りあクト!

TypeScriptで極める現場のReact開発

大岡由佳

React 16.8対応!

現場で使えるReactの考え方やツールを徹底解説

「Reactの流儀」とは

Redux DevTools, Emotion,
Storybook, StoryShots, Jest, Cypress...

シリーズ累計
1.8万部
突破!

BOOTH
技術書売上
12位

りあクト!
TypeScript で極める現場の React 開発

大岡由佳　著

くるみ割り書房

まえがき

本書は 2018 年 10 月の技術書典 5 で頒布された『りあクト！ TypeScript で始めるつらくない React 開発』の続編となります。前作が図らずも筆者の当初の期待を大きく上回る好評を博してしまったため、続編を書くにあたり何を題材に書けばいいのか、かなり悩みました。Gatsby のような React ベースの静的サイトジェネレータ、ポスト Restful API との期待が高まっている GraphQL クライアントの Apollo、React のパフォーマンスチューニングやサーバーサイドレンダリングの話題と、いろいろ検討してみたのですが、どれもあの続編としてはあまりふさわしいと思えませんでした。

そこで前作『りあクト！』のどこが評価されたのか、そこに立ち返ってみることにしました。シニアと若手エンジニアの対話による説明が理解しやすかったという読者からの声もありました。ただ、あの本を書くのに筆者の一番のモチベーションとなっていたのは、「読者に React の思想を理解してもらった上で、綺麗な設計・綺麗なコードを書いてほしい」という思いでした。それは筆者がフリーランスとしていくつかの現場を経験した中で、設計が破綻していたり、著しく可読性の低いコードに出会って苦労させられた経験からのものです。前作のあの内容は、「React で綺麗な設計、かつ綺麗なコードを書くために、最小限必要なところまでさかのぼって、最低限必要な要素を網羅した」ものであり、そしてそこが評価されたのではないかと考えました。

もしそうであるならば、続編もその「React で綺麗な設計、綺麗なコード」を実現することに即した内容であるべきでしょう。キーワードは「現場」です。現実の業務開発の場においてはチーム開発が当たり前となっていますが、それぞれのスキルも経験も信条も異なる複数のメンバーが共同で作業する中で、極力、技術的負債を生じさせず将来にわたってメンテナンス性を確保していくのは非常にタフな仕事です。しかしそれを助けてくれる考え方やツールがすでにいくつもあり、正しいポリシーの元で運用できればその問題への有効な打開策となりえます。

そこで筆者がこれまでいくつかの現場にそれらを導入し、運用してきた経験をベースにして、さらに最新の情報や知見を盛り込んだ内容で本書を書き上げることにしました。たとえば、コンポーネントのスタイルガイドを作成しておくことで、メンバー間でのコンポーネントの再利用性が高まります。また何をテストし何をテストしないかの方針を決め、ユニットテストや E2E テストを必要な範囲で作成することで、将来にわたって可動性と品質を担保できます。意識の高い開発者層はすでにやってい

ることかもしれませんが、React を始めとするモダンフロントエンド開発そのものの歴史が浅く経験者も少ない中、まだ多くの現場ではなかなかそこまで実践できていないのではないでしょうか。そういった情報を前作同様、取っつきやすい会話形式の切り口で紹介できれば、多くの人たちの役に立ってくれるような本になると考えました。

例によって技術選定はもちろんのこと、今回は何をドキュメントとして残すか、何をテストするかといったポリシーについて、筆者の個人的な考えに色濃く影響された内容になっています。そのあたりはあらかじめご理解いただいた上で、読者の方々が客観的な判断を元に、あくまで参考として読んでいただければと思います。

そして今回も、前作から引き続き芝崎雪菜と秋谷佳苗のふたりの対話によって、説明が進められていきます。ふたりの関係なども気にしつつ、ぜひ楽しんで読んでいただければ幸いです。

本書について

登場人物

柴崎雪菜（しばさき・ゆきな）

とある都内のインターネットサービスを運営する会社のフロントエンドエンジニア。React 歴は二年半ほど。本格的なフロントエンド開発チームを作るための中核的人材として、今の会社に転職してきた。チームメンバーを集めるため採用にも関わり自ら面接も行っていたが、彼女の要求基準の高さもあってなかなか採用に至らない状態が続いていた。そこで「自分が React を教えるから他チームのエンジニアを回してほしい」と上層部に要望を出し、希望者の社内公募が実行された。

秋谷香苗（あきや・かなえ）

柴咲と同じ会社のエンジニアで新卒二年目。入社以来もっぱら Ruby on Rails によるサーバーサイド開発に携わっていたが、柴崎のメンバー募集に志願してフロントエンド開発チームに参加した。そこで柴崎から「一週間で戦力になって」と言われ、マンツーマンで教えを請うことになったのだが……。

本文中で使用している主なソフトウェアのバージョン

- React 16.8.6
- Create React App 2.1.8
- TypeScript 3.4.2
- React Router 5.0.0
- Redux 4.0.1
- React Redux 6.0.1
- Redux-Saga 1.0.2
- React Helmet 6.0.0-beta
- @emotion/core 10.0.10
- Storybook 1.0.0
- Redux Saga Test Plan 4.0.0-beta.2
- Cypress 3.2.0

目次

まえがき ……2

本書について ……4

登場人物 ……4
本文中で使用している主なソフトウェアのバージョン……5

目次 ……6

プロローグ ……8

第 1 章 デバッグをもっと簡単に……10

1-1. VSCode でらくらくデバッグ ……10
1-2. React Developer Tools を使ったデバッグ……15
1-3. Redux DevTools を使ったデバッグ ……21

第 2 章 コンポーネントのスタイル戦略……26

2-1. グローバルな CSS との闘い ……26
2-2. CSS Modules で CSS をカプセル化する ……28
2-3. 群雄割拠の CSS in JS ライブラリ ……30
2-4. Emotion を UI フレームワークと組み合わせて使う ……36

第 3 章 スタイルガイドを作る……40

3-1. なぜコンポーネントのスタイルガイドが必要か ……40
3-2. スタイルガイド作成ツールの比較 ……42
3-3. Storybook を使おう ……43

第 4 章 ユニットテストを書く……50

4-1. 私たちは何をテストするべきか ……50
4-2. Jest で API ハンドラをテストする ……52
4-3. Redux Saga Test Plan で Saga をまるごとテストする ……56
4-4. StoryShots でスナップショットテスト ……61

