インプレスR&D［NextPublishing］

技術の泉 SERIES
E-Book / Print Book

React 環境構築の教科書

井手 優太 著

そのプラグインってどうして必要なの？

Impress R&D
An Impress Group Company

GIJUTSU no IZUMI SERIES
POWERED by NEXTPUBLISHING
技術の泉 SERIES

インプレスR&D［NextPublishing］

技術の泉 SERIES
E-Book / Print Book

React 環境構築の教科書

井手 優太 著

そのプラグインってどうして必要なの？

GIJUTSU no IZUMI SERIES
POWERED by NEXTPUBLISHING
技術の泉

Impress R&D


An Impress
Group Company
SERIES

電子書籍閲覧に関するご注意

　本書では、プログラムリストに専用の等幅フォントを使用しています。ビューアによって以下の作業が必要になります。
・Kindle Paperwhiteの場合：フォント設定画面で「出版者のフォント」を選択
・kobo Androidアプリの場合：フォント画面で「オリジナル」を選択

目次

はじめに

本書は、コピー&ペーストに頼らない環境構築ができるようになるための本です。技術書典応援祭[1]で出版した、「JavaScript環境構築の設定をひとつずつ丁寧に-そのプラグインってどうして必要なの！？-[2]」という同人誌が原案になっています。私自身がフロントエンド環境構築がとても苦手で、克服するために勉強したことを、その同人誌にまとめました。私はそのときの執筆やレビューを通して、**なぜそのツール・プラグインが必要なのか** を明らかにすると、理解が深まることに気付きました。そこで、「プラグインの役割に着目しながら、環境構築をおさらいできるような教材を作れないか」と思い、本書を執筆しています。本書が、環境構築が苦手という方にとって、よき教材や手引きとなればうれしいです。

環境構築ができるようになろう

フロントエンドにおける環境構築は、年々複雑化しています。昔は、スクリプトタグでライブラリを読み込むだけで、開発に必要なものをそろえていました。しかし今では、ビルドや静的検証のパイプラインを構築する、といったこともするようになっています。それに伴い、Package Manager 、 Transpiler 、 Bundler 、 Linter 、 Formatter 、 Test Runner、AltJSなど、たくさんの周辺ツールも生まれました。もちろん、ビルドという観点だけで見ると全部は不要ですが、開発効率を上げるという点では、押さえておきたいツールたちです。それらを押さえた上で、ビルドのパイプラインを組み立てることが要求されることもあり、ただHello Worldするだけの難易度は上がっています。

なぜ環境構築を覚える必要があるのか

環境構築に要求されるツールが増える一方で、便利なプリセットも生まれており、簡単に構築を済ませられるツールも生まれています。たとえばcreate-react-appやNext.jsは、環境構築を代替できる便利なツールです。しかし、手組みで環境構築した経験がなければ、いざ手組みで整備しなければいけなくなった場合に苦労します。もしかしたら、将来的にビルドを別のツールと連携しながら行うことや、何らかの最適化を行うことが必要になってくるかもしれません。そのような事態に備え、環境構築は手組みでゼロからできるようになっておいた方が良いと、筆者は考えています。

環境構築スキルの習得は難しい

　そこで、環境構築スキルの習得を目指していきたいのですが、挫折するポイントが多く潜んでいます。そのため、独学では困難だと、筆者は感じています。

経験値を得づらい

　そもそも、仕事で環境構築をする機会は少ないです。環境構築を行う時期は、プロジェクトが立ち上がるときか、非機能要件の磨き込みに時間を割けるときです。

デバッグのしづらさと成長のしづらさ

　環境構築をデバッグするためには、ある程度が動くようにしてからではないと難しいです。そのため、どこが原因で動かないかという原因の切り分けがしづらい分野です。デバッグの末に、仮にビルドに成功しても、どの修正によって成功したかの判断がつかないこともあります。たとえば、「実は間違ったことをしていたけど、ビルドだけは通るようになった」ということもあります。

対象読者

そこで本書では、環境構築に挑戦したい人に向けて、少しずつ動かしながら「こう設定すればこうなる」を理解できるような説明を心がけました。

そのため本書は、**ビルド設定に困っている、自分でもビルド環境を作れるようになりたい**という方向けに書かれており、**JavaScript の初学者に向けては書かれていません。**なるべく基本的な用語や概念も解説するようにはしましたが、それでも説明不足に感じるところがあるかもしれません。

また、各章は独立しているため、自分の読みたい章から読めます。ただし、環境構築の全体感を想像できない方は、最初から読むことをお勧めします。第9章の「0から環境を作ってみる」は、いわゆる「やってみた」という内容で、実践的なハンズオンです。写経をすれば、全体の雰囲気をつかむことができるでしょう。ソースコードも公開しています。[3]

タイトルではReactを扱うと書いていますが、ここで紹介するツールやエコシステムは、React以外のフレームワークやライブラリを使った開発においても使われているものです。Reactを使わない開発者にとっても、CLIやプリセットに頼らない開発をする場面では、本書が役に立つかもしれません。もし気になる場合は、目次を確認してください。

本書が扱う内容

本書では、フロントエンドアプリケーションのビルドと検証をするための方法を学びます。各章では次のような内容を扱います。

・実行基盤となるNode.js（エコシステムの基盤）
・Babelによるトランスパイル（ビルド）
・TypeScript Compilerによるトランスパイル（ビルド）
・webpackによるバンドル（ビルド）
・ESLintによるリント（検証）
・Prettierによるフォーマット（検証）
・Jestによるテスト（検証）
・Storybookによるコンポーネント管理（検証）

環境構築に登場するツールの役割は、大きくビルドか検証ツールかに分けられ、それぞれに代表的なライブラリが存在します。そのライブラリの数はさほど多くはありませんが、それらに多くのプラグインを挿していくので、結果的には利用するライブラリの数が膨大になります。**ビルド複雑化の原因は、大量に挿すプラグインにあると言っても過言ではありません。**本書では、プラグインを挿される側のライブラリの粒度で、章を分けて整理します。そして、それぞれで利用するプラグインの役割を紹介していくことで、説明を図っています。そのため、章ごとにツールとプラグインの役割を確認しながら読み進めると、役割に関する混乱は緩和できます。

筆者の動作環境とサンプルコードについての注意事項

・筆者のPC環境は、macOS Catalina(Version:10.15.2)です。Node.jsの環境はv12.16.3で、nvm(v0.35.3)で管理しています。
・第9章は、完全な形でのサンプルコード[4]を提供しています。本書で使ったライブラリのバージョンは、ここに書かれているpackage.jsonのものです。
・各章ではコマンドの出力を表記していますが、誌面の都合で適宜編集を加えています。

謝辞

・クレスウェア代表の奥野賢太郎さん(@okunokentaro)、長年TypeScriptを利用されている経験や、過去の執筆経験からアドバイスをいただき、ありがとうございました。レビューを通してお互いに異なる方針を持っていたことがわかり、それを議論できたことも楽しかったです。
・TechBowl CEOの小澤政生さん(@zawamasa)、日本最大級のエンジニアコミュニティTechTrain のメンターやメンティーにレビューを依頼できる仕組みを作ってくださり、ありがとうございました。上級者の方から初級者の方まで、幅広い意見を取り入れることができ、とても助かりました。
・TechTrainメンターの、大木優さん(@gurusu_program、株式会社TechBowl)、今川裕士さん(@ug23_、弁護士ドットコム株式会社)、徳田祥さん(@haze_it_ac)、寺嶋祐稀さん(@y_temp4)、吉野雅耶さん(@ayasamind、株式会社Fusic)、ありがとうございます。日ごろの業務の経験談にもとづいたアドバイスをいただき、ブラッシュアップさせていくことができました。
・TechTrain メンティーの門田朋己さん(@tmk815)、tktcorporation さん(@tktcorporation)、西田吉克さん(@nsd244)、小越雄太さん(@ykotti1)、ありがとうございます。説明や手順の不足を指摘していただいたことで、当初の荒削りな原稿を読みやすくできました。特に小越雄太さんには実際に説明やコードの検証をしていただいたことで、執筆する上での不安を和らげることができました。本当にありがとうございます。

1. https://techbookfest.org/market

2. その本では、Webフロントエンド開発の環境構築手順をひとつずつ丁寧に見ていき、設定が足りない状態で動かすとどうなるのか、といった実験をしていました。たとえばbabel-cliを使わずにトランスパイルしたり、css-loaderを使わずにstyle-loaderだけを使おうとしたりして、どうしてそれらのライブラリが必要なのかを説明しました。https://techbookfest.org/product/6209306726760448
3. https://github.com/sadnessOjisan/js-build-book-support
4. https://github.com/sadnessOjisan/js-build-book-support

第1章　Node.jsを使ったビルド環境整備

1.1 Node.jsとは

Node.js® は、ChromeのV8 JavaScriptエンジンで動作するJavaScript環境です。[1]Node.jsは、サーバやCLIツールとして使われています。クライアントサイドJavaScriptの環境構築は、このNode.jsのエコシステム上で行います。本章では、そのNode.jsについて学びます。

1.2　Node.jsのインストール

Node.jsは、公式HP[2]からダウンロードできます。各プラットフォームごとにインストーラが用意されており、インストールできます。また、公式HPではHomebrewなどのパッケージマネージャーを使ったインストール方法も紹介されています。本書はNode.js v12.16.3を利用しますが、releaseページ[3]からバージョンを細かく指定して、ダウンロードすることもできます。[4]ご自身にあった手法でインストールしてください。

また、公式にのっとった方法ではありませんが、バージョンマネージャーを使っても良いです。本書はNode.js v12.16.3を利用しますが、バージョンマネージャーを使うと、該当バージョンをピンポイントで入れることが容易になります。既にNode.jsを持っている人にとっては、該当バージョンへの切り替えが容易になるメリットもあります。[5]Node.jsのVersion Managerには、さまざまなものがありますが、筆者はnvm[6]を利用しています。本書ではnvmのインストール方法や利用方法は説明しませんが、詳しいことはnvm公式に全て書かれているため、そちらを参照してください。[7]

1.3　JavaScriptプロジェクトを管理する

JavaScriptプロジェクトはpackage.jsonと呼ばれるファイルで、その構成を管理します。このファイルではたとえば、次の項目を管理できます。

・パッケージ名
・バージョン
・依存ライブラリ
・エントリポイント
・script
・ライセンス

依存ライブラリをpackage.jsonファイルで管理できるため、Gitの管理下に含めておくと、依存ライブラリの更新や、その更新に失敗した際の差し戻しが容易になります。またパッケージ名・バージョン・エントリポイントは、プロジェクト自体をライブラリとして提供する際に、利用者が参照する大切な情報です。たとえば、nameはそのままnpm公式サイトでのパッケージの検索に使われます。**開発やエコシステムとの連携という観点で、このファイルはとても大切なものです。**

1.3.1　package.jsonを生成する

package.jsonは、npm initコマンドで作成できます。もちろん、ただのjsonファイルなので、すべて手書きしても問題ありません。

```
$ npm init
```

このコマンドを実行すると、次のようなファイルが作成されます。

リスト1.1: package.json

```
{
  "name": "npmtst",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
}
```

1.3.2　packageを追加する

Node.jsには、npmと呼ばれるpackage managerとコマンドが付随しています。[8]このnpmを使うことで、ライブラリをダウンロードできます。

npmを使ってpackageを追加します。npm install reactなどとすれば、packageを入手できます。

どのようなpackageを保存したか[9]は、package.jsonに記録され、packageの実体は node_modulesというフォルダに保存されます。またそのpackage自体が持つ依存の管理には、package-lock.jsonファイルが使われます。これはlockファイルと呼ばれているものです。**そのためpackage.jsonとpackage-lock.jsonさえあれば、node_modulesを削除しても環境を再現できます。**

# package.json と package-lock.json を元に package を DL して node_modulesに保存する
$ npm install

パッケージを追加するとpackage.jsonが次のようになります。たとえば、reactを追加したときはこのようになります。

リスト1.2: dependencies

```
{
  "dependencies": {
    "react": "^16.13.1"
  }
}
```

これはreactを依存として含めることを意味します。
一方で、このようなpackage.jsonもあります。

リスト1.3: devDependencies

```
{
  "devDependencies": {
    "typescript": "^3.8.3"
  }
}
```

ここでの違いは、dependenciesかdevDependenciesかという違いです。devDependenciesに指定されているpackageはdependenciesと違って、**このプロジェクト自体がライブラリとして使われる際に依存を含めません**。今回の例だと、TypeScriptコンパイラは開発用のツールであり、ライブラリとしての機能には関わってきません。そのため、**dependenciesではなくdevDependencies**に指定します。**このように開発時のみに使うツールかどうかで、dependenciesとdevDependenciesを使い分けます**。全部dependenciesに含めると、ライブラリとして使われるときに利用者に不要なインストールを強いることになるため、気を付けましょう。devDependenciesとしてライブラリをダウンロードするためには、-Dオプションを使います。

```
$ npm install -D typescript
```

　ここまでに出てきた、ライブラリをダウンロードするために利用するnpmコマンドを復習していきます。

```
# アプリで使うライブラリとしてreactをDL
$ npm install react

# 開発用ライブラリとしてTypeScriptをDL
$ npm install -D typescript

# package.jsonとpackage-lock.jsonを元に、そのプロジェクトに必要なライブラリをすべてDL
$ npm install

# installと打たなくても短縮形が使える
$ npm i
```

1.4　ライブラリに付随するコマンドを実行する

　ここまで、ライブラリをダウンロードする方法を学びました。それらは、これからたくさんのビルドツールをダウンロードするための知識です。この節では、ダウンロードしたライブラリをどのようにして呼び出すかを学びます。

1.4.1　bin

　node_modulesにダウンロードしたライブラリの中には、直接コマンドとして使えるものがあります。たとえば、npm install -g @babel/cliとすれば、babelコマンドがどこでも使えます。

```
$ npm install -g @babel/cli

$ babel
babel:
  stdin compilation requires either -f/--filename [filename] or --no-babelrc
```

　どこでも使える理由は、-gオプションを使うことでglobal installしたからです。一方でnpm install -D @babel/cliとした場合は、babelコマンドは使えません。

```
$ npm install -D @babel/cli

$ babel
```

command not found: babel

なぜならこちらはglobal installしていないためです。しかし、npx babelとすれば実行できます。

```
$ npm install -D @babel/cli

$ npx babel
babel:
  stdin compilation requires either -f/--filename [filename] or --no-babelrc
```

これが実行できる仕組みについて、ダウンロードしたライブラリとコマンドの関係について学びましょう。node_modulesの中には.binディレクトリがあり、これは特別な意味を持ちます。

図1.1: node_modules/.binの中身

この.binの中にあるファイルは、package.jsonが管理するプロジェクトの下では、「npx ファイル名」として実行できます。たとえば、この中に

含まれるatobというファイルも、コマンドとして実行できます。ちなみにatobは、Base64形式にエンコードされている文字列をデコードするライブラリです。

```
$ npx atob "SGVsbG8sIFdvcmxkIQ=="
Hello, World!
```

　このようにnpx ファイル名とすることで実行できました。[10]ちなみに、この.binに含まれているatobのコードは、次のようになっていました。

リスト1.4: node_modules/.bin/atob

```
#!/usr/bin/env node
'use strict';

var atob = require('../node-atob');
var str = process.argv[2];
console.log(atob(str));
```

　ここから、node-atobというライブラリにあるatob関数に、コマンドラインから渡された引数を渡して、その結果を出力していることが分かります。
　ファイルが.binに含まれるかどうかは、インストールしたライブラリのpackage.jsonのbinフィールドにあるコマンド名と実行ファイルに依存します。**つまり、ライブラリの提供者はbinフィールドに実行内容を登録することで、コマンドを提供できます。**
　たとえば、先ほどのatobのpackage.jsonはこのようになっています。(※ 誌面の都合上、一部省略しています。)

リスト1.5: package.json

```
{
  "name": "atob",

  ...

  "main": "node-atob.js",
  "bin": {
    "atob": "bin/atob.js"
  },
  "version": "2.1.2"
}
```

これから使うビルドや開発支援ツールは、このようなコマンドをたくさん使います。そのコマンドの実体がどこにあるかを知っていれば、デバッグもしやすくなるので、この.binについて覚えておきましょう。

1.4.2 npm scripts

package.jsonにはscriptsフィールドがあります。

リスト1.6: package.json

```
{
  ...
  "scripts": {
    "decode": "atob"
  },
  ...
}
```

ここには、コマンド名と実行内容を登録できます。そしてここの実行内容として、.binで登録されている関数を指定できます。その際はnpxをつける必要はありません。上記の場合だと、atobの前にnpxは不要です。

そして、その実行はscriptsのキー名を使います。つまり、npm run key名として実行できます。

```
$ npm run decode "SGVsbG8sIFdvcmxkIQ=="
Hello, World!
```

　複数からなる処理をまとめたり、処理にわかりやすい名前をつけるためのテクニックとしても使えるので、覚えておきましょう。

1.5　まとめ

　フロントエンドのビルドや開発エコシステムは、Node.jsを利用しています。一般的には、サーバサイドで使う言語の印象が強いですが、ツールチェインを提供する環境としても大きく活躍しています。Node.jsは公式サイトからダウンロードできるほか、nvmのようなVersion Managerを活用してもインストールできます。ただし、Node.jsは早いサイクルでバージョンが上がっていくため、何らかのVersion Managerを利用することを推奨します。Node.jsにはnpm(Node Package Manager)が付随しており、ライブラリを導入することも容易です。

1. https://nodejs.org/ja/
2. https://nodejs.org/ja/download/
3. releaseページからのダウンロード. https://nodejs.org/ja/download/releases/
4. もしどうしてもv12.16.3が手に入らない場合は別のバージョンを使って動かしてください。動作確認をしていませんがv10以上の環境に対応しています。
5. Node.jsのバージョンは半年に1回ずつ上がり、プロジェクトによって使っているNode.jsのバージョンが異なることもあるため、複数のバージョンを切り替えて開発する仕組みが便利です。
6. https://github.com/nvm-sh/nvm
7. nvmを使ったインストール方法は、Node.js公式で紹介されているものではありません。あえてnvmを選択するのは、筆者の好みによるところがあるため、意図的に説明を省いています。ただし、バージョンマネージャーの利用自体、はNode.jsを使った開発においては有用なため、何かしらのツールを使うことは推奨します。
8. Node.jsのpackage managerにはnpm以外も存在します。たとえばYarnが有名です。Yarn、npm双方ともpackage.jsonを利用して、バージョン管理を行います。そのため、どちらを選択しても、本書で扱う内容には影響しません。ただYarnにしかない機能もあるため、package managerの選択を考える場面もあります。 https://classic.yarnpkg.com/ja/
9. パッケージ名とバージョンが保存されます。
10. npxはローカルにコマンドが存在しなければリモートを見ます。そのためnpm installしていないコマンドを実行でき、とても便利なコマンドです。

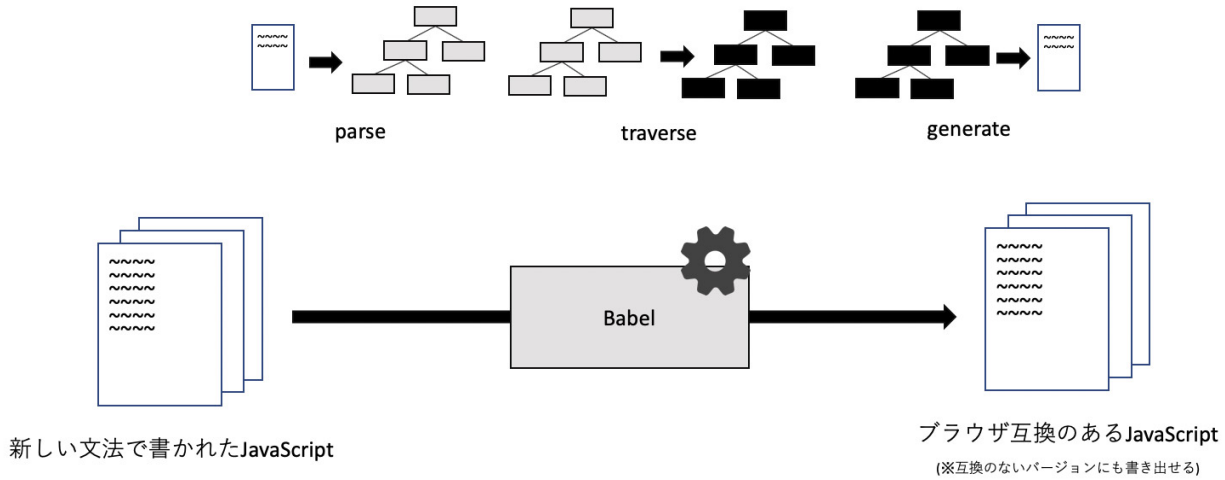
第2章　Babelを使ったトランスパイル

2.1　Babel とは

BabelはJavaScript compilerです。[1]新しい仕様のJavaScriptで書かれたソースコードを、ブラウザがサポートするJavaScriptに変換できるため、Babelを使うと新しいJavaScriptの文法で開発ができます。

Babelはソースコードを AST(abstract syntax tree、抽象構文木)と呼ばれる木構造に変換し[2]、ASTに対する変換操作を行い[3]、変換後のASTからコードを生成する[4]といった変換形式をとっています。そのため、ソースコードの変換はASTに対する変換操作を通して行います。そして、この変換操作は関数として定義でき、プラグイン形式で導入できるほか、自作した関数を利用することも可能です。ここではBabelそれ自体の仕組みについては扱いませんが、**何らかの関数を導入してソースコードを変換できる**ということを意識してください。

図2.1: Babelのイメージ



さっそくBabelの使い方を学びましょう。

2.2　Babelを実行する

　Babelは本体とCLIがライブラリとして分かれているため、インストールするときは注意しましょう。

2.2.1　@babel/coreのインストール

　Babelそのものは@babel/core[5]というライブラリに含まれています。

```
$ npm install -D @babel/core
```

　しかし**これは変換コマンドではない**ため、変換するためには次のようなコードを書く必要があります。

リスト2.1: @babel/coreを利用

```
const babel = require("@babel/core");

babel.transform(
  `
const a = () => {}
`,
  {},
  function (err, result) {
    // codeが変換されたことを確認
    console.log("generated code:", result.code);
  }
);
```

このままだと使い勝手が悪いので、この変換をコマンドラインからも行えるようにしたいです。

2.2.2 @babel/cliのインストール

Babelをコマンドとして使えるようにするものが、@babel/cliです。

```
$ npm install -D @babel/cli
```

これによりbabelコマンドを利用できます。

2.2.3 CLIを使う

では、次のarrow function[6]を変換してみましょう。

リスト2.2: src/main.js

```
const hoge = () => {};
```

ブラウザ互換のある変換を期待するため、arrow functionはただのfunctionへと変換されてほしいです。

```
# src/main.jsに書かれているファイルを変換してdistディレクトリに書き出す。
$ npx babel src/main.js --out-dir dist
```

