

TechFeedのつくりかた

Angular2 / Webpack / Ionic2 / Cordova 実践入門

2016/09/03

株式会社オープンウェブ・テクノロジー
TechFeed統括
白石 俊平

今日の結論

今後Webはコンポーネント
指向になる！（断言）

本日のアジェンダ

- 自己紹介
- TechFeedとは
- 要素技術の概要／採用の経緯
 - Cordova
 - Angular2
 - Ionic2
 - Webpack
- それぞれの技術のメリット／デメリット

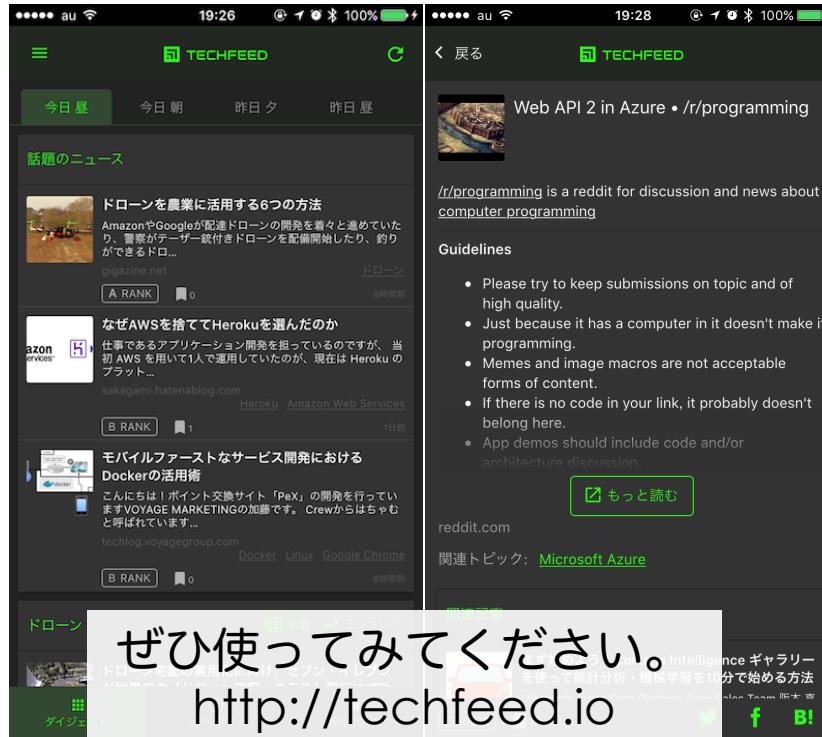
自己紹介

- html5j ファウンダー
- HTML5 Experts.jp 編集長
- 株式会社オープンウェブ・テクノロジー CEO
 - TechFeedというサービスをやっています