第 5 章 E2E テストを自動化する……68

5-1. E2E テストツールの比較 ……68
5-2. Cypress を導入しよう ……71

第 6 章 プロフェッショナル React の流儀 ……79

6-1. Advanced Thinking in React ……79
6-2. Store のスケーリング戦略 ……82

エピローグ ……88

あとがき ……89

著者紹介 ……91

第１章 デバッグをもっと簡単に

1-1. VSCode でらくらくデバッグ

「秋谷さんは、Rails で開発してたときはデバッグはどんなふうにやってたの？」

「Ruby 用には Byebug っていうツールがあって、ブレークポイントを置きたいところに byebug って一行追加すると、そこでサーバの実行が止まって対話的に任意の変数の中身をダンプできたりするんですよ。最初の内は便利と思って使ってたんですが、でも結局やりたいのってほとんど特定の変数のダンプくらいなので、よっぽどのことがない限り p foo みたいに普通に表示させる方法に戻っちゃいましたね。そっちのほうが手っ取り早いので」

「なるほど。React なら任意の行に console.log(foo) みたいに書き入れれば同じようなことができるよね。これまでも何回かやって見せたけど。でも VSCode と Chrome を連携させることで、もっと簡単にデバッグすることができるようになる。GUI インタフェースなので、いちいちコマンドを入力しなくても変数の中身を確認するくらいなら、ちょっとしたマウスを操作するだけで可能になる」

「へ――、便利そう」

「じゃ、実際にやってみようか。まずはデバッグに使うサンプルコード[*1] をダウンロードしてきて」

「はい。どれどれ？　えっ、これって雪菜さんが作ったものですか？」

「うん、先週使った GitHub API で会社のメンバーリストを取得・表示するコードをさらに機能追加して、リポジトリの検索もできるようにしたものだよ。今週は基本的にこのコードを使って学んでいく予定」

「いやいや、これけっこういい感じに作り込まれてますよね。React Router、Hooks、Redux、Redux-Saga と先週習った機能がほぼ全部盛り込まれて連携が取れてて、さらに Hooks を使ったフォームハンドリングまでやってるじゃないですか。正直、このコードの説明をお願いしたいくらいなんですけど……」

「まあ、現状のウチでの新規プロジェクトを始めるボイラープレートになり得る内容だからね。でも

---

*1　https://github.com/oukayuka/ReactOnTheJob/tree/master/01-debug/01-vscode-chrome

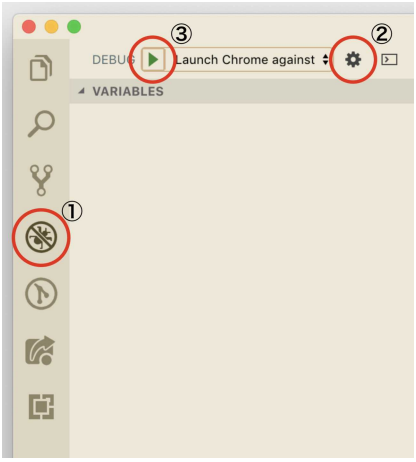
先週までの知識があれば、独力で全部読み解けるはずだよ。だからこれの中身を理解するのは秋谷さんの宿題。今はこれを使ってデバッグをやっていきましょう」

「……はーい、わかりました……」

「じゃ、まずは VSCode の Debugger for Chrome っていう拡張をインストールして」

「はい、VSCode の左のアクティビティバーの拡張のアイコンをクリックして、『Debugger for Chrome』で検索っと。へー、二千万ちかくもダウンロードされてるんですねこれ。……はい、インストール完了しました」

「じゃ同じく左アクティビティバーの、虫を丸く囲んで斜線が入ったデバッグアイコンをクリック①。開いたサイドバーの右上部分に歯車の設定アイコンがあるよね。それをクリックする②と設定ファイルが開く」



「その設定ファイルの内容を以下のように書き換えておいて。サンプルコードではすでにこうなってるけどね」

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Launch Chrome against localhost",
      "type": "chrome",
```

```
        "request": "launch",
        "url": "http://localhost:3000",
        "webRoot": "${workspaceRoot}/src",
        "userDataDir": "${workspaceRoot}/.vscode/chrome",
        "sourceMapPathOverrides": {
          "webpack:///src/*": "${webRoot}/*"
        }
      }
    ]
  }
```

「これは Create React App のドキュメント[*2] で推奨されている内容をそのままコピったもの。ここに書かれた内容はプロジェクトルートに `.vscode/launch.json` として保存されるようになってるの。これで設定完了なので、サイドバー上の緑の ▶ボタンを押して③、デバッグサーバを走らせてみよう」

「あれ？　なんかいつもの Chrome とちがう雰囲気のウィンドウが別に立ち上がりましたよ？」

「うん、それが VSCode のデバッグセッションと連携した Chrome ウィンドウね。これ以外のウィンドウや、同じウィンドウでも別のタブは実行中のデバッグセッションとつながってないので注意して」

「あ、そうなんですね。メモメモ……」

「じゃ、さっそくデバッグのデモをしてみよう。API ハンドラのファイル `src/services/github/api.ts` を開いて、66 行目にブレークポイントを設定してみて。行番号の左をクリックするだけね」