この変換結果は次の通りです。

リスト2.3: dist/main.js

```
const hoge = () => {};
```

ここではarrowのまま残っています。**なぜなら、どのように変換するか、といった設定をまだ書いていないからです。**つまり、変換を行うためには、どのように変換するかを別途設定する必要があります。

2.2.4 設定ファイルを書く

Babelの設定ファイルはbabel.config.js[7]です。昔は.babelrcというファイル名が使われていましたが、いまはbabel.config.jsの方が使われています。JavaScriptとして読み込むと、コメントを書けたり、PrettierやESLintのチェック対象にも含められる利点があります。最近では、公式もJavaScriptとして読み込むように推奨しているため、設定を書く場合は.js形式で書くと良いでしょう。[8]

設定ファイルは次のように書きます。

リスト2.4: babel.config.js

```
module.exports = {
  plugins: []
};
```

ここにarrow functionを変換する処理を書きます。

2.2.5 pluginを利用する

変換処理を行う関数を作って、それを読み込んでもよいのですが、すでにサードパーティのプラグインがあるため、それを活用します。arrow functionを変換するプラグインは@babel/plugin-transform-arrow-functionsです。[9]

これをインストールします。

```
$ npm install -D @babel/plugin-transform-arrow-functions
```

そして、利用するプラグイン名を設定ファイルで指定します。

リスト2.5: arrowを変換できるプラグインを追加

```
module.exports = {
  plugins: ["@babel/plugin-transform-arrow-functions"]
};
```

そしてbabelコマンドを実行します。

```
$ npx babel src/main.js --out-dir dist
```

distの中を確認すると、このようにarrow functionは変換されました。

リスト2.6: dist/main.js

```
const hoge = function() {};
```

ここで利用したpluginという仕組みがどのようなものか、見ていきましょう。

2.3　pluginとは

　Babelそれ自体はconst babel = code => codeとして動作する、薄い仕組みです。その中で受け取ったコードを自由に変換する機構が、pluginです。

　Babel Pluginには、Transform PluginsとSyntax Pluginsがあります。先ほどの@babel/plugin-transform-arrow-functionsはTransform Pluginです。その実体は、ASTに対して何らかの処理をする関数です。

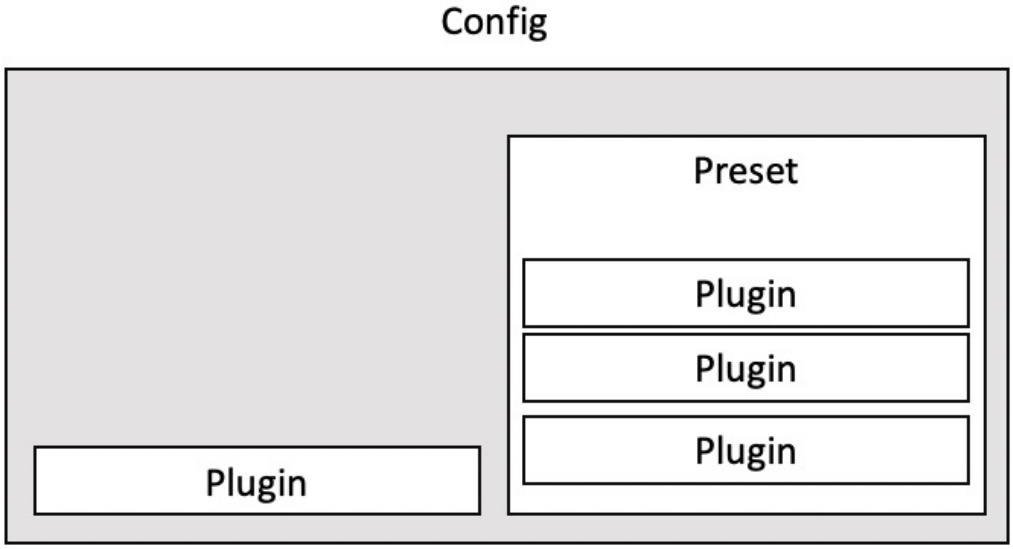
2.4　複数のpluginをpresetで管理する

　先ほどの例では、@babel/plugin-transform-arrow-functionsプラグインを入れることで変換をしました。最新のJavaScriptに追従すべく、機能ごとにプラグインを入れることは骨が折れます。そこで複数のpluginをセットにした、presetを利用して変換処理を行うことが一般的です。

2.4.1　presetとは

　preset[10]は、複数のpluginをセットにしたものです。presetを使うことで、複数のpluginを簡単に導入できます。
　たとえば、次のようなpresetが用意されています。
　・@babel/preset-env
　・@babel/preset-flow
　・@babel/preset-react
　・@babel/preset-typescript
　presetもpluginと同じく配列で管理できるため、複数のpresetを組み合わせられます。これにより、たとえばTypeScriptで書かれたReactアプリケーションのビルドなどもできます。

図2.2: Presetのイメージ



2.4.2　ES5に変換するpreset

ここでは、設定を書くことで必要なpluginを判別し、指定したターゲットに向けたJavaScriptを生成してくれる@babel/preset-envを紹介します。[11]

@babel/preset-envとは

@babel/preset-env は 、 browserslist 、 compat-table[12] 、 electron-to-chromium[13]を参照し、各環境で動くJavaScriptへと変換してくれます。

browserslist を活用する

browserslist は Share target browsers between different front-end tools, like Autoprefixer, Stylelint and babel-preset-envとあるように、**さまざまなフロントエンドツールに対して「私たちはこのようなブラウザをサポートするので、それに合わせて設定をしてください」**

と伝えるためのツールです。[14]Babel以外だと、Autoprefixer[15]などで使われています。通常は、package.jsonや.browserslistrcに書きます。

リスト2.7: browserslistをpackage.jsonに書く

```
  "browserslist": [
    "defaults",
    "not IE 11",
    "not IE_Mob 11",
    "maintained node versions",
   ]
```

もし、babel.config.jsの中でbrowserslistを指定したり、ignoreBrowserslistConfigというフラグを立てなければ、package.jsonや.browserslistrcが参照されます。

browserの指定に関して、browserslistの公式はpackage.jsonに明示することを推奨し[16]、Babelの公式は.browserslistrcファイルに書くことを推奨しています。ここでは.browserslistrcファイルに書きます。

リスト2.8: .browserslistrc

```
> 0.25%
not dead
```

@babel/preset-envはこの.browserslistrcを見て、どの構文にすべきかを判断して変換します。そのため、@babel/preset-envでは、この.browserslistrcが鍵となってきます。

@babel/preset-envの設定を試す

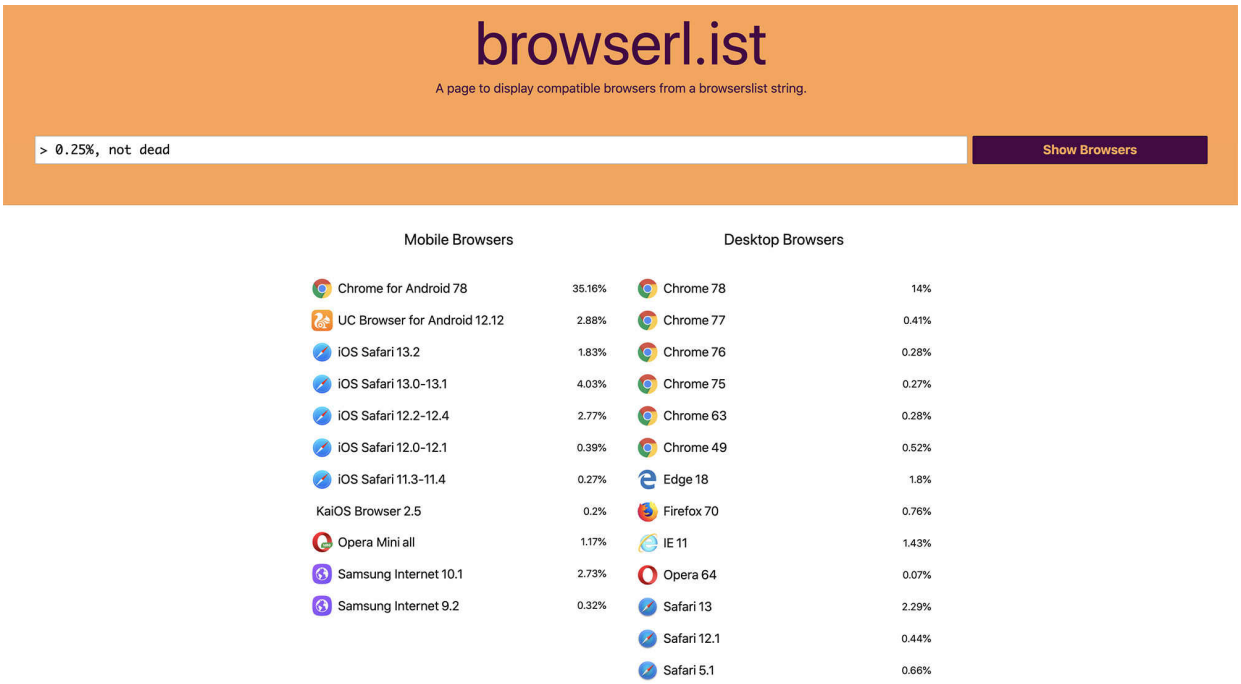
@babel/preset-envにはどのような設定が必要かを、実験してみましょう。.browserslistrcに設定を書きます。例として、使用率が0.25%より上のブラウザを指定します。

リスト2.9: .browserslistrcを使って変換

```
# Browsers that we support

> 0.25%
not dead
```

ちなみに、この設定がサポートしている対象ブラウザは次の通りです。[17]

図2.3: サポートブラウザ



先ほどの設定で、次のようなES6のコードがどう変形されたかを見ます。

リスト2.10: classファイルがどう変換されるか

```
class A {
  constructor() {}
```

```
}
```

変換します。

```
$ npx babel src/main.js --out-dir dist
```

実行した結果がこちらです。

リスト2.11: classファイルを変換した結果

```
"use strict";

function _classCallCheck(instance, Constructor) {
  if (!(instance instanceof Constructor)) {
        throw  new  TypeError("Cannot  call  a  class  as  a
function");
  }
}

var A = function A() {
  _classCallCheck(this, A);
};
```

classはES5では関数で実現されますが、それをnewを使わずに関数として呼び出していないかを確認しています。classをサポートしていないブラウザに向けたコードなので、このような確認をしています。では、class構文がサポートされているGoogle Chromeの最新バージョンに向けてトランスパイルすると、どうなるでしょうか。

リスト2.12: .browserslistrcをchrome向けに変換

```
# Browsers that we support

```

```
  chrome 80
```

$ npx babel src/main.js --out-dir dist

リスト2.13: chrome80用に書き出されたJS

```
"use strict";

class A {
  constructor() {}
}
```

今度は、class構文がそのまま使われています。これは、Chrome80はclass構文をサポートしているからです。このように、サポートブラウザに合わせてトランスパイルされることが確認できました。

2.5　ブラウザの差異をpolyfillで管理する

2.5.1　polyfillとpreset-env

ところで、@babel/preset-envの公式が出している設定ファイルを見てみましょう。

リスト2.14: babel.config.jsの設定

```
{
  "presets": [
    [
      "@babel/preset-env",
      {
        "useBuiltIns": "entry"
      }
    ]
  ]
}
```

公式にある設定では、presetsが二重配列になっています。これは@babel/preset-envにオプションを指定するために、このような書き方になっています。そのオプションとは、polyfillの指定です。

polyfillとは

polyfillとは、最近の機能をサポートしていない古いブラウザーでもその機能を使えるようにするためのコードです。[18]たとえば、Internet Explorer 7でHTML Canvas要素の機能を疑似的に実現したり、CSSのrem [19]単位やtext-shadowなどを疑似的に実現できます。

polyfillを使うためには、必要なpolyfillをダウンロードして差し込むほか、Polyfill.ioのようなサービスを使います。[20]

このpolyfillは、ソースコードを変換する役割を持つBabelにおいても、ブラウザの互換性を維持する上で関係してくる仕組みです。Babelには自動でpolyfillを入れてくれる仕組みが用意されているため、その設定について見ていきましょう。

BabelとPolyfill

@babel/preset-envには、useBuiltIns: "usage"というオプションがあり、これにより必要なpolyfillを自動で入れられます。実際にコードを変換して確認してみましょう。

リスト2.15: babel.config.js

```
module.exports = {
  presets: [
    [
      "@babel/preset-env",
      {
        useBuiltIns: "usage"
      }
    ]
  ]
};
```

この設定で、次のようなasyncが混じったコードをビルドしてみましょう。async/awaitはES2017の機能です。

リスト2.16: asyncを利用したコード

```
const hoge = async () => {
  console.log("hallo world");
};

hoge();
```

ビルドすると次のような警告が出ます。

```
$ npx babel src/main.js --out-dir dist

WARNING: We noticed you're using the `useBuiltIns` option wi
thout declaring a core-
js version. Currently, we assume version 2.x when no version
is passed. Since this default version will likely change in future
 versions of Babel, we recommend explicitly setting the core-
js version you ar
e using via the `corejs` option.

You should also be sure that the version you pass to the `corej
s` option matches the version specified in your `package.json`'
s `dependencies` s
ection. If it doesn't, you need to run one of the following com
mands:

  npm install --save core-js@2    npm install --save core-js@3
  npm install core-js@2           npm install core-js@3

Successfully compiled 1 file with Babel.
```

　これはBabelが自動でpolyfillを挿入するので、そのためのpolyfillライブラリを各自でダウンロードするように言われています。**Babelはあくまでもpolyfillを読み込む関数を差し込むだけで、polyfillは自分で入れる必要があります。**
　試しに、このビルドしたコードを実行してみましょう。

```
$ node dist/main.js

internal/modules/cjs/loader.js:796
```

```
    throw err;
    ^

Error: Cannot find module 'core-js/modules/es6.promise'
Require stack:
- dist/main.js
    at Function.Module.runMain (internal/modules/cjs/loader.js:
1043:10) {
  code: 'MODULE_NOT_FOUND',
  requireStack: [
    'dist/main.js'
  ]
}
```

「core-jsが見つかりませんでした」とエラーが出ました。では、このビルドされたコードを見てみましょう。

リスト2.17: buildされたファイル

```
"use strict";

require("core-js/modules/es6.promise");
require("core-js/modules/es6.object.to-string");
require("regenerator-runtime/runtime");

省略

var hoge =
  /*#__PURE__*/
  (function() {
    省略
  })();

console.log(hoge(hey));
```

どうやら、core-jsとregenerator-runtimeがあれば動きそうです。追加しましょう。

```
$ npm install core-js@3 regenerator-runtime
```

このとき、-Dは不要なことに注意してください。polyfillはアプリケーションコードで動くものですので、開発用ライブラリではありません。

そして、babel.config.jsで利用するpolyfillのバージョンを指定してください。**指定がない場合は、version2が使われます**が、ここでは**3**を使うので、指定をします。

リスト2.18: core-jsを指定

```
module.exports = {
  presets: [
    [
      "@babel/preset-env",
      {
        useBuiltIns: "usage",
        corejs: 3
      }
    ]
  ]
};
```

ではビルドして実行します。

```
$ npx babel src/main.js --out-dir dist
Successfully compiled 1 file with Babel.
  Done in 0.59s.
```

```
$ node dist/main.js
hello world
```

このように、エラーが出ることなく動作しました。[21]

2.6　まとめ

　本章では、Babelを使ったトランスパイルについて学びました。Babelはブラウザでの互換性を保ったソースコードを出力できるため、利用すると新しい文法のJavaScriptを使えるようになる利点があります。ソースコードの変換方法は、pluginを使って拡張していけます。pluginをセットにしたpresetと呼ばれる機構で、まとめて拡張することも可能です。

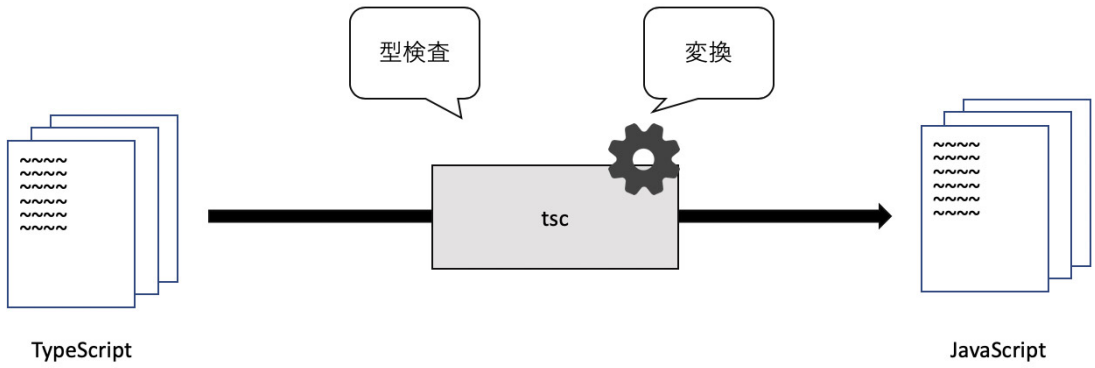
1. https://babeljs.io/
2. parse
3. traverse
4. generate
5. https://github.com/babel/babel/tree/master/packages/babel-core
6. arrow functionはES2015の新構文で、Babelによる変換対象としてよく使われていました。
7. **.babel.config.js**ではなく**babel.config.js**であることに注意。昔はよく.babelrcというファイルが使われていたので、書き間違いをしやすいです。
8. https://babeljs.io/docs/en/configuration
9. https://babeljs.io/docs/en/babel-plugin-transform-arrow-functions
10. https://babeljs.io/docs/en/presets
11. https://babeljs.io/docs/en/babel-preset-env
12. ES5 以上の文法とブラウザの対応を一覧できるテーブル. https://kangax.github.io/compat-table
13. Electron と chromium のバージョンの組み合わせを調べられるツール。https://www.npmjs.com/package/electron-to-chromium
14. https://github.com/browserslist/browserslist
15. ベンダープレフィックスを付与するツール。https://github.com/postcss/autoprefixer
16. https://github.com/browserslist/browserslist
17. browserl.istというサイトで調べられる。https://browserl.ist/
18. https://developer.mozilla.org/ja/docs/Glossary/Polyfill
19. ルート要素のfont-size.レスポンシブデザインやアクセシビリティを考慮した時に、文字サイズの切り替えをしやすいといった利点がある。
20. https://polyfill.io/v3/
21. このような実験を行いました。 https://github.com/ojisan-toybox/is_need_regenerator-runtime

第3章　TypeScriptを使ったコンパイル

3.1　TypeScriptとは

TypeScript[1]は、JavaScriptのスーパーセット言語です。JavaScriptに型を付けることができ、型検査を行うことで、実行時に起きうる不具合を事前に見つけ出せます。また、TypeScript自体が最新のJavaScriptの機能を取り入れており、変換ターゲットを柔軟に設定できるため、Transpilerという側面もあります。[2]利用することによる制限はほとんどないため、これからWebアプリケーション開発をする際には、利用を推奨します。

図3.1: TypeScriptのイメージ



このとき、外部から取り入れたライブラリを含めて型検査を行うためには、型定義ファイルと呼ばれるファイルが必要になります。[3]TypeScriptで開発されたライブラリであれば、ライブラリに付随しているはずなので[4]、それを使います。もし配布されていない場合は自分で作るほか、DefinitelyTyped[5]からインストールすることで利用可能です。

DefinitelyTypedに登録されている型定義ファイルは、npm install -D @types/hoge といったふうに、npm経由でインストールできます。

3.2　tscのインストール

　TypeScriptからJavaScriptへの変換は、tscコマンドを使います。このコマンドによって、TypeScriptからJavaScriptに変換できます。これはTypeScriptインストール時に、CLIとして付いてきます。

```
$ npm install -D typescript

$ npx tsc
```

　ビルド対象や結果の出力先は、設定ファイルで指定できます。
　設定はtsconfig.jsonに書きます。tsc --initを実行すると設定ファイルを生成できるため、初回設定では活用しましょう。（※誌面の都合で、コメントアウトされている行の一部とコメントを削除しています。）

リスト3.1: tsconfig.json

```
{
  "compilerOptions": {
    /* Basic Options */
    // "incremental": true,
    "target": "es5",
    "module": "commonjs",
    // "lib": [],
    // "allowJs": true,
    // "checkJs": true,
    // "jsx": "preserve",
    // "outFile": "./",
    // "outDir": "./",
    // "rootDir": "./",
```

```
      // "composite": true,
      // "removeComments": true,
      // "noEmit": true,
      // "downlevelIteration": true,
      // "isolatedModules": true,

      /* Strict Type-Checking Options */
      "strict": true,
      // "noImplicitAny": true,
      // "strictNullChecks": true,
      // "strictFunctionTypes": true,
      // "strictBindCallApply": true,
      // "strictPropertyInitialization": true,
      // "noImplicitThis": true,
      // "alwaysStrict": true,

      /* Additional Checks */
      // "noUnusedLocals": true,
      // "noUnusedParameters": true,
      // "noImplicitReturns": true,
      // "noFallthroughCasesInSwitch": true,

      /* Module Resolution Options */
      // "moduleResolution": "node",
      // "baseUrl": "./",
      // "paths": {},
      // "rootDirs": [],
      // "typeRoots": [],
      // "types": [],
      "esModuleInterop": true,

      /* Advanced Options */
      "forceConsistentCasingInFileNames": true
    }
  }
```

　このファイルのうち、よく触ることになるであろう項目について確認します。

3.2.1 target

どのバージョンにトランスパイルするかを選択できます。候補は次の通りです。

・ES3 (default)
・ES5
・ES6/ES2015
・ES2016
・ES2017
・ES2018
・ES2019
・ES2020
・ESNext

Babelを利用していたときは、preset-envと.browserslistrcを使うことでブラウザやシェアを細かく指定できましたが、このオプションではそれができません。

どのバージョンに変換したらよいかは、compat-table[6]を参考にすると良いです。これまではES5への変換がデフォルトとして考えられていましたが、（Internet Explorerへの対応が不要な場合に限って）最近はそれ以上のバージョンに変換しても、ブラウザで動かせます。

3.2.2 module

どのモジュールパターンで出力するかを決められます。

・commonjs: Node.jsで利用される。require('name')形式で読み込める。
・amd: ブラウザで利用される。モジュールを非同期でロードする仕組みが提供される。
・umd: AMDとCommonJSの両方をサポート
・ES6: ES Module