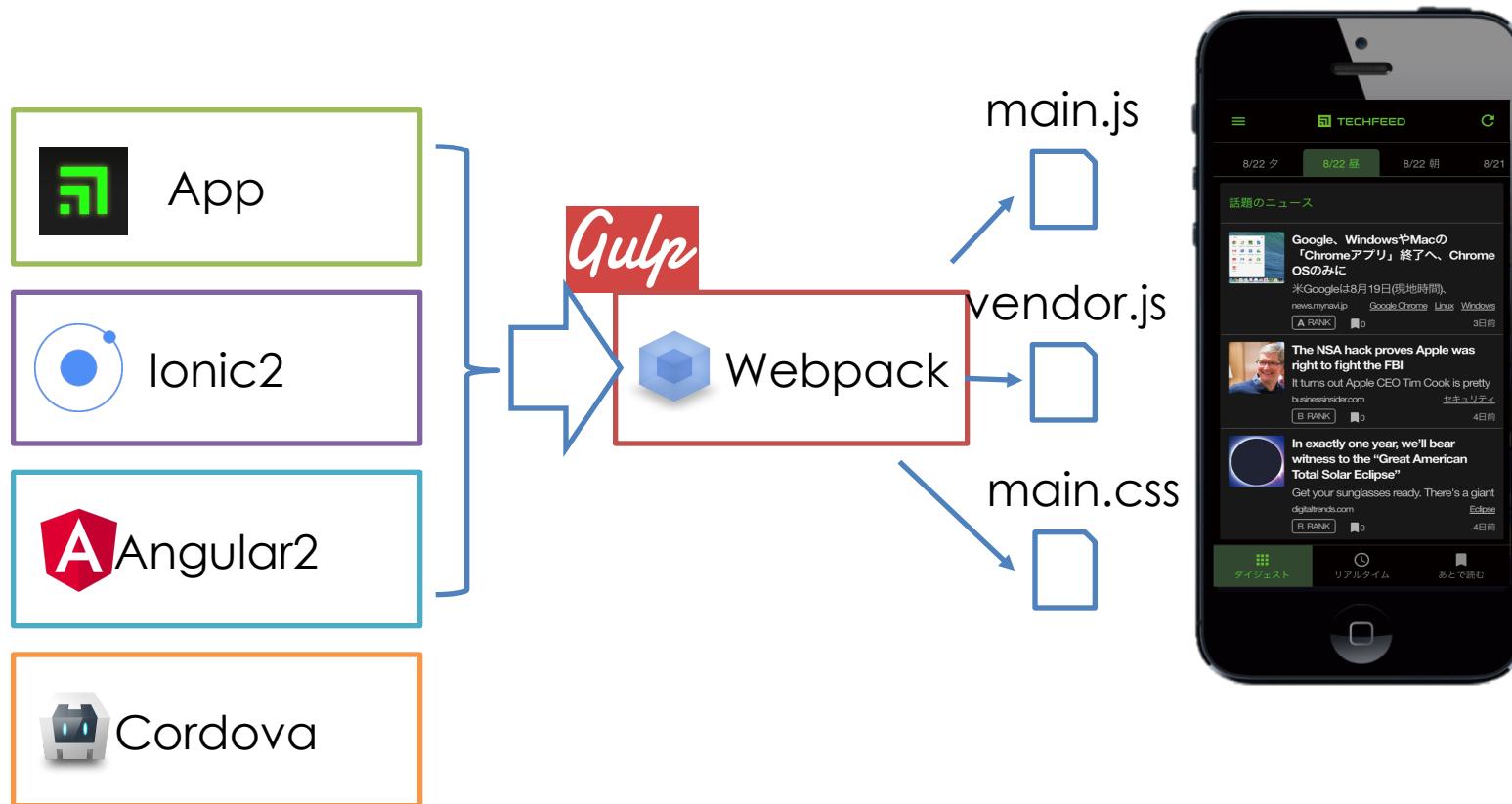


TechFeed

- エンジニア向けキュレーションサービス
 - 世界中から、エンジニア向けの情報収集 & ランキング
- エンジニアがウォッチしなくてはならない情報は多様 & ディープ
 - 人力でやるべきではない→サービスつくった



TechFeed Mobileのアーキテクチャ



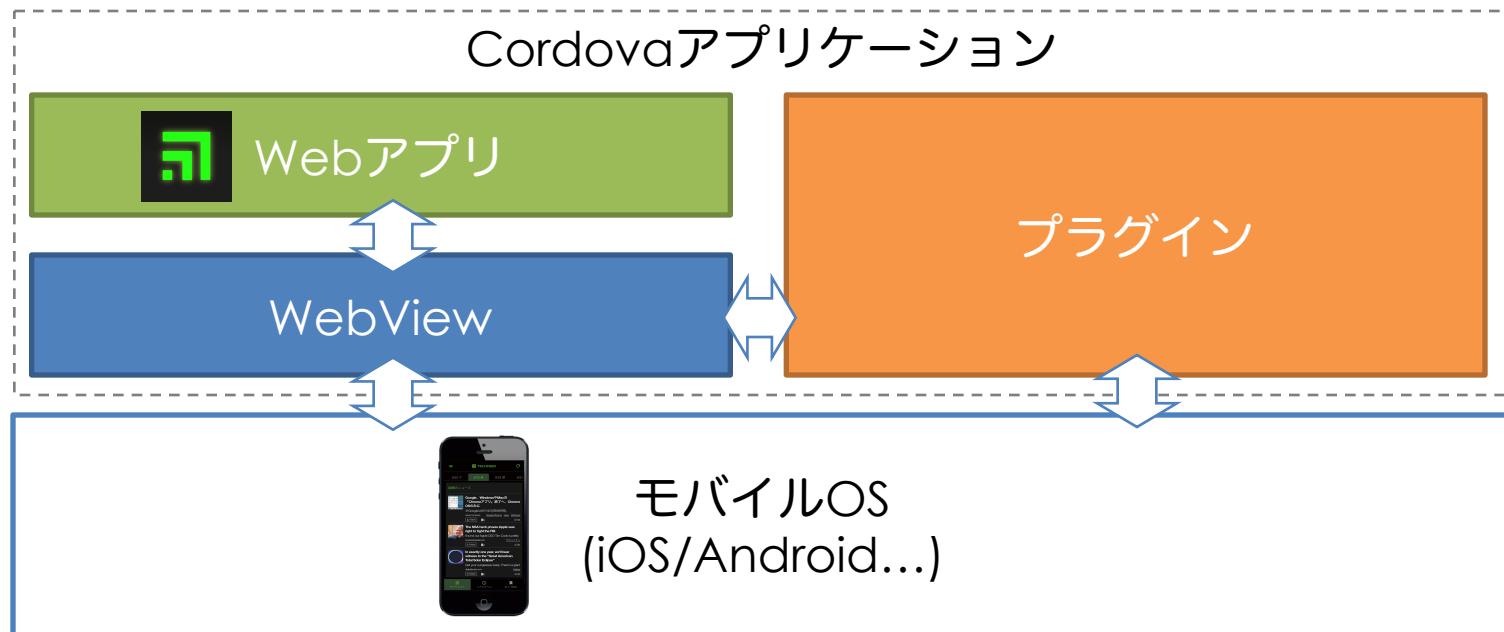
要素技術の概要／採用の経緯



Cordova

Cordovaとは？

- モバイルアプリをHTML/CSS/JavaScriptで開発できるようしてくれるプラットフォーム
- 各種モバイルOSが備えるWebViewの上で動作する
- プラグインを使用することで、ネイティブの機能も呼び出せる



なぜCordovaだったか？

- Webフロントエンドに関する知識を活用したかった
 - エンジニアが全員Web技術者
- モバイルアプリのコードベースを統一したかった
- できればWebとモバイルアプリのコードも統一したかった
 - PWAppsも見据えつつ

TechFeedでのCordova利用

- TechFeedでは、以下のプラグインを利用（一部）
 - プッシュ通知
 - ディープリンク（ユニバーサルリンク）
 - アプリ内ブラウザ（InAppBrowser）
 - SafariViewController/ChromeCustomTabs
 - SafariやChromeをアプリと統合できる
 - Crosswalk
 - ...