```
52  export const searchRepositoriesFactory = (optionConfig?: ApiConfig) => {
53    const instance = createAxiosInstance(optionConfig);
54
55    const searchRepositories = async (
56      q: string,
57      sort?: 'stars' | 'forks' | 'updated' | '',
58    ) => {
59      try {
60        const params = qs.stringify({ q, sort });
61        const response = await instance.get(`/search/repositories?${params}`);
62
63        if (response.status !== 200) {
64          throw new Error('Server Error');
65        }
66        const repositories: Repository[] = response.data.items;
67
68        return repositories;
69      } catch (err) {
70        throw err;
71      }
72    };
```

*2　https://facebook.github.io/create-react-app/docs/setting-up-your-editor#visual-studio-code

「はい、できました」

「じゃ、ブラウザから http://localhost:3000/repositories/search のリポジトリ検索のページにアクセスして、テキストフォームの中に『redux-saga』とでも入力して検索ボタンをクリック」

「おおっ、ページが『読み込み中...』でローダーがぐるぐる回ってる状態のまま止まってます。VSCodeのほうは、ブレークポイントが設定された 66 行目がハイライトされてますね」

「その状態で、response.data.items の items の上にカーソルを持ってきてみて」

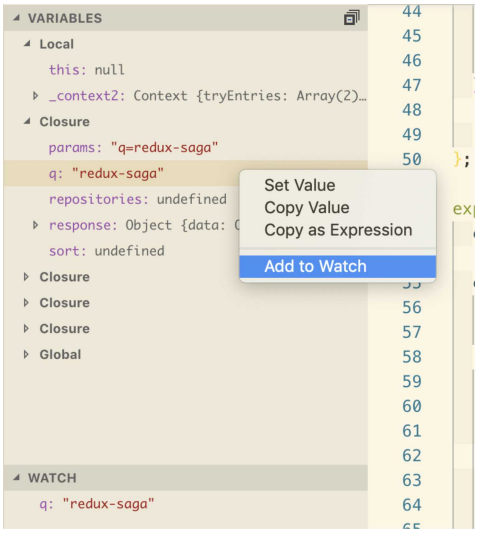
「あっ、これ API から取得してきたレスポンスの中身ですよね。▷ をクリックして展開すると個別の中身まで確認できます」

「同様に、このスコープにあるローカル変数の中身は全部マウスホバーで見られるよ。また、ペイン下の DEBUG CONSOLE パネルのコマンドラインに変数名を入力して Enter でも同様に確認できる。内容をテキストデータとしてコピペしたいときなんかはこっちを使ったほうがいいかもね」

「うーん、便利ですねえ」

「じゃ、いったん最後まで実行させようか。デバッグサーバ起動中は VSCode のペイン上部に操作パレットが表示されてるから一番左のボタンを押して、そこから動作を再開させて」

「じゃあ、また同じくブラウザで検索ページからボタンを押して検索を再実行しよう。さっきと同様ブレークポイントで止まったら、左サイドバーの『◢ VARIABLES』パレット内のひとつめの『◢ CLOSURE』をクリックして、その中の変数一覧を表示させて」



「q のところでマウスを左クリックして『Add to Watch』を選択。すると、その下の『◢ WATCH』パレットに変数 q が追加されるよね。また再始動して検索を最後まで終わらせて、今度はフォームの中身を『create-react-app』に変えて検索してみよう。そうしたら『◢ WATCH』パレットの q の中身が "create-react-app" に変わったのがわかるよね。こうやって任意の変数の値をトレースすることができるの」

「へー、へー、へー」

「これまであげた VSCoce で行った一連のデバッグは、Chrome の Developer　Tool からも同じように

操作できるよ。まあ VSCode でやったほうが圧倒的に使いやすいので、わざわざ Chrome でやる人はあまりいないだろうけど。こういう Developer Experience を共通化したいのもあって、うちのチームではエディタを VSCode に統一してるの。

それじゃ、以上で Debugger for Chrome 拡張を使った VSCode でのデバッグの説明はおしまい。操作パレット一番右の赤い ■ボタンを押すと、デバッグサーバを停止できるよ。今度からはこんな感じでデバッグしていきましょう」

「確かに、実際のコードに手を全く加えなくてよくて、マウスホバーだけで変数の中身が次々確認できるの、超便利ですね。これならめんどくさがりの私でも使っていけそうです！」

「うん、既存のコードに手を加えずにデバッグできるのがいいよね。`console.log()` とかでやってると色んなところに仕込んだ後、消し忘れが発生することもあるし」

「あ、それ私もよくやってました（笑）」

1-2. React Developer Tools を使ったデバッグ

「Web アプリケーション開発には、『デバッグ』といっても複数の側面からのものが存在してるよね。代表的なところで言えば、Rails でもそうだったと思うけど、プログラミング言語レベルのものと HTML や CSS のレベルのもの。さっき紹介した Debugger for Chrome と VSCode の組み合わせではプログラミング言語レベルのデバッグがサポートできるわけだけど」

「HTML や CSS レベルでのデバッグというと、Chrome のデベロッパーツールですか？」

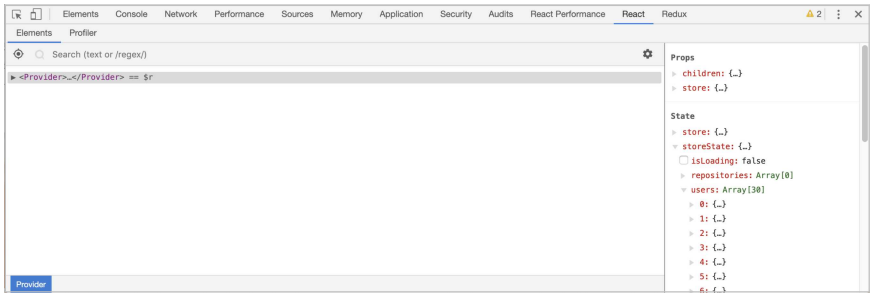
「そうそう、Chrome DevTools 。定番だよね。これから説明するのは Chrome DevTools をさらに機能拡張して React のコンポーネントレベルでのデバッグを可能にしてくれる Chrome 拡張。名前もそのまんま『React Developer Tools』っていうんだけど、Facebook から正式に提供されてるの。Chrome ウェブストアで検索してみて」

「これ[*3]ですね。さっそくインストールと。……って、あれ？　これどうやって使うんですか？」

「まずChromeメニューの『表示 ＞ 開発 / 管理 ＞ デベロッパーツール』からデベロッパーツールのウィンドウを開いて。デフォルトではブラウザを縦に分割して右半分がデベロッパーツールになってしまうので見えないんだけど、これタブ群の一番右に『React』ってタブが追加されてるのね。縦長だと使いにくいので「⋮」をクリックして『Dock side』から横分割で下がデベロッパーツールのウィンドウになるようにしてみて」

