やる夫で学ぶ React、Reduxだお‥

知識ゼロから環境構築するお。
Redux-toolkitまで学ぶお。

2021年1月版

まだReduxで消耗してんの？ Redux-Toolkitで楽するお…

おおおお　redux-toolkitやて～　wwwwwww
キターーー(* v*)ーーー　明日からはフロントエンド担当だお･･･
Typescript･･･　ちょwww　吹いたwwwww　フロント
！おめでとう👏👏👏👏　俺も知りたいす！
javascript･･････　お勉強は、楽しいお･･･

やる夫で学ぶ「react-redux」

— redux-toolkit で、簡単・完璧理解だお··· —

気分はもう　著

技術書典 10（2021 年夏）新刊
2021 年 6 月 25 日　ver 1.0

■免責

本書は情報の提供のみを目的としています。
本書の内容を実行・適用・運用したことで何が起きようとも、それは実行・適用・運用した人自身の責任であり、著者や関係者はいかなる責任も負いません。

■商標

本書に登場するシステム名や製品名は、関係各社の商標または登録商標です。
また本書では、™、Ⓡ、Ⓒ などのマークは省略しています。

はじめに

このたびは、「やる夫で学ぶ-Redux- Redux Toolkit は簡単だお･･･」を、手にしていただき誠にありがとうございます。

私は 15 年近く Visual Studio 上で C#を使って UI のあるプログラムを書いていました。最近は、同一コードでマルチプラットフォーム上で動作するアプリケーションの構築は幾通りもの方法がありますが、以前は、方法が限られていました。

あるとき、あるプロジェクトでは、「Windows、Mac 上にて同一 UI で動作」が要求されました。そのために選択したのが、ブラウザ上で動作する Web アプリケーションです。

さらに、Web アプリケーションではローカルファイルにアクセスできないため、Electron を使用することで、Web アプリケーションをマルチプラットフォームで動作するアプリケーションにしました。

当時は、Web アプリケーションのフレームワークとして Angular と React が候補に挙がりましたが、React の方が学習コストが低いと言われていたため、React を学びました。
学習コストが低いとはいえ、壁にブチ当たると時間をかけて調べることを繰り返しました。

本書は、私が React や Redux を使い始めたときに「こんな本があれば良かったのに･･･」を目指して書きました。React,Redux の初学者から中級の方のお役に立てれば幸いです。

React、Redux の解説書やチュートリアルは、書籍・インターネット上にたくさんあります。しかし、つぎつぎと新機能が追加され本家以外の情報は、あっという間に古くなってしまいます。この書籍は、掲載情報を最新にするために電子書籍のみとし、古くなった情報は随時更新していきたいと思っています。

お気付き点がございましたら、Guthub 上へお寄せください。

本書は、フロントエンド開発で使用されている

- react

- redux(redux-toolkit)

を習得するために、開発環境の作成 (つまりゼロ) から始め、日記アプリケーションを作成します。

すでに開発環境を整えている方は、第 1 章、第 2 章を飛ばしてもかまいません。

また、ゼロからの環境構築、サンプルアプリケーションは GutHub に公開していますが、こちらも最新の情報に更新していくつもりです。GutHub では、章毎に別ブランチにしてありますので、写経がメンドウな方は章に対応したブランチを使ってください。

