CLIENT SIDE OF ■■■■■FRESH.TV

2016.04.05 - Node学園 20時限目 @ahomu

@ahomu です

- 株式会社サイバーエージェント
- Web クライアントサイドエンジニア
- http://aho.mu
- Frontend Weekly キュレーター とか
- HTML5 Experts 幽霊部員 とか

Frontend Weeklyは、
毎週フロントエンド開発に関連する
記事、チュートリアル、動画/スライドなどを、
キュレーションしてメールでお届けします。

frontendweekly.tokyo

アジェンダ

- FRESH! について
- SSR について
- SPA について
- Component について

FRESH! について

生放送配信プラットフォーム

時節的なもの

- 2015 年の春ごろに開発スタート
- 紆余曲折 (イロイロ) あって、2016 年 1 月にリリース
- 今回紹介する大枠の構成も 1 年くらい前に作られたもの
- ライブラリのアップデートとかは可能な範囲で実施中

Web クライアント開発者の分担 (4〜5人)

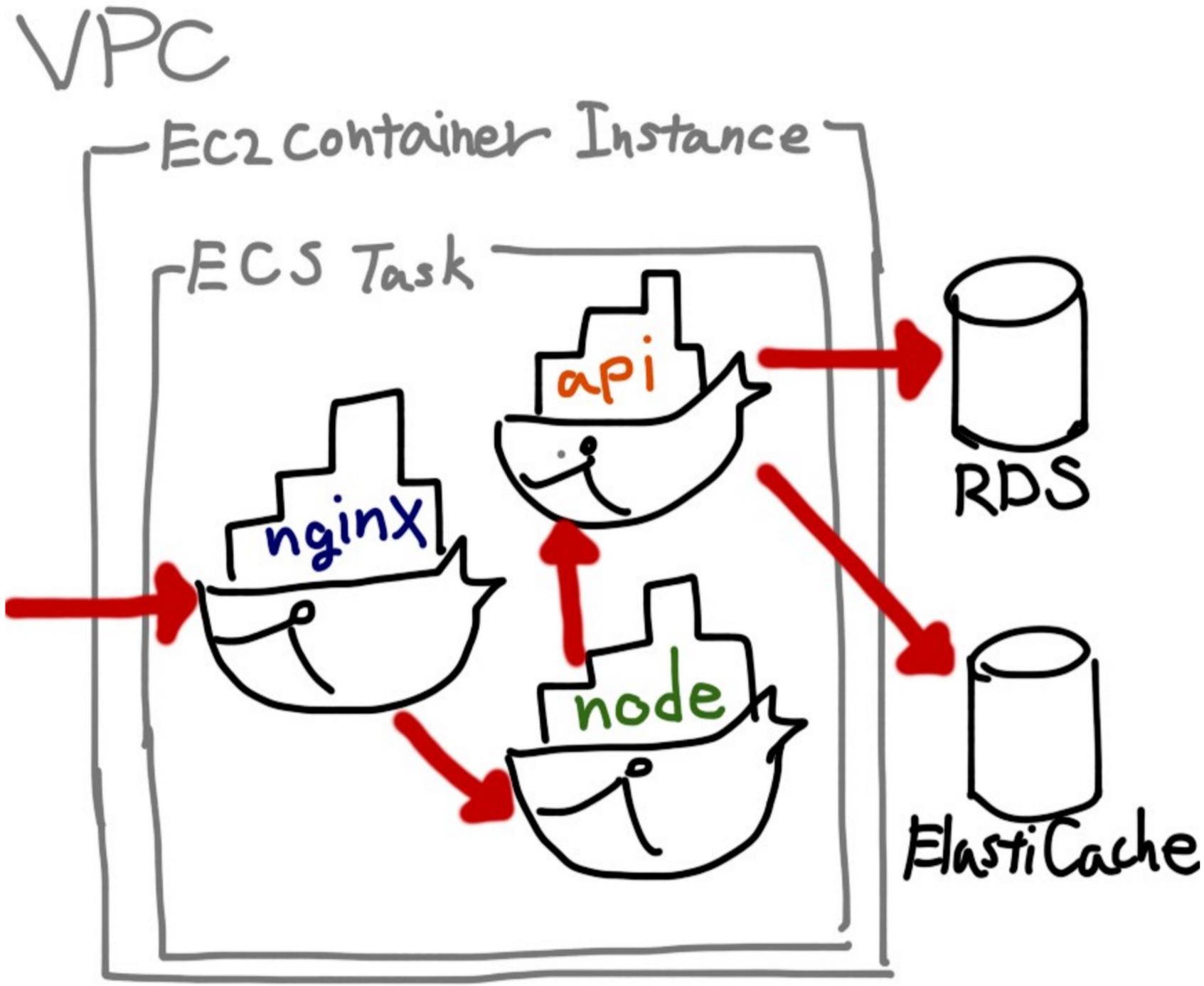
- Desktop Web 担当
- Mobile Web 担当
- **Web Backend 担当** (ﾜｲ)
- **SPA 基盤 担当** (ﾜｲ)
- HLS Player 担当 (ﾜｲ)
- 配信管理 Web UI 担当