「あっ、いちばん右に『React』ありました！」

「そう、それね。じゃあ今回もさっき使ったサンプルコード[*4]をデバッグのデモに使いましょう。サーバを起動したら、ブラウザのトップページから『いろんな会社のメンバー ＞ Facebookのメンバー』で`http://localhost:3000/facebook/members`にアクセスして、そこでReact DevToolsウィンドウを開いて」



「デフォルトでは一番トップレベルのコンポーネントが閉じた状態で表示されてる。これは`src/index.tsx`で定義されてるReduxのProviderコンポーネントだね。右ペインにはそのコンポーネントのPropsとLocal Stateが表示される。ここでは`storeState`というのがReduxが提供しているStoreのState値だね。`src/reducer.ts`で定義されている`GithubState`がその型になってる。コードを引用するとこれね」

---

*3 https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi

*4 https://github.com/oukayuka/ReactOnTheJob/tree/master/01-debug/01-vscode-chrome

第 2 章 コンポーネントのスタイル戦略

2-1. グローバルな CSS との闘い

「これまでのサンプルコードでは、CSS フレームワークである Semantic UI の React 向け実装である Semantic UI React を使って見た目を整えてきたわけだけれど、これってスタイリングの実体としてはただ Semantic UI CSS が提供するところの `semantic.min.css` を、トップレベルのコンポーネントファイル `src/index.tsx` でグローバルに読み込んでるものなんだよね。ただその上で、個別にパーツのマージンとかを調整したい場合には各コンポーネントから CSS をインポートして、そこで任意のクラスを追加設定したりしてた」

「はい、そうなってましたね」

「そうやってコンポーネント別に JavaScript から任意の CSS をインポートして見た目のカスタマイズできるのは、Create React App が内部で利用してる Webpack の恩恵を受けてるから可能なことなんだけども。Webpack のような高機能なモジュールバンドラー登場以前の開発の現場では、ひとつのアプリケーションに対しておおよそひとつの CSS ファイルを当ててた。それをみんなで編集するもんだから各々が好き勝手に書いたりすると、ここを変更したら別のところが崩れたとか、スタイルの優先順位が複雑になってしまって収拾がつかなくなり、`!important` がそこかしこに書き散らされたりと、依存関係も可読性も破綻した巨大で複雑怪奇な CSS ファイルができあがってしまうこともめずらしくなかった」

「ついこの間までサーバーサイドアプリケーションを開発してたので、よくわかります。けっきょく前のチームでは、CSS は絶対的にデザイナーさんだけのものでエンジニアがさわっちゃいけないことになってて、その用意してくれた巨大な CSS ファイルは魔術書みたいに難解で禁断の存在になってました」

「そういった状況の中で生まれたのが BEM とか OOCSS とか FLOCSS とか SMACSS といった、厳密な命名規則を適用してクラスを細分化し、できるだけ破綻を防ごうという CSS 設計の思想というか手法だね。BEM は `<Block>_<Element>__<Modifier>` という規則で一意となる長いクラス名を用いる手法で、シングルクラスなので記述量が多くなるが各々の独立性が高い。対照的にオブジェクト指

向をクラス設計に持ち込んだ OOCSS は再利用性が高いけど、変更の影響範囲が大きくなる」

「『BEM』って言うんですか？　このめっちゃ長いクラス名は見おぼえがありますね。デザイナーさんから渡された CSS の中がこんな感じだったと思います。そのときは『なんでこんなに長ったらしい名前なんだろう』とあんまり深く考えてなかったんですけど、そういう意味があったんですね」

「うん。たぶん BEM そのものじゃなくて FLOCSS や SMACSS に代表される後発の手法だったと思う。BEM のように独立性を高めれば記述量が増えるし、OOCSS のように再利用性を高めようとすれば変更の影響範囲の予測がつきづらくなる。ほとんどの後発の手法は、そのいいところどりをしようとふたつを独自のやり方で組み合わせたものと言っていいからね。

　でも、そもそもなぜこんなやっかいな問題が起きるのかというと、CSS のクラス名ってプログラミング言語と違って常にグローバルだからなんだよね。どこからでも名前を指定すればクラス内容を上書きできて、そうしたらそのクラスを参照している全ての HTML ファイルに影響が出てしまう。だからこの CSS の名前問題に悩まされるのは、一リクエストに対して一枚岩の HTML データを返すことを基本思想にしていた従来型の Web アプリケーションには逃げられない宿命だったと言える」

「なるほど」

「だからこそコンポーネント指向をベースとする React の開発者から、別の視点によるアプローチが提案されたのは必然だったのかもしれないと私は思う。コンポーネント指向はロジックに加えて見た目もコンポーネントという独立した単位に押し込めるものだからね。2014 年に、後の React Native の共同開発者でもある Christopher Chedeau が行った『React: CSS in JS』[*1] というタイトルのプレゼンが非常に大きな反響を生んだの。CSS を JavaScript のオブジェクトにして、コンポーネントの style に入れ込むというアプローチで、従来の CSS 周りの問題が解決できるのではという提案だった」

「ふむふむ」

「それを受けてその後、たくさんの CSS in JS ライブラリが誕生したわけだけども。その中でまず頭ひとつ抜け出したのが CSS Modules だったの」

---

*1　https://speakerdeck.com/vjeux/react-css-in-js

2-2. CSS Modules で CSS をカプセル化する

「CSS Modules は、CSS のローカルスコープを JavaScript を用いて自然な形で利用できるようにしたライブラリ。くわしく説明する前に、まずはその簡単な使い方から見てみよう」