git、GitHub の使い方については、本書では取り扱いません。

本書は、チュートリアルによくある ToDo リストを日記アプリケーションとし

1. Redux なしで作成
2. Redux を導入
3. Redux Toolkit を導入

と、書き換えていくことで Redux を用いた「状態管理 (アプリケーション全体でのデータ)」を理解してもらえるようになっています。

必要なものは、すべて無料でそろえることができる以下のものです。これらが何なのか、そして、インストール方法は第 1 章にて解説しています。

- Node.js
- yarn（npm を使われる方は不要）
- Microsoft Visual Studio code
- Google Chrome

それでは、始めていきましょう。

目次

はじめに i

第 1 章 ゼロから始める (開発環境構築) 1

1.1 Node.js . . . 2

1.1.1 Node.js について . . . 2

1.1.2 Node.js のインストールの前に . . . 7

1.1.3 nvm . . . 8

1.2 Microsoft Visual Studio Code + 拡張機能 . . . 15

1.2.1 VSCode のインストール . . . 15

1.2.2 VSCode の拡張機能 . . . 16

1.2.3 Eslint . . . 18

1.2.4 Prettier . . . 18

1.3 Google Chrome + 拡張機能 . . . 19

1.3.1 Google Chrome のインストール . . . 19

1.4 第 1 章のまとめ . . . 24

第 2 章 スタートプロジェクトの作成 25

2.1 create-react-app コマンド . . . 26

2.1.1 アプリケーションを実行 . . . 27

2.1.2 create-react-app で作成された中身 . . . 29

2.2 ゼロから構築してみる . . . 32

2.2.1 ステップ 1 Node.js プロジェクト作成 . . . 32

2.2.2 webpack のインストールと設定 . . . 33

2.2.3 webpack の動作確認 . . . 35

2.2.4 webpack の設定ファイル . . . 42

2.2.5 webpack 設定ファイルを分割する . . . 53

2.2.6 Babel.js のインストールと設定 . . . 57

2.3 eslint、prettier とは？ . . . 66

2.3.1 eslint、prettier のインストール . . . 67

2.4 eslint、prettier の指摘を修正 . . . 74

2.5 第 2 章のまとめ . . . 79

第 3 章　日記アプリケーションの作成 (React のみ)　80
3.1　React とは？ 80
3.2　表示するデータの型を決める 81
3.3　データ表示画面 82
3.4　React hooks を使用して、データの追加・編集・削除 83

第 1 章

ゼロから始める (開発環境構築)

▲図 1.1: やる夫が、突然の移動を命じられたお！

本章では、React、Redux の開発環境を作成します。

「なぜ、これらが必要なのか？」

は、とりあえず置いといて

- Node.js
- Microsoft Visual Studio Code + 拡張機能
- Google Chrome + 拡張機能

をインストールしてください。これらは、すべて無償で提供されています。

なお、これらの準備が整っている方は、本章を読み飛ばしていただいてもかまいません。

1.1 Node.js

♡| 1.1.1 Node.js について

Node.js とは？

「Node.js」は、通常ブラウザ上で実行される JavaScript をサーバや PC 上で実行できるようする「**JavaScript 実行環境**」です。たとえば、Windows に Python をインストールすると「python.exe」を使い python ファイルを Windows 上で実行できます。

▲図 1.2: python を実行

同じように、**Node.js** をインストールすると、「node.exe」を使い JavaScript ファイルを実行できます。たとえば、以下のように「test01.js」ファイルを作成します。

▼リスト 1.1:

```
const name='やる夫';
const message='こまけぇこたぁいいんだよ';

console.log(`${name}が、こんなこと言ってます。${message}`);
```

このスクリプトをターミナル上で実行します。「node」コマンドに実行したい JavaScript

ファイルを引数として渡します。

▼ リスト 1.2: app.js を実行

```
⌧ node test01.js
やる夫が、こんなこと言ってます。こまけぇこたぁいいんだよ
```

このように、今までは JavaScript の実行環境はブラウザでしたが、Node があれば JacaScript を PC、サーバで実行できます。

Node.js について

▲図 1.3: Node.js をインストールするお！

▲図 1.4: Node.js のバージョンには注意

では、Node.js の本家トップページ:`https://nodejs.org/ja/` へアクセスします。

▲図 1.5: Node.js トップページ

ここでダウンロード可能なのは、「16.13.1 LTS(Long Term Support) 推奨版」と「17.3.0 最新版」[*1]の 2 つがあります。

LTS 版、最新版は以下のロードマップにより更新されます。

[*1] 2022/12/17 現在

リリース

Node.js のメジャーバージョンは 6 ヶ月間 *現行* リリースの状態になり、ライブラリの作者はそれらのサポートを追加する時間を与えられます。 6 ヶ月後、奇数番号のリリース (9, 11 など) はサポートされなくなり、偶数番号のリリース (10, 12 など) は *アクティブ LTS* ステータスに移行し、一般的に使用できるようになります。 *LTS* のリリースステータスは「長期サポート」であり、基本的に重要なバグは 30 ヶ月の間修正されることが保証されています。 プロダクションアプリケーションでは、*アクティブ LTS* または *メンテナンス LTS* リリースのみを使用してください。