Web クライアントの構成要素

- Node.js v4 + Express v4
- **React v0.14 + Fluxible v1**
- Video.js v4 + contrib-hls v0.17改
- Socket.io v1.4
- Babel v5, TypeScript v1.6
- cssnext, csscomb, stylelint
- mocha, power-assert, karma
- browserify, gulp

開発環境の構成要素

- Amazon Web Services
- Microservices with Docker
- GitHub Enterprise
- Circle CI
- ChatOps with Slack
- New Relic, Speed Curve, Mackerel
- InVision, Pixate


VPC
EC2 container Instance
ECS Task
api
nginX
node
RDS
ElastiCache

SSR について

Server Side Rendering

初期のモチベーション

1. Angular v1 が肌に合わなかったので道具を変えたい
2. コンポーネントライブラリなら何でもいいんだけど
3. deku とか Riot.js よりはエコシステム成熟してそう
4. よし、**React でサーバーサイドレンダリング**しよう
5. ES Proposals や Observable は見送り (副産物は作った)

SSR の真価は SEO ではない

- 2重テンプレートを避けつつ、**コンテンツの表示を高速化したい**
- JS のみでレンダリングすることのペナルティは少なくない
    - メインコンテンツのクリティカルレンダリングパスが長くなる
    - レンダリング前に JS のロードと実行、API からのデータ取得が必要
        - 初期データをHTMLに埋めても、JS の初期化が遅延しない保証はない
- SSR なら CSSOM と DOM が構築できれば JS を待たず表示できる
- ページ全体の SSR に時間かかるなら、一部分を非同期化すればOK
- ソーシャル流入が多くなりがち、初期表示が遅いことは機会損失

Isomorphic なパーツ選び

- **Flux** : yahoo/fluxible

  必要なパーツの一通りが詰まったライブラリ群 (細かい話はココ)