プッシュ通知

- phonegap-plugin-pushというプラグインを使用
- プッシュ通知に関する処理を、OSに関わらず同じコードで書ける

```
// プッシュAPIの初期化
const push = Push.init({
  android: {icon: 'icon'},
  ios: {alert: true}
});
push.on('registration', data => {
  // デバイストークン取得時の処理
  this.registerDevice(data.registrationId)
});
push.on('notification', data => {
  // プッシュ通知がタップされた際の処理
  ...
});
```

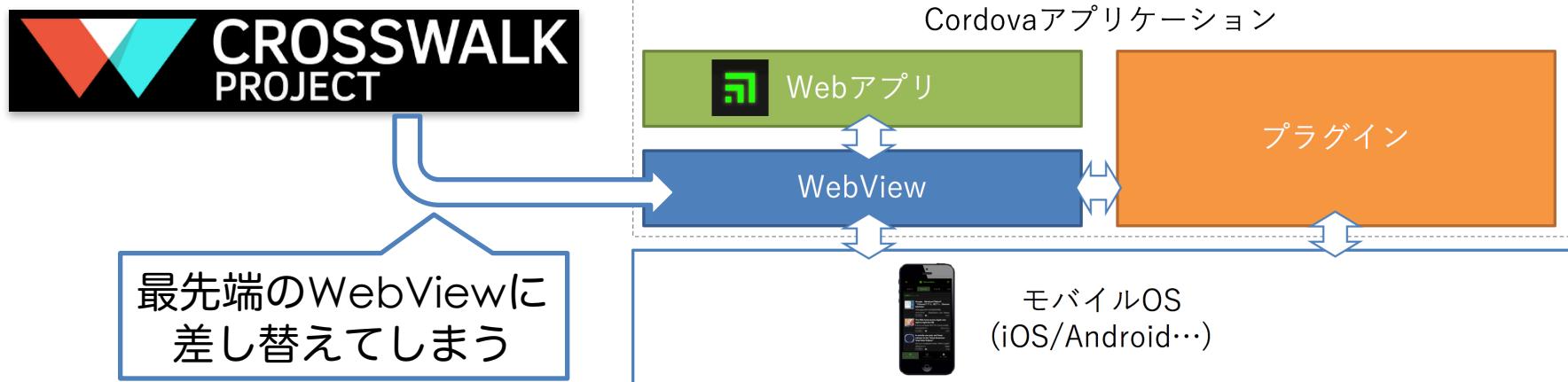
ディープリンク (ユニバーサルリンク)

- cordova-universal-links-pluginを使用
- 特定のURLに反応して、モバイルアプリが立ち上がるようになる。
 - TechFeedの場合、メールのリンクをタップしたらアプリが起動する

```
// プラグインから発生するイベントを監視
universalLinks.subscribe(null, (event) => {
  const {url} = event;
  // URLから、アプリの特定のビューを開く処理
  ...
});
```

Crosswalk

- <https://crosswalk-project.org/>
- (主にAndroid)デバイスやOSの差異に関わらず、最先端のWebViewを提供してくれるプラグイン
 - Chromiumの独自ビルドを内包することで実現
- Androidデバイスのフラグメンテーションとおさらばできる！
 - これはかなり嬉しい





Angular2とは？

- Angular.jsの後継フレームワーク
- 1系と互換性はないが、移行は可能
- 1系との最大の違いはコンポーネント指向であること

Angular2の特徴

- TypeScriptを推奨
- コンポーネント指向
- 包括的なアプリケーションアーキテクチャ
 - モジュール
 - Dependency Injection

コンポーネントのコード例

```
Angular2のコンポーネントを読み込み
ES7のデコレータを使用
@Component({
  selector: 'my-alert-button',
  template: `
    <button (click)="click()">Click me!</button>
  `,
  styles: [
    button { width: 100%; }
  ]
})
export class AlertButtonComponent {
  click(): void {
    window.alert('Hello');
  }
}
```

コンポーネントのテンプレート

コンポーネントのタグ名を指定

コンポーネントのイベントを捕捉

コンポーネントのスタイル

Angular2のDependency Injection

- DI=依存性注入
 - 依存関係にあるクラスのインスタンス生成と注入をフレームワークに任せる
- テスト容易性、メンテナンス性に優れたコードを素早く書けるようになる