リリース	ステータス	コードネーム	初回リリース	アクティブ LTS 開始	メンテナンス LTS 開始	サポート終了
v12	メンテナンス LTS	Erbium	2019-04-23	2019-10-21	2020-11-30	2022-04-30
v14	メンテナンス LTS	Fermium	2020-04-21	2020-10-27	2021-10-19	2023-04-30
v16	アクティブ LTS	Gallium	2021-04-20	2021-10-26	2022-10-18	2024-04-30
v17	現行		2021-10-19		2022-04-01	2022-06-01
v18	次期		2022-04-19	2022-10-25	2023-10-18	2025-04-30

日程は変更される場合があります。

▲図 1.6: Node.js ロードマップ

Node.js の Releases:https://nodejs.org/ja/about/releases/ にあるように、Node.js は、各年の 4 月、10 月にリリースされ、

- Current
- Active
- Maintenance

のフェーズを経ますが、メジャーバージョン番号が偶数のものだけが、Active 期間を経て長期サポートされます。

上記トップページにある Node.js 16 は、2024/4/30 までの長期サポートとなります。実際のプロジェクトで使用する場合は、よほどの理由がない限りは最新の LTS 版を使用します。

♡| 1.1.2　Node.js のインストールの前に

Node.js は、ロードマップにより定期的にバージョンアップされます。又、Node.js 自体の不具合の修正などでマイナーバージョンアップも行われます。

プロジェクト開発中のマイナーバージョンアップでも検証が必要になりますが、メジャーバージョンアップの場合はさらに大きな検証が必要になります。場合によっては、ソース

コードの大幅な改良をしなければならなくなります。

それを避けるためにも、プロジェクト毎に Node.js のバージョンは固定して開発します。

通常は、OS にインストールできる Node.js のバージョンはひとつですが、長期にわたるサポートや新規プロジェクト開発のためには、複数の Node.js のバージョンを切り替えて使用できるしくみを用意しましょう。

私が使用しているのは、nvm(node version manager):https://github.com/nvm-sh/nvm です。いろいろなバージョンの Node.js を、簡単にインストール・アンインストール・切替ができます。

「nvm」も含めた Node.js バージョン管理ツールについては、こちらの@heppokofrontend さんの良記事が参考になります。

Node.js のバージョン管理ツールを改めて選定する【2021 年】[*2]

♡| 1.1.3　nvm

nvm(node version manager) を使えば、複数バージョンの Node.js を 1 台の PC にインストールし、バージョンを切り替えることが簡単にできます。

GitHub 上の nvm は、Shellscript(sh, dash, zsh, bash) 上で動作するため、Linux(UNIX 系)、macOS にインストールできます。

Windows 版:https://github.com/coreybutler/nvm-windows は、別な方が GitHub 上で公開されています。コマンドが本家と少し違いますが、複数バージョンのインストール・バージョンの切り替えなど機能は問題ありません。

[*2] https://qiita.com/heppokofrontend/items/5c4cc738c5239f4afe02

nvm のインストール

macOS

お使いの Terminal から以下のコマンドを実行してください。

▼ リスト 1.3: nvm のインストール

```
  >curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.0/install.sh | bash
```

インストール完了後には、.zshrc へ以下を追加してください。

▼ .zshrc へ追加

```
  export NVM_DIR="$([ -z "${XDG_CONFIG_HOME-}" ] && printf %s "${HOME}/.nvm" || pr›
›intf %s "${XDG_CONFIG_HOME}/nvm")"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh" # This loads nvm
```