- **Rendering** : facebook/react

  みんなだいすき Virtual DOM 生成器 + レンダラ

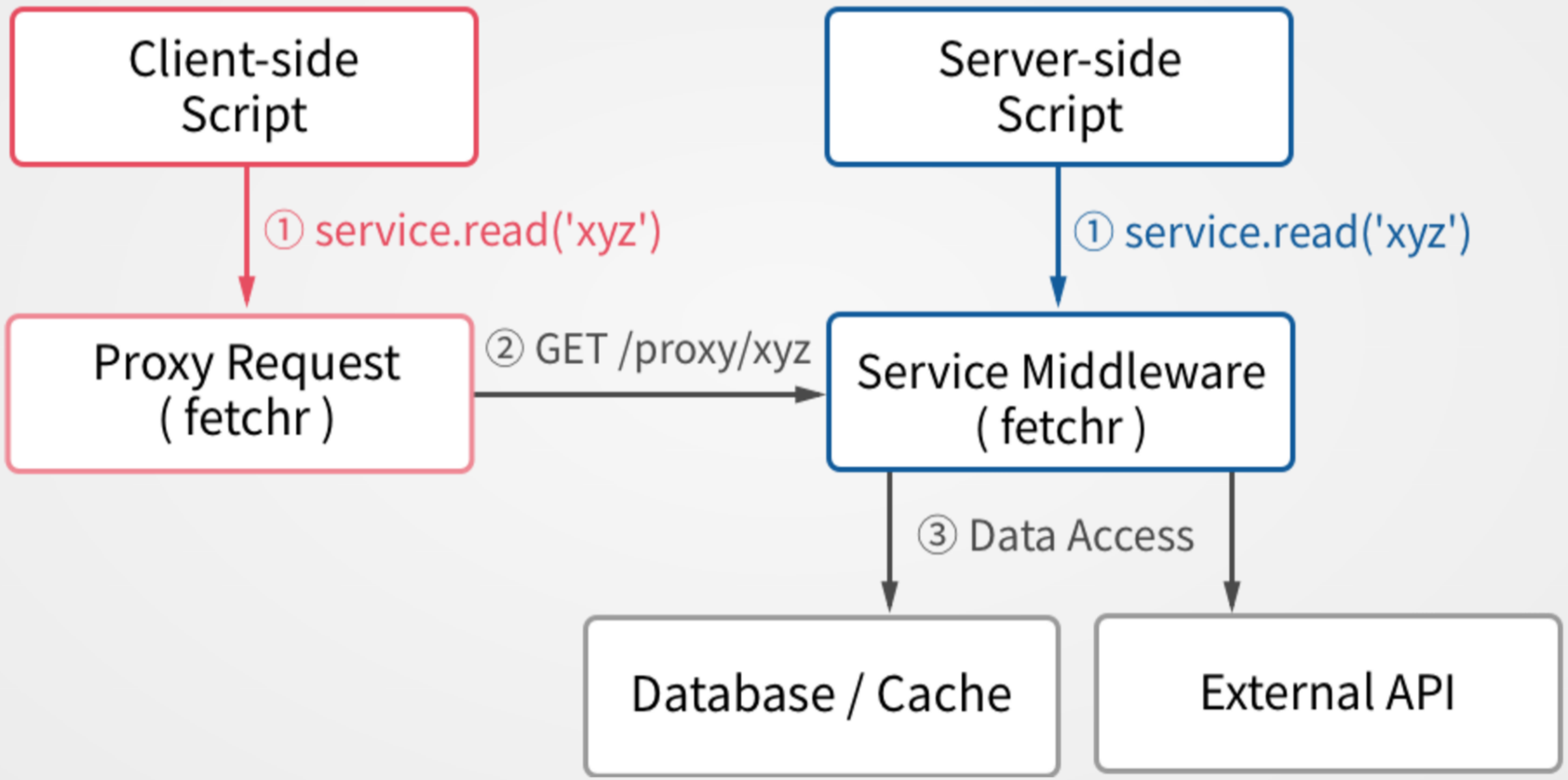
- **Routing** : yahoo/fluxible-router

  pushState や History API 、スクロール制御もあって優秀

- **Data Loading** : yahoo/fluxible-plugin-fetchr + mzabriskie/axios

  fetchr (次で説明) + HTTP クライアントは Promise ベースの axios

fetchr はサーバ用の Service Middleware と
クライアント用の Proxy Request を提供



- `actions/` : Fluxible の Action
- `components/` : React の Component
- `stores/` : Fluxible の Store
- `routes/` : URL ごとの Route 処理
- `server/` : サーバー固有の処理
  - `controllers/` : OAuth など Routr 外のルーティング
  - `handlers/` : Express にくわせるミドルウェア
  - `services/` : Fetchr の Service
- `app.js` : サバクラ共通 Fluxible アプリケーション本体
- `client.js` : browserify 実行時のブートストラップ
- `server.js` : node 実行時のブートストラップ