出力したコードに対するimportやrequireに影響し、主にライブラリを作るときに意識するオプションです。

3.2.3　lib

　コンパイルに含める組込みライブラリを指定できます。どのlibが必要になるかは、targetや入れたライブラリや型定義ファイルに依存します。たとえば、ES6の組込み関数（たとえばSet、Map）を使う際には、libにES6を指定する必要があります。[7]

　libを指定しない場合は、targetに従って自動で設定されます。ES5をtargetにしているときは、DOM/ES5/ScriptHost、ES6をtargetにしているときはDOM/ES6/DOM.Iterable/ScriptHostが入ります。ただし、もしどれか1つでもlibを手動で設定していると、自動で設定されなくなります。1つでもlibを設定したら、必要なlibは全部設定しましょう。

3.2.4　strict

　型検査の振る舞いを制御できるオプションです。trueにすると検査が厳しくなります。tscのオプションでは、noImplicitAny、strictNullChecks、noImplicitThis、alwaysStrictという項目があり、strictをtrueにすると、これら4つすべてを検査します。このオプションをtrueにしておくと、型推論がより信頼できます。既存プロジェクトにTypeScriptを導入していくときにこのオプションをつけると、警告がたくさん出てたいへんな目に遭いますが、新規開発の場合はぜひtrueにしておきましょう。

3.2.5　noEmit

　このオプションは、コンパイラでの型検査だけを行います。自分の書いたコードの型が合っているか確かめたり、CI上での静的検証として使えます。

// 作業中のdirectoryがproject rootとして、-pでそのプロジェクト位置を指定している

```
$ npx tsc -p . --noEmit
```

3.2.6　forceConsistentCasingInFileNames

ファイルの大文字・小文字の違いをエラー報告するオプションです。たとえば、実際にあるファイル名はHelper.tsなのに、import{calc}from'./helper'などと書くとエラーを発生させるオプションです。なぜこれが必要であるかというと、macOSにて採用されるファイルシステムでは、大文字・小文字を区別しなくても解決できるからです。そのため、開発中のMacではテストもビルドも通るのに、CIサーバでは失敗し、その原因がよく分からないといった事態を防げます。この値はdefaultでtrueです。

3.2.7　jsx

--jsxオプションはその名の通り、tsx(TypeScriptがサポートするJSX)形式を変換するためのオプションです。取りうる値はpreserve、react、react-nativeです。ここでreactを指定するとjsを出力し、preserveを指定するとjsxを出力します。[8]reactを指定してjsを出力すると変換が一度で済みますが、preserveを利用すると、jsxをjsに変換する必要が生まれます。ただしデメリットではなく、jsxからjsの変換をBabelに任せて多段構成にするなど、柔軟なビルドが可能になります。

3.2.8　outDir

ビルドしたファイルを置く場所を指定できます。

3.2.9　outFile

ビルドしたファイル名を指定できます。

3.3　Reactを動かす

　では、設定のイメージをつかむために、ReactをTypeScriptで動かす設定をしてみましょう。ただしここでは誌面の都合上、設定ファイルに書いていたことをCLIのオプションで渡して設定します。tsconfig.jsonの設定内容は、tscコマンドの引数でも同様の表現ができます。

　検証用の環境構築を楽に済ませるため、ReactやReactDOMはCDN経由で読み込みます。[9]めったにやらない方法ですが、公式にも説明はあります。Reactをサクッと試したいときなどには使えるので、覚えておくと良いでしょう。

リスト3.2: index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>only tsc react app</title>
  </head>
  <body>
    <script
      crossorigin

src="https://unpkg.com/react@16/umd/react.development.js"
    ></script>
    <script
      crossorigin
              src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js"
    ></script>
    <div id="root"></div>
    <script src="./main.js"></script>
  </body>
</html>
```

これにより、クライアント側でReact、ReactDOMが見えるようになりました。ただしtscの実行に必要になるため、型定義ファイルだけはインストールしておきます。

```
npm install -D @types/react @types/react-dom
```

次に、このようなコンポーネントを表示させてみましょう。

リスト3.3: main.tsx

```
interface Props {
  name: string;
}

const El: React.FC<Props> = props => <h1>Hello, {props.name}</h1>;

ReactDOM.render(<El name="taro"></El>, document.getElementById("root"));
```

このとき、ビルドに必要なオプションは次の通りです。

```
$ npx tsc src/main.tsx --outDir dist --jsx react --lib es2015,dom
```

これらのオプションについて、そのオプションがないと何が起きるのかを見ていきましょう。たとえば、--jsx reactがないと次のエラーが出ます。

```
$ npx tsc src/main.tsx --outDir dist --lib es2015,dom
warning package.json: No license field
$ js-build-book-code/node_modules/.bin/tsc src/main.tsx --outDir dist --lib es2015,dom
src/main.tsx:5:39 - error TS17004: Cannot use JSX unless the '--jsx' flag is provided.

5 const El: React.FC<IProps> = props => <h1>Hello, {props.name}</h1>;
                                        ~~~~

src/main.tsx:7:17 - error TS17004: Cannot use JSX unless the '--jsx' flag is provided.

7 ReactDOM.render(<El name="taro"></El>, document.getElementById("root"));
                  ~~~~~~~~~~~~~~~~~~

Found 2 errors.
```

--jsxオプションはその名の通り、jsxを扱うときに使うオプションで、ここではjsファイルが出力されるreactを選びます。
また、--lib es2015がないと次のエラーが出ます。

```
$ npx tsc main.ts --outFile bundle.js --outDir dist --jsx react
$ need-babel__template-when-ts/node_modules/.bin/tsc main.ts --outFile bundle.js
node_modules/@types/react/index.d.ts:388:23 - error TS2583: Cannot find name 'Set'. Do you need to change your target li
```

brary? Try changing the `lib` compiler option to es2015 or later
.

388 interactions: Set<SchedulerInteraction>,

これは、@types/reactにSetが含まれており、それを解決できなかったことによるエラーです。そのためSetを使っていなかったとしても、--lib es2015という指定が必要です。--lib dom の指定も同じような理由です。

これらを設定ファイルに落とし込むと、次の通りになります。

リスト3.4: tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "jsx": "react",
    "lib": ["es2015", "DOM"]
  }
}
```

この設定ファイルでビルドすると、無事Reactアプリケーションが表示されます。

Babelでの設定

変換の責務はBabelが担うべきという思想であれば、BabelでTypeScriptを変換したくなるでしょう。Babelで変換する場合は、@babel/plugin-transform-typescriptを利用します。BabelでTypeScriptを変換するメリットもあり、たとえばTypeScriptにまだ含まれていない機能を使いたいときは、Babelで変換した方が良いです。また、Polyfillの追加や、ビルドターゲットの指定も柔軟にできます。ちなみにTS->ESをtscで、ES->ES5への変換をBabelで行うといった、多段変換も可能です。

3.4　まとめ

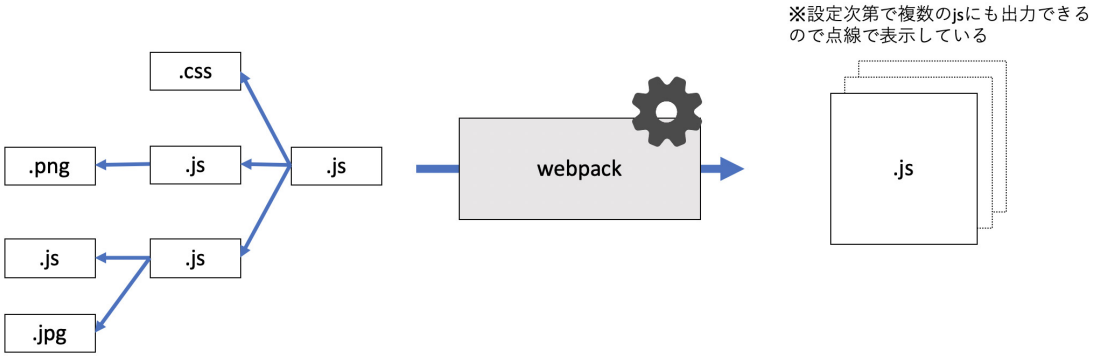
　TypeScriptはJavaScriptのスーパーセット言語です。型が提供され、フロントエンドプロジェクトにおける静的検証の幅を広げてくれます。TypeScript Compilerが提供されており、これによりJavaScriptに変換できます。基本的には、TypeScriptを採用するデメリットが小さくなってきているので、採用することを推奨します。ただし設定項目はとても多く、特殊な設定が必要となった場合は、詰まることもあるでしょう。そこで、公式のCompiler Optionの説明を理解しておくなど、ある程度の訓練は必要です。そのぶん恩恵も大きいので、頑張ってキャッチアップすることを推奨します。[10]

1. https://www.typescriptlang.org/
2. そのためBabelと同じように、レガシーブラウザに向けて変換する役割を担えます。
3. 型定義ファイルを用意できなくても、tscの設定を緩めることでコンパイルエラーを防ぐことはできます。
4. ここで「はず」と書いたのは、型定義ファイルを出力せずに配布することも可能だからです。ただしそのようなケースに、筆者は出会ったことはありません。
5. https://github.com/DefinitelyTyped/DefinitelyTyped
6. http://kangax.github.io/compat-table/es2016plus/
7. アプリケーションに含まれなくても、利用ライブラリに含まれているなら、この指定は必要です。
8. react-nativeを指定すると、ビルド結果はpreserveと同じで、拡張子が.jsとなります。つまり.js形式にjsxが出力されます。
9. CDNを使わない一般的な方法は、次のwebpackの章で説明します。webpackはモジュールの依存解決を行えるツールです。
10. 公式のドキュメントがとても充実しています。https://www.typescriptlang.org/docs/home.html

第4章　webpackを使ったバンドルとビルド

webpack[1]はmodule bundlerです。Babelがコンパイラとすれば、webpackはリンカ[2]です。複数のJavaScriptや静的アセット[3]の依存解決を行います。webpackは静的アセットの依存解決もできるため、JavaScriptの世界にCSSやバイナリをもimportし、それらをJavaScriptから利用可能にします。

図4.1: webpack のイメージ



このwebpackによってソースコード内にあるexport/importを解決でき、ライブラリのimportが容易になります。反対にwebpackがなければ、ライブラリを利用するための手間が増えます。なぜなら依存解決ができなければ、importしたいライブラリを実行ファイルに含められないからです。もし依存解決を行うツールがなければ、依存の順番を意識しなが

らscriptタグで順番に読み込むといったことをする必要があります。webpackはそういった依存解決の難しさを解決してくれるツールです。

webpackは一見するとモジュールを解決しているだけなので、役割や使い方は単純そうにも見えますが、その実態は複雑なものです。なぜならwebpackは依存解決をする目的以外の役割も担え、多様な使い方が可能だからです。たとえば、webpackにはloaderやpluginという仕組みがあり、次のような使い方ができます。

・Babelを呼び出せる
・tscを呼び出せる
・開発用サーバを立てられる
・HTMLにJavaScriptを埋め込める
・etc...

どういうことができるのかは、具体例で見ていきましょう。

4.1　webpackの使い方

4.1.1　webpackのinstall

webpackをインストールします。

```
$ npm install -D webpack
```

webpackもBabelのようにCLIツール[4]が本体から切り離されているので、別途ダウンロードします。

```
$ npm install -D webpack-cli
```

これでwebpackコマンドが使えるようになりました。[5]webpackはモジュールバンドラーであるため、依存を持つJavaScriptファイルのentry pointを受け取り、その依存を解決したファイルを成果物としてoutputできます。webpackは多くの場合で、そのようにして使われます。それでは、依存解決がどう行われるのか見てみましょう。これらのファイルを、webpack経由でbundleしてみましょう。

リスト4.1: src/index.js

```
import { hello } from "./sub";

hello();
```

リスト4.2: src/sub.js

```
export const hello = () => {
  console.log("hello");
};
```

バンドルします。

```
$ npx webpack
  Asset       Size  Chunks             Chunk Names
index.js  972 bytes       0  [emitted]  main
Entrypoint main = index.js
[0] ./src/index.js + 1 modules 97 bytes {0} [built]
    | ./src/index.js 41 bytes [built]
    | ./src/sub.js 56 bytes [built]
```

ここでindex.jsをentry pointとして読み込んだ際、sub.jsのconsole.log("hello");が生成物に含まれていると、依存解決が成功したことになります。書き出されたファイルは次の通りです(見やすくするために整形しています)。

リスト4.3: dist/main.js

```
!(function(e) {
  var t = {};

  省略,

  function(e, t, r) {
    "use strict";
    r.r(t);
    console.log("hello");
  })
```

ちゃんとconsole.log("hello");が組み込まれていることが確認できました。つまり、依存解決に成功しています。

webpackはv4からzero configでの実行も可能で、設定ファイルを書かなければビルドターゲットがsrc/index.jsで出力先をdist/main.jsとし、諸々の最適化を加えて出力します。そのため、設定ファイルを指定せずともビルドできました。もちろん、設定ファイルを活用した方が、差分管理や便利な機能を導入する上で好ましいため、設定ファイルについて次の節で見ていきます。

4.2　webpackの設定ファイル

wepbackは、設定ファイルとしてwebpack.config.jsを見ます。

リスト4.4: webpack.config.js

```
const HtmlWebpackPlugin = require("html-webpack-plugin");
const Dotenv = require("dotenv-webpack");
const path = require("path");

module.exports = {
  mode: process.env.NODE_ENV,
  entry: "./src/main.tsx",
  output: {
    path: path.resolve(__dirname, "./dist"),
    filename: "build.js",
  },
  module: {
    rules: [
      {
        test: /\.(ts|tsx)$/,
        use: [
          {
            loader: "ts-loader",
          },
        ],
      },
    ],
  },
  resolve: {
    extensions: [".js", ".ts", ".tsx", ".css"],
  },
  plugins: [
    new HtmlWebpackPlugin(),
    new Dotenv(),
```

```
    ],
};
```

その設定ファイル内で指定できる設定は、次の通りです。

4.2.1 mode

ビルド時の最適化について設定できます。取りうる値は、none、development、productionです。指定しなかった場合のデフォルト値は、productionです。productionは最適化をしてくれ、noneは何も最適化を行いません。developmentを設定すると、Source Map[6]を有効にできます。

4.2.2 entry

entryではentry pointを指定できます。SPAとしてサイトを開発している場合は、基本的に1つしか指定しません。ですが、もしMPA[7]を開発している際は、複数のエントリポイントを指定し、それぞれにoutputを作ることができます。

リスト4.5: entry

```
entry: {
  home: './home.js',
  about: './about.js',
  contact: './contact.js'
}
```

4.2.3 output

outputでは、ビルド後のファイルをどう保存するかを指定できます。

リスト4.6: output

```
output: {
  path: path.resolve(__dirname, "./dist"),
  filename: "build.js",
},
```

この例では、distディレクトリにbuild.jsという名前で保存します。

4.2.4 loader

loaderはwebpackのpre-process機構です。つまり、bundle前にソースコードへ何らかの前処理を行える機能です。よくある使われ方は、bundle前にBabelやtscでコンパイルする使い方です。このような使い方がよくされるので、**webpackがビルドの処理を背負っているようにも見え、Babelとwebpackの違いについて混乱しやすい**ポイントでもあります。

ts-loader

tscでの変換ができるloaderです。TypeScriptプロジェクトをwebpackでビルドする際に必要になります。

loaderをinstall後、*.tsと*.tsxにloaderを利用する設定を付け加えることで、利用できます。

```
$ npm install -D ts-loader
```

リスト4.7: webpackにts-loaderを設定

```
module: {
    rules: [
      {
        test: /\.(ts|tsx)$/,
```

```
          use: [
            {
              loader: "ts-loader"
            }
          ]
        }
      ]
    },
```

ちなみに、このとき内部ではTypeScriptが使われるので、tsconfig.jsonが必要です。仮にtsconfig.jsonがない場合は、このようなエラーが出ます。

```
$ npx webpack

ERROR in [tsl] ERROR
      TS18002: The 'files' list in config file 'tsconfig.json' is empty
.
```

---

tsconfig.jsonを追加して実行しましょう。

```
$ npx webpack

Built at: 02/24/2020 6:58:47 PM
    Asset      Size  Chunks             Chunk Names
bundle.js  1.09 KiB       0  [emitted]  main
Entrypoint main = bundle.js
[0] ./src/main.ts 305 bytes {0} [built]
 Done in 3.09s.
```

---

ts-loaderにはconfigFileというオプションがあり、これを使うことでTypeScriptのビルドに使う設定ファイルを切り替えることができます。ビルドファイルを複数使い分けたい場合に使います。

babel-loader

babel-loaderは、webpack実行時にBabelを実行できるようにするloaderです。つまり、ES6で書いたコードをwebpackでビルドできるようにするloaderです。

リスト4.8: babel-loaderを利用

```
module: {
  rules: [
    {
      test: /\.js$/,
      exclude: /(node_modules|bower_components)/,
      use: {
        loader: "babel-loader",
        options: {
          presets: ["@babel/preset-env"]
        }
      }
    }
  ];
}
```

この例では、.jsにbabel-loaderを適用させています。また、optionsの設定にbabelの設定を書いていますが、もちろん別ファイルに切り出すこともできます。

file-loader

file-loader[8]はファイルのimportを解決し、ビルド時に出力先フォルダへ該当ファイルを出力します。このloaderを利用することで、JavaScript内でimportしたファイルを、ビルド後のファイルからの参照が

可能です。画像や動画といったassetを読み込む際に利用します。file-loaderの設定には、読み込みたいファイルの拡張子を指定してください。

リスト4.9: file-loaderを利用

```
  module: {
      rules: [
        {
          test: /\.(png|jpg)$/,
          use: [
            {
              loader: "file-loader"
            }
          ]
        }
      ]
    },
```

raw-loader

raw-loader[9]は、ファイルの内容をそのままテキストとして読み込むloaderです。CSSやGLSLを使いたいときに使えます。

style-loader

style-loaderは、JavaScriptファイルに埋め込まれたCSSの情報を、htmlのstyleタグに加えることができます。これにより、raw-loaderやcss-loaderでimportしたCSSを反映させることができます。

リスト4.10: style-loader

```
  module: {
      rules: [
        {
          test: /\.css$/,
          use: ["style-loader", "raw-loader"]
```

```
        }
      ]
    },
```

ここでは、CSSファイルを読み込むためにraw-loaderを使っています。

css-loader

css-loaderはraw-loaderと同じく、ファイルを読み込めるようにするプラグインです。[10]raw-loadrとの違いはCSSへのサポートの有無であり、公式のオプションを見ると、さまざまな機能があることを確認できます。

たとえば、css-loaderのoptionでmodulesをtrueにすると、CSS Moduleを利用できます。

リスト4.11: css-loader

```
{
  test: /\.css$/,
  use: [
    "style-loader",
    {
      loader: "css-loader",
      options: {
        modules: true
      }
    }
}
```

4.2.5 plugin

webpackは、pluginを使うことで便利に扱えます。公式が便利なpluginの一覧とその説明を見やすい形で提供していますが、利用度の高いものは本書でも解説します。[11]

html-webpack-plugin

ビルド後のファイルと、それを読み込むhtmlファイルの連携を取りやすくするためのプラグインです。これまではビルド済みファイルを読み取るために、/distディレクトリにあらかじめindex.htmlを配置する構成でした。しかしその構成だと、bundleファイル名が変わるとビルドが落ちるようになる、といった懸念や問題がありました。

html-webpack-pluginは、そのような問題を解決できるプラグインです。これは読み込ませたいhtmlを指定するだけで、ビルド時に一緒にoutputディレクトリにコピーしてくれます。さらには、scriptタグでの読み込み行まで埋め込みます。

たとえば、次のようなhtmlを用意します。

リスト4.12: build対象のHTML

```
<!DOCTYPE html>
<html>
 <head>
    <meta charset="UTF-8" />
    <title>webpack App</title>
  </head>
  <body></body>
</html>
```

そして、次のような設定ファイルを準備します。

リスト4.13: webpack.config.js

```
plugins: [new HtmlWebpackPlugin()]
```

これをビルドすると、次のようになります。

リスト4.14: build後のHTML

```
<!DOCTYPE html>
<html>
 <head>
```

```
      <meta charset="UTF-8" />
      <title>webpack App</title>
    </head>
    <body>
       <script src="bundle.js"></script>
    </body>
  </html>
```

<script src="bundle.js"></script>が、ビルド後のファイルに含まれました。これによりビルド後のhtmlを開くだけで、アプリケーションの実行を確かめられます。

Dotenv

環境変数は.envというファイルで管理することが多いです。環境変数には、APIのキーやパスワードなどの機密性の高いものが含まれており、ソースコードに直接含めることが危険だからです。よくあるのは、.envに環境変数を書き、Gitの管理下には置かず、.env.sampleのようなファイルにサンプル値を書いて、Gitで管理するといったやり方です。

プログラミング言語によっては、.envから環境変数にセットするライブラリがあり、JavaScript、Node.jsも例外ではありません。

たとえば、dotenv-webpackは.envにある環境変数を、ビルド時に埋め込んでくれます。[12]

リスト4.15: dotenvをプラグインに追加

```
  const Dotenv = require("dotenv-webpack");

  module.exports = {
    ...
    plugins: [
      new Dotenv(),
    ],
  };
```

この状態で.envにXXX="hello"などを用意してビルドすれば、アプリケーション内でprocess.env.XXXとして読み込むことができます。

フロントエンド開発においては、結局クライアントのコンソール実行ファイルを見れるので.envで管理しないデメリットは少ないです。ただ環境変数が一覧で保存しやすいという利点もあるので、筆者はこのプラグインを使っています。開発環境が複数あるときに、.envに環境変数をコピーすれば入れ替えが簡単で開発しやすくなります。

4.2.6 resolve

resolve[13]オプションはその名の通り、モジュールの解決方法を指定するためのオプションです。たとえば、import時のpathを短縮するためのaliasや、拡張子を省略できるextensionといった機能を提供します。ここでは、extensionについて解説します。

extension

extensionを指定することで、import時に拡張子の記述を省略できます。

リスト4.16: extensionの指定

```
module.exports = {
  //...
  resolve: {
    extensions: ['.wasm', '.mjs', '.js', '.json']
  }
};
```

拡張子を省略するため、異なる拡張子で同一名のファイルは識別できなくなります。その場合、extensionsで指定したもののうち先頭にあるものから優先されるため、注意しましょう。

4.3　まとめ

　webpackはモジュールバンドラです。最近のフロントエンド開発は、たくさんのライブラリを入れて開発するため、ソースコード間の依存は膨らんでいきます。webpackはそれらの依存関係を解決してくれます。

　あくまで依存解決が役割ですがloaderと呼ばれるpre processor機構を用意してくれており、ここでソースコードに対して変換処理を入れられます。つまり、Babelやtscを実行できます。そのため、webpackはトランスパイルはしないものの、ビルドの一端を担っています。よく役割を混乱しますが、そのときはトランスパイル・ビルド・バンドルといった言葉を意識してみてください。