Windows

nvm-windows のリリースページ:https://github.com/coreybutler/nvm-windows/releases/tag/1.1.8 より、最新版をダウンロードしインストールしてください。

nvm の使い方

macOS ではターミナルを起動し、Windows ではコマンドプロンプト、または、Windows Terminal を起動してください。

nvm と nvm-windows では、コマンドが少し違いますが、「nvm --help」を入力することで使用できるコマンドが表示されます。

▼ Mac OSX

```
> nvm --help

Node Version Manager (v0.37.2)
←中略
Example:
  nvm install 8.0.0                     Install a specific version number
  nvm use 8.0                           Use the latest available 8.0.x release
  nvm run 6.10.3 app.js                 Run app.js using node 6.10.3
  nvm exec 4.8.3 node app.js            Run `node app.js` with the PATH pointing t›
›o node 4.8.3
  nvm alias default 8.1.0               Set default node version on a shell
  nvm alias default node                Always default to the latest available nod›
›e version on a shell

←中略
Note:
  to remove, delete, or uninstall nvm - just remove the `$NVM_DIR` folder (usually›
› `~/.nvm`)
```

もし、32bit 版 Windows をお使いの場合には、インストールの際に 32bit 版を指定する「32」をコマンドの最後につけてください。

▼windows

```
PS C:\Users\inabakazuya> nvm --help

Running version 1.1.7.

Usage:

  nvm arch                     : Show if node is running in 32 or 64 bit mode.
  nvm install <version> [arch] : The version can be a Node.js version or "latest" ›
›for the latest stable version.
                                 Optionally specify whether to install the 32 or 6›
›4 bit version (defaults to system arch).
                                 Set [arch] to "all" to install 32 AND 64 bit vers›
›ions.
                                 Add --insecure to the end of this command to bypa›
›ss SSL validation of the remote download server.
  nvm list [available]         : List the Node.js installations. Type "available" ›
›at the end to see what can be installed. Aliased as ls.
 ←中略
  nvm uninstall <version>      : The version must be a specific version.
  nvm use [version] [arch]     : Switch to use the specified version. Optionally s›
›pecify 32/64bit architecture.
                                 nvm use <arch> will continue using the selected v›
›ersion, but switch to 32/64 bit mode.
  nvm root [path]              : Set the directory where nvm should store differen›
›t versions of Node.js.
                                 If <path> is not set, the current root will be di›
›splayed.
  nvm version                  : Displays the current running version of nvm for W›
›indows. Aliased as v.
```

インストール可能な Node.js を表示

まずは、インストール可能な Node.js のバージョンを表示してみます。macOS の場合には、古いバージョンから最新バージョンまでが表示されます。

▼Mac OSX

```
>nvm ls-remote
←古いバージョンから全て表示されるので中略
v14.13.1
v14.14.0
v14.15.0   (LTS: Fermium)
v14.15.1   (LTS: Fermium)
v14.15.2   (Latest LTS: Fermium)
```

```
v15.0.0
v15.0.1
v15.1.0
v15.2.0
v15.2.1
v15.3.0
v15.4.0
```

一方、Windows の場合には、表形式で表示されます。新しいバージョンが上に表示されます。

▼ Windows

```
PS C:\Users\inabakazuya> nvm list available

|   CURRENT    |     LTS      |  OLD STABLE  | OLD UNSTABLE |
|--------------|--------------|--------------|--------------|
|    15.4.0    |   14.15.2    |   0.12.18    |   0.11.16    |
|    15.3.0    |   14.15.1    |   0.12.17    |   0.11.15    |
|    15.2.1    |   14.15.0    |   0.12.16    |   0.11.14    |
|    15.2.0    |   12.20.0    |   0.12.15    |   0.11.13    |
|    15.1.0    |   12.19.1    |   0.12.14    |   0.11.12    |
|    15.0.1    |   12.19.0    |   0.12.13    |   0.11.11    |
|    15.0.0    |   12.18.4    |   0.12.12    |   0.11.10    |
|   14.14.0    |   12.18.3    |   0.12.11    |    0.11.9    |
|   14.13.1    |   12.18.2    |   0.12.10    |    0.11.8    |
|   14.13.0    |   12.18.1    |    0.12.9    |    0.11.7    |
|   14.12.0    |   12.18.0    |    0.12.8    |    0.11.6    |
|   14.11.0    |   12.17.0    |    0.12.7    |    0.11.5    |
|   14.10.1    |   12.16.3    |    0.12.6    |    0.11.4    |
|   14.10.0    |   12.16.2    |    0.12.5    |    0.11.3    |
|    14.9.0    |   12.16.1    |    0.12.4    |    0.11.2    |
|    14.8.0    |   12.16.0    |    0.12.3    |    0.11.1    |
|    14.7.0    |   12.15.0    |    0.12.2    |    0.11.0    |
|    14.6.0    |   12.14.1    |    0.12.1    |    0.9.12    |
|    14.5.0    |   12.14.0    |    0.12.0    |    0.9.11    |
|    14.4.0    |   12.13.1    |   0.10.48    |    0.9.10    |

This is a partial list. For a complete list, visit https://nodejs.org/download/rel›
›ease
```