Flux 周辺の雰囲気

- Component はスマート UI 気味に作り、必要に応じて改修
- 他のコンポーネントと共有しない UI の状態は state 使う
- Context 注入を使って、props バケツリレーの一部を避けてる
- Dispatcher 隠蔽型でふつうに Action と Store がある
- EventEmitter 的だけど、べつに困ることはない

```
<Document> <!-- サーバで HTML 全体を返すときだけ動作 -->
  <Application> <!-- サバクラ共通ルートコンテナ -->

    <!-- ★ UI をもつコンポーネント -->
    <Header />
    <PageHandler /> <!-- 画面遷移ごとのページ差し替え -->
    <Footer />

    <!-- ★ UI をもたないコンポーネント -->
    <Metadata /> <!-- 動的な <meta> <link> の書き換え -->
    <ScrollHandler /> <!-- 画面遷移ごとのスクロール制御 -->

  </Application>
</Document>
```

```
let app = new Fluxible({ component : Application });
let ctx = app.createContext();

// Route 処理 ( API からのデータ取得など ) を実行
await ctx.executeAction(navigateAction, {url: req.url});

// <Application> の実行 & HTML 文字列化
let root = renderToString(createElementWithContext(ctx));

// クライアントに提供するデータの serialize
let expose = serialize(app.dehydrate(ctx));

// <Document> の実行 & HTML 文字列のレスポンス
res.send(renderToStaticMarkup(createElement(Document, {
  root     : root,
  $$expose : `window.$$expose=${expose};`,
  context  : ctx.getComponentContext()
})));
```

```
let app = new Fluxible({ component : Application });

// サーバから提供されたデータを deserialize
let context = await app.rehydrate(window.$$expose)

// <Application> を指定された要素にマウントして実行
let mountNode = document.getElementById('app');
let root = createElementWithContext(context);
ReactDOM.render(root, mountNode);
```

Isomorphic の細かい話は
ブログに書いた

- コンフィグのクラサバ横断共有
- ユーザーエージェントの評価
- ブラウザオブジェクトに触れるコード
- `<head>` への操作または副作用の管理
- などなど

SPA について

Single Page Application

SPA の メリデメ

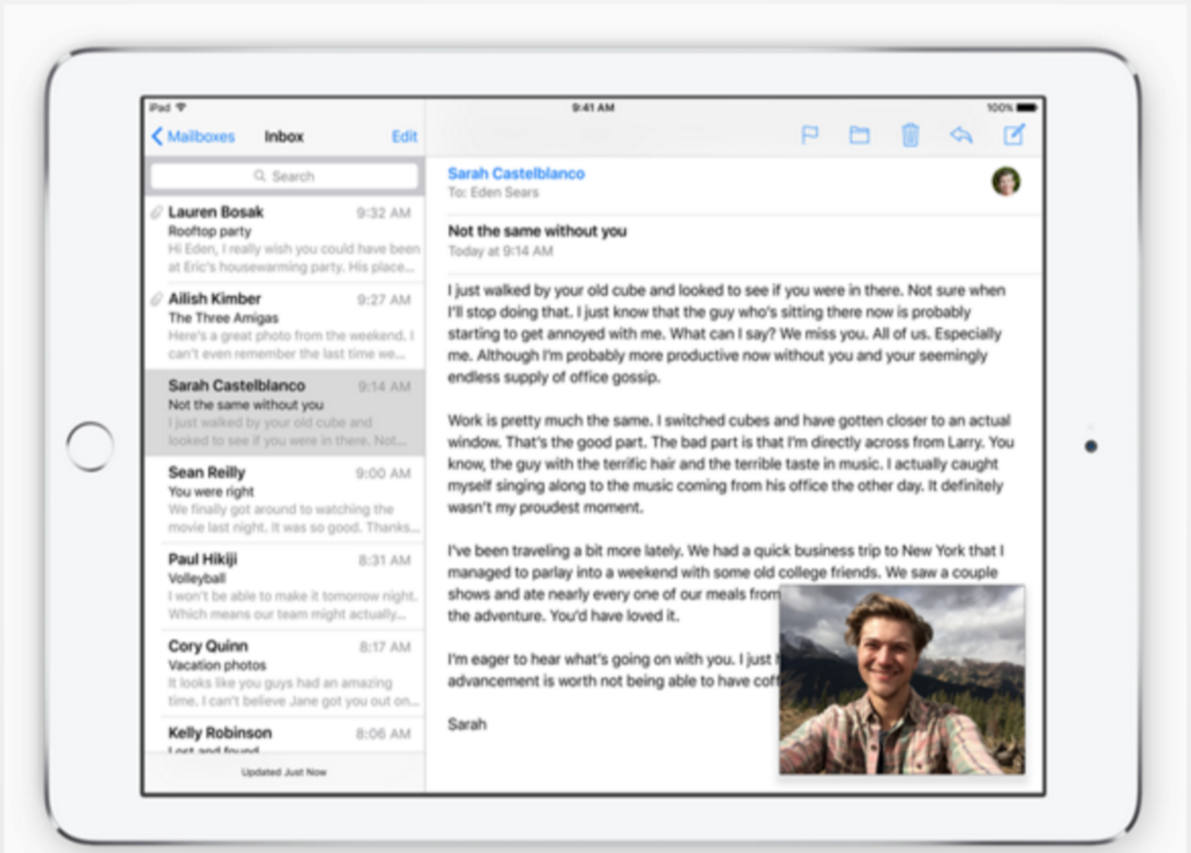
- メリット
  - なんとなく画面遷移が速くなる
  - URL をまたいだ表現が可能になる
- デメリット
  - **ブラウザナビゲーションを破壊してしまう**
  - まあまあ面倒なので半端にやると痛い目に