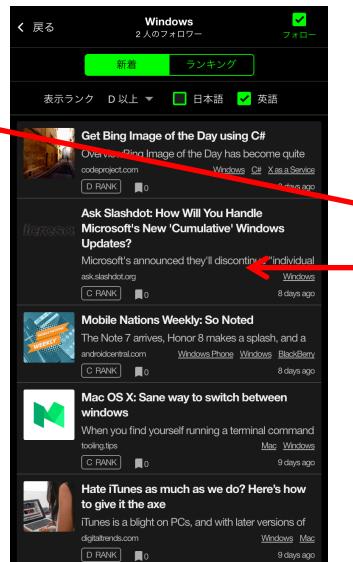
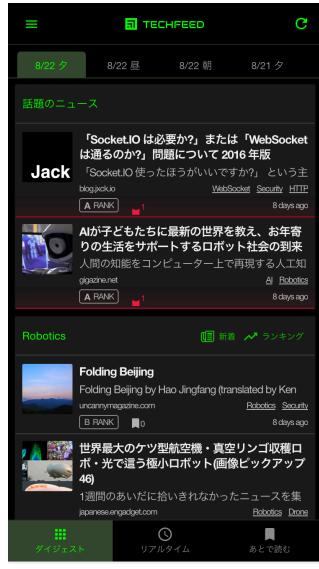
```
import {Injectable} from
'@angular/core';

@Injectable()
export class MyService {
  loadData(): string[] {
    return ['a', 'b', 'c'];
  }
}
```

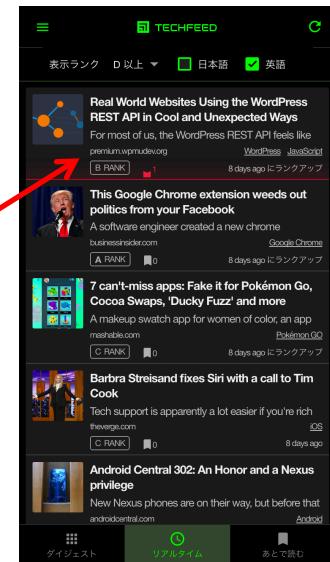
```
@Component({
  providers: [MyService]
})
export class AppComponent {
  constructor(
    myService: MyService) {
  }
}
```

なぜAngular2を選んだか？

- コンポーネント指向を採用したかった
 - 最初に作ったWeb版で、共通化したい部分が多数あることに気付いたため
- TypeScript
 - 型なしで大規模アプリを書くことに少し疲れを感じていた。。
 - 今の時代どうせビルドが必要なので、コンパイルしてほしい



一覧の項目はいたる
ところで再利用する



なぜReactではなくAngular2を選んだか？

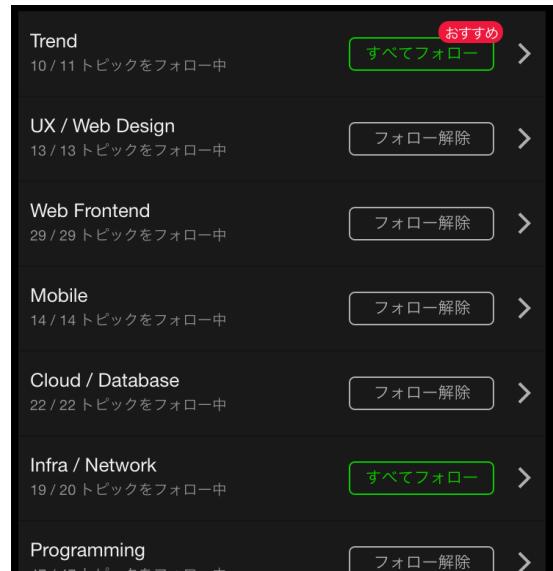
- 当時、Reactにはモバイルアプリ用UIフレームワークのデファクトが存在しなかった
 - 今ならOnsen UIが使える
- 「フレームワーク」が欲しかった。
 - 割と強めの規律でチームのコーディングスタイルを統一したかった
 - アーキテクチャを模索している時間が惜しい
- React+Reduxが難しかった。
 - かなりのパラダイムシフト+学習コストが必要に感じた。
 - 一方Angular2は、Angular1を知っていると、アーキテクチャの理解は割とすんなり。
 - DI / ディレクティブ / パイプ…



Ionic2とは

- Angular2上に構築されたUIフレームワーク
- iOS/Android/Windows Phone Mobileに対応し、スタイルの自動的な切り替えにも対応
- 最初から数多くのUIコンポーネントを備える
 - UIの開発効率はかなり高い

Ionic2のコンポーネント例



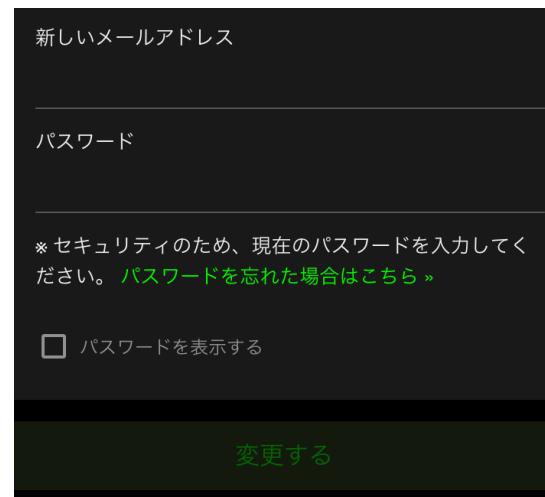
リスト



タブバー



サイドメニュー



フォーム

Ionic2のコード例

- Ionic2独自のコンポーネントを使用してテンプレートを記述していく



```
<form [formGroup]="form" (ngSubmit)="changeUsername()">
  <ion-label stacked>ユーザーID</ion-label>
  <ion-input type="username" required
  autocapitalize="none"></ion-input>
  <button type="submit" full [disabled]={!isValid}>
    変更する</button>
</form>
```