```
/* style.css */
.common {
  font-size: 10px;
}
.normal {
  composes: common;
  color: green;
}
```

```
import styles from "./style.css";

const MyAwesomeComponent = () => (
  <div className={styles.normal}>Awesome!</div>
);
```

「このアプリを起動すると、ブラウザからは以下のように見える」

```
<div class="style-normal__227xg">Awesome!</div>
```

「実際のクラス名は <ファイル名>-<クラス名>__<ハッシュ> というフォーマットで自動生成され、末尾に付与されるランダムなハッシュ値によって一意であることが保証される。だから開発者は何も考えなくても、クラス名をローカルなものとして安心して使うことができるのよね」
「なんかちょっと BEM の命名規則と似てますね」
「たぶんそれは意識的にそうしたのかもしれないね。ちなみに、ファイル名とクラス名がハイフンでつながれてるので、CSS Modules では CSS のクラス名にハイフンが使えないのは注意してね」
「あ、そうなんですね。知らないと普通に使っちゃうところでした。でもそうか、参照するときは

第 3 章 スタイルガイドを作る

3-1. なぜコンポーネントのスタイルガイドが必要か

「サーバーサイドアプリケーション開発の現場ではなかった文化だと思うけれど、SPA 開発においてはデザイナーさんが用意してくれるものとは別に、フロントエンドエンジニアが UI のスタイルガイドを作ることがままあるの」

「へー、スタイルガイドってデザイナーさんの領域だと思ってました。エンジニアが実装フェーズに移る前に、色味やフォントの指定とか、主だった UI パーツの見本とかが掲載された素材集が Photoshop や Sketch で作られたみたいなの。でも結局、CSS もデザイナーさんが用意してくれるので、単に初期段階で UI スタイルをなんとなく把握する感じのものになってて、正直なところエンジニア的にはそれほど重要視されてませんでしたね」

「そういえばここまで言ってなかったかもしれないけど、SPA 開発の現場では CSS を書いたりメンテナンスしたりするのはフロントエンドエンジニアの領分だからね。CSS が書けないデザイナーさんもめずらしくないし」

「ええっ？！　そうなんですか？」

「うん、他の会社でもだいたいそんな傾向だよ。スマホのネイティブアプリ開発の文化が持ち込まれていたり、元々『コーダー』と呼ばれていた人たちがジョブチェンジしてフロントエンドエンジニアになってたりといった要因もあると思うけど、一番の理由はモダンフロントエンド開発ではコンポーネント指向がもはやデファクトスタンダードになってるからだろうね」

「やはりそこもコンポーネント指向ですか……」

「ロジックと見た目を独立したコンポーネントに閉じ込めるわけだから、ロジックが書けないデザイナーさんには手を出せないよね。CSS がフロントエンドエンジニアの領分になってるからこそ、先日やった CSS in JS が流行したりするわけだよ」

「……ううん、そういう文脈だったんですね。でもデザイナーさんがロジックを書けないのと同じように、私も CSS 書けないんですけど……」

「今の CSS は Flexbox 主体だから、昔ほど組み上げるのはつらくないよ。それに最近のデザインツー

ル、Sketch とか Figma といったものは HTML / CSS 書き出し機能があるから、それを利用すれば自分で一から書く必要もないし」
「ほっ、そうなんですね。それを聞いてちょっとだけ気が楽になりました」
「さかのぼって要約すると、こんな感じになるかな」

1. スマートフォンのネイティブアプリの普及により、Web アプリも同様に高度な UI・UX を求められるようになる。
2. その要求に応えるために JavaScript による SPA が登場、さらにその開発における複雑さを解決するためにコンポーネント指向が導入されていった。
3. ロジックと見た目が融合したコンポーネントは、フロントエンドエンジニアにしか手を出せない。
4. デザイナーはもっぱら見た目だけを厳密に定義する本来の役割に戻り、CSS はフロントエンドエンジニアの領分になる。

「なるほど。最初は無茶ぶりに聞こえましたけど、こうやって順を追って説明されると必然的な流れのように思えて納得せざるをえないですね……」
「そういう理由からモダンフロントエンド開発の現場では、もっぱら見た目だけ厳密に定義するデザイナーと CSS を組み上げる複数のフロントエンドエンジニアが協力しあってアプリケーションの UI を作っていくことになるの。だからこそ、そのコミュニケーションを円滑にするためにコンポーネント単位のスタイルガイドが必要になってくるのよ」
「ふむふむ」
「デザイナーとエンジニアのコミュニケーションだけでなく、エンジニア同士の間でもコンポーネントの再利用性を高めるためにスタイルガイドは有用だよ。それがないと他のメンバーがすでに同じようなコンポーネントを作っていたのに気づかずに自分でも作ってしまって、リポジトリ内に役割の重複したコンポーネントがあふれたりすることになる」
「確かに。そうか、そうなんですね、その状況は全然わかってませんでした。そんなことも知らずにフロントエンド開発に志願した私って、ひょっとして恥ずかしい人だったんでしょうか？」
「いや、まあ SPA 開発自体が歴史が浅い技術だからね。会社によっては違うやり方をやってるところもあるだろうし。これからおぼえていけばいいんじゃないかな」

3-2. スタイルガイド作成ツールの比較

「でも、コンポーネント単位のスタイルガイドって、具体的にどんなものなんですか？」

「うん、じゃあこれを見てもらおうかな。React Cosmos っていうスタイルガイド作成ツールのデモページ[*1] なんだけど」

「あはは、なにこれおもしろーい。テトリスが遊べるー」

「これはちょっと題材が変わりダネだけどね。左のツリーパネルから各コンポーネントの内容を個別に指定して見ることができたり、パラメータを変更して見た目がどう変化するかを確認できるようになってたりするのは、だいたいこういったスタイルガイド作成ツールに共通してる」

「スタイルガイド作成ツールも、CSS in JS みたいにたくさんあってしのぎを削ってる感じなんですか？」

「ん――――、CSS in JS とは状況がちょっと違うかな。一強の絶対的存在があって、あとはまあドングリの背比べみたいな」

「えー、じゃあこの React Cosmos もその他大勢の内のひとつなんですか。ちょっと気に入ったのに」

「スタイルガイド作成ツールには大きく分けてふたつの種類があってね、ひとつは JavaScript で記述するもの。もうひとつは Markdown で記述するもの。大まかに言うと JavaScript で記述していくタイプは作るのが多少面倒だけど柔軟性があって、機能も豊富。Markdown で記述するタイプは手軽に書けるけど機能が絞られていて、動的な要素が少ない」

「どっちが主流派なんですか？」

「Markdown 記述タイプは、数は多いんだけどね。最古参の Styleguidist[*2] を始め、Docusaurus[*3]、最近

---

*1 https://react-cosmos.github.io/

*2 https://react-styleguidist.js.org/

*3 https://docusaurus.io/

伸びてきてる Docz[*4] なんかがある。主流派、というか一強なのが Storybook[*5]。いつものように npm trends のグラフを見せようかと思ったけど、2019 年 3 月時点の週間ダウンロード数だと Storybook の 144 万件に対して Markdown 記述タイプでトップの Styleguidist が 8 万件なので、二位以下が地に張り付いてて意味のあるグラフにならない」

「えー？　そこまでなんですね。なんでそんなに差がついちゃってるんでしょう？」

「Markdown 系は手軽に書けるのはいいんだけど、それだけだとスタイルガイドを作ることが自己目的化してしまって気がついたら面倒になって作る人がいなくなってた、という事態になるのもめずらしくないからね。その点 Storybook は、後で別途説明するけどテストと連携したりとか、スタイルガイドを作るモチベーションをエンジニアに感じさせてくれる仕組みがあるんだよね」

「なるほど、興味が出てきました」

「じゃ、既存のサンプルコードに Storybook を導入するのをやってみようか」

3-3. Storybook を使おう