その Web サイト は
本当に SPA にするべき?

- Web サイト的な情報設計/デザインで SPA ってどうなの
  - ユーザー操作の大半が URL 遷移なら SSR だけでもいいのでは
  - 逆に Web アプリ的なら、SPA は必要だが SSR は不要なのでは
- ブラウザのナビゲーションを破壊するなら責任をもって
  - スクロール制御や前画面のキャッシュは車輪の再発明に等しい
  - クロスブラウザで完璧な個別実装はそれなりに面倒

ではなぜ SPA しているのか

PinP (Picture in Picture) が要件にあったから



ルーティングの後に画面更新

画面遷移の開始と同時にページが真っ白とかは NG
処理中は現状を残して、完了してから表示更新

```
// 架空のコード、こうあってほしい雰囲気
router.addEventListener('changeRoute', () => {
  await justRouting(location.path);
  updatePageContent();
});
```

ルーティングの後にスクロール

fluxible-router は画面遷移の開始とスクロール制御が同時
ルーティングには非同期処理が入るので先走り甚だしい

```
// 架空のコード、こうあってほしい雰囲気
router.addEventListener('changeRoute', () => {
  await justRouting(location.path);
  updatePageContent();
  adjustScroll();
});
```

ブラウザバックは即座に表示

ブラウザバックなら情報は既知であるべき
Routing 処理が発生しても fetch はスキップ

```
// ルーティング中に popState とキャッシュを確認して分岐
Promise
  .all(isPopStateAndCached(context) ? [/* noop */] : [
    executeAction(fetchFooFromApi),
    executeAction(fetchBarFromApi)
  ])
  .then(() => done())
```

社内の後期型 SSR + SPA 勢

- 7gogo (ホリ■モンと始めたアレ) : Pure Flux で Isomorphic
- 某巨大サービス (システム刷新中) : Redux で Isomorphic