なぜIonic2を選んだか？

- Angular2に対応したUIフレームワークは、当時唯一の選択肢だった。
 - 今ならOnsen UIもある
- 「モバイルアプリ」を作りたかった。
 - モバイルアプリのUXを自動的に実現してくれる
 - 以前のWeb版はBootstrapなどを使用していた



Webpackとは

- モジュールバンドラーの一種
 - JavaScriptモジュールをまとめ上げ、1つ（以上）のファイルを出力するツール
- すさまじく多機能



SystemJS

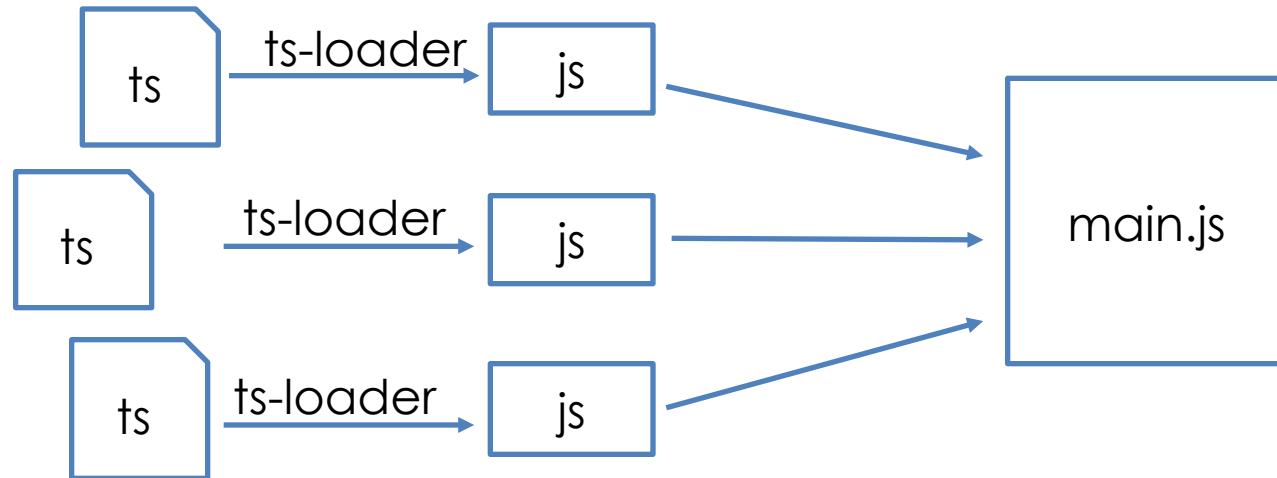
モジュールバンドラー

- モジュールバンドラーで、import/exportをJSコードに変換

```
import {Component} from '@angular2/core'  
↓ (TypeScriptコンパイラ)  
var Component = require('@angular2/core').Component;  
↓ (モジュールバンドラーがrequireをブラウザ上で使えるよう変換)  
var component_1 = __webpack_require__(1);
```

Webpackの設定

- Webpackの設定は、 webpack.config.js に記述する。
- 基本的な考え方は、
 - ローダーがファイルを読み込み
 - JSコード内の require から依存関係を解釈し、
 - 最終的なファイル出力



Webpackは超多機能

- ローダーも、プラグインも、死ぬほどある
 - [ローダーの一覧](#)
 - [プラグインの一覧](#)

List of loaders

basic

- json: Loads file as JSON
- hjson: Loads HanSON file (JSON + comments)
- raw: Loads raw content of a file (as a string)
- val: Executes code as module and returns its value
- to-string: Executes code as a module and returns its string representation
- imports: Imports stuff to the module
- exports: Exports stuff from the module
- expose: Exposes exports from a module
- script: Executes a JavaScript file
- apply: Executes a exported JavaScript function
- callback: Parses your JS, calls it as a function
- if-loader: This is a preprocessor
- ifdef-loader: Preprocessor for .js files
- source-map: Extract sourceMap from the module
- checksum: Computes the checksum of the module
- null: Emits an empty module
- cov: Emits a module with a coverage report
- dsv: Loads csv/tsv files
- glsl: Loads glsl files and supports GLSL
- render-placement: Adds React.renderToStaticMarkup or renderToNode
- xml: Loads XML as JSON
- svg-react: Loads SVG files as JSX
- svg-url: Loads SVG file as utf-8 encoded string
- svg-as-symbol: Wraps content of a SVG element in a symbol element
- base64: Loads file content as base64
- ng-annotate: A loader to annotate AngularJS code
- node: Loads .node files that are pre-compiled
- required: Require a whole directory
- icons: Generates iconfont from .svg files
- markup-in-line: Inline SVGs to HTML
- block-loader: Generic loader for non-module files
- bundler-configuration: Bundler configuration loader
- console: Prints the resolved require statement
- solc: Compiles Solidity code (.sol files)
- web3: Deploys Ethereum VM bytecode
- includes: Load any text file and include its content
- combine: Combine results from multiple loaders
- regexp-replace: Replace RegExp with a string