1. https://webpack.js.org/
2. 単体で実行可能なファイルを生成するもの
3. たとえばCSSや画像ファイル
4. https://github.com/webpack/webpack-cli/tree/next/packages/webpack-cli
5. 正確には、webpackコマンドはwebpackの持ち物です。ただ、そのwebpackコマンドが内部でwebpack-cliを呼び出すので、webpack-cliをDLしなければ動きません。
6. ビルド後のソースとビルド前のソースを関連付けるファイル。これを用いれば、webpackでビルドしたファイルのデバッグが容易になる。
7. マルチページアプリケーション。複数のエントリポイントを持つアプリケーション。機能ごとに画面を分けられるメリットがある。
8. https://webpack.js.org/loaders/file-loader/
9. https://github.com/webpack-contrib/raw-loader
10. https://github.com/webpack-contrib/css-loader
11. https://webpack.js.org/plugins
12. https://github.com/mrsteele/dotenv-webpack
13. https://webpack.js.org/configuration/resolve/

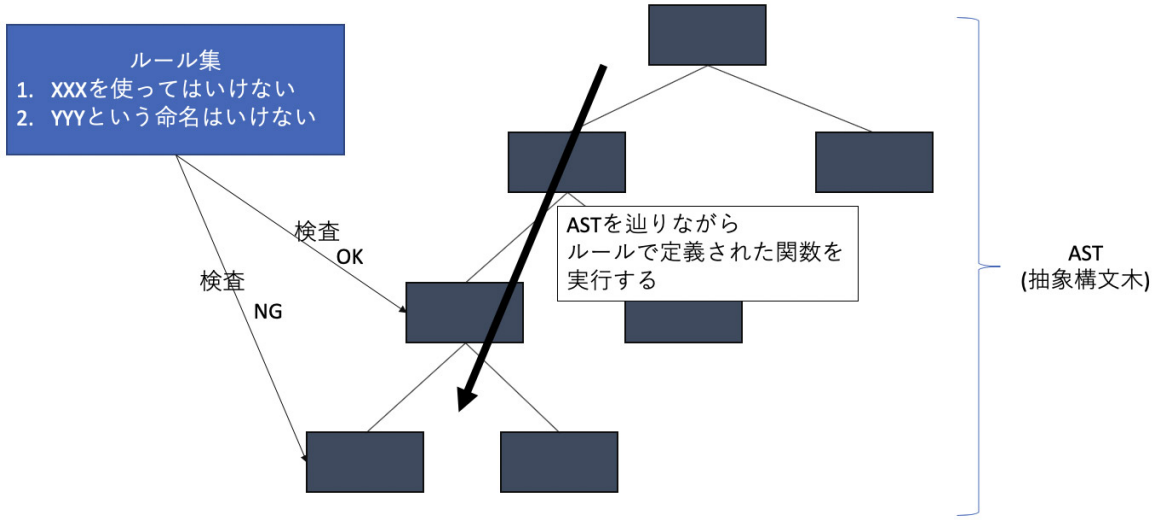
第5章　ESLintを使った静的解析

　lintもしくはlinterは、ソースコードの構文や品質を検査できるツールです。JavaScriptにもそのようなツールがあり、ESLintは代表的なlinterです。ESLintは公式の言葉を借りると、Pluggable JavaScript linterであり、pluginによって自由に拡張できます。[1]

5.1　ESLintのしくみ

JavaScriptのソースコードは、AST（抽象構文木）という形式で表現できます。ここではASTについては深く解説しませんが、ソースコードは何らかの木構造で表現できると考えてください。ESLintはこのASTを巡回し、その各ノードであらかじめ設定されたルールに合っているかを検証します。

図5.1: ASTとルール



ルール自体は関数として定義でき、そのルールを許可するかどうかは設定ファイルに記述しておきます。ルールの実体は**ただの関数**ですので、自作したり、第三者が作ったルールをインストールして使えます。たとえば私は以前、ソースコード内から俳句を見つけだし、それを禁止するルール

を作ったこともあります。[2]このようにESLintでは、**ルールの自作やインストールによって、柔軟な拡張ができます。**ルールが根幹にあるため、ESLintを理解するためには、まずルールに着目すると良いです。

5.2　ESLintの使い方

ESLintは次のコマンドで導入できます。

```
$ npm install -D eslint
```

ESLintには設定ファイルがあり、その設定が必要です。ここでは.eslintrc.jsに設定を書いていきます。[3]

5.2.1　チェックしたいルールを設定する

チェックしたいルールを設定ファイルに書いておけば、ESLintはlintを行います。ruleは以下のように設定できます。

リスト5.1: .eslintrc.js

```
module.exports = {
  rules: {
    "no-console": ["warn"],
  }
};
```

このように、ESLintではrule名とそれに対するふるまいを書くことで、ruleを設定できます。ふるまいとしては、"error"、"warn"、"off"があります。

no-consoleは、console.*[4]を禁止するルールです。このルールに該当したら、warn(警告)を出すというのがこの設定ファイルが指し示す意味

です。

5.2.2　ルールをセットにしたpluginを導入する

ESLintのルールは自分でも定義できますが、そのルールを自作することは大変な作業です。そこで、公式やサードパーティが配布しているルールを活用します。そのようなルールはpluginとも呼ばれており、設定ファイルを通して導入できます。たとえばReactに関するルールを入れたければ、eslint-plugin-reactをinstallし、それを設定ファイルで登録すれば使えます。

```
$ npm install -D eslint-plugin-react
```

リスト5.2: .eslintrc.js

```
module.exports = {
  "plugins": ["react"],
};
```

ただし注意しないといけないのは、**この時点ではルールのオン／オフは指定されていません。**あくまでルールを登録しただけであり、実際にルールを使うためにはrules配下に設定を書くか、次に紹介するextendsを使う必要があります。

5.2.3　どのルールを切り替えるかのextends

pluginsで入れたルールのオン／オフをセットする手順を、すべて手で行うと大変です。そこで、それらのオン／オフの設定をしてくれるextendsという仕組みを使います。extendsはpluginが提供するルール一覧に対

して、recommendedやallといったconfigセットを使った設定を可能にします。

リスト5.3: .eslintrc.js

```
module.exports = {
  "extends": [
        "plugin:react/recommended",
  ],
  "plugins": ["react"],
};
```

　このように設定することで、eslint-plugin-reactで入れたルールにおける推奨設定をしたことになります。

5.2.4　initコマンドを利用して生成する

　ここまで.eslintrc.jsに設定を直書きしましたが、これらの設定を生成する方法もあります。ESLintにはinitオプションがあり、これを利用します。

```
$ npx eslint --init
? How would you like to use ESLint? To check syntax and find problems
? What type of modules does your project use? JavaScript modules (import/export)
? Which framework does your project use? None of these
? Does your project use TypeScript? No
? Where does your code run? Browser
? What format do you want your config file to be in? JavaScript
Successfully created .eslintrc.js file
```

```
Done in 13.20s.
```

上記の表示になるように回答することで、出力された設定ファイルは次の通りです。

リスト5.4: .eslintrc.js

```
module.exports = {
  env: {
    browser: true,
    es6: true
  },
  extends: "eslint:recommended",
  globals: {
    Atomics: "readonly",
    SharedArrayBuffer: "readonly"
  },
  parserOptions: {
    ecmaVersion: 2018,
    sourceType: "module"
  },
  rules: {}
};
```

今回はフロントエンドのJavaScriptプロジェクトに対して、推奨設定を使うような設定を吐き出しました。ここで聞かれた回答の結果次第では、TypeScriptやReactやNode.jsに対する設定も出力できます。

この設定ファイルには、これまでに解説したpluginsやextends以外のものも多く登場しています。ここでESLintでは、何を設定できるのかを一望しましょう。

5.3　ESLintで設定する項目

5.3.1　rules

　rulesは、ESLintで検証したいルールそのものを指定できる箇所です。**ESLintでは、このrulesが根幹にあると考えてください。**公式による提供、第三者によるプラグイン提供、自作することによってruleを入手できます。デフォルトや公式がどのようなルールを提供しているかは、ドキュメントを参照してください。[5]
　このルールの正体はASTを検証する関数です。その関数の中で検査を行い、その結果に応じてルール違反かどうかをレポートできます。
　そのルールに対して、それを許可または禁止するかの設定を、この項目で行えます。

リスト5.5: eslintのrule

```
{
    "rules": {
        "semi": ["error", "always"],
        "quotes": ["error", "double"]
    }
}
```

　ルールには次の3つのレベルがあります。
　・なし: "off" or 0
　・警告: "warn" or 1
　・エラー: "error" or 2
　これらは、objectのvalueの配列の0番目に指定します。また、ルールのオプションはvalueの配列の1番目以降に指定します。たとえば、semiというルールだとオプションとしてalways、neverなどを指定できます。[6]

5.3.2　extends

　extendsは設定を拡張するセットで、第三者の.eslintrcを読み込めます。主にどのruleをオン／オフにするかを一括で設定するために使われています。

　extendsを提供するライブラリには、有効にするルールを設定したconfigファイルが同梱されていることもあります。たとえば、eslint:recommendedというルールセットがよく使われていますが、すべてのルールをONにするeslint:allもあり、extends内の宣言で切り替えることができます。

リスト5.6: eslint-recommended.js

```
/**
  *  @fileoverview  Configuration  applied  when  a  user
configuration extends from
 * eslint:recommended.
 * @author Nicholas C. Zakas
 */

"use strict";

/* eslint sort-keys: ["error", "asc"] */

/** @type {import("../lib/shared/types").ConfigData} */
module.exports = {
    rules: {
        "constructor-super": "error",
        "for-direction": "error",
        "getter-return": "error",
        "no-async-promise-executor": "error",
        "no-case-declarations": "error",
        ...
        "no-useless-catch": "error",
        "no-useless-escape": "error",
        "no-with": "error",
        "require-yield": "error",
        "use-isnan": "error",
```

```
            "valid-typeof": "error"
        }
    };
```

ちなみにpluginを入れずとも、ルールを追加できるextendsも存在しており、このextendsはpluginと混乱しやすいものでもあります。たとえば、eslint-plugin-node[7]は"extends": ["plugin:node/recommended"]と設定するだけで、configの設定だけでなくルールの追加まで行ってくれます。公式の説明にはadd this plugin into plugins.とあり、pluginの追加もしています。[8]extendsは設定ファイル自体を拡張する機能なので、この挙動は正常ですが、extendsとpluginの役割を混乱させる挙動になっているため、注意しましょう。

5.3.3 plugin

pluginは、ESLintの設定を追加する役割を担います。主にルールを追加するために利用します。[9]pluginをダウンロードし、設定ファイルのplugins項目で指定することで利用可能です。

リスト5.7: .eslintrc.js

```
module.exports = {
    "plugins": [
        "react",
        "@typescript-eslint"
    ],
};
```

pluginを入れることで、第三者が作成したルールを手に入れることができます。ただしそのルールを許可・禁止するかどうかは、自分でrulesで設定するか、extendsの指定が必要です。[10]

5.3.4　env

　envプロパティを使うことで、実行環境に関する前提条件を設定できます。たとえば、「これはブラウザで動くコードである」、「これはES6を使っている」といったことを宣言できます。ブラウザで動くことを指定していれば、windowといった変数が突然現れても、未定義変数である警告[11]を発しないなどの利点があります。

　このenvは、eslint --initの回答結果からも影響します。eslint --initから作られるパターンは少ないですが、実際には多くの設定可能な値があります。

・browser
・node
・commonjs
・es6
・es2017
・serviceworker
・etc...

　自分のソースコードを動かす環境と同じ設定を必ずしておきましょう。

　たとえば、ES6を使ったフロントエンドのコードではこのような設定をしておきます。

リスト5.8: .eslintrc.js

```
module.exports = {
    "env": {
        "browser": true,
        "es6": true
    }
};
```

5.3.5　globals

　globalsでは、特定のグローバル変数を許可・禁止できます。globalsにはwritableとreadonlyを指定でき、上書き可能・読み取り可能を定

義できます。

リスト5.9: globalsの例

```
  "globals": {
    "var1": "writable",
    "var2": "readonly"
  }
```

　ただ、一般的に使うグローバル変数はenvの設定で事足りるでしょう。そのため、ほかのJavaScriptファイルから読み込んだグローバル変数を使いたい場合などに出番があるオプションです。[12]
　また、globalsにはoff設定もあり、global変数の利用を禁止できます。この値を使えば、envで許可したグローバル変数の一部だけを禁止できます。

リスト5.10: offの例

```
  {
    "env": {
      "es6": true
    },
    "globals": {
      "Promise": "off"
    }
  }
```

5.3.6　parser

　parserのバージョンを指定します。ESLintは標準では、ES5を想定してparseを行います。そのため、lint対象となるコードにあったparserをインストール・選択しないと、ESLintは実行できません。たとえば、TypeScriptを対象にlintを行う場合は、@typescript-eslint/parserが必要です。

また、parserOptionsでは、さらに細かくparserの挙動を指定できます。これを使うと、JSXを受け入れることもできます。

リスト5.11: .eslintrc.js

```
module.exports = {
    ...
    "parser": "@typescript-eslint/parser",
    "parserOptions": {
        "ecmaFeatures": {
            "jsx": true
        },
        "ecmaVersion": 2018,
        "sourceType": "module"
    },
};
```

5.4　TypeScriptとReact用の設定をする

それでは、TypeScriptとReact用の設定をしてみましょう。

5.4.1　eslint init

TypeScriptとReactの設定も、ESLintのinit機能で作れます。使用フレームワークや使用言語については、init時の質問で聞かれます。

```
$ npx eslint --init
? How would you like to use ESLint? To check syntax and find problems
? What type of modules does your project use? JavaScript modules (import/export)
? Which framework does your project use? React
? Does your project use TypeScript? Yes
? Where does your code run? Browser
? What format do you want your config file to be in? JavaScript
Local ESLint installation not found.
The config that you've selected requires the following dependencies:
eslint-plugin-react@latest @typescript-eslint/eslint-plugin@latest @typescript-eslint/parser@latest eslint@latest
? Would you like to install them now with npm? Yes
```

上記の通りに回答した結果、次のような設定ファイルが作成されます。

リスト5.12: .eslintrc.js

```
module.exports = {
    "env": {
        "browser": true,
        "es6": true
    },
    "extends": [
        "eslint:recommended",
        "plugin:react/recommended",
        "plugin:@typescript-eslint/eslint-recommended"
    ],
    "globals": {
        "Atomics": "readonly",
        "SharedArrayBuffer": "readonly"
    },
    "parser": "@typescript-eslint/parser",
    "parserOptions": {
        "ecmaFeatures": {
            "jsx": true
        },
        "ecmaVersion": 2018,
        "sourceType": "module"
    },
    "plugins": [
        "react",
        "@typescript-eslint"
    ],
    "rules": {
    }
};
```

5.4.2　必要なプラグインをinstall

initコマンド実行時、Would you like to install them now with npm?と聞かれて、Yesと回答しました。その結果、次のコマンドが実行されたのと同じ結果になっています。

```
$ npm install -D eslint-plugin-react @typescript-eslint/eslint-plugin @typescript-eslint/parser
```

これらは、上記の設定ファイルを動かすにあたって、必要となるライブラリです。それぞれ、どういう役割のライブラリなのか見てみましょう。

eslint-plugin-react

eslint-plugin-react[13]は、React用のルールセットです。extendsも含まれており、推奨設定のみONにするplugin:react/recommendedと全ルールをONにするplugin:react/allが含まれています。

@typescript-eslint/eslint-plugin

@typeScript-eslint/eslint-pluginをインストールすると、typeScript-eslintプラグインを利用できます。このプラグインは、ESLintにないTypeScript固有の設定を担います。

リスト5.13: eslint-pluginの設定

```
module.exports = {
  ...,
  extends: [
    'eslint:recommended',
     'plugin:@typescript-eslint/eslint-recommended', // <=
NEW
    'plugin:@typescript-eslint/recommended', // <= NEW
  ]
};
```

plugin:@typescript-eslint/recommendedは、型を必要としない基本設定を詰め込んだものです。plugin:@typescript-eslint/eslint-recommendedは、TypeScriptでチェックされる項目を除外する設定です。両方ともnpx eslint --init実行時に設定ファイルへと追加されます。

@typescript-eslint/parser

TypeScriptを解析するためには、このparserが必要です。これまでのJavaScript用のparserだと、次のようなコードをlintしようとしてエラーになります。

リスト5.14: main.js

```
export class Hoge {
  private _id: string;

  constructor(id: string) {
    this._id = id;
  }

  getId() {
    return this._id;
  }
}
```

ES5にはアクセサ修飾子やクラスフィールドがないため、上記のようなコードはparseできません。実行すると、Parsing error: Unexpected token _id eslintというエラーが出るでしょう。

5.4.3　設定された項目をみてみる

initで生成された設定がどのようなものか、1つずつ見てみましょう。

env

envではbrowserとES6が許可されています。そのため、ブラウザやES6で登場するメソッドをソースコード内で未定義のまま使っても、エラーは出ません。

リスト5.15: env

```
"env": {
  "browser": true,
  "es6": true
}
```

extends

ESLint標準の推奨設定とReactプラグインの推奨設定と、TSプラグインの推奨設定がされています。

リスト5.16: extends

```
"extends": [
  "eslint:recommended",
  "plugin:react/recommended",
  "plugin:@typescript-eslint/eslint-recommended"
],
```

この項目は、1問目のHow would you like to use ESLint?の回答に影響しています。
このとき提示された選択肢は、次の通りです。
・To check syntax only
・To check syntax and find problems
・To check syntax, find problems, and enforce code style
このうちTo check syntax and find problemsを選び、その結果"eslint:recommended"が設定ファイルに記述されました。
ちなみにTo check syntax onlyの場合、"eslint:recommended"は入りません。またTo check syntax, find problems, and enforce

code styleを選んだ場合はstyle guide[14]の選択肢が出てきます。

globals

SharedArrayBufferとそれを扱えるAtomic APIがreadableです。[15]

リスト5.17: globals

```
 "globals": {
  "Atomics": "readonly",
  "SharedArrayBuffer": "readonly"
},
```

parser

parserは@typescript-eslint/parserが指定されています。TypeScriptを静的解析するためには、このparserが必要です。

リスト5.18: parser

```
"parser": "@typescript-eslint/parser",
```

parserOptions

ここではES2018の文法とjsxが指定されており、ES Moduleであることをparserに伝えています。ecmaVersionの指定は新しい構文を使う上では必要なので、設定忘れに注意しましょう。[16]

リスト5.19: parserOptions

```
"parserOptions": {
  "ecmaFeatures": {
    "jsx": true
  },
  "ecmaVersion": 2018,
```

```
    "sourceType": "module"
  },
```

plugins

eslint-plugin-reactと@typescript-eslint/eslint-pluginが指定されています。この2つのプラグインを入れることで、TypeScriptとReactに関するルールを追加できます。

リスト5.20: plugins

```
  "plugins": [
    "react",
    "@typescript-eslint"
  ],
```

rules

ここでは何もruleが指定されていません。init時では、個別にルールのオン／オフは指定されません。extendsで設定されたルールはここで上書きできるため、適宜利用しましょう。

リスト5.21: rules

```
  "rules": {}
```

5.5　まとめ

　Linterは、ソースコードの品質を担保するしくみです。ESLintでは、利用したいルールとその可否を指定することで規約を作り、静的解析を行います。規約をすべて手で書くことは大変ですが、Extendsという仕組みである程度の設定を自動で行えます。またルールそれ自体はただの関数で、自分で定義できるほか、第三者が作ったルールをplugin経由で入れられます。設定ファイルの記述はたいへんですが、initコマンドでひな型を生成できるので、有効活用しましょう。

1. https://eslint.org/
2. https://github.com/sadnessOjisan/eslint-plugin-detect-haiku
3. こちらもBabel同様このファイル名に限りません。たとえば.eslintrc といったファイル名でも設定ファイルを書けます。
4. console.log や console.info などが該当します。ソースコードにこれらのコマンドが残っていればユーザーにデバッグログが見えたりもするので、入れておきたいルールです。
5. https://eslint.org/docs/rules/
6. https://eslint.org/docs/rules/semi
7. https://github.com/mysticatea/eslint-plugin-node
8. https://github.com/mysticatea/eslint-plugin-node/blob/master/lib/configs/recommended-module.js
9. 厳密には、environmentsなどの設定も行えます。 https://eslint.org/docs/developer-guide/working-with-plugins
10. pluginの提供者はextendsも提供していることが多いので、提供者が用意した設定を使うことがほとんどです。
11. no-undefルールが設定されていれば警告が出るため、その警告を抑えることができます。
12. たとえば、トラッキングに使うSDKを呼び出すときなどに使います。
13. https://github.com/yannickcr/eslint-plugin-react
14. コーディング規約。組織ごとに作られることも多い。有名なところだとAirbnbの規約がある。https://github.com/airbnb/javascript
15. この設定を付け加える理由は明記されていませんがコミッターの方に問い合わせたところ、歴史的背景とのことでした。

16. global変数の有効化はenvで行えますが、構文に対する有効化は、この設定が必要となります。

第6章　Prettierを使ったフォーマット

Prettier[1]は、code formatterです。ソースコードを読みやすく整形します。導入すると、読みやすさを担保するというメリットがありますが、運用を徹底することで**チーム開発時に開発者どうしで不要な差分を出さないというメリット**もあります。

6.1　prettierの使い方

6.1.1　prettierのインストール

Prettierをインストールします。

```
$ npm install prettier --dev
```

インストール後、prettierコマンドを利用できます。

```
$ npx prettier test.js
```

実行すると結果が標準出力に出力されますが、format結果はファイルに保存されてほしいです。また1ファイルだけではなく、プロジェクト全体のコードをformatにかけたいです。そのために実行方法を工夫します。

6.1.2　prettierの実行

prettierの実行オプションで、ファイル保存や複数ファイルを対象とした実行ができます。それらのオプション付きのコマンドが少し長いので、npm scriptsに保存しておきます。

リスト6.1: package.json

```
  "scripts": {
    "format": "prettier --write './src/**/*.{js,ts,tsx}'"
  }
```

これで、"prettier --write './src/**/*.{js,ts,tsx}'" をnpm run formatとして実行できます。--writeは、フォーマットしたファイルを上書きするためのオプションです。'./src/**/*.{js,ts,tsx}'はフォーマット対象です。glob形式はpathをsingle quote('')で囲まないと使えないので、注意しましょう。

6.1.3 prettierの設定

Prettierでどのようなformatをするかも、設定ファイルに書くことができます。たとえばindentの幅を指定できたり、折り返す上限文字数を指定できます。これもBabelのように、.prettierrcやprettier.config.jsといったファイルに書くことができます。拡張子や記法については、公式サイトでご確認ください。[2]ここではJSの記法で作成します。

リスト6.2: prettier.config.js

```
module.exports = {
  trailingComma: "es5",
  tabWidth: 4,
  semi: false,
  singleQuote: true
};
```