Node.js 最新 LTS 版をインストール

それでは、最新の LTS 版をインストールします。インストールは、Mac、Windows とも、「nvm install xx.yy.zz(インストールするバージョン番号)」でインストールできます。

▼ Mac OSX

```
> nvm install v14.15.2
Downloading and installing node v14.15.2...
Downloading https://nodejs.org/dist/v14.15.2/node-v14.15.2-darwin-x64.tar.xz...
##################################################################################›
›############################## 100.0%
Computing checksum with shasum -a 256
Checksums matched!
Now using node v14.15.2 (npm v6.14.9)
```

▼ Window

```
PS C:\Users\inabakazuya> nvm install v14.15.2
Downloading Node.js version 14.15.2 (64-bit)...
Complete
Creating C:\Users\inabakazuya\AppData\Roaming\nvm\temp

Downloading npm version 6.14.9... Complete
Installing npm v6.14.9...

Installation complete. If you want to use this version, type

nvm use 14.15.2
```

インストールされている Node.js のバージョンの確認

インストールされている Node.js のバージョンは、「nvm ls」で表示させ確認できます。

▼ Mac OSX

```
$ > nvm ls
          v8.17.0
         v12.19.0
         v14.15.0
  ->     v14.15.2
           system
  default -> v12.18.3
  iojs -> N/A (default)
  unstable -> N/A (default)
  node -> stable (-> v14.15.2) (default)
  stable -> 14.15 (-> v14.15.2) (default)
  lts/* -> lts/fermium (-> v14.15.2)
  lts/argon -> v4.9.1 (-> N/A)
  lts/boron -> v6.17.1 (-> N/A)
  lts/carbon -> v8.17.0
  lts/dubnium -> v10.23.0 (-> N/A)
  lts/erbium -> v12.20.0 (-> N/A)
  lts/fermium -> v14.15.2
```

▼ Windows

```
PS C:\Users\inabakazuya> nvm ls

    14.15.2
    14.15.1
    12.13.1
  * 8.16.1 (Currently using 64-bit executable)
```

使用する Node.js のバージョン切り替え

先ほどの「インストールされている Node.js のバージョン表示」で、現在使われている

Node.js のバージョンも表示されています。使用する Node.js のバージョンを変更する場合には、「nvm use xx.yy.xx(使用するバージョン番号)」で切り替えます。

▼Node.js のバージョン確認と切替

```
$ > node -v
v14.15.2
[~]
$ > nvm use v12.18.3
Now using node v12.18.3 (npm v6.14.6)
[~]
$ > node -v
v12.18.3
```

以上で、複数のバージョンの Node.js を切り替えて使える環境が構築できました。

1.2 Microsoft Visual Studio Code + 拡張機能

Microsoft 社が無料で提供している「テキストエディタ」です。Electron:https://www.electronjs.org/ をベースにしたオープンソースで開発されています。

Electron は、GitHub 社が「Atom(テキストエディタ)」を開発するために構築したフレームワークで、HTML・CSS・JavaScript を使用して、Windows、Mac、Linux のマルチプラットフォームで動作するアプリケーションを開発できます。

Visual Studio Code(以後は、VSCode と表記します。) は、コードを記述する「テキストエディタ」として非常に優秀ですが、拡張機能 (Google Chrome や Firefox などのブラウザ同様に拡張機能が多くの開発者により公開されています。) を追加することで JavaScript 以外の言語 (C#、python など) でも使えます。