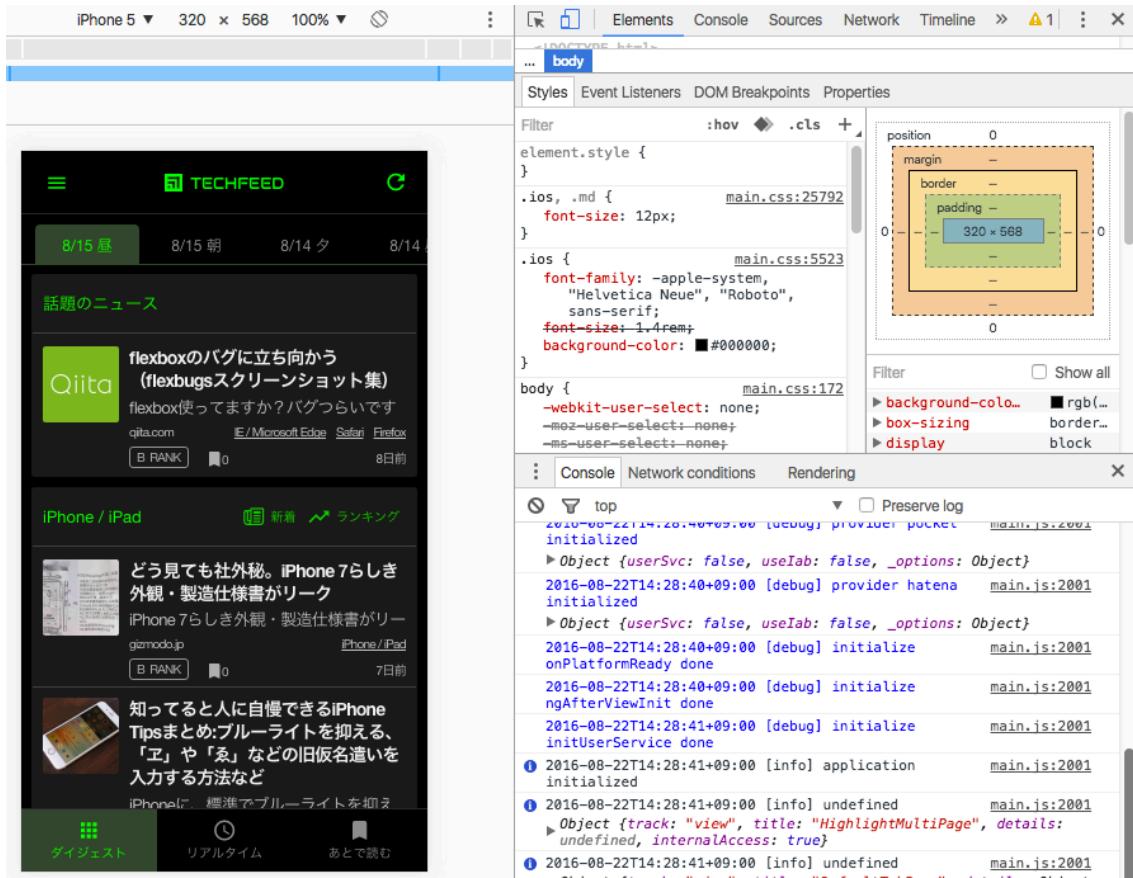
packaging

- file: Emits the file into the output
- url: The url loader works like the file loader
- extract: Prepares HTML and CSS for extraction
- worker: The worker loader creates a separate worker thread
- shared-worker: Like the worker loader but for shared workers
- service-worker: Like the worker loader but for service workers
- bundle: Wraps request in a require block
- promise: Wraps request in a promise
- async-module: Same as bundle, but for asynchronous modules
- react-proxy: Code Splitting for React components
- react-hot: Allows to live-edit React components
- image: Compresses your images
- img: Load and compress images via image-webpack-loader
- base64-image: Load image as base64
- responsive: Create multiple resized versions of images
- svgo: Compresses SVG images via svgo
- svg-sprite: Like style-loader but for SVGs
- svg-fill: Changes colors in SVGs
- line-art: Inlines SVG files, converts them to data URIs
- baggage: Automatically require any dependencies of a module
- polymer: Process HTML & CSS with polymer
- uglify: Uglify contents of a module
- html-minify: Minifies HTML using html-minifier
- vue: Load single-file Vue.js components
- toJSON: Serialize module exports as JSON
- zip-it: Convert files and directories to ZIP archives

なぜWebpackを選んだか

- 実績豊富
- 機能豊富
- AngularのチュートリアルではSystem.js
が使われているので、悩ましくはあった
 - JSPMと組み合わせなくてはならなそうなの
が面倒に感じた