このようにして設定できますが、筆者は設定項目にはあまり関心がなく、チームでフォーマットが決まっていれば何でもよいという立場です。なので、どういう設定がよいかといった議論には、ここでは立ち入りません。[3]最適な設定は、プロジェクト内容やチームによって違ってくるはずで

す。どういう項目を設定できるかは、こちらを確認した上でご自身で決められると良いでしょう。[4]

6.2　ESLintとの協調

ところで、linterとformatterは衝突しないのでしょうか。たとえば、ESLintには--fixというオプションがあり、これを実行することでソースコードを書き換えられます。この書き換えとPrettierでの書き換えは、衝突するときがあります。これに対して、公式は「**eslintでのformatをやめる**」「**eslintからprettierを実行する**」ように推奨しています。[5]その方法を実現するためには、次の2つのライブラリを利用します。

6.2.1　eslint-config-prettier

eslint-config-prettier[6]はPrettierとESLintの衝突を避けるために、eslintでの styleに関するルールを全部offにします。

```
$ npm install -D eslint-config-prettier
```

リスト6.3: .eslintrc.js

```
{
  "extends": [
    "airbnb-base", // 例なのでなんでもいい
    "prettier",
  ]
}
```

これにより、ESLint側でスタイルに関して指摘されなくなります。

6.2.2　eslint-plugin-prettier

　eslint-plugin-prettier[7]はeslintからprettierの実行を可能にするプラグインです。このプラグインがなければ、npx eslint src/main.js --fixしてから、npm formatしなければいけません。コマンドを2つ実行することは億劫なので、eslint実行時にformatもできるようにします。

```
$ npm install -D eslint-plugin-prettier
```

　eslint-plugin-prettier 自体は次のように使います。

リスト6.4: .eslintrc.js

```
{
  "plugins": ["prettier"],
  "rules": {
    "prettier/prettier": "error"
  }
}
```

　このrulesはeslint-plugin-prettierに定義されているもので、prettierの実行結果をESLintのエラーとして扱えるようにしています。つまり、formatが間違っていれば、ESLintのエラーとして扱えます。
　上記がeslint-plugin-prettierの設定ですが、**実際には上記の設定はしません。**eslint-plugin-prettierはextendsも提供しており、それを利用すると上記の設定が足されるため、extendsに任せます。**つまりextendsを利用することで、上記のpluginとrulesの設定が不要になります。**

リスト6.5: eslint-plugin-prettierにおけるlinterの設定

```
{
 ...,
  extends: [
    "airbnb-base'", // 例なので何でもいい
    "plugin:prettier/recommended" // <= 追加
  ],
}
```

ここでprettierのextends設定は、**extendsの最後で読み込むように**注意してください。ESLintのルールは後勝ちするため、もし後ろのルールがstyle設定をonにしていれば、上書きされてしまうからです。このplugin:prettier/recommended extendsは次の3つを行っています。

・eslint-plugin-prettierを利用可能にする
・prettierでのルールに反した場合、ESLintのエラーとして扱う
・eslint-config-prettierのextendsを読み込む

つまり、ESLintとPrettierを共存させる設定を、すべてこのextendsが行っています。[8] ただし、eslint-config-prettierをeslint-plugin-prettierが読み込んでいても、その実体は自分でダウンロードする必要があるので注意しましょう。あくまでも、eslint-plugin-prettierは設定ファイルをextendsできるだけです。

そのため、ESLintとPrettierを共存させる設定は次の通りです。

リスト6.6: ESLintとPrettierの共存

```
{
 ...,
  extends: [
    "airbnb-base'", // 例なので何でもいい
    "plugin:prettier/recommended" // <= 追加
  ],
}
```

6.3　TypeScriptとの協調

6.3.1　prettier/@typeScript-eslint

PrettierとESLintの競合を消すときに導入したeslint-config-prettierには、prettier/@typeScript-eslintというextendsが含まれています。これは、ESLintのTypeScript対応で導入したプラグイン@typescript-eslint/eslint-pluginとのルールの競合を抑えるための設定です。

抑えるルールは次の通りです。[9]

リスト6.7: 抑えられるルール

```
module.exports = {
  rules: {
    "@typescript-eslint/quotes": 0,
    "@typescript-eslint/brace-style": "off",
    "@typescript-eslint/comma-spacing": "off",
    "@typescript-eslint/func-call-spacing": "off",
    "@typescript-eslint/indent": "off",
    "@typescript-eslint/member-delimiter-style": "off",
    "@typescript-eslint/no-extra-parens": "off",
    "@typescript-eslint/no-extra-semi": "off",
    "@typescript-eslint/semi": "off",
          "@typescript-eslint/space-before-function-paren":
"off",
    "@typescript-eslint/type-annotation-spacing": "off"
  }
};
```

TypeScriptを併用する場合は、このextendsも利用しましょう。

リスト6.8: TypeScriptとESLintとPrettierの共存

```
{
 ...,
  extends: [
    "plugin:prettier/recommended",
    "prettier/@typeScript-eslint"
  ],
}
```

6.4　まとめ

　Prettierは、ソースコードのフォーマットを整えてくれるツールです。一方でESLintは、ソースコードの誤りや品質を指摘してくれるツールです。この2つは同時に登場する機会が多いので、混乱しないように注意しましょう。

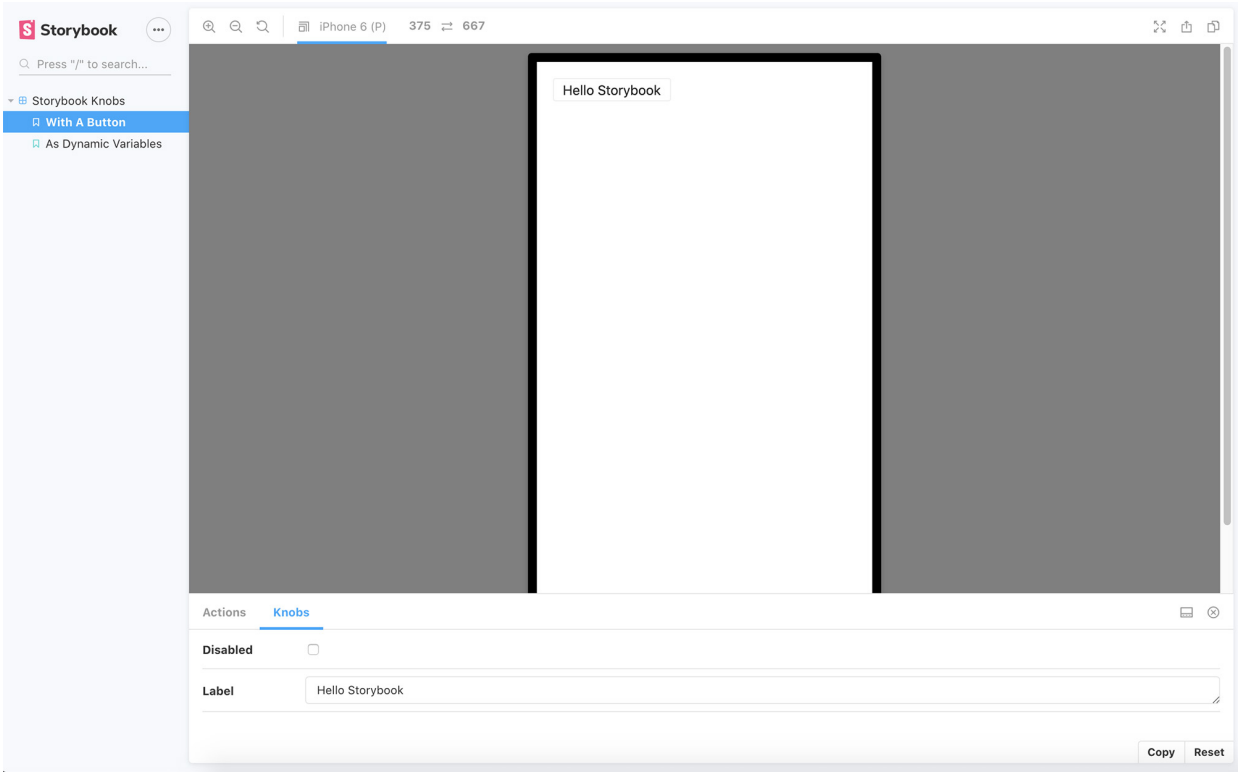
　PrettierとESLintはときに競合します。それはESLintがスタイルを指摘するためです。そのため、eslint-config-prettierでESLintにおけるスタイルのルールをOFFにしましょう。また、PrettierをESLintから実行できるように、eslint-plugin-prettierも活用できます。似たようなライブラリ名ですが、混乱しないように役割を意識しましょう。

1. https://prettier.io/
2. https://prettier.io/docs/en/configuration.html
3. どんなルールであれ、チームで同じルールを使えば差分はでないため。
4. https://prettier.io/docs/en/options.html
5. https://prettier.io/docs/en/integrating-with-linters.html
6. https://github.com/prettier/eslint-config-prettier
7. https://github.com/prettier/eslint-plugin-prettier
8. https://github.com/prettier/eslint-plugin-prettier/blob/master/eslint-plugin-prettier.js
9. https://github.com/prettier/eslint-config-prettier/blob/master/%40typescript-eslint.js

第7章　Storybookを使ったコンポーネント管理

Storybookは、UIコンポーネントを管理できるプレイグラウンド環境です。[1]公式はBuild bulletproof UI components fasterと宣伝しており、今ではただのプレイグラウンド以上に、様々な機能があります。たとえばテストに利用したり、ドキュメントの出力にも使えます。また、サーバが完成していない状況でもダミーデータを入れてUIを確認したり、stateを切り替えるaddonを使うことで、様々な分岐を網羅しながら開発を進められます。そのため、ただのプレイグラウンドに止まらず、開発を支援する様々なツールが入ったライブラリと見なせます。

図7.1: Storybook



ただし、設定には慣れが必要で、独自の設定方法も覚える必要があります。プレイグラウンド環境の構築は、初見だと容易ではありません。そのため、Storybookを入れるのであれば、開発初期で入れるようにしたいです。

7.1　Storybookを設定する

　Storybookのプレイグラウンド環境は、ライブラリをインストールして設定ファイルを書くことで作れます。便利なことに、その環境を生成してくれるCLIツールがあるため、本書ではそのツールを使います。

```
$ npx -p @storybook/cli sb init
```

　このコマンドを利用すると、利用フレームワークやライブラリに応じて、Storybookの環境ができます。[2]上記のコマンドにより、storybookコマンドがnpm scriptsに登録されました。

リスト7.1: package.json

```
{
  ...,
  "scripts": {
    "storybook": "start-storybook -p 6006",
    "build-storybook": "build-storybook"
  }
}
```

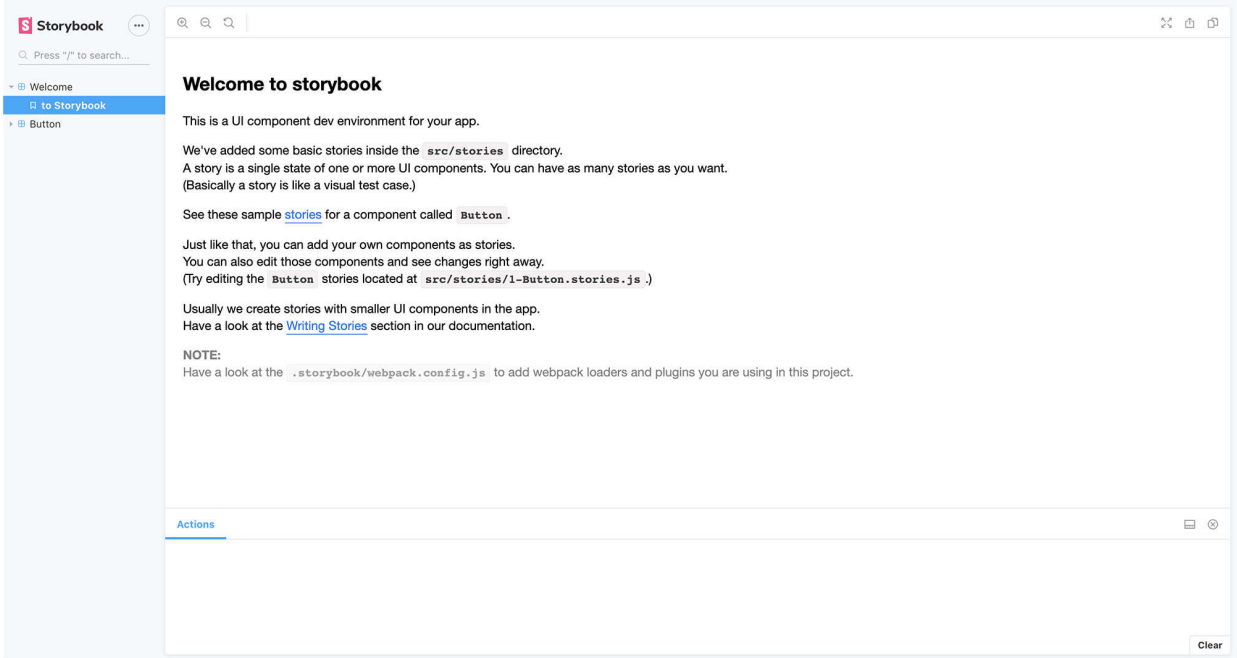
　このStorybook環境を立ち上げるコマンドを実行します。

```
$ npm run storybook
Storybook 5.3.19 started
5.12 s for manager and 4.85 s for preview
```

```
Local: http://localhost:6006/
On your network: http://192.168.0.2:6006/
```

Storybookの環境が立ち上がりました。

図7.2: CLI環境が生成したStorybook環境



このように、CLIツールを使うことで簡単にStorybook環境を立ち上げられました。次に、その環境を拡張するために設定の手順を確認していきます。

7.1.1 story fileを作る

Storybook環境は、story fileと呼ばれるファイルによって構築されます。story fileはたとえばこのようなものです。

リスト7.2: app.stories.jsx

```
import React from 'react';
import { action } from '@storybook/addon-actions';
import Button from './Button';

export default {
  component: Button,
  title: 'Button',
};

export const text = () => <Button onClick=
{action('clicked')}>Hello Button</Button>;

export const TextButton = () => (
  <Button onClick={action('clicked')}>
    <span role="img" aria-label="so cool">
      TEXT
    </span>
  </Button>
);
```

story fileではコンポーネントをimportし、そのコンポーネントに説明を付け加えています。Storybookはこのファイルを読み取り、そこに書かれた情報を元に、プレイグラウンド環境を構築します。

Component Story Format (CSF)

では、story fileの作り方を詳しく見ていきましょう。story fileは、Component Story Format (CSF)[3]と呼ばれる形式に沿って書くことが推奨されています。Component Story Formatは、コンポーネントとそのメタデータオブジェクトがES Moduleとして定義されたフォーマットです。

リスト7.3: Component Story Format (CSF)

```
export default {
  title: 'Path/To/MyComponent',
  component: MyComponent,
  decorators: [ ... ],
```

```
    parameters: { ... }
  }
```

CSFではexport defaultしたオブジェクトは、story fileのメタデータを表します。この中に、story名やaddonと呼ばれる拡張の設定を書きます。

一方でnamed exportsしている関数は、Storybookで動かす対象となるコンポーネントです。named exportsされていると、Storybookの管理対象になります。また、named exportsしたオブジェクト（ここではFunction Component）のプロパティを通して、コンポーネント単位でメタ情報を追加していくこともできます。たとえば、nameの指定をすると、左サイドバーに表示される名前を変えられます。

リスト7.4: Storyを拡張する

```
export const TextButton = () => (
  <Button>
    <span role="img" aria-label="so cool">
      TEXT
    </span>
  </Button>
);

TextButton.story = {
  name: 'with TEXT',
};
```

decorator

decoratorは、componentやstory fileをwrap (修飾)します。

リスト7.5: decoratorの例

```
export default {
  title: 'Button',
```

```
  decorators: [storyFn => <Center>{storyFn()}</Center>],
};
```

スタイリングだけでなく、addon(のちに紹介する拡張機能) の設定にも使います。たとえば、@storybook/addon-knobsなどはこの設定が必要です。

リスト7.6: @storybook/addon-knobsの設定

```
import React from "react";
import { withKnobs } from "@storybook/addon-knobs";

export default {
  title: "Storybook Knobs",
  decorators: [withKnobs]
};
```

Componentに共通した何か（スタイリングや機能）を埋め込みたいときに使う機能と考えると良いです。

Parameters

Parametersは、storyオブジェクトに対するカスタムメタデータです。これを利用する場面の1つには、addonに対する設定の埋め込みが挙げられます。

たとえば、様々なデバイスで確認できるようにする@storybook/addon-viewportにおける、「どのようなデバイスを利用するか」という設定は次のように埋め込まれます。

リスト7.7: parametersの指定

```
export default {
  title: "Button",
  component: Button,
  parameters: {
```

```
      viewport: {
        viewports: INITIAL_VIEWPORTS,
      },
    },
  };
```

7.1.2　story fileをロードする

Storybookのconfigファイルでは、story fileとしてロードするファイルを指定できます。その設定は.storybook/main.jsに書きます。読み込み対象としては、hoge.stories.tsxや、__stories/hoge.tsxといった形式がよく使われます。

リスト7.8: .storybook/main.js

```
module.exports = {
  addons: [
    ...
  ],
  stories: ["../**/*.stories.tsx"],
  webpackFinal: async (config) => {
    ...
  },
};
```

7.1.3　ビルドの設定をする

configファイルである.storybook/main.jsのwebpackFinalには、ビルドの設定を書いていきます。たとえば、TypeScript対応の設定としてts-loaderの読み込みを行います。

リスト7.9: .storybook/main.js

```
module.exports = {
  ...
  webpackFinal: async config => {
```

```
    config.module.rules.push({
      test: /\.(ts|tsx)$/,
      use: [
        {
          loader: require.resolve("ts-loader")
        },
      ]
    });
    config.resolve.extensions.push(".ts", ".tsx");
    return config;
  }
};
```

7.1.4　UIの設定をする

　もしコンポーネント本体が何らかのreset.cssなどのグローバルスタイルが当たっている場合、Storybook側のUIでも、それを反映させる必要があります。そのために利用するのは、preview-body.html[4]です。たとえばリセットしたいCSSを追加したい場合は、このようなstyleタグを追加すると実現できます。

リスト7.10: .storybook/preview-body.html

```
<style>
  *,
  *:after,
  *:before {
    margin: 0;
    padding: 0;
    box-sizing: inherit;
  }

  html {
    font-size: 62.5%;
  }

  body {
```

```
      box-sizing: border-box;
    }
  </style>
```

StorybookそのもののUIに対して調整したい場合は、options API[5]を利用できます。たとえば、ナビゲーションの表示の有無やパネルの位置を指定できます。

7.2　Storybookをaddonで拡張する

　Storybookにはaddon[6]と呼ばれる、拡張を受け付ける仕組みがあります。addonを使うとデバッグやドキュメンテーションなどが容易になり、利便性が向上します。
　利用するaddonは、設定ファイルで登録できます。ここで登録したaddonの順序が、Storybook環境でのaddonタブ(画面の下に表示されるタブ)での並び順です。[7]

リスト7.11: .storybook/main.js

```
module.exports = {
  addons: [
    '@storybook/addon-actions',
    '@storybook/addon-knobs'
  ],
  ...
};
```

　ここでは利用するaddonの登録だけをしており、addon自体の設定は別途story fileごとに行います。また、ここに登録せずに利用するaddonもあるので、設定は注意が必要です。たとえば、@storybook/addon-infoはここに登録しません。
　Storybookではどのようなaddonが使えるかを見てみましょう。

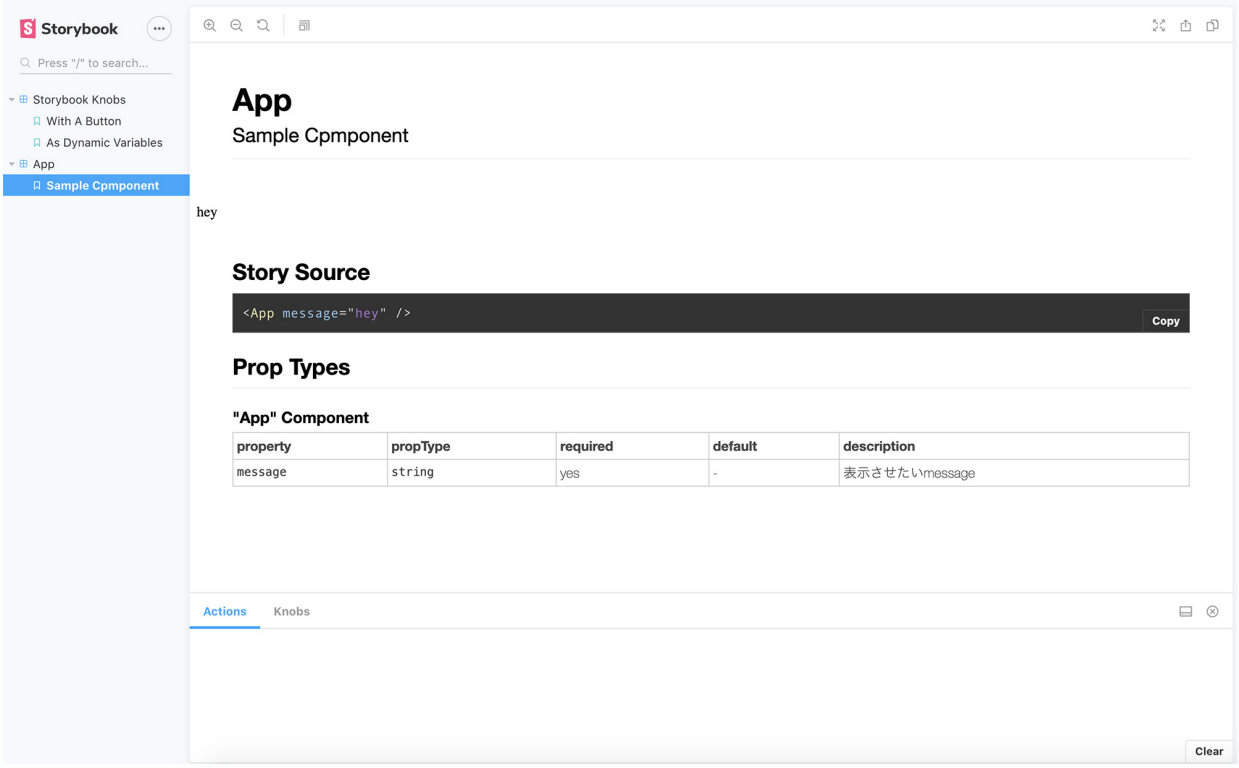
7.2.1　@storybook/addon-info

　Storybook Info Addon[8]を使うと、コンポーネントのドキュメントやpropsを表示させられます。propsの説明はPropsTypeやTypeScriptから生成できます。