デバッグなども行えるため、Web 開発では、事実上の標準と言っても良いでしょう。有料では、Jetbrains 社:https://www.jetbrains.com/ の Webstorm がありますが、今回は、無償の VSCode を使用します。

♡| 1.2.1　VSCode のインストール

VSCode のインストールは
本家サイト https://code.visualstudio.com/
から、ダウンロード後インストールしてください。
または、以下の方法でもインストールできます。

Windows

パッケージマネージャー「Chocolatey」をお使いの方は、

▼Chocolatey

```
choco install vscode
```

にて、インストールできます。
パッケージマネージャー「winget」をお使いの方は、

▼ winget

```
winget install -e --id Microsoft.VisualStudioCode
```

にて、インストールできます。

macOS

パッケージマネージャーに「brew」をお使いの方は、

▼ Homebrew

```
$ > brew update
$ > brew cask install visual-studio-code
```

にて、インストールできます。

♡| 1.2.2　VSCode の拡張機能

VSCode は、プラグイン形式で拡張機能を追加できます。
React、Redux を使用したプロジェクトでは、以下をインストールすると便利です。
VSCode を起動し、左ツールバーの拡張機能アイコンをクリックします。

▲図 1.7: VSCode の拡張機能

ここの検索窓に拡張機能の名前、キーワードを入力して検索します。

Debugger for Chrome

デバッグの際に、PC にインストールされている Chrome を自動で起動してくれます。Chrome の DevTools は、非常に強力です。また、Chrome にも React、Redux 用の拡張機能を追加すると更に便利になります。

▲図 1.8: Beautify

♡| 1.2.3　Eslint

コード記法の間違いを指摘・修正してくれます。

▲図 1.9: Beautify

♡| 1.2.4　Prettier

Eslint と同じように、コード記法の間違いの指摘・修正やコードフォーマットを行います。Eslint を合わせて使うと最強です。

▲図 1.10: Beautify

1.3 Google Chrome + 拡張機能

ご存じ Google 社が提供するブラウザです。
PC へインストールされていない方は、

♡| 1.3.1 Google Chrome のインストール

Google Chrome:https://www.google.com/intl/ja/chrome/

▲図 1.11: Google Chrome

Google Chrome の拡張機能

こちらも、VSCode と同様に拡張機能を追加することで、さらに便利に使うことができます。

React、Redux の開発では、以下の拡張機能は必須と言っても良いほどです。

拡張機能のインストールは、下記の Chrome Web store で検索してください。

Chrome Web store:https://chrome.google.com/webstore/category/extensions?

```
hl=ja
```

▲図 1.12: Chrome Web store

React Developer Tools

React を使用して作成したページは、最終的にはページ出力用 JavaScript に変換され、ブラウザで表示されるときには HTML として出力されます。この拡張機能を使うと、Google Chrome の DevTools に Components タブが作成され、Props、State を確認できます。

▲図 1.13: React Developer Tools

▲図 1.14: React DevTools で App を表示

Redux DevTools

のちほど、Redux の章であらためて説明しますが、「タイムトラベルデバッグ (実行されたアクションをさかのぼる)」が簡単にできます。また、実行されたアクション、変更された State が「新」「旧」とあり、どの部分が変更されたのかも確かめるのも簡単です。

▲図 1.15: React DevloperTools 拡張機能

1.4 第 1 章のまとめ

React、Redux の開発環境は、できましたでしょうか？

- nvm
- node
- VSCode + 拡張機能
- Google Chrome + 拡張機能

のインストールを完了してください。

第 2 章
スタートプロジェクトの作成

React アプリケーションを作成するための最初のステップとして、トップページのみを持つスタートアッププロジェクトを作成します。

スタートアッププロジェクトを作成する方法として、

1. create-react-app
2. ゼロから構築

の 2 つの方法を解説します。

「create-react-app」は、コマンド一発で React アプリケーション開発が数分で始められます。

ただし、Facebook(Meta 社) を中心に開発されている便利なものなのですが、メンドウな設定などが隠されているためバージョンの合わないライブラリを導入すると整合性が崩れ手に負えなくなることもあります。