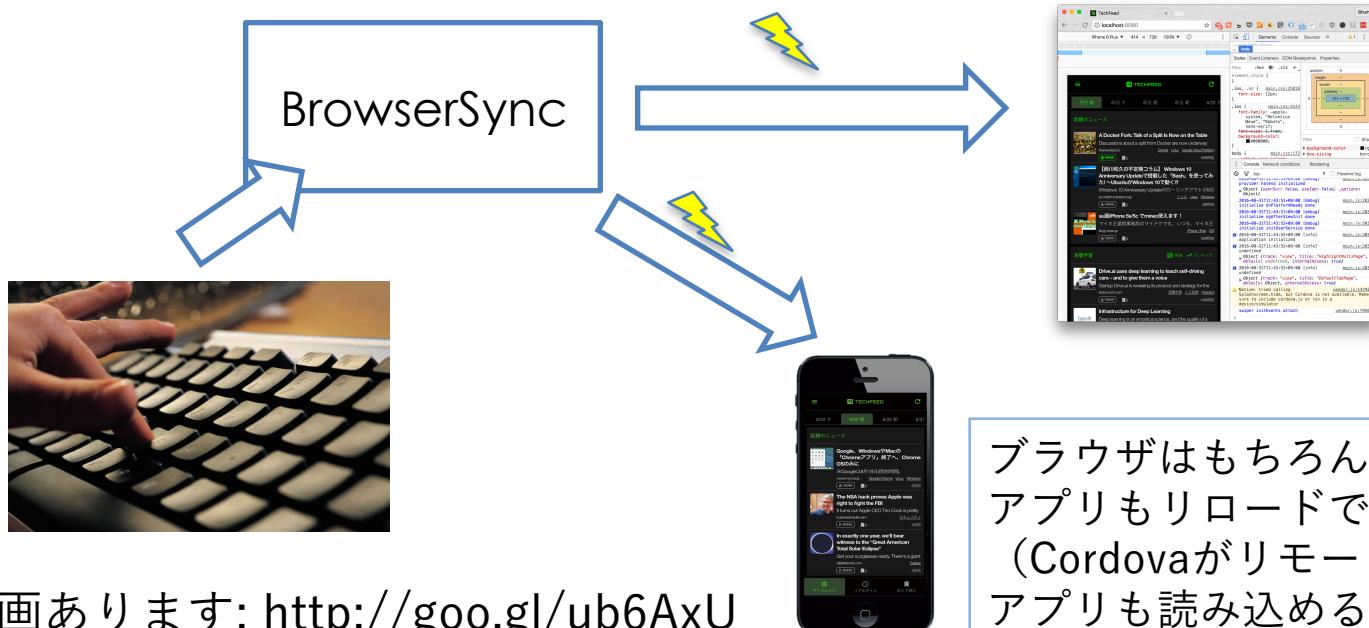
公開！TechFeedの開発環境



- Angular2
- Ionic
- gulp
- Webpack
- (Cordova)

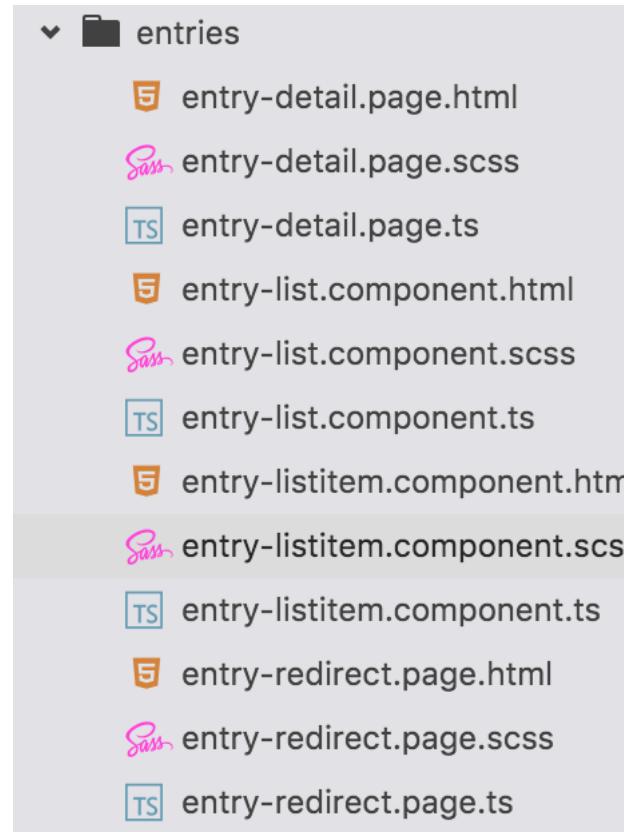
開発フロー上のポイント

- 基本的にはブラウザで開発
- 実機上の開発版アプリも、コードを修正すると自動的にリブートする



ソースコード上のポイント

- コンポーネントはすべて、以下のファイルの組み合わせからなる
 - HTML
 - SCSS
 - TypeScript
- Webpackがこれらをまとめ上げてくれる
 - HTML/CSSのコードもJSに一体化
 - SCSS→CSS変換も自動的に



The screenshot shows a file explorer window with a tree view. At the top is a header bar with a search icon and a refresh icon. Below the header is a toolbar with icons for file operations like New File, Copy, Paste, Cut, Delete, Find, and Select All. The main area displays a folder named 'entries'. Inside the 'entries' folder, there are several files listed:

- entry-detail.page.html
- entry-detail.page.scss
- entry-detail.page.ts
- entry-list.component.html
- entry-list.component.scss
- entry-list.component.ts
- entry-listitem.component.html
- entry-listitem.component.scss
- entry-listitem.component.ts
- entry-redirect.page.html
- entry-redirect.page.scss
- entry-redirect.page.ts

The file 'entry-listitem.component.scss' is highlighted with a light gray background.