「Storybook は TypeScript を正式にサポートしていて公式ドキュメントにもその導入方法が書いてある[*6] のでその『Setting up TypeScript with babel-loader』ってとこに気をつけながらその通りにやればおおむねいいんだけど、ところどころ注意点があるし、実際に使うにあたってカスタマイズしといたほうがいいところもいくつかある」

「うーん。そのページ、Webpack のめんどそうな設定がたくさんありますね。create-react-app 使ってるから Webpack パズルとは無関係でままでいられると思ったのに」

「それでも Eject することを考えたら全然楽だよ。昔は Eject しないと Storybook 使えなかったんだか

*4　https://www.docz.site/

*5　https://storybook.js.org/

*6　https://storybook.js.org/docs/configurations/typescript-config/

第 4 章 ユニットテストを書く

4-1. 私たちは何をテストするべきか

「フロントエンドのテストっていろいろ難しいのよね。一番外側の入出力だけをとっても、たとえば対象がサーバーサイドの REST API なら、入力が単なる HTTP リクエストで最終出力がプレーンテキストな JSON データなので、テストするのに困ることはほとんどない。でもフロントエンドは入力が UI の操作なら最終出力も UI の状態でどっちもつかみどころのないものという、テストするには非常に不都合な宿命を背負ってるわけ」

「確かにそうですね。でも内部の機能ごとのユニットテストはその限りじゃないのでは？」

「『難しい』と言ったのは、計測が難しいというのもあるけど、できたとしてそこにどれだけの意義を見いだせるか、という意味もあるの。フロントエンド、特に React の場合はほとんどの関心事が View なわけじゃない？

純粋なロジック部分については、もちろんテストは必要だと思う。でも個々の View コンポーネントをテストしようとするなら、`title` という Props に `'Foo'` という文字列を与えたら `<title>` タグの中に `Foo` が表示される、みたいなテストになるわけだけど、秋谷さんはそんなテストを延々と書きたいと思う？」

「……徒労感が大きそうですね。『1 + 3 = 3 + 1』を書き続けてるみたいな気になるかも」

「今挙げたのは極端に単純化した例なわけだけど。テストというのはやみくもに書けばいいってものじゃない。効用の低いテストの記述を義務づけるのはチームメンバーのモチベーションを削ぐし、積み上がったテストの実行時間も馬鹿にならないし、そもそもプロダクトが大きく変わっていく局面では膨大なテスト自体が変化への負債になりかねない。

だから他の開発者たちに聞いても、人によって『テストは一切してない』という意見から、『Enzyme[*1] によるコンポーネントテスト出力のテストをしてるから E2E テストは不要』「E2E テストをしてるか

*1 https://airbnb.io/enzyme/

らコンポーネントのテストは不要』とフロントエンドのテストに対する態度はまちまちなんだよね。それもフロントエンド開発において、何をテストするべきか、テストする労力に対してその効果が見合うのかが明確じゃないせいだと思う」

「なんだか聞いてる私もよくわからなくなってきました……」

「そもそも私たちエンジニアがテストを書く意味に立ち返ってみると、私見だけどその理由はこの四つに集約されると考えてるのね」

1. **設計者の意図通りに機能が実現されているかの確認**
2. **新規に追加した全ての処理に破綻がないかの確認**
3. **既存の機能を破壊していないかの確認**
4. **モジュラリティの確保**

「一言で『テスト』というと 1 のイメージが大きいけど、ソフトウェアテストを書くことによる効用は、私はむしろ 2・3・4 のほうが大きいと思う。だって極端な話、1 はテストを書かなくても使ってみれば一発でわかることだから」

「うーん、確かに。でもそれを言うのは反則な気もしますけど（笑）」

「2 は『安全性』とも言い換えられる。全ての分岐を通って途中で予期せず処理が止まったりすることがないかということだね。これは TypeScript でコードを書いていることによる Null 安全性や型整合性の確保によって、すでにある程度実現できていると思う。ソフトウェアのバグで最も多いものが Null アクセスの例外だからね」

「以前、TypeScript について学んだときに、その話をしてくれましたね」

「だから残るのは 3 の既存のコードを破壊していないかの確認と、4 のモジュラリティの確保」

「ちょっと待ってください。『モジュラリティ』ってどういう意味ですか？」

「モジュラリティとは端的に言えば、不要な他からの依存が切れていて、他からの境界が自明になっている状態。つまりモジュールとして分離されていて、独立性が高く扱いやすいことだね」

「何か聞いたことがある気がしますね。あっ、Storybook にスタイルガイドを登録するためにコンポーネントの設計を見直したことがありましたよね。あれがそうなんじゃないですか？」

「そう、勘がいいね。テストをするためにはそれ自体が独立していてインターフェースが整理されている必要があるけど、スタイルガイドを作ることも同様に View のモジュラリティを確保することにつな

がるんだよね。だからここまでやってきたことが2と4の、完全にとは言わないけどある程度の担保につながってる。残るのは、3の既存の機能を破壊していないかの確認」

「ふむふむ」

「これも実は、ここまでにやったことで半分以上すでに環境が整ってるの。Storybookのプラグインに StoryShots というものがあって、Storybookにストーリー登録したコンポーネントのスナップショットテストを自動的にやってくれる。『スナップショットテスト』とは、既存の UI のスナップショットを DOM テキストや画像ファイルで取得しておいて、比較して変更があった場合にアラートを上げてくれるもののこと。これを利用すれば、テストコードを記述することなく既存の View を破壊していないかを確認することができる」

「おお――、そんな便利なものがあるんですね」

「話が長くなったのでいっぺんまとめるよ。まずロジックのテストはちゃんとやる。ここまでで作ったアプリにおける純粋なロジック部分って、API ハンドラや Redux-Saga の Saga 群になるけど、これらのユニットテストを用意して実施する。

コンポーネントに関しては、費用対効果を考えて最小限、Storybookにストーリー登録した Presentational Component のスナップショットテストを行う。さらに全体的な動作の保証のために、自動化された E2E テストを正常系に限って行う。これがウチのチームにおけるテストの方針になってる」

「いいんじゃないでしょうか。すごくバランスが取れてて、チームメンバーとしても納得感がある方針だと思います」

「ありがとう。まあ私もこれまで、有用性の低いテストの記述を押しつけたりして軽くひんしゅくを買ったりしながら試行錯誤した結果、この方針に行き着いたんだけどね」

「あはは、雪菜さんと私の出会いがその試行錯誤の後でよかったです（笑）」

4-2. Jest で API ハンドラをテストする

「Create React App にはすでにユニットテストを書くための環境がすでにお膳立てされてるので、そ

れをそのまま利用させてもらうよ。Jest[*2] というオールインワンのテストフレームワークが react-scripts の中に統合されていて、TypeScript も Babel を通して使えるようになってる。最初のアプリケーションのひな形を生成するときに、`src/App.tsx` に対応する `src/App.test.tsx` というファイルが作られてるんだよね。まあこれは使わないので消してしまおう」

「JavaScript には、Jest 意外に他のテストツールとかないんですか？」

「あるにはあるんだけど、Jest が便利すぎるからね。普通、テストツールはテストランナー、アサーション、モック、カバレッジ計測などに機能別に分類されるものなんだけど、Jest はその全てがひとつにまとめて提供されてる。Facebook が提供する React 公式のテストツールという看板もあるし、ドキュメントも充実してるし、わざわざこれを外す理由はどこにもないよね」

「なるほど。じゃあ安心して使えるんですね」

「うん、そうだね。それじゃこれからテストを書いていくけど、今のサンプルコードには特にヘルパーユーティリティみたいなものは作っていないので、API ハンドラをテストしていこう。非同期通信には axios[*3] を使っていたわけだけど、その通信リクエストをモックしてくれる axios-mock-adapter というライブラリがあるので、それをインストールしておく」