2021 年 12 月 14 日にリリースされた「create-react-app V5.0.0」では、webpack、eslint などは最新のものが使われています。

「ゼロから構築」を選択すると、最新のライブラリが使用できますが、webpack、ESLint などの設定ファイルは自分で書かなくてはなりません。使用するライブラリの設定自体は難しくないので、ここで勉強しておけば必ず役に立つはずです。

どちらの方法も GitHub にテンプレートとしてアップロードしてありますので、ご自由にお使いください。

2.1 create-react-app コマンド

React アプリケーションをゼロから作成するためには、

- 「node プロジェクト」に必要な package.json を作成
- react など必要なライブラリのインストール
- 作成したアプリケーションが、古いブラウザでも実行できるようにコードを変換 (Babel 使用)
- 出力するファイルをまとめる (バンドルする - webpack 使用)

など、react ライブラリのインストール以外にも、Babel や webpack をインストールして設定ファイルを作成しなくてはなりません。

また、使用するライブラリによっては、プラグインのインストールや設定など、アプリケーションのコードを書き始める前の作業がたいへんです。

しかし、「そんなメンドウなことは、やってられない。」と誰しもが思ったか、すぐにでもコードを書き始めることのできるスタート用アプリケーションが、react 開発元の Facebook(Meta) から提供されています。

さらに、そのスタート用アプリケーションは、コマンド一発でインストールできます。

では、実際に手を動かしましょう。ターミナルを起動し、プロジェクトを作成するフォルダへ移動します。

▼create-react-app でスタート用アプリケーション作成

```
> npx create react-app プロジェクト名 --template typescript
```

エンターキーを押すと、作業が始まり「プロジェクト名」のフォルダが作成され、以下のように表示されればスグにでも開発に取りかかれます。

▼create-react-app 完了時

```
  Success! Created yaruo-cra-template at /Users/kazuyainaba/Documents/Devs_kz/Proj›
›ect_技術書展202012/yaruo_react_sample/yaruo-cra-template
  Inside that directory, you can run several commands:

    npm start
      Starts the development server.
```

```
    npm run build
      Bundles the app into static files for production.

    npm test
      Starts the test runner.

    npm run eject
      Removes this tool and copies build dependencies, configuration files
      and scripts into the app directory. If you do this, you can't go back!

  We suggest that you begin by typing:

    cd yaruo-cra-template
    npm start

  Happy hacking!
```

> 「create-react-app」は、2021 年 12 月 14 日に V5.0.0 がリリースされました。このリリースでは、メジャーバージョンアップされていた「webpack5」、「eslint8」が採用されています。

github

ここまでの作業は、GitHub にあります。

▼ GitHub から

```
> git clone -b 01_create-react-app-executed https://github.com/yaruo-›
›react-redux/yaruo-cra-template.git
```

♡| 2.1.1　アプリケーションを実行

アプリケーションが作成できましたので、実行してみます。

ターミナルの表示に従い、プロジェクトフォルダへ移動し、スタート用のコマンドを入力します。

▼ プロジェクトの実行

```
> cd プロジェクト名
> npm run start ←もしくは、yarn start
```

すると、webpack に同梱されている開発用の web server が起動し、デフォルトでは、port:3000 でアプリケーションへアクセスできます。

▼ npm run start 時

```
Compiled successfully!

You can now view yourproject in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://pcのローカルIPアドレス:3000

Note that the development build is not optimized.
To create a production build, use yarn build.
```

Google Chrome が起動し、http://localhost:3000 へアクセスし以下のページが表示されます。

▲図 2.1: create-react-app の画面

このページが表示されれば成功です。

♡| 2.1.2　create-react-app で作成された中身

create-react-app で作成された中身は、以下となります (使用するテンプレートにより作成されるファイル・フォルダは異なる)。

▼create-react-app で作成されたファイル・フォルダ

```
.
├── node_modules
├── README.md
├── package-lock.json
├── package.json
```

```
├──── public
│    ├──── favicon.ico
│    ├──── index.html
│    ├──── logo192.png
│    ├──── logo512.png
│    ├──── manifest.json
│    └──── robots.txt
├──── src
│    ├──── App.css
│    ├──── App.test.tsx
│    ├──── App.tsx
│    ├──── index.css
│    ├──── index.tsx
│    ├──── logo.svg
│    ├──── react-app-env.d.ts
│    ├──── reportWebVitals.ts
│    └──── setupTests.ts
└──── tsconfig.json
```

package.json ファイルは、Node.js を使用するプロジェクトの設計図にあたるものです。

Node.js を使うプロジェクトを開始する場合には、プロジェクトフォルダで「npm init」を行うと対話形式で「package.json」を作成しますが、create-react-app コマンドを使用すると、package.json も以下のように作成されます。