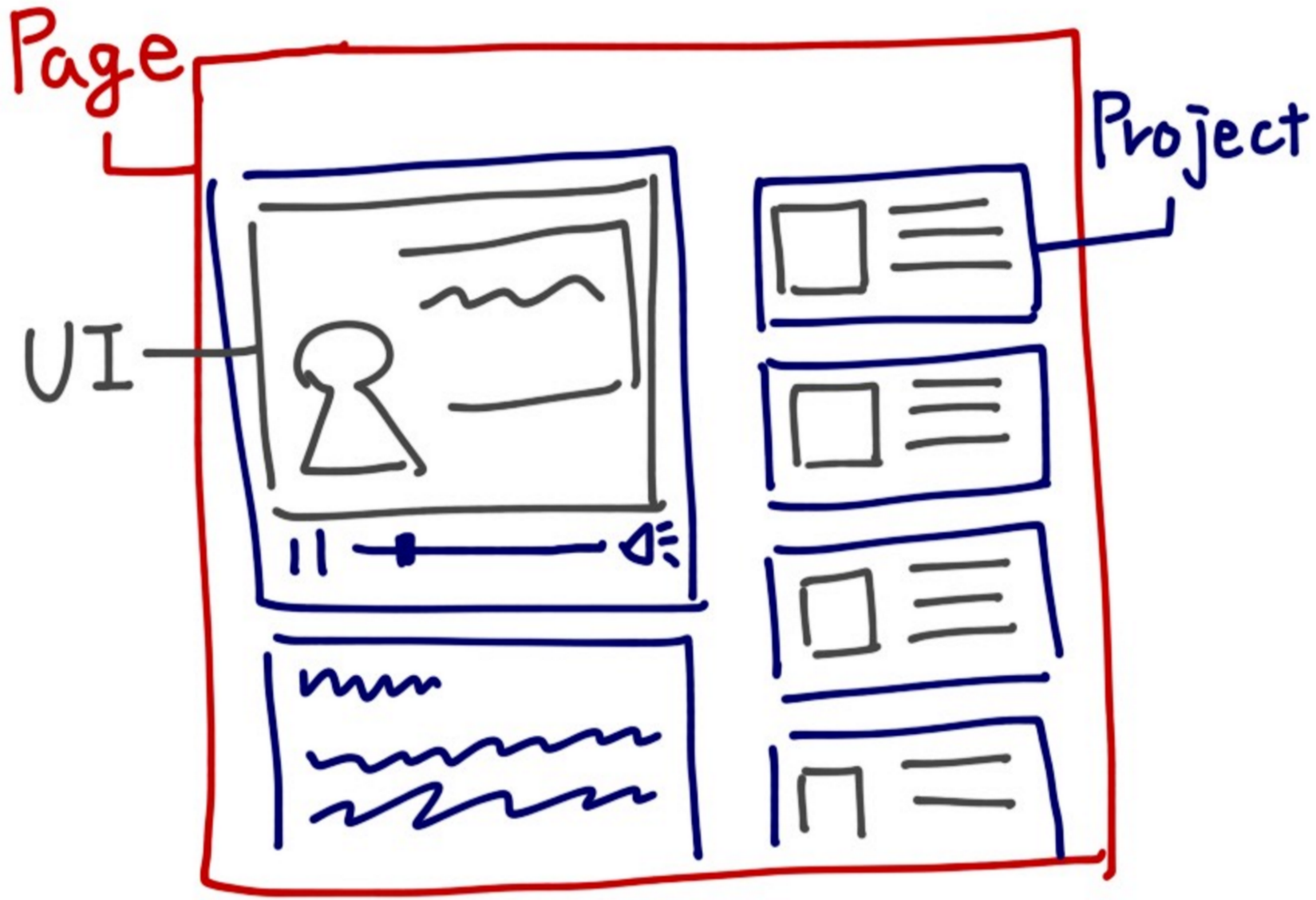
たぶんそのうち中の人がどこかでアウトプットする

Component について

HTML/CSS/JavaScript のパッケージ単位

FRESH! における分類

- **Page** : Store から状態を受け取る
- **Project** : プロジェクトのロジックを含む
- **UI** : 単機能のプリミティブな要素
- **Utils** : 文字校正や副作用管理のユーティリティ


Page
Project
UI

Component 内のファイル構成

- `Component/index.js` : コンポーネントの JSX
- `Component/index.css` : コンポーネントの CSS
- `Component/index.sg.js` : コンポーネントのスタイルガイド
- `Component/index.test.js` : コンポーネントのテスト
- `Component/SubComponent.js` : サブコンポーネントの JSX

components/UI

"再利用性を考慮し、独立したコンポーネントとして存在できるようにデザインします。例えば、Components・Bootstrap に見られるプリミティブなコンポーネントを指します。"

```
<!-- 画像プロキシのパスを足したり、整形パラメータを付与 -->
<Image src="/i.png" width="30" height="20" crop />

<!-- グリッドレイアウトをアレするやつ -->
<Grid align="center" valign="middle"> ... </Grid>

<!-- SVG アイコンの <use> 参照や PNG fallback を提供 -->
<Icon name="ameba" title="アメーバくん" color="green" />

<!-- props で状態制御するための Video.js ラッパー -->
<Video src="/playlist.m3u8" volume="0.8" autoplay />
```