```
yarn add -D axios-mock-adapter
```

「また、Storybook のストーリーファイルのときもそうだったけど、そのままだと devDevpndencies に入れられたライブラリをインポートすると ESLint に怒られるので、.eslintrc.js にテストファイルも例外扱いするように設定を追加しておく」

```
    'import/no-extraneous-dependencies': [
      'error',
      {
        devDependencies: [
          '.storybook/**',
          'stories/**',
```

*2 https://jestjs.io/ja/

*3 https://github.com/axios/axios

```
            '**/*/*.story.*',
            '**/*/*.stories.*',
            '**/__specs__/**',
            '**/*/*.spec.*',
            '**/__tests__/**',
            '**/*/*.test.*',
          ]
        }
      ],
```

これで、src/services/github/api.ts のテストを書く準備が整った。同階層のディレクトリに src/services/github/api.spec.ts を作って getMembers() の正常系のテストを書いてみよう」

```
import axios from 'axios';
import MockAdapter from 'axios-mock-adapter';

import { getMembersFactory } from './api';
import userData from './__mocks__/users.json';

describe('GitHub API handlers', () => {
  const mock = new MockAdapter(axios);

  afterEach(() => {
    mock.reset();
  });

  describe('Getting organization members', () => {
    it('should succeed', async () => {
      const companyName = 'facebook';
      mock.onGet(`/orgs/${companyName}/members`).reply(200, userData);

      const getMembers = getMembersFactory();
      const members = await getMembers(companyName);

      expect(members[0].login).toBe(userData[0].login);
    });
  });
});
```

「書き方はほぼ RSpec と同じですね。describe()　があって it()があって expect()でアサーションする形」

第 5 章 E2E テストを自動化する

5-1. E2E テストツールの比較

「じゃ、E2E テストツールについて見ていこうか」
「Rails では Capybara ＋ Selenium が定番でしたけど、あんな感じなんでしょうか？」
「基本的には同じだね。でも JS 界の E2E テストツールはもっと進化したものもある。基本的に Web フロントエンドにおいて E2E テストの主な選択肢は以下の三つだと思う」

- **Puppeteer**[*1]
- **Cypress**[*2]
- **TestCafe**[*3]

「Puppeteer は聞いたことあります。GUI のないヘッドレスモードで動く Chrome を使ってるんですよね」
「そう。Puppeteer は、Google の Chrome DevTools チームが開発した Node のライブラリで、Chrome 系のブラウザをコントロールする API を提供してる。だから正確にはテストツールじゃなくて、Jest などのテストスイートと組み合わせることでテストにも使えるようになるってことだね。だからテストを記述する際は、よく言えば素直な JavaScript で書ける、悪く言えば煩雑で冗長な書き方になる」
「雪菜さんの好み的には？」
「あとのふたつはテストに特化しているおかげで記述するコード量も少なくて済むのもあるけど、その他にもテストに便利な機能が充実してるので、Puppeteer を使う選択肢はストイックすぎるかな」

---

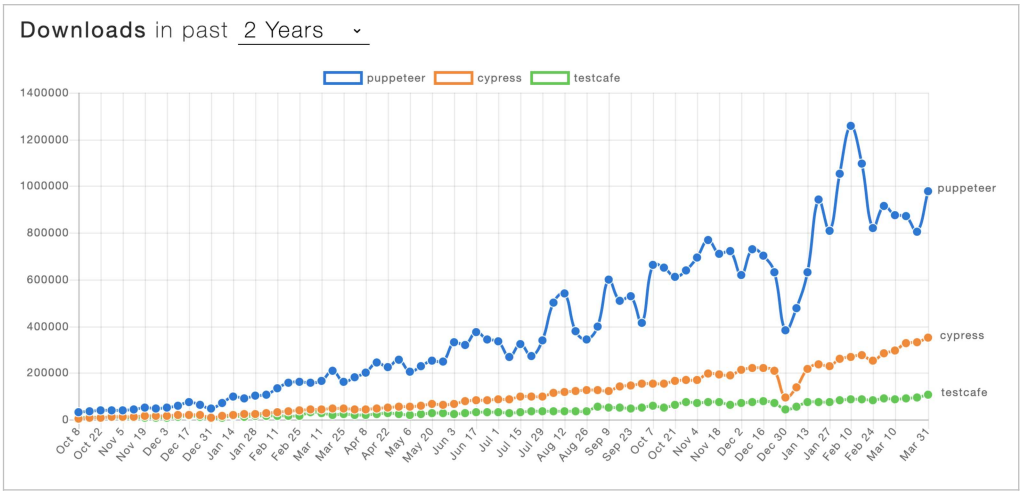
*1 https://github.com/GoogleChrome/puppeteer

*2 https://www.cypress.io/

*3 https://devexpress.github.io/testcafe/

「なるほど。じゃ Cypress と TestCafe では？」

「その前にいつもの npm trends グラフ[*4] を見てみようか」



「Puppeteer がいちばん人気ですね』

「さっきも言ったけど Puppeteer はテストだけに使われてるわけじゃない。スクレイピングだったり、スクリーンショットを撮るのだったり、色んな用途があるからね。いくらかさっぴいて考えるべき」

「Cypress と TestCafe もよく見ると差が大きいですね。三倍くらい違う」

「だんだん差が開いていってるよね。私も昔は TestCafe を使ってたんだけど、やっぱり Cypress のほうが便利だなと思って移行したクチなので」

「あ、雪菜さんは Cypress 派なんですね。Cypress は TestCafe よりどこがいいんでしょうか？」

「TestCafe のほうが優れてるところもあるよ。まずマルチブラウザ対応。TestCafe は主要ブラウザの他に Remote モードというのがあって、スマホなどの実機を操ってテストさせることができる」

「えっ、すごいじゃないですか」

「あとは Cypress では iframe の入ったページは扱えないんだけど、そういう制限もない。OAuth のページ組み込みで iframe を使うことがたまにあるけど、そういうアプリのテストには Cypress は使えな

*4 https://www.npmtrends.com/puppeteer-vs-cypress-vs-testcafe

いよね。あとどうしても IE だったりスマホの実機での E2E テストが必須といったケースも TestCafe を採用する理由になるかな」

「けっこう TestCafe が有利に聞こえてきたんですが、Cypress にはそれを上回る良さがあるわけですよね？」

「そうだね、たとえば Redux DevTools のようなタイムトラベルデバッギングができたり、指定したエレメントがハイライトで見れたりとブラウザ上のツールがすごく使いやすいこと。さらに UI 操作とは別に任意の HTTP リクエストが打てるので、あらかじめ API 操作で任意のデータを仕込んでおくことでテストの条件を整えるなんてことができる。さらにテストの実行速度が TestCafe より体感で二倍は速い」

「へ――――」

「あとどちらもテストの書き方にはクセがあるけど、Cypress はテストランナーとアサーションに Mocha と Chai を採用しているのでまだ馴染みのある記法になるし、非同期処理の検証も TestCafe のようにメソッドチェーン記述を強制されることがないというのもある。両者の簡単なテストのサンプルを引用してみよう。まず Cypress から」

```
describe('My First Test', () => {
  it('clicking "type" navigates to a new url', () => {
    cy.visit('https://example.cypress.io');
    cy.contains('type').click();
    cy.url().should('include', '/commands/actions');
  });
});
```

「次に TestCafe はこう」