▼package.json

```
{
  "name": "作成時に入力したプロジェクト名",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.16.1",
    "@testing-library/react": "^12.1.2",
    "@testing-library/user-event": "^13.5.0",
    "@types/jest": "^27.0.3",
    "@types/node": "^16.11.14",
    "@types/react": "^17.0.37",
    "@types/react-dom": "^17.0.11",
    "react": "^17.0.2",
    "react-dom": "^17.0.2",
    "react-scripts": "5.0.0",
    "typescript": "^4.5.4",
    "web-vitals": "^2.1.2"
  },
  "scripts": {
    "start": "react-scripts start",
```

```
      "build": "react-scripts build",
      "test": "react-scripts test",
      "eject": "react-scripts eject"
    },
    "eslintConfig": {
      "extends": [
        "react-app",
        "react-app/jest"
      ]
    },
    "browserslist": {
      "production": [
        ">0.2%",
        "not dead",
        "not op_mini all"
      ],
      "development": [
        "last 1 chrome version",
        "last 1 firefox version",
        "last 1 safari version"
      ]
    }
  }
```

package.json 内にある「scripts」にあるものがコマンドになります。react-scripts は、npm スクリプトを連続、または、並列に実行してくれるものです。

package.json の「dependencies」には、実行に必要でインストール済みの npm パッケージが記載されています。必要な npm パッケージをインストールすると、ここに自動的に追記されます。

また、開発時のみ必要なパッケージ (build したときには組み込まれない) は、「devDependencies」に追加されます。

2.2　ゼロから構築してみる

本章では、最新のライブラリを使用してゼロから React/TypeScript の環境を構築します。

ステップ毎に GitHub 上でブランチを作成してありますので、どこからでも始めていただけます。

♡| 2.2.1　ステップ 1 Node.js プロジェクト作成

新しくプロジェクト用のフォルダを作成し移動します。

コンソールで「npm init -y」コマンドを実行します。オプションの「-y」なしで実行すると、対話形式で「package.json」を作成できます。

▼node プロジェクトの開始

```
  ⌧ npm init -y
  Wrote to /Users/yaruo/Documents/yaruo_react_sample/yaruo-start-template/package.›
›json:

  {
    "name": "yaruo-start-template",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1"
    },
    "repository": {
      "type": "git",
      "url": "git+https://github.com/yaruo-react-redux/yaruo-start-template.git"
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "bugs": {
      "url": "https://github.com/yaruo-react-redux/yaruo-start-template/issues"
    },
    "homepage": "https://github.com/yaruo-react-redux/yaruo-start-template#readme"
  }
```

作成された「package.json」が表示されます。

ここまでの内容は、GitHub 上で、以下のコマンドでクローンできます。

▼ GitHub

```
> git clone -b 01_start-node-project https://github.com/yaruo-react-r›
›edux/yaruo-start-template.git
```

♡| 2.2.2　webpack のインストールと設定

▲ 図 2.2: desc

webpack とは、(本家[*1]) のトップにある上図が示しているように、

- JavaScript ファイル
- CSS(SASS,SCSS) ファイル
- 画像ファイル

[*1] https://webpack.js.org/

などをすべて JavaScript ファイルとして扱い、インストールしているライブラリファイルなどもすべて含めて 1 つのファイルとして出力するバンドラー (まとめる) です。

しかし、すべてを 1 つのファルとするよりも「html ファイル」、「css ファイル」、画像ファイルを別ファイルとして出力し、ブラウザがファイルを並列にダウンロードできると効率がよくなり、表示速度も速くなります。そのため、上図のