの基本

- テストコードを準備する
    - ブラウザーを操作する`browser`オブジェクト
    - 目的の要素を取得するelement/element.allメソッド
    - ロケーターと組み合わせる
        - ロケーターとは、要素をどのように検索するかを表す情報で、byオブジェクトのメソッドとして生成

- NOTE: ショートカット関数 $,$$
    - よく利用するという理由で、Protractorではelement/element.allメソッドについて以下のようなエイリアスを持っている
        - element(by.css('...')) → $('...')
        - element.all(by.css('...')) → $$('...')

- 要素の情報を取得する

- テストを実行する

第11章関連ライブラリ／ツール

11.1 Angularで利用できる関連ライブラリ

- 役立つサイト/ページ
    - Angular Scripthttp://angularscript.com/
    - Angular Modules, Plugins, Directiveshttp://ngmodules.org/
    - EXPLORE ANGULAR RESOURCEShttps://angular.io/resources/
        - 利用にあたっては、対応バージョンを確認するようにしてください

11.1.1 Bootstrap を Angular アプリで活用する ― ngx-bootstrap

- ngx-bootstrap
    - bootstrapのラッパーで、bootstrapの機能をコンポーネントやディレクティブの形式で使用できるようにしてくれている

11.1.2 標準の検証ルールを拡張する ― ng2-validation

- Angular本体にも定型的な入力値検証の機能は用意されている

- 本格的な開発ではより充実した検証ルールが欲しくなる

- 自作もできるが、まずは`ng2-validation`で提供されていないかを確認しておくと良い
- モデル駆動形フォームでの実装例を記載

11.1.3 国際化対応ページを実装する ― ng-xi18n

- TODO: 必要になったら再読

11.2 開発に役立つソフトウェア／ツール

11.2.1 アプリの雛形を自動生成する ― Angular CLI

- Angularのスケルトンを作成してくるCLIツール
- 開発時は必ず利用する
- コマンドの紹介
- `ng generate`コマンドで利用できるサブコマンド
    - ng g module my
    - ng g component my
    - ng g directive my
    - ng g pipe my
    - ng g service my
    - ng g guard my
    - ng g class my
    - ng g interface my
    - ng g enum my
- フラットオプションも場合により使用する
- TODO: フォルダ構成を作成する方法
    - フラットをつけて作成する場面
    - 一般的なフォルダ構成について確認

11.2.2 Angular アプリの構造を視覚化する ― Augury

11.2.3 アプリを事前コンパイルする ― AoT コンパイル

- AoT(Ahead-of-Time)コンパイル
    - アプリサイズを抑え、起動/実行の速度が高まる
    - テンプレートエラーを早期に検出できる
    - コンパイル済みのJavascriptを直接実行するので、インジェクション攻撃の可能性を軽減できる

AppendixTypeScript簡易リファレンス

A.1 TypeScript Playground

A.2 変数

A.3 定数

A.4 型アサーション

A.5 関数

A.6 アロー関数

A.7 関数のオーバーロード

A.8 共用型

- パイプで型をor指定する

A.9 クラス

A.10 get／set アクセサー

- メリット
    - 値の出し入れに伴う任意の処理を実装できる

A.11 継承

A.12 インターフェイス

A.13 ジェネリック