```
import { Selector } from 'testcafe';

fixture `My First Test`
  .page `http://devexpress.github.io/testcafe/example`;

test('Includes submitted text', async t => {
  await t
    .typeText('#developer-name', 'John Smith')
    .click('#submit-button')
```

第 6 章 プロフェッショナル React の流儀

6-1. Advanced Thinking in React

「秋谷さんの学習の仕上げとして、React アプリケーション設計のプラクティスについて学んでおいてもらいたいのね」

「確かにこれまで色んな技術を教えてもらいましたけど、じゃ今からコード書いてと言われると、何から手をつけていいか戸惑ってしまいそうです……」

「うん。これまで設計については Presentational Component と Container Component を分けて作りましょう、くらいしか説明してこなかったからね。

React の公式ドキュメントには『React の流儀（Thinking in React)』[*1] という記事があるんだけど、そこにはどうすれば React らしく上手にコンポーネント設計ができるかが説明されてる。React のドキュメントもやっと日本語化されたので、必ず一度は読んでおいてほしい内容。サマリーを抜粋するとこんな感じ」

Step 1: UI をコンポーネントの階層構造に落とし込む
Step 2: React で静的なバージョンを作成する
Step 3: UI 状態を表現する必要かつ十分な state を決定する
Step 4: state をどこに配置するべきなのかを明確にする
Step 5: 逆方向のデータフローを追加する

「なるほど。まずコンポーネントの階層構造を決める。次にそれぞれの Props を、さらに Local Staate を決定する。それから必要であれば、コールバック関数による子から親へのデータフローを定義するという順番で作っていくわけですね。これなら綺麗な設計になりそう」

---

*1 https://ja.reactjs.org/docs/thinking-in-react.html

「でもこれは基本中の基本。Redux も非同期通信処理も必要ない場合の話ね。ここまで色んな技術を学んできたわけだけど、それを包括して業務に耐えうる複雑なアプリケーションをどう作るかという、『Thinking React』をさらに推し進めた方法論が必要になってくる」

「確かに Redux が絡んでくると複雑になりそうですよね」

「でも最初のステップはほぼそのまま変わらない。Step 1 から Step 4 までは同様、でも Redux を使う場合は Action を発行してその結果を Sotre State で参照するので、子から親へのデータフローを使う場面がほとんどないから Step 5 は必要ないよね。

Step 4 まで作った状態で、どのコンポーネントを Container Component にするべきかを次に考える。昔はできるだけトップレベルのコンポーネントを Redux に connect する Container にするように公式ドキュメントにも書かれていたけど、最近はもっと柔軟に設計するべきという意見が主流になってるみたいだね。親コンポーネントが子コンポーネントの具体的なデータや発行する Action を知りすぎないよう、またひとつのコンポーネントの Props が5個や6個以上にならないよう調整していくといいと思う。あと、Presentaitona Component が Container Component を、Container Component が Presentational Component を呼ぶのはいいけど、Container が Container を呼ぶのはどこでデータが上書きされるかが複雑に絡み合ってややこしくなるので、できれば避けたほうがいい」

「雪菜さんのコードは確かにそうなってましたね。親 Container が子 Container を呼ぶのは、たぶん私の頭がこんがらがりそうです」

「Container にするべきコンポーネントが決まったら、ページを構成する主要な Presentational Component を Storybook にスタイルガイドとして登録する。これは同時にスナップショットテストが用意されることにもなるよね」

「StoryShots ですね。あれ便利ですよね」

「次に、その Container が発行する Action とその発行に使う Action Creator 関数を作る。そしてその Action に対応する Reducer も併せて作る。

さらにその Action が新しく API 通信処理を必要とする場合、対応する API ハンドラとそのユニットテストを作る」

「Axios Mock Adaptor を使ったモック通信によるテストでしたね」

「そして次はその Action に対応した Saga を作成する。それができたら Redux DevTools から生テキストの Action を Dispatch してみて、その Saga が正しく動作することを確認。その上で Redux Saga Test Plan を用いて Saga と Reducer のユニットテストを書く。

あとがき

「私が書いたものを誰よりも読みたいと思ってるのは私なのに、私が書いてくれないとそれが読めないというジレンマ」とは、本書の執筆に入る前の筆者のツイートです。そしてやってくれました「私」。そう、まさにこの内容の本が読みたかった……。

本書は筆者がフリーランスとしていくつかの現場を経験する中で、チームの開発を助けるために導入したツールの経験やそこで磨かれた考え方をベースに、最新の技術や知見をあらためて調べた上で書き下ろしたものです。選定した技術も過去に導入したものにこだわらず、いま現在の勢いを参考の上で新機能を体験し使い勝手の比較をして、個人的な好みが入りながらもできるだけ公正な目で評価したつもりです。また、Create React App や TypeScript の環境において相性やバグの問題でそのままでは動作しないツールも多く、その解決にも手を焼かされました。しかしそのおかげで、本書のサンプルコードは 2019 年 4 月現在における最新の React ＋ TypeScript 環境における、最強の全部入りボイラープレートになっていると言えます。筆者が苦労して環境整備したノウハウの塊です。

また、内容は当初の予定では第 5 章までで終えるつもりだったのですが、筆が乗ったといいますか、ツールの紹介とその導入だけでは終われないと思い、急遽、第 6 章を追加しました。React 公式ドキュメントの「Thinking in React（React の流儀）」は、あの中でも筆者が最も繰り返し読んで実際に参考にした記事ですが、Redux や非同期通信が絡んだ複雑なアプリケーションで、さらにテストも書く必要のあるチーム開発においては物足りない内容でした。それを自分なりにアレンジし、複数の現場の中で磨いていったのが第 6 章にある手法です。複数メンバーが関わるチーム開発において、ちゃんとしたガイドラインにのっとって秩序を保ちながら開発していきたいと考えている方々に、ぜひ参考にしていただければ幸いです。

なお今回の表紙イラストは、秋谷佳苗さんでした。どうですか？　かわいいでしょう？　クール美人な芝崎雪菜さんとは対照的に、元気なスポーツ少女（？）のイメージだったんですが、イラスト担当の黒木めぐみ様のすばらしい腕により、このような姿に具現化しました。ちょっといたずらっぽい笑顔がなんとも言えません。

そして最終章からエピローグにかけて、筆者はちょっと感無量で泣きかけました。秋谷さん、がん

ばったねえ。そして雪菜さんへの態度が健気だねえ。ふたりが信頼し合って仕事をしている姿が書けて、嬉しかったです。

本書の内容についてのご意見・ご感想、またはご指摘などがあれば Twitter やブログなどで言及していただけると、筆者としてはとても励みになります。また、Twitter ハッシュタグ #りあクト をつけてツイートしていただくと、必ず目を通します。なおイラストについてのご感想もお待ちしております。

それではまた、次の機会に。

著者紹介

大岡由佳（おおおか・ゆか）

フリーランスで現在、React を専門にするフロントエンドエンジニア。過去には PHP や Rails のサーバーサイドエンジニアや、人材系サービスのプロダクトマネージャーなどの経歴を持つ。直近でのお気に入りの漫画は『かげきしょうじょ!!』と『ブルーピリオド』。二年ほど毎日続けているストレッチの成果で最近、360 ° 開脚と I 字バランスができるようになったのが自慢。Twitter アカウントは @oukayuka。

黒木めぐみ（くろき・めぐみ） ◎表紙イラスト

漫画家、イラストレーター。

りあクト! TypeScript で始めるつらくない React 開発 第2版

2019 年 4 月 14 日　第 2 版第 1 刷発行

著者　大岡由佳

印刷・製本　日光企画