それぞれの技術を振り返る

Cordovaを振り返る

Cordovaを振り返る

- メリット
 - コードベースが統一できる
 - 頑張ればモバイルとWebアプリも統一可能
 - 「Progressive Web Apps + モバイルアプリ」が可能
 - Webに関するスキルがフル活用できる
 - Webブラウザでモバイルアプリを開発できる
 - Crosswalkのおかげで、Androidのバージョン差異が問題にならない

Cordovaを振り返る

- デメリット
 - やはり、ネイティブに比べると動作が遅い
 - 一人のエンジニアが全OS面倒見なくてはならなくなる
 - ネイティブの機能を使う場合、いろいろしんどい
 - Crosswalkでかい（20M超）
 - iOSのWebViewがしょぼい（WKWebViewであっても）
 - 遅い、バグる
 - IE6を彷彿とさせる…

Angular2を振り返る

Angular2を振り返る

- メリット
 - Angular1の反省がよく活かされている
 - 全体的に整理されていて美しい
 - パフォーマンス
 - アーキテクチャ上迷うことがあまりない
 - Dependency Injectionは便利
 - CSSのカプセル化も最初から利用可能
 - 静的型付け言語をベースとした、縛り強めなフレームワークなので、コードがぐちゃぐちゃになりにくくい

Angular2を振り返る

- デメリット
 - 学習コストが高い
 - TypeScript
 - RxJS
 - 当然Angular2そのものも
 - 静的型付け言語が嫌いな人にはつらい
 - バージョンアップがつらい（今だけだけど）

TypeScriptを振り返る

TypeScriptのメリット

- 最新の言語仕様を思う存分使える
- 静的型付け言語である
 - コンパイラによるコードの静的解析
 - コードがより文書的
 - ツールによるコーディング補助
 - 入力補完
 - リファクタリング

TypeScriptのデメリット

- ツールチェインがまだ未成熟
 - WebStormといえどもまだまだ
- JavaScriptコードとの統合はやはりつらい
 - 基本的には型定義ファイル (d.ts) 必要
 - JSとTSの間には浅くない溝がある
 - なので、プロジェクトの一部をTSで…というのは厳しい
- 静的型付け言語である
 - 記述が冗長になりがち
 - JSほど気楽に書き下させない
 - ビルドが速くない（これはBabelでも同じだが…）

Ionic2を振り返る

Ionic2のメリット

- 開発効率上がる
- ネイティブアプリに近いUXを得られる
- ionic-nativeは地味に嬉しい
 - CordovaのプラグインをTypeScriptでラップしてくれている
 - これがなかったら、全部anyでやるか、d.tsを書かねばならないところ

Ionic2のデメリット

- 遅い
 - 正式リリース前だからだと信じたい
- サーバサイドレンダリングは諦めざるを得ない
- カスタマイズが面倒
 - これはUIフレームワーク全般に言える話
- バージョンアップがしんどい（正式リリース前だからだと信じたい）

Webpackを振り返る

Webpackのメリット

- import/exportができるようになる
 - <script>地獄からの開放
- 機能・プラグインが豊富
 - 欲しいと思ったものはたいがい、ある。
- コミュニティ・実績が豊富
 - ググればだいたい出てくる

Webpackのデメリット

- 機能が豊富すぎて、何でもやらせくなってしまう
 - 実際にはGulpとかに任せたほうが楽なことも多い
- 謎の難しさがあり、学習コストが高い
 - 出来上がったwebpack.config.jsは単純でも、そこにたどり着くまでがしんどい。。

TechFeedでのビルド環境

- 結局、適材適所の組み合わせ
 - npm scripts…ビルドコマンドの定義
 - Gulp…ビルドフロー全体をコントロール
 - Webpack…モジュールバンドル（だけ）
- 「○○だけでOK、もう△△は古い」という言説は疑ってかかれ



コンポーネント指向開発 を振り返る

コンポーネント指向開発 のメリット

- 適度にコード分割されるので、コードの見通しが良い
- 外部コンポーネントの活用で開発効率アップ
- コンポーネントの再利用性も高まる

コンポーネント指向開発 のデメリット

- 開発を始めるまでの準備が結構かかる
 - フレームワークなどのセットアップ
 - コンポーネント設計
- かなり未来感があるので、勉強が大変

準備せよ。
向学心を燃やせ。

まとめ

今後Webはコンポーネント
指向になる！（断言）

ご清聴ありがとうございました。

“エンジニアなら、使っとけ。”



<http://techfeed.io>



<http://facebook.com/techfeedapp>



<http://twitter.com/techfeedapp>