リスト7.12: app.stories.jsx

```
import React from "react";
import { withInfo } from "@storybook/addon-info";
import { App } from "./app";

export default {
  title: "App",
  decorators: [withInfo],
  parameters: {
    info: { inline: true },
  },
};

export const SampleCpmponent = () => <App message="hey">
</App>;
```

図7.3: Storybook



parametersの設定は、どのようにドキュメントを表示させるかの設定です。ソースコードの表示を行うか、デフォルト状態でドキュメントを表示させるか、といった設定ができます。

ここではstory fileのUIを拡張するためにdecoratorが使われており、story fileごとにこの設定が必要になります。もしそれが手間だという場合は、preview.jsを利用して一括で設定もできます。

リスト7.13: .storybook/preview.js

```
import { addDecorator } from "@storybook/react";
import { withInfo } from "@storybook/addon-info";

addDecorator(withInfo);
```

リスト7.14: app.stories.jsx

```
import React from "react";
import { App } from "./app";

export default {
  title: "App",
  parameters: {
    info: { inline: true },
  },
};

export const SampleCpmponent = () => <App message="hey">
</App>;
```

7.2.2 @storybook/addon-actions

Storybook Addon Actions[9]を使うと、Storybook上で発火したイベントをログに出力できます。これにより、たとえばbuttonやinputタグの挙動をStorybook上で確認できます。

利用するためには、まずaddonを利用することをmain.jsに登録します。

リスト7.15: .storybook/main.js

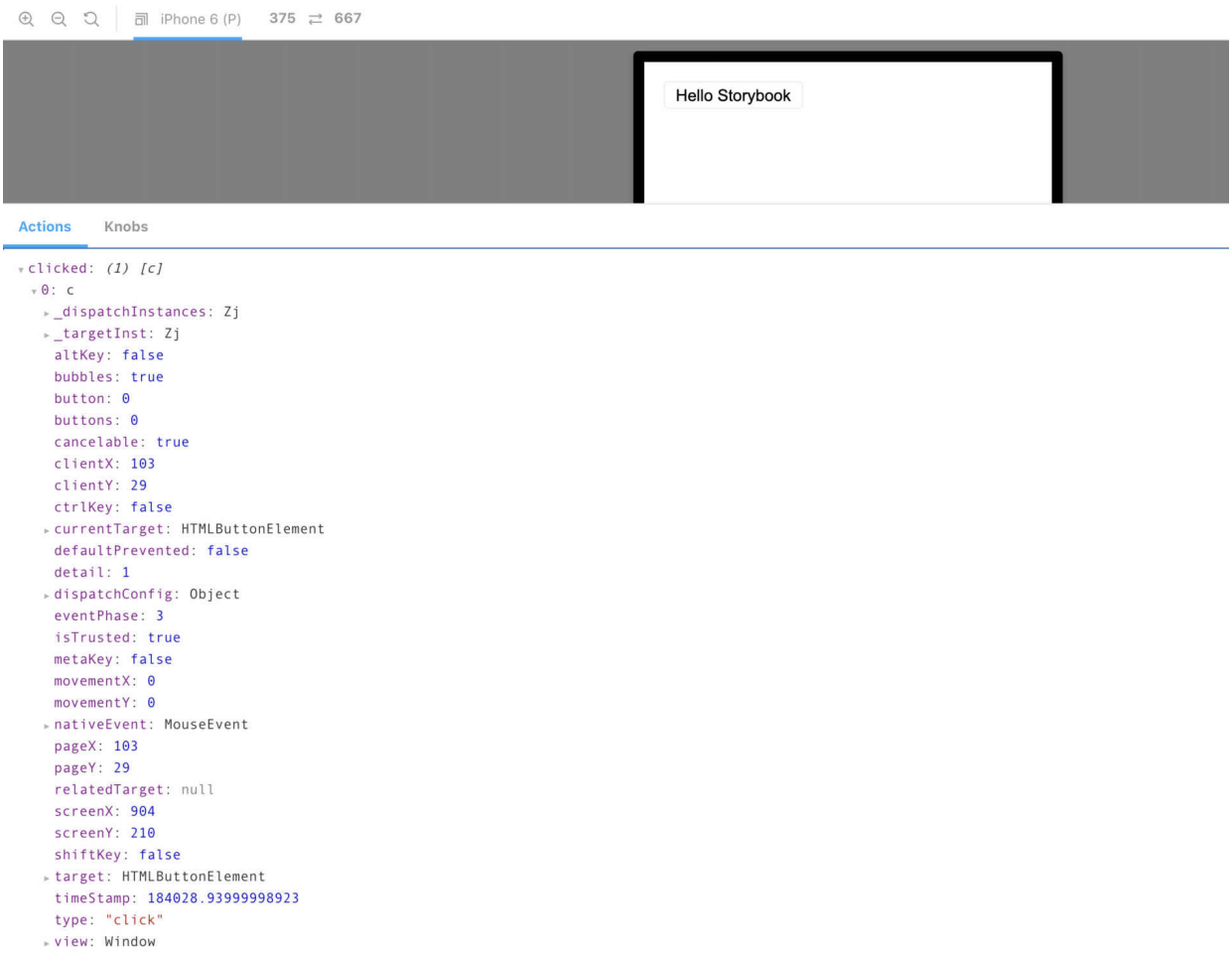
```
module.exports = {
  addons: [
    ...
    "@storybook/addon-actions", // <= 追加
  ],
  ...
};
```

そしてstory fileでは、action関数をイベントハンドラの中で実行します。

リスト7.16: app.stories.jsx

```
import { action } from '@storybook/addon-actions';
import Button from './button';

export default {
  title: 'Button',
  component: Button,
};

export const defaultView = () => (
    <Button  onClick={action('button-click')}>Hello  World!
</Button>
);
```

図7.4: addon-actions



このように、クリック時に発火したイベントをログとして確認できます。

7.2.3 @storybook/addon-knobs

Storybook Addon Knobs[10]を使うと、Storybook上でpropsの値を書き換えることができます。つまり、Componentのstateを切り替えて、挙動を確認できます。

decoratorを指定して、knob(text、boolean、numberなど)を呼ぶだけで使えます。

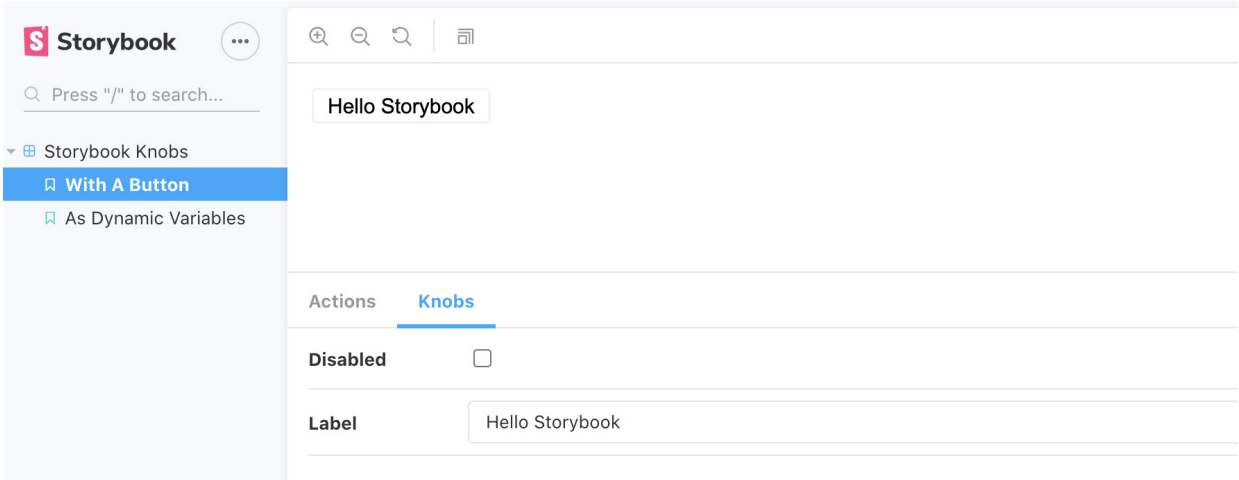
リスト7.17: .storybook/main.js

```
module.exports = {
  addons: [
    ...
    "@storybook/addon-knobs", // <= 追加
  ],
  ...
};
```

リスト7.18: Button.stories.tsx

```
import React from "react";
import { withKnobs, text, boolean, number } from
"@storybook/addon-actions";

export default {
  title: "Storybook Knobs",
  decorators: [withKnobs]
};
// Add the `withKnobs` decorator to add knobs support to
your stories.
// You can also configure `withKnobs` as a global
decorator.

// Knobs for React props
export const withAButton = () => (
  <button disabled={boolean("Disabled", false)}>
    {text("Label", "Hello Storybook")}
  </button>
);

// Knobs as dynamic variables.
export const asDynamicVariables = () => {
  const name = text("Name", "James");
  const age = number("Age", 35);
  const content = `I am ${name} and I'm ${age} years old.`;

  return <div>{content}</div>;
};
```

図7.5: knobs



7.2.4 @storybook/addon-viewport

Storybook Addon Viewport[11]を使うと、Storybook上でViewportを切り替えられます。つまり、好きなデバイスサイズで表示を確認できます。

リスト7.19: Storybook Addon Viewport

```
import * as React from "react";
import Button from "./Button";
import { INITIAL_VIEWPORTS } from "@storybook/addon-
viewport";

export default {
  title: "Button",
  component: Button,
  parameters: {
    viewport: {
      viewports: INITIAL_VIEWPORTS,
    },
  },
};

export const ButtonComponent = () => (
```

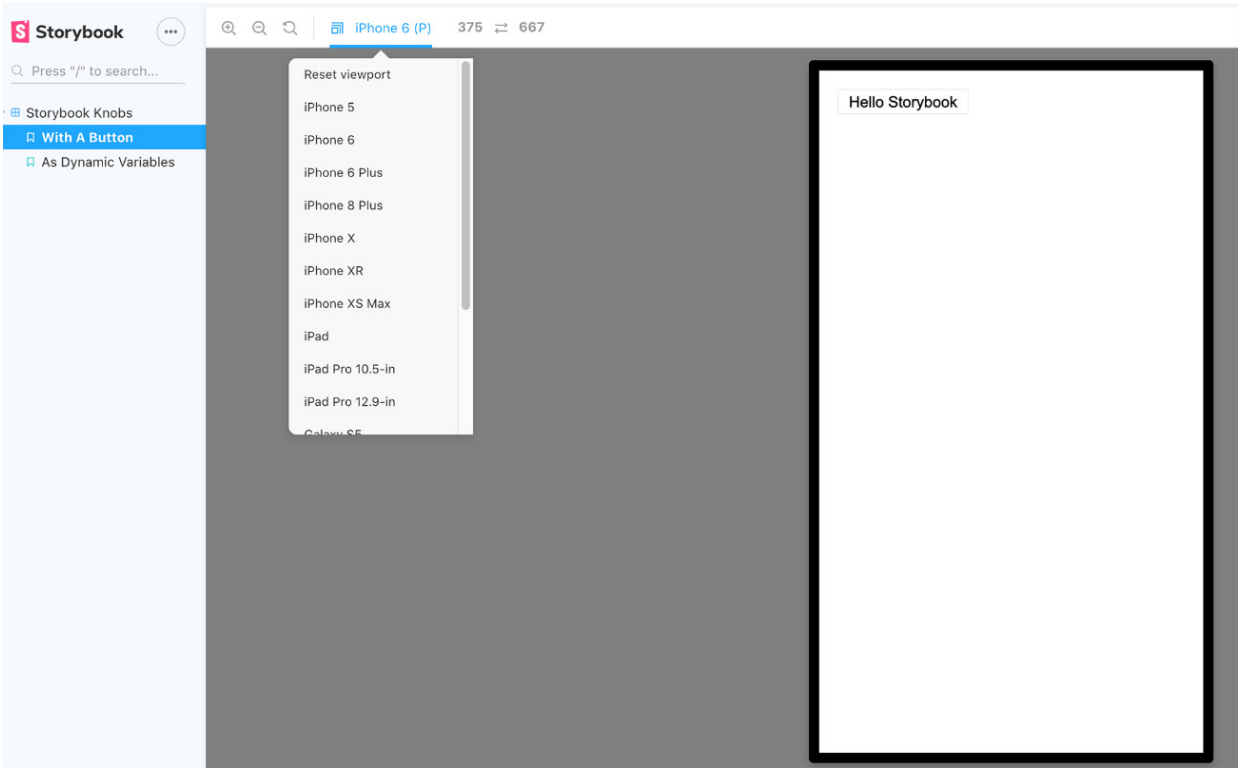
```
    <Button>
      HeyHeyHey
    </Button>
  );

  ButtonComponent.story = {
    name: "default",
  };
```

また、利用するためには、設定ファイルでの指定も必要です。

リスト7.20: .storybook/main.js

```
  module.exports = {
    addons: [
      ...
      "@storybook/addon-viewport", // <= 追加
    ],
    ...
  };
```

図7.6: addon-viewport


Storybook
Press "/" to search...
Storybook Knobs
With A Button
As Dynamic Variables
iPhone 6 (P)
375
667
Reset viewport
iPhone 5
iPhone 6
iPhone 6 Plus
iPhone 8 Plus
iPhone X
iPhone XR
iPhone XS Max
iPad
iPad Pro 10.5-in
iPad Pro 12.9-in
Hello Storybook

7.3　まとめ

　Storybookはコンポーネントカタログです。addonが充実しており、UIサンドボックスとしても扱えます。たとえばコンポーネントに好きな値を注入したり、デバイスサイズを変えたときの挙動を確認できます。またdocumentationにもサポートしており、TypeScriptの型から、コンポーネントのI/Fを記述することもできます。

1. https://storybook.js.org/
2. もし利用フレームワークの判別がうまくいかなければ、手動でセットできます。また、npxコマンドを使うのであれば、手元にこのコマンドをinstallする必要はありません。
3. https://storybook.js.org/docs/formats/component-story-format/
4. https://storybook.js.org/docs/configurations/add-custom-body/
5. https://storybook.js.org/docs/configurations/options-parameter
6. https://storybook.js.org/docs/addons/introduction/
7. https://storybook.js.org/docs/addons/using-addons/
8. https://www.npmjs.com/package/@storybook/addon-info
9. https://www.npmjs.com/package/@storybook/addon-actions
10. https://github.com/storybookjs/storybook/tree/next/addons/knobs
11. https://github.com/storybookjs/storybook/tree/next/addons/viewport

第8章　Jestを使ったテスト

Jest[1]は、JavaScript Testing Frameworkです。Testing Frameworkの名の通り、テストランナー・モック・カバレッジなど、テストを行うために必要な機能が一式でそろっています。Jestの登場以前は、それぞれの役割ごとにライブラリを入れていました。それがJest単体でテスト環境を揃えられるようになり、テスト環境の構築がしやすくなりました。[2]

個人的な意見ですが、テストは絶対に書いた方が良いので、テスト環境構築もそれはそれでやるべきだと思っています。Jestの設定は、BabelやTypeScriptの設定に依存するところもあります。そのため、ソースコードが育ってからテスト環境を構築しようとすると、つまるポイントも出てきます。そこで、ビルド環境を構築するときに一緒にやることを推奨します。

8.1　Jestの使い方

　それでは、さっそくテスト環境を構築しましょう。Jestをインストールします。

```
$ npm install -D jest
```

　これでjestコマンドが使えるようになりました。このときいきなりjestコマンドを実行すると、次のようなエラーが出ます。

```
$ npx jest
No tests found, exiting with code 1

  3 files checked.
   testMatch:  **/__tests__/**/*.[jt]s?(x),  **/?(*.)+(spec|test).[tj]s?(x) - 0 matches
  testPathIgnorePatterns: /node_modules/ - 3 matches
  testRegex:  - 0 matches
Pattern:  - 0 matches
```

　当然ですが、テストコードを作っていないので失敗します。デフォルトでは**/__tests__/**/*.[jt]s?(x), **/?(*.)+(spec|test).[tj]s?(x) に該当するファイルがテストされます。つまり、__test__に入っているファイルか、hoge.test.jsのようなファイルが要求されます。そこで、main.test.jsというファイルを作ってテストしてみましょう。

8.1.1　テストコードを書く

テスト対象としてmain.jsにこのような1を返す関数を定義します。

リスト8.1: main.js

```
export const returnOne = () => {
  return 1;
};
```

この関数に対するテストを作成します。

リスト8.2: main.test.js

```
import { returnOne } from "./main";

it("should be 1", () => {
  expect(returnOne()).toBe(1);
});
```

returnOneという関数を実行したら、1が返ることをテストしています。このテストを実行すると、次の通りに失敗します。

```
$ npx jest
 FAIL  src/main.test.js
  ● Test suite failed to run
                                                                    ({"Object.
<anonymous>":function(module,exports,require,__dirname,__
filename,global,jest){import { returnOne } from "./main";

    SyntaxError: Unexpected token {

Test Suites: 1 failed, 1 total
```

```
Time:        2.987s
```

ここでのエラーの原因は、依存解決です。今のままでは依存解決の仕組みが何もないので、importを解決できていません。そこで、CommonJS形式で書き直してみましょう。Node.jsは、requireを使って依存解決ができます。

リスト8.3: テスト対象

```
const returnOne = () => {
  return 1;
};

module.exports = returnOne;
```

リスト8.4: テストコード

```
const returnOne = require("./main.js");

it("should be 1", () => {
  expect(returnOne()).toBe(1);
});
```

テストを実行してみましょう。

```
$ npx jest
 PASS  src/main.test.js
  ✓ should be 1 (3ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
```

```
Time:        1.876s
Ran all test suites.
```

このように、テストに成功しました。

8.1.2　initで設定を生成する

先ほどimportが解決できなかった問題は、Jestの設定を施すことで解決できます。そこで、Jestの設定について見ていきましょう。設定ファイルは、次のコマンドで生成できます。

```
$ npx jest --init
```

ここでは次のことが聞かれます。
・test コマンドでテストを実行できるようにしますか？
・テストの実行環境は、Node.jsかブラウザどちらですか？
・カバレッジレポートを作りますか？
・テストの実行ごとにモックをクリアしますか？
これに回答すれば、次のような設定ファイルが生成されます（※誌面の都合で大幅に省略しています）。

リスト8.5: jest.config.js

```
module.exports = {
  clearMocks: true,
  coverageDirectory: "coverage",
  // preset: undefined,
};
```

それでは、この設定ファイルを利用して、import文を使えるようにしましょう。

8.1.3 ES6への対応

import文はES6からの構文なので、どんな環境でも使えるわけではありません。そのためBabelを使って、テスト環境であるNode.jsの形式（CommonJS形式）に変換します。[3]BabelとJestの連携のためには、babel-jestを使います。

しかし、最近のJestはbabel-jestが標準で組み込まれているため、新しくダウンロードする必要はありません。そのため本当は設定ファイルを書かずともテストを通すことができました。[4]では、なぜ先ほどはimportの解決がうまくいかなかったかというと、肝心のBabelが入っていなかったためです。**そのため、この設定ファイルがなくても、Babelさえインストールしていれば先ほどのテストは成功していました。**

8.1.4 TypeScriptへの対応

TypeScriptをJestで用いる際も、設定が必要です。Babelを使ったときでも問題になりましたが、TypeScriptでテストを書く際も、モジュールの解決が問題になります。babel-jestは標準利用できるため、**BabelでTypeScriptの依存を解決していれば**エラーが起きません。tscを利用している場合は、依存解決が問題となります。そこで、babel-jestのような役割を持つts-jestを利用します。

ts-jest

ts-jestは、**JestのTypeScript preprocessor**です。

```
$ npm install -D ts-jest
```

設定ファイルのpresetに指定すれば、Jest上でTypeScriptを利用できます。よく似た設定にtransformというものがありますが、それについてはあとで見ていきます。

リスト8.6: ts-jestをセット

```
preset: "ts-jest"
```

jestの型定義ファイルを設定

テストコードをTypeScriptで書くと、Jestの型定義ファイルを入れないとエディタ上で警告が出ます（実行はできます）。そこで、型定義ファイルをダウンロードします。

```
$ npm install -D @types/jest
```

これで、TypeScriptを利用したJestの環境構築ができました。試しに実行してみます。テスト対象は、先ほど作ったファイルの拡張子を.tsに変更してください。

リスト8.7: main.ts

```
export const returnOne = () => {
  return 1;
};
```

リスト8.8: main.test.ts

```
import { returnOne } from './main'

it("should be 1", () => {
```

```
    expect(returnOne()).toBe(1);
  });
```

```
$ npx jest
PASS  ./main.test.ts
  ✓ should be 1 (2 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.526 s, estimated 2 s
Ran all test suites.
```

8.2　Reactのテスト

　次にコンポーネントのテストをする設定を行います。Reactのソースコードをテストするために、testing-library/reactというライブラリを利用します。

8.2.1　testing-library/react

　testing-library/reactは、ページ単位での結合テストを可能にするライブラリです。[5]「このボタンをクリックしたときに、この表示がこうなる」「このページにアクセスすると、APIからこのような値が返ってきて、このように表示される」といったテストケースを作ることができます。ユースケースそのものをテストケースにできるため、ケース数を抑えつつも、実態に沿ったケースを網羅していけます。[6]
　そのようなテストを行うために、まずコンポーネントの取得やイベントを発火させるユーティリティである@testing-library/reactをインストールします。[7]

リスト8.9: react testing libraryのinstall

```
npm install -D @testing-library/react
```

8.2.2　jest-dom

　さらに、そのテストの結果比較のために使うcustom matcherを含んだjest-domを導入します。

リスト8.10: jest-domのinstall

```
npm install -D @testing-library/jest-dom
```

　このライブラリによって、toHaveTextContentやtoContainHTMLといった要素に対する、matcherが利用可能になります。

8.2.3　mountのテスト

　このようなコードをテストしましょう。マウント時に表示が変わるコードです。

リスト8.11: マウント時に表示が変わるcode

```
import * as React from "react";

interface State {
  text: string;
}

class App extends React.Component<{}, State> {
  state = {
    text: "initial text"
  };

  componentDidMount() {
    this.setState({ text: "mounted text" });
  }

  render() {
    return (
      <div>
        <p data-testid="text">{this.state.text}</p>
      </div>
    );
  }
}
```

```
export default App;
```

これがマウント時に"mounted text"と表示されるかをテストします。まず、テスト対象を取得できるQueryを作成します。

リスト8.12: セレクタを取得

```
import { render } from "@testing-library/react";

const { getByTestId } = render(<App></App>);
```

ここで得られるgetByTestIdQueryに、該当するDOMのidを渡すとその要素を取得できます。

リスト8.13: 要素を取得

```
getByTestId("text")
```

この戻り値を、カスタムマッチャでテストします。

リスト8.14: テキストをテスト

```
expect(getByTestId("text")).toHaveTextContent("mounted
text");
```