components/Project

"プロジェクト固有のレイヤーであり、いくつかの UI と、それに該当しない要素によって構成されます。例えば、チャンネルリストやユーザープロフィールなどがそれに該当します。このレイヤーでは UI の定義をオーバーライドすることを許容します。"

```
<!-- 番組リスト -->
<ProgramList programs={programs} theme="red" vertical />

<!-- フォローボタン -->
<FollowButton channel={channel} size="lg" />

<!-- <Video> をインフィード再生 -->
<InFeedPlayer program={program} />
```

components/Page

"URL と対応するページのレイヤーであり、UI・Project を組み合わせて完結させるか、必要であればページ固有のスタイルを追加します。このレイヤーでは、UI・Project の定義をオーバーライドすることを許容します。"

```
<Page theme="darkestEx">
  <section className="u-mt40 u-mrA u-mlA u-wContainer">
    <Heading className="u-mb40">サインアップ</Heading>
    <div className="u-mrA u-mlA u-mb60 u-wForm">
      <ProfileForm submitLabel="FRESH! をはじめる"
                   defaultValues={socialProfile}
                   currentValues={profileInputs}
                   errors={profileErrors}
                   onSubmit={onSubmit} />
    </div>
  </section>
</Page>
```

components/Utils

"特定のUIを表現するのではなく、テキストの整形や、タイムスタンプからの相対時間の表示、URL文字列に対する自動リンクなどのユーティリティが該当します。"

```jsx
<!-- 文字列内の URL をリンクにする -->
<AutoLink>{description}</AutoLink>

<!-- 相対時間の表示 -->
<RelativeTime datetime={program.onAirAt} />

<!-- 長い文字列を指定した行数の末尾で ... にして切る -->
<LineClamp line={3}>{longdesc}</LineClamp>
```

SUIT CSS naming conventions をベースに
Chainable BEM modifiers を Mix (@pocotan001 談)

```
<div class="Component -modifier is-stateType">
  <div class="Component__subComponent -modifier">
    ...
  </div>
</div>
```

つまり、CSS はこうなる

```
/* Component */
.Component { }
.Component.-modifier { }

/* Subcomponent */
.Component__subComponent { }
.Component__subComponent.-modifier { }

/* State */
.Component.is-stateType { }
```

レイアウト情報の所有は再利用性を妨げるので
使用時にユーティリティクラス (u-***)を併用

```
<Panel className="u-mb28">
  <p className="u-tSm-dark u-mb8">
    <RelativeTime className="u-vaM u-mr12"
                  datetime={program.onAirAt} />
    <Icon className="u-mr4" name="watch" />
    <span className="u-vaM">
      <FormattedNumber value={program.viewCount} />
    </span>
  </p>
  <p><AutoLink>{program.description}</AutoLink></p>
</Panel>
```

FRESH! における
コンポーネント設計の心得

- 関数合成のように単機能を組み合わせること
- 求められる再利用性の程度を明確にすること
- 規模に応じた簡潔なレイヤリングをもつこと
- `shouldComponentUpdate` を信じて細分化すること
- @pocotan001 と議論すること

About Atomic Design

Atomic design is a methodology used to construct web design systems.

abema.tv と Atomic Design

Functional Stateless Components をうまく使ってる風

たぶん、そのうち中の人がどこかでアウトプットする

There are five distinct stages in atomic design:

まとめ

エッジな構成を運用した学び

- SSR は最高です
- SPA は向き不向きあります
- コンポーネントの設計だいじ

食べ残されたトピック :P

- **Web 技術で HLS を再生することへの呪詛**
- コンポーネント時代のセマンティクス管理
- 超速でデータが流れる Flux のパフォチュー
- Babel と TypeScript が混在する過渡期構成の辛み


THANK YOU
aho.mu
@ahomu