このテストコードを実行します。

リスト8.15: テストに成功

```
$ npx jest
 PASS  src/main.test.tsx
  ✓ mounted text (29ms)
```

```
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        3.47s
Ran all test suites.
  Done in 4.89s.
```

このように、react-testing-libraryとjest-domを用いて、コンポーネントのふるまいをテストできました。

8.2.4 eventをテスト

DOMの操作についてもテストしてみましょう。このようなコードがあるとします。

リスト8.16: ボタンで状態を切り替えるコード

```
import * as React from "react";

interface State {
  text: string;
  isClicked: boolean;
}

class App extends React.Component<{}, State> {
  state = {
    text: "initial text",
    isClicked: false
  };

  componentDidMount() {
    this.setState({ text: "mounted text" });
  }

  handleClick = () => {
              this.setState({    ...this.state,    isClicked:
!this.state.isClicked });
```

```
  };

  render() {
    return (
      <div>
        <p data-testid="text">{this.state.text}</p>
                <button  onClick={this.handleClick}  data-
testid="btn">
          CLICK
        </button>
        <p>
          status:
          <span data-testid="clicked-status">
            {this.state.isClicked ? "on" : "off"}
          </span>
        </p>
      </div>
    );
  }
}

export default App;
```

この画面は、ボタンのクリックに応じて要素の表記を切り替えられます。このコードを用いて、クリックしたときに表記が切り替わるのかをテストします。

まず、テスト対象を取得できるQueryを作成します。

リスト8.17: セレクタを取得

```
import { render } from "@testing-library/react";

const { getByTestId } = render(<App></App>);
```

そして、クリックイベントを発生させます。これは、fireEventという@testing-library/reactの組込み関数を利用し、その要素でクリック

イベントを発火させます。

リスト8.18: イベントを発行

```
import { fireEvent } from "@testing-library/react";

fireEvent(
  getByTestId("btn"),
  new MouseEvent("click", {
    bubbles: true,
    cancelable: true
  })
);
```

この結果は、マウントのときのテストと同じようにテストできます。ここではクリックした結果、表記がoffからonになっていることをテストします。

リスト8.19: テスト

```
expect(getByTestId("clicked-
status")).toHaveTextContent("on");
```

テストコードを実行すると、このようになります。

リスト8.20: テストに成功

```
$ npx jest
 PASS  src/main.test.tsx
  ✓ mounted text (34ms)
  ✓ click once (18ms)
  ✓ click twice (7ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        3.679s
```

```
Ran all test suites.
  Done in 5.17s.
```

成功しました。全体としては、このようなコードになります。

リスト8.21: テストコード

```
import React from "react";
import { render, fireEvent } from "@testing-
library/react";
import "@testing-library/jest-dom/extend-expect";
import App from "./App";

it("mounted text", () => {
  const { getByTestId } = render(<App></App>);
     expect(getByTestId("text")).toHaveTextContent("mounted
text");
});

it("click once", () => {
  const { getByTestId } = render(<App></App>);
  fireEvent(
    getByTestId("btn"),
    new MouseEvent("click", {
      bubbles: true,
      cancelable: true
    })
  );
                                expect(getByTestId("clicked-
status")).toHaveTextContent("on");
});

it("click twice", () => {
  const { getByTestId } = render(<App></App>);
  fireEvent(
    getByTestId("btn"),
    new MouseEvent("click", {
      bubbles: true,
```

```
        cancelable: true
      })
    );
    fireEvent(
      getByTestId("btn"),
      new MouseEvent("click", {
        bubbles: true,
        cancelable: true
      })
    );
                                    expect(getByTestId("clicked-
status")).toHaveTextContent("off");
});
```

@testing-library/reactを利用すると、「ユーザーがこのような操作をしたときに画面はこうなる」といったことをテストできるため、シナリオベースでのテストがしやすいです。UIのテストは難しいとされますが、シナリオベースでのテストは比較的書きやすいです。

8.3　Jestで設定する項目

　Jestの設定項目は多岐に渡ります。そのすべてを説明することは難しく、その必要もありません。そこで、重要な概念とテストが実行できないときの原因になりやすい項目について挙げます。

8.3.1　transform

　transformerとの対応を宣言できます。transformerとは、ソースコードを変換する同期的な関数です。平たくいうと、トランスパイラとそれに対応するファイルパスを指定できます。
　たとえば、次のように指定できます。

リスト8.22: jest.config.js

```
...,

"transform": {
  "^.+\\.[t|j]sx?$": "babel-jest"
},
...
```

　ここで、babel-jestはbabelを使った変換を行うtransformerです。上記の設定では、.js、.jsx、.ts、.tsxに対して、babel-jestによる変換を行っています。

8.3.2　preset

　prestはjest configのベースです。ts-jestでは、presetの内部でtransformerが設定されるため、TypeScriptへの変換を行いたいとき

によく使うオプションです。presetを使うとtransformだけでなく、testMatchとmoduleFileExtensionsも設定されます。[8]
　そのため、transformの設定で行っていた設定は、こちらのpresetに書くことでも実現可能です。

8.3.3　testMatch

　テスト対象を指定できます。デフォルトでは**/__tests__/**/*.[jt]s?(x)、**/?(*.)+(spec|test).[tj]s?(x) が指定されています。

8.3.4　moduleNameMapper

　画像やスタイルといった静的アセットのstubへの対応です。test対象に画像などが含まれていたとき、テストはそのバイナリをjsとして解釈しようとするため失敗します。それを防ぐために、そのファイルをモックとして置き換えることができます。そのため、この設定ではファイルパスとモックへのパスを書いておきます。

リスト8.23: jest.config.js

```
  moduleNameMapper: {
                                                          "\\.
  (jpg|jpeg|png|gif|eot|otf|webp|svg|ttf|woff|woff2|mp4|webm
  |wav|mp3|m4a|aac|oga)$":
       "<rootDir>/__mocks__/fileMock.js",
     "\\.(css|less)$": "<rootDir>/__mocks__/styleMock.js",
  },
```

リスト8.24: __mocks__/styleMock.js

```
  module.exports = {};
```

8.4　まとめ

　Jestはテストランナーです。テストの実行環境だけでなく、テストの構造やMacher、Mock機能までも提供してくれます。あくまでもJavaScriptのテスト環境であるため、TypeScriptプロジェクトの場合はpre processorとの連携が必要です。testing-libraryシリーズやjest-domを利用することで、UIに関するテストも行えます。テストの導入は後回しにするほどつらくなりがちですので、環境構築のタイミングで一緒に入れてしまうことをお勧めします。

1. https://jestjs.io
2. Jestの登場以前、僕の友人はJSのテスト環境を「モカ・チャイ・ジャスミンって、コーヒーチェーン店のオプションかよ！」と叫んでいました。Mocha: https://mochajs.org/ 、Chai: https://www.chaijs.com/ 、Jasmine: https://jasmine.github.io/
3. BabelはNode.js形式に変換するので、ブラウザで実行ができない。そのため、webpackといったライブラリがビルド設定に必要だったわけです。
4. https://github.com/facebook/jest/tree/master/packages/babel-jest
5. https://github.com/testing-library/react-testing-library
6. 一方で、コンポーネント単位でテストを書けるenzymeというライブラリも有名です。https://github.com/enzymejs/enzyme
7. DOM Testing Libraryのラッパです。
8. https://github.com/kulshekhar/ts-jest/blob/61d31b48dc633bb415eb2d0d39f97dab771327c4/src/config/create-jest-preset.ts

第9章　0から環境を作ってみる

本章ではこれまで学んだことを元に、通しで0から環境構築をしてみます。TypeScriptとReactを使ったアプリケーションのビルドを行います。サンプルリポジトリはこちらです。[1]

9.1　Node.jsの設定

　Node.jsをnvmからinstallします。執筆時点ではv14が使えましたが、出たばかりということもあり、v12を使っています。

```
$ nvm install v12

$ node -v
v12.16.3
```

　Node.jsにはnpmコマンドが付属しています。最初に、npmコマンドでプロジェクトを作ります。

```
$ npm init

{
  "name": "practice",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo ¥"Error: no test specified¥" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

これを実行すると、package.jsonファイルが作成されます。このファイルでプロジェクトを管理します。

9.2　TypeScriptの設定

　次に、TypeScriptを導入します。開発ツールなので、インストール時には-Dオプションを付けます。

```
$ npm install -D typescript
```

　これでnpx tsc (TypeScript Compilierコマンド)が使えるようになり、ビルドができるようになりました。global installしていないため、npxをつける必要があります。次に、TypeScriptの設定を作ります。

```
$ npx tsc --init
```

　これで、設定ファイルが生成されました。これからReactを利用するので、jsxオプションをreactに変更しておきます。

リスト9.1: tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "jsx": "react",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
```

```
    }
}
```

9.3　webpackの設定

　それでは、Reactプロジェクトをビルドできるように、webpackの設定を進めていきます。webpack本体とそれを実行するためのCLIツールを入れます。

```
$ npm install -D wepback webpack-cli
```

　これで、npx webpackコマンドを使ってビルドできるようになりました。次に、webpack経由でTypeScriptをビルドできるように、プリプロセッサーであるts-loaderを入れます。

```
$ npm install -D ts-loader
```

　では、それらを使うように設定ファイルを書いていきましょう。設定ファイルの名前は、webpack.config.jsです。

リスト9.2: webpack.config.js

```
const path = require("path");

module.exports = {
  mode: process.env.NODE_ENV,
  entry: "./src/index.tsx",
  output: {
    path: path.resolve(__dirname, "./dist"),
    filename: "build.js",
```

```
  },
  module: {
    rules: [
      {
        test: /\.(ts|tsx)$/,
        use: [
          {
            loader: "ts-loader",
          },
        ],
      },
    ],
  },
};
```

Reactプロジェクトをビルドできるようになったため、次にReactの設定をしましょう。

```
$ npm install react react-dom
```

TypeScriptを利用するため、型定義ファイルも入れます。型定義ファイルは実行時には不要なので、-Dオプションを付けます。

```
$ npm install -D @types/react @types/react-dom
```

ビルドのエントリポイントとして指定したsrc/index.tsxにコードを書きましょう。

リスト9.3: src/index.tsx

```
import * as React from "react";
import * as ReactDOM from "react-dom";

ReactDOM.render(<div>hello                          world</div>,
document.getElementById("root"));
```

ビルドします。

```
$ npx webpack
  Asset     Size  Chunks             Chunk Names
build.js  128 KiB       0  [emitted]  main
Entrypoint main = build.js
[2] ./src/index.tsx 1.08 KiB {0} [built]
```

無事、ビルドしたものがdistディレクトリに吐き出されました。では、これをHTMLで確認できるようにしましょう。そのためには、html-webpack-pluginを使います。

```
$ npm install -D html-webpack-plugin
```

html-webpack-pluginは、webpackでバンドルしたファイルを読み込むHTMLファイルを作るライブラリです。元になるHTMLファイルを指定するだけで、ビルド生成先のフォルダにHTMLファイルも作ってくれます。
次に、このようなテンプレートとなるHTMLファイルを用意します。

リスト9.4: src/index.html

```
<!DOCTYPE html>
<html>
```

```
    <head>
      <meta charset="UTF-8" />
      <title>js-build-book-support</title>
    </head>
    <body>
      <div id="root"></div>
    </body>
  </html>
```

ReactアプリケーションはReactDOMの設定でid=rootを持つ要素にmountするように設定しているため、その要素を作りました。このファイルを使ったwebpack.config.jsは次の通りです。

リスト9.5: webpack.config.js

```
  const path = require("path");
  const HtmlWebpackPlugin = require("html-webpack-plugin");

  module.exports = {
    mode: process.env.NODE_ENV,
    entry: "./src/index.tsx",
    output: {
      path: path.resolve(__dirname, "./dist"),
      filename: "build.js",
    },
    module: {
      rules: [
        {
          test: /\.(ts|tsx)$/,
          use: [
            {
              loader: "ts-loader",
            },
          ],
        },
      ],
    },
          plugins:    [new    HtmlWebpackPlugin({    template:
```

```
  "./src/index.html" })],
  };
```

これをビルドします。

```
$ npx webpack
    Asset       Size  Chunks             Chunk Names
 build.js    128 KiB       0  [emitted]  main
index.html  166 bytes          [emitted]
Entrypoint main = build.js
[2] ./src/index.tsx 1.08 KiB {0} [built]
   + 7 hidden modules
```

無事に成功しました。ファイルの成果物を確認してみましょう。

```
$ open dist/index.html
```

ちゃんと表示されました。

図9.1: ビルドされた例

ビルドの確認のたびに毎回HTMLを開くのも億劫なので、開発サーバを立てましょう。weppack-dev-serverはlive reloading機能を備えた開発用サーバです。コードの変更を即座に反映してくれる利点があります。

```
$ npm install -D webpack-dev-server
```

起動しましょう。

```
$ npx webpack-dev-server
```

これで表示されるようになりました。src/index.tsxに変更を加えたら、即時に反映されることも確認できます。

次に、静的ファイルもReactで扱えるような設定をします。プロジェクトによっては、reset.cssのようなグローバルなCSSを読み込みます。そこで外部CSSを読み込めるようにしましょう。

そのためには、css-loaderとstyle-loaderを使います。css-loaderはJSにCSSファイルそのものを文字列として読み込む機能を提供し、style-loaderはそれをstyleタグとして挿入してくれる機能を提供します。

```
$ npm install -D css-loader style-loader
```

リスト9.6: webpack.config.js

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {
  mode: process.env.NODE_ENV,
  entry: "./src/index.tsx",
  output: {
    path: path.resolve(__dirname, "./dist"),
    filename: "build.js",
  },
  module: {
    rules: [
      {
        test: /\.(ts|tsx)$/,
        use: [
          {
            loader: "ts-loader",
          },
        ],
      },
      {
```

```
        test: /\.(css)$/,
        use: ["style-loader", "css-loader"],
      },
    ],
  },
        plugins:    [new    HtmlWebpackPlugin({    template:
"./src/index.html" })],
};
```

では、このようなreset.cssを読み込んでみましょう。

リスト9.7: src/reset.css

```
*,
*:after,
*:before {
  margin: 0;
  padding: 0;
  box-sizing: inherit;
}

html {
  font-size: 62.5%;
}

body {
  box-sizing: border-box;
}
```

リスト9.8: src/index.tsx

```
import * as React from "react";
import * as ReactDOM from "react-dom";
import "./reset.css";

ReactDOM.render(<div>hello                          world</div>,
document.getElementById("root"));
```

ついでに、画像ファイルも読み込めるようにしましょう。静的assetを読み込めるfile-loaderを使います。

```
$ npm install -D file-loader
```

loaderにはfile-loaderを指定します。

リスト9.9: webpack.config.js

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {
  mode: process.env.NODE_ENV,
  entry: "./src/index.tsx",
  output: {
    path: path.resolve(__dirname, "./dist"),
    filename: "build.js",
  },
  module: {
    rules: [
      {
        test: /\.(ts|tsx)$/,
        use: [
          {
            loader: "ts-loader",
          },
        ],
      },
      {
        test: /\.(css)$/,
        use: ["style-loader", "css-loader"],
      },
      {
        test: /\.(png)$/,
        use: ["file-loader"],
```

```
        },
      ],
    },
          plugins:    [new    HtmlWebpackPlugin({    template:
"./src/index.html" })],
};
```

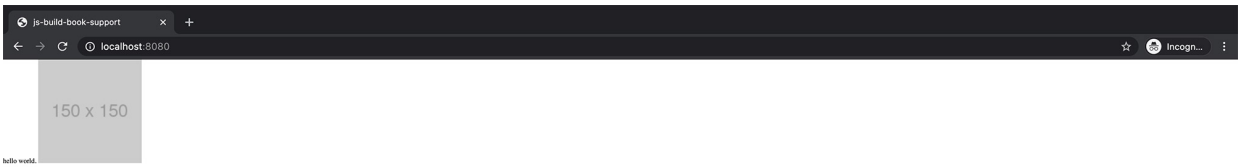
JSX内で画像を読み込みます。imgタグでは、ファイルパスではなくimportしたデータを読み込みます。また、このときTypeScriptが警告を出しますが、説明の都合上いったん目をつむります。

リスト9.10: src/index.tsx

```
import * as React from "react";
import * as ReactDOM from "react-dom";
import Img from "./example.png";
import "./reset.css";

ReactDOM.render(
  <div>
    hello world. <img src={Img}></img>
  </div>,
  document.getElementById("root")
);
```

ビルドすると画像が表示されます。

図9.2: ビルドされた例



先ほど出たであろう警告は、Cannot find module './example.png'. といったものですが、この解決のために次のようなアンビエント宣言したファイルを追加してください。型定義ファイルを自作します。

リスト9.11: import-file.d.ts

```
declare module "*.png";
```

これはAmbient Modulesの短縮記法であり、.png がついたファイルからany型でimportできます。このファイルはどの階層においても問題ありません。

これでTypeScript+webpackの設定が終わりました。次は、開発支援ツールの設定をしていきましょう。

9.4　Prettierの設定

　コードフォーマッターであるPrettierをインストールします。

```
$ npm install -D prettier
```

　.prettierrc.jsといったファイルに整形ルールを書くことができますが、いまはデフォルトルールのまま使う予定なので、ファイルは作りません。
　フォーマットは次のコマンドで行います。

```
$ npx prettier --write 'src/**/*.{js,ts,jsx,tsx}'
```

　--writeは、フォーマットした結果をファイルに書き込むオプションです。設定しなければ、整形結果が標準出力に流れるだけです。設定しない場合でもフォーマットの不備に気付けるので、CI上でのテスト目的に使えます。
　'src/**/*.{js,ts,jsx,tsx}'は、フォーマット対象をglob形式で表しています。glob形式を使う場合は、''(シングルクオート)が必要なので注意しましょう。

9.5　ESLintの設定

次は、linterの設定をします。

```
$ npm install -D eslint
```

ESLintは、--initオプションで設定ファイルを作成できます。作成したいファイルについて質問をされるので、それに答えていきましょう。

```
$ npx eslint --init
? How would you like to use ESLint? To check syntax and find problems
? What type of modules does your project use? JavaScript modules (import/export)
? Which framework does your project use? React
? Does your project use TypeScript? Yes
? Where does your code run? Browser
? What format do you want your config file to be in? JavaScript
The config that you've selected requires the following dependencies:

eslint-plugin-react@latest @typescript-eslint/eslint-plugin@latest @typescript-eslint/parser@latest
? Would you like to install them now with npm? Yes
```

準備ができたので実行します。あらかじめlintに引っかかるコードを追加して、動作を確かめてみましょう。

リスト9.12: src/index.tsx

```
while (true) {
  break;
}
```

このコードは、eslintにエラーとして識別されます。ESLintのrecommendedルールでは、whileの条件に定数は使えないためです。

```
$ npx eslint 'src/**/*.{ts,tsx}'
  1:8  error  Unexpected constant condition  no-constant-condition

1 problem (1 error, 0 warnings)
```

これにより、きちんとルールが動作することを確認できました。エラーが出ることを確かめるために、追加したコードは消しておきましょう。

9.6　ESLintとPrettierを共存させる設定

　これでESLintとPrettierの設定が終わりました。しかし、ルールの設定次第によっては、お互いの設定が衝突してしまいます。そのため、フォーマットはPrettierが行い、それをESLintから実行するという仕組みを作ります。

　そのために、eslint-config-prettierとeslint-plugin-prettierを使います。eslint-config-prettierはeslintでのstyleに関するルールを全部offにします。eslint-plugin-prettierは、eslintからprettierを実行します。

```
$ npm install -D eslint-config-prettier eslint-plugin-prettier
```

　そして、それらの設定を.eslintrc.jsへ加えます。

リスト9.13: .eslintrc.js

```
module.exports = {
  env: {
    browser: true,
    es6: true,
  },
  extends: [
    "eslint:recommended",
    "plugin:react/recommended",
    "plugin:@typescript-eslint/eslint-recommended",
    "plugin:@typescript-eslint/recommended",
    "plugin:prettier/recommended", // <- 追加
    "prettier/@typescript-eslint", // <- 追加
  ],
```

```
    globals: {
      Atomics: "readonly",
      SharedArrayBuffer: "readonly",
    },
    parser: "@typescript-eslint/parser",
    parserOptions: {
      ecmaFeatures: {
        jsx: true,
      },
      ecmaVersion: 11,
      sourceType: "module",
    },
    plugins: ["react", "@typescript-eslint"],
    rules: {},
  };
```

ここではpluginsにeslint-plugin-prettierが設定されていませんが、問題なく動きます。なぜならば、plugin:prettier/recommendedというextendsが、中でpluginを入れているためです。ただし、このextendsはeslint-plugin-prettierが提供しているため、eslint-plugin-prettierは入れる必要があります。

plugin:prettier/recommendedはeslint-plugin-prettierを入れてくれ、prettierのエラーを eslintのエラーとして扱ってくれる役割も持ちます。さらにeslint-config-prettierをextendsに追加もしてくれ、ESLintにおけるスタイルチェックのルールをOFFにしてくれます。また、最後に追加しているprettier/@typescript-eslintは、TypeScript用のルールにおけるスタイルをOFFにしてくれるものです。@typescript-eslint/eslint-pluginで追加されるルールのうち、スタイルにおけるルールをOFFにしてくれます。これが動作するか、試してみましょう。

tsxファイルのスタイルを適当に崩します。

リスト9.14: src/index.tsx

```
  import * as React from "react";
  // スタイルを崩す
```

```
  import * as ReactDOM from "react-dom";
import Img from "./example.png";
import "./reset.css";

ReactDOM.render(
  <div>
    hello world. <img src={Img}></img>
  </div>,
  document.getElementById("root")
);
```

2行目を崩しました。そしてeslintを実行します。

# 警告を出すために--fixを意図的に外しています。
$ npx eslint 'src/**/*.{ts,tsx}'

2:1 error Delete `··` prettier/prettier

1 error and 0 warnings potentially fixable with the `--fix` option.

すると、Delete`··`eslintprettier/prettierといった警告が表示されるようになりました。

これで、ESLintとPrettierの共存ができました。

9.7　Jestの設定

次にテストを書きましょう。テストランナーにはjestを使います。

```
$ npm install -D jest @types/jest
```

jestの型定義ファイルを入れることで、テスト自体もTypeScriptで書いていけます。TypeScriptを実行するために、preprocessorを導入します。それがts-jestです。

```
$ npm install -D ts-jest
```

ではjest上でts-jestを使ってテストを実行できるように、設定ファイルを書いていきましょう。設定ファイルの生成は、initオプションから行えます。ここでも質問が表示されるので、それに答えましょう。

```
$ npx jest --init

The following questions will help Jest to create a suitable confi
guration for your project

  Would you like to use Jest when running "test" script in "pac
kage.json"? … no
```

```
  Choose the test environment that will be used for testing › jsdom (browser-like)
  Do you want Jest to add coverage reports? … yes
  Automatically clear mock calls and instances between every test? … yes

  Done in 12.33s.
```

　実行すると、このような設定ファイルができます（コメントアウトをすべて削除しています）。

リスト9.15: jest.config.js

```
module.exports = {
  clearMocks: true,
  coverageDirectory: "coverage",
};
```

　ではts-jestの設定を加えます。それはpresetという項目を設定することで加えられます。

リスト9.16: jest.config.js

```
module.exports = {
  clearMocks: true,
  coverageDirectory: "coverage",
  preset: "ts-jest",
};
```

　これで実行の準備が整いました。次にテスト対象を作ります。jestの機能をテストするために、このようにコードを書き換えました。

リスト9.17: src/index.tsx

```
import * as React from "react";
import * as ReactDOM from "react-dom";
import { App } from "./app";

ReactDOM.render(<App></App>,
document.getElementById("root"));
```

リスト9.18: src/app.tsx

```
import * as React from "react";
import Img from "./example.png";
import "./reset.css";

// テストしやすいようにわざと作った関数
export const generateHelloWorld = () => {
  return "hello world.";
};

export const App = () => {
  return (
    <div>
      <div>
        {generateHelloWorld()}
        <img src={Img}></img>
      </div>
    </div>
  );
};
```

ついでに、webpackも少し書き換えます。tsxのimport時に拡張子を毎回書くのも億劫なので、webpackの設定で拡張子を不要にします。

リスト9.19: webpack.config.js

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
```

```
module.exports = {
  mode: process.env.NODE_ENV,
  entry: "./src/index.tsx",
  output: {
    path: path.resolve(__dirname, "./dist"),
    filename: "build.js",
  },
  module: {
    rules: [
      {
        test: /\.(ts|tsx)$/,
        use: [
          {
            loader: "ts-loader",
          },
        ],
      },
      {
        test: /\.(css)$/,
        use: ["style-loader", "css-loader"],
      },
      {
        test: /\.(png)$/,
        use: ["file-loader"],
      },
    ],
  },
  // 追加
  resolve: {
    extensions: [".ts", ".js", ".tsx"],
  },
        plugins:    [new    HtmlWebpackPlugin({    template:
"./src/index.html" })],
};
```

presetでts-jestが指定されたプロジェクトでは、jestが*.test.tsと形式のファイルをテストしてくれます。そのため、テストファイルをindex.test.tsと

して作ります。

リスト9.20: index.test.ts

```
import { generateHelloWorld } from "./app";

test("generateHelloWorldがHelloWorldを返す", () => {
  const actual = generateHelloWorld();
  expect(actual).toBe("hello world.");
});
```

これで実行の準備が整いましたが、これを実行すると画像のimportで失敗します。

```
$ npx jest
  SyntaxError: Invalid or unexpected token

    1 | import * as React from "react";
    2 | import * as ReactDOM from "react-dom";
  > 3 | import Img from "./example.png";
      | ^
    4 | import "./reset.css";
    5 |
    6 | export const generateHelloWorld = () => {
```

これは、画像などのバイナリファイルをJSとして解釈しようとしたことによるエラーです。そのため、このような静的アセットはMockのjsで置き換えましょう。それは、moduleNameMapperという設定で実現できます。置き換え対象の拡張子と置き換えるファイルの対応を書きます。

リスト9.21: jest.config.js

```
module.exports = {
  clearMocks: true,
  coverageDirectory: "coverage",
  preset: "ts-jest",
  moduleNameMapper: {
      "\\.(png|css)$": "<rootDir>/__mocks__/fileMock.js",
    }
};
```

リスト9.22: __mocks__/fileMock.js

```
module.exports = {};
```

これでテストを実行しましょう。

```
$ npx jest
 PASS  src/app.test.ts
  ✓ generateHelloWorldがHelloWorldを返す (1 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Ran all test suites.
 Done in 2.77s.
```

成功しました。
ついでに、コンポーネント自体のテストも書きましょう。コンポーネントのテストにはtesting libraryを使います。

```
$ npm install -D @testing-library/react @testing-library/jest-dom
```

---

installした@testing-library/reactは、react用のtesting libaryです。@testing-library/jest-domは、比較に使うcustom matcherを含んだライブラリです。

ここでは、componentがmountされた後にhello world.が表示されることをテストします。テストの対象となるDOMにidとして、data-testid="text"をセットします。

リスト9.23: src/app.tsx

```
import * as React from "react";
import Img from "./example.png";
import "./reset.css";

export const App = () => {
  const [isMounted, setMount] = React.useState(false);
  React.useEffect(() => {
    setMount(true);
  }, []);
  return (
    <div data-testid="text">
      {isMounted && (
        <div>
          {generateHelloWorld()}
          <img src={Img}></img>
        </div>
      )}
    </div>
  );
};
```

このコンポーネントに対するテストは、次のように書きます。

リスト9.24: src/app.test.tsx

```
import * as React from "react";
import { render } from "@testing-library/react";
import "@testing-library/jest-dom/extend-expect";
```

```
import { generateHelloWorld, App } from "./app";

test("mounted text", () => {
  const { getByTestId } = render(<App></App>);
        expect(getByTestId("text")).toHaveTextContent("hello
world.");
});
```

ファイル名が.tsではなく、.tsxになっていることに注意してください。.tsxを使わないと、JSXの構文であることを認識できません。

```
$ npx jest
 PASS  src/app.test.tsx
  ✓ generateHelloWorldがHelloWorldを返す (2 ms)
  ✓ mounted text (30 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Ran all test suites.
  Done in 1.95s.
```

テストを実行すると、このように成功しました。

9.8　StoryBookの設定

　先ほどの節では、コンポーネントを分割して、テストができるようになりました。では、そのコンポーネントをカタログとして管理しましょう。そのためには、Storybookを使います。React用の設定とTypeScript用の設定があるため、設定量は多く、分割しながら環境構築します。
　まず、React用の設定を作ります。StoryBookのCLIツールで、まずはひな型を作ります。

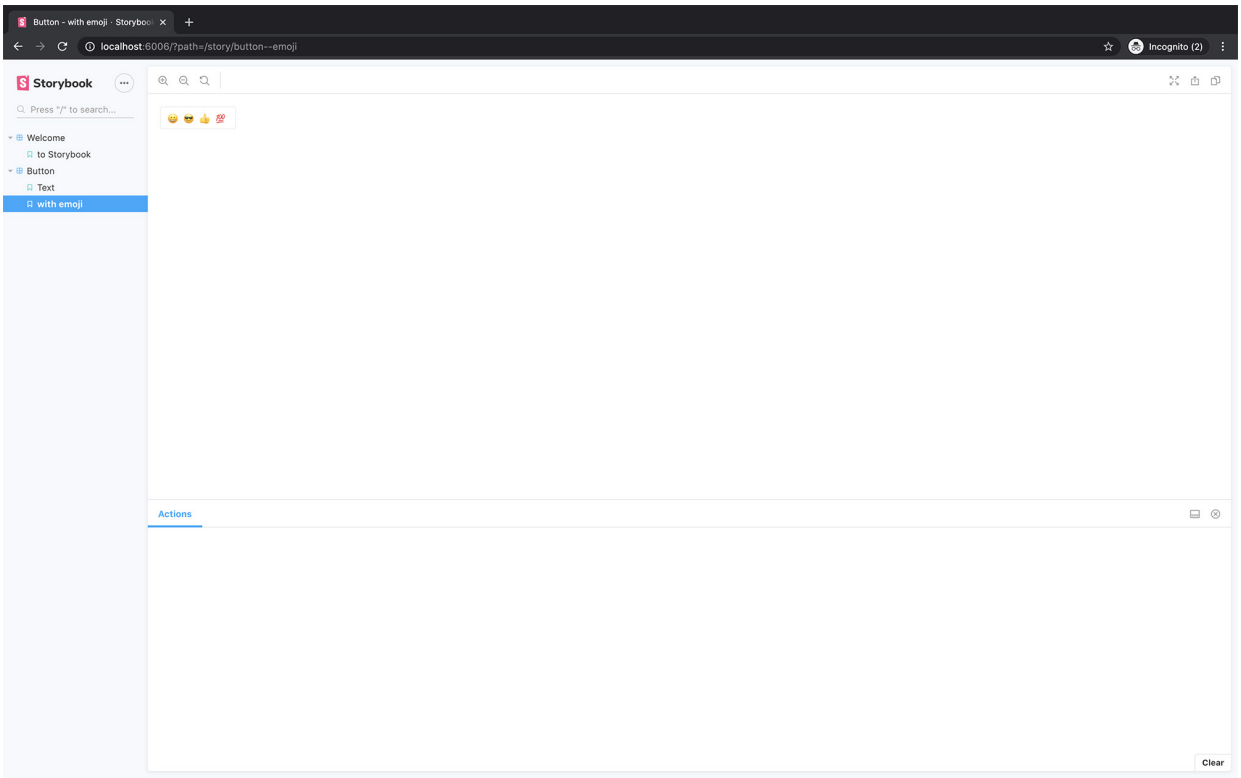
```
$ npx -p @storybook/cli sb init
```

　これで、必要なパッケージのインストールと設定ファイルの作成が終わりました。試しに起動してみます。

```
$ npx storybook
```

　このように表示されました。

図9.3: storybook



では、この上で自作のTypeScript製のコンポーネントが動くようにしましょう。そのためにはStoryBook上でTypeScriptが動くように、ts-loaderを入れてその設定を加えます。

リスト9.25: .storybook/main.js

```
module.exports = {
  stories: ["../src/**/*.stories.tsx"],
  webpackFinal: async (config) => {
    config.module.rules.push({
      test: /\.(ts|tsx)$/,
      use: [
        {
          loader: require.resolve("ts-loader"),
        },
      ],
    });
    config.resolve.extensions.push(".ts", ".tsx");
```

```
    return config;
  },
};
```

story fileは*.stories.tsxという名前で作ります。

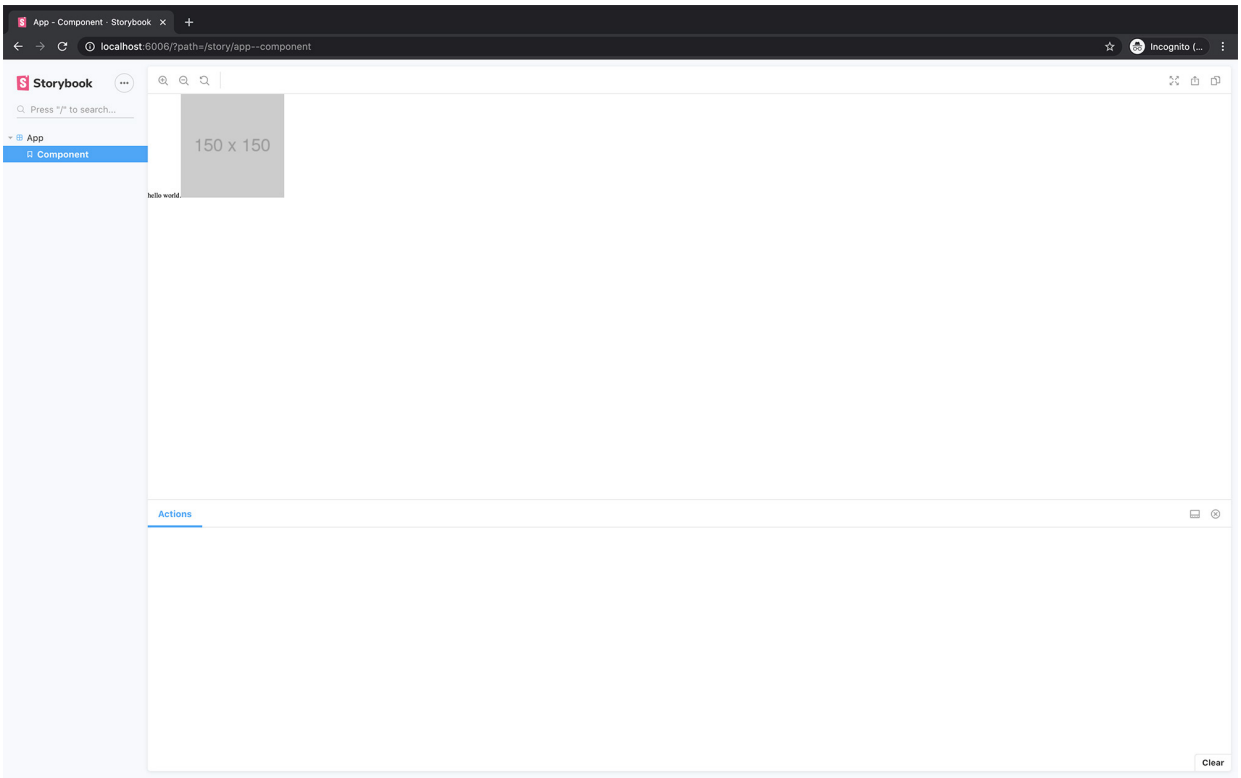
リスト9.26: src/app.stories.tsx

```
import React from "react";
import { App } from "./app";

export default {
  title: "App",
  component: App,
};

export const Component = () => <App></App>;

Component.story = {
  name: "Component",
};
```

これを実行すると、次のように読み込むことができました。

```
$ npx storybook
```

図9.4: storybook


App - Component - Storybook
localhost:6006/?path=/story/app--component
Incognito (...
Storybook
Press "/" to search...
App
Component
150 x 150
hello world
Actions
Clear

9.9　まとめ

　TypeScript+React+webpackで環境構築し、無事にHello Worldを出力できました。さらに、ESLint、Prettier、Jest、Storybookを活用し、開発エコシステムとの連携を取ることもできました。実際に手を動かして構築してみると、意外と簡単だったのではないでしょうか。ただしカスタマイズするとなると、これらの設定を深く理解しなければ詰まるので、ご自身でも素振りとして環境構築をすることをお勧めします。

1. https://github.com/sadnessOjisan/js-build-book-support

付録A　バージョンの追従

A.1　環境構築で増えすぎた依存

　これまでに扱った環境構築の手順をそのまま行うと、package.jsonの依存がとても膨らみます。たとえば、私が普段スターターコードして利用しているpackage.jsonは、このような依存を持っています。

リストA.1: package.json

```
{
  "devDependencies": {
    "@babel/core": "^7.9.0",
    "@storybook/addon-actions": "^5.3.18",
    "@storybook/addon-info": "^5.3.18",
    "@storybook/addon-knobs": "^5.3.18",
    "@storybook/addon-links": "^5.3.18",
    "@storybook/addon-viewport": "^5.3.18",
    "@storybook/addons": "^5.3.18",
    "@storybook/react": "^5.3.18",
    "@testing-library/jest-dom": "^5.3.0",
    "@testing-library/react": "^10.0.1",
    "@types/jest": "^25.1.4",
    "@types/react": "^16.9.25",
    "@types/react-dom": "^16.9.5",
    "@types/styled-components": "^5.0.1",
    "@typescript-eslint/eslint-plugin": "^2.25.0",
    "@typescript-eslint/parser": "^2.25.0",
    "babel-loader": "^8.1.0",
    "css-loader": "^3.4.2",
    "dotenv-webpack": "^1.7.0",
    "eslint": "^6.8.0",
    "eslint-config-prettier": "^6.10.1",
    "eslint-plugin-prettier": "^3.1.2",
    "eslint-plugin-react": "^7.19.0",
    "file-loader": "^6.0.0",
    "html-webpack-plugin": "^4.0.1",
```

```
    "husky": "^4.2.3",
    "jest": "^25.2.0",
    "lint-staged": "^10.0.9",
    "prettier": "^2.0.2",
    "react-docgen-typescript-loader": "^3.7.2",
    "style-loader": "^1.1.3",
    "ts-jest": "^25.2.1",
    "ts-loader": "^6.2.2",
    "typescript": "^3.8.3",
    "webpack": "^4.42.1",
    "webpack-cli": "^3.3.11",
    "webpack-dev-server": "^3.10.3"
  },
  "dependencies": {
    "react": "^16.13.1",
    "react-dom": "^16.13.1",
    "styled-components": "^5.0.1"
  }
}
```

これらのバージョン管理にどう立ち向かうか、本章で補足します。

A.2　package gardening

　依存パッケージのアップデートは、「パッケージにアップデートがあることを検知し、それをアップデートする」といった作業をします。ですが、アップデートした結果、アプリケーションが動かなくなることは避けなければいけません。そして、**バージョンを上げても問題ないことの保証がないと上げづらいです。**しかし、そのために全ページで全パターンを動作確認することは非現実的であり、パッケージのアップデートは一筋縄ではいきません。「毎週のように依存パッケージを上げ続ける努力」といった記事[1]にもあるとおり、努力しなければいけない仕事です。

A.3　CI

そのため、少しでも工数を減らし、安全にアップデートを行う仕組みとして、CIを活用します。ここまでStorybookやtestを整備した理由は、ここにあります。それらをCIサーバで運用すれば、パッケージアップデート後の確認コストを下げられます。

最近では、GitHub Actions(GHA)などのサービスを活用することで、低コスト（金額面・手間面）でCIサーバを運用できます。

A.3.1　automated dependency updates

バージョンを上げ続ける努力をするためには、パッケージのアップデートがあることを検知できなければいけません。これは、automated dependency updatesと呼ばれているツールを使うことで実現できます。この手のツールを利用すると、何かバージョンアップデートがあるたびにBotがバージョンをアップデートするPRを作ってくれます。

たとえば、Dependabot [2]、Snyk [3](Greenkeeper)、Renovate[4]などがあります。

A.3.2　TestとStorybook

依存をアップデートしたときに、既存コードが動かなくなることは避けなければいけません。そのためにjestやStorybookを整備しました。もしテストが失敗したり、ページやコンポーネントがビルドできなければ、そのアップデートは中断すべきです。中断が必要か気付けるためには、日ごろからテストやStorybookを整備しておくことが効果的です。

A.4　各種レポートのホスティングとプレビュー

テストやStorybookを網羅できているかを確かめるために、カバレッジレポートやカタログをどこかにデプロイしておくと良いでしょう。最近では、GitHubと連携してホスティングできるサービスが増えているので、それらを活用できます。

たとえば、Netlify[5]、Vercel[6](Now)、Amplify[7]などはブランチ連携だけでデプロイできるので、適しています。

1. package gardeningという命名もこのブログからいただきました。https://techlog.voyagegroup.com/entry/2016/06/27/080000
2. https://dependabot.com/
3. https://snyk.io/
4. https://renovate.whitesourcesoftware.com/
5. https://www.netlify.com/
6. https://vercel.com/home
7. https://aws.amazon.com/jp/amplify/

さいごに

お疲れさまです。最後までお読みいただき、ありがとうございます。本書でも扱った通り、フロントエンドの開発環境エコシステムには、Package Manager、Transpiler、Bundler、Linter、Formatter、Test Runner、AltJSなどがあります。本書では、それらの代表的なツールや、その周辺プラグインについて紹介しました。しかし、紹介したツールやプラグインはあくまでも筆者独自の選定[1]によるものであり、唯一の選択肢ではありません。そのため、皆さんが環境構築をする場合は、他のツールを使うこともあるかもしれません。また、いわゆる流行り廃りやベストプラクティスの変遷によって、皆さんが本書を読む頃には、紹介したツールが使われていないかもしれません。しかしどのようなライブラリを選定するにしても、役割という観点では共通するはずで、本書での学びが無駄になることはないでしょう。今後新しいライブラリを使っていく上でも、本書で学んだエコシステムの枠組みの知識が活きるはずです。**そのライブラリが何をしているか、そのライブラリがなければなぜ動かないか**ということに着目すれば、難しい環境構築も一歩一歩前進できるはずです。環境構築はつまずきがちな領域ですが、諦めずに少しずつ取り組んでいきましょう。

1. 筆者が独自に選んではいるものの、広く利用実績があるものを選んでいます。

著者紹介

井手 優太（いで ゆうた）

約3年間、大手事業会社にて営業や開発を経験し、現在はフリーランスとして独立。2019年2月から株式会社TechBowlが提供するTechTrainでメンターを始める。2019年7月には株式会社Kaizen Platformにて社内横断のフロントエンド開発基盤・テスト基盤の設計・開発・保守を担当する。現在は新アーキテクチャ移行に向けたテストやドキュメントの整備、パッケージガーデニングとその調査に従事している。

◎本書スタッフ
アートディレクター/装丁：岡田章志＋GY
編集協力：深水 央
デジタル編集：栗原 翔

〈表紙イラスト〉
錆缶
イラストレーター。グラフィックデザインを学びながら背景からキャラまで様々なタッチで絵描きます。

技術の泉シリーズ・刊行によせて

技術者の知見のアウトプットである技術同人誌は、急速に認知度を高めています。インプレスR&Dは国内最大級の即売会「技術書典」（https://techbookfest.org/）で頒布された技術同人誌を底本とした商業書籍を2016年より刊行し、これらを中心とした『技術書典シリーズ』を展開してきました。2019年4月、より幅広い技術同人誌を対象とし、最新の知見を発信するために『技術の泉シリーズ』へリニューアルしました。今後は「技術書典」をはじめとした各種即売会や、勉強会・LT会などで頒布された技術同人誌を底本とした商業書籍を刊行し、技術同人誌の普及と発展に貢献することを目指します。エンジニアの"知の結晶"である技術同人誌の世界に、より多くの方が触れていただくきっかけになれば幸いです。

株式会社インプレスR&D
技術の泉シリーズ　編集長　山城 敬

●お断り
掲載したURLは2020年5月1日現在のものです。サイトの都合で変更されることがあります。また、電子版ではURLにハイパーリンクを設定していますが、端末やビューアー、リンク先のファイルタイプによっては表示されないことがあります。あらかじめご了承ください。

●本書の内容についてのお問い合わせ先
株式会社インプレスR&D　メール窓口
np-info@impress.co.jp
件名に「『本書名』問い合わせ係」と明記してお送りください。
電話やFAX、郵便でのご質問にはお答えできません。返信までには、しばらくお時間をいただく場合があります。
なお、本書の範囲を超えるご質問にはお答えしかねますので、あらかじめご了承ください。
また、本書の内容についてはNextPublishingオフィシャルWebサイトにて情報を公開しております。
https://nextpublishing.jp/

技術の泉シリーズ

React環境構築の教科書

2020年8月21日　初版発行Ver.1.0（リフロー版）

著　者　井手 優太
編集人　山城 敬
企画・編集　合同会社技術の泉出版
発行人　井芹 昌信
発　行　株式会社インプレスR&D
〒101-0051
東京都千代田区神田神保町一丁目105番地
https://nextpublishing.jp/

ISBN978-4-8443-7877-8

NextPublishing®

◉本書はNextPublishingメソッドによって発行されています。
NextPublishingメソッドは株式会社インプレスR&Dが開発した、電子書籍と印刷書籍を同時発行できるデジタルファースト型の新出版方式です。 https://nextpublishing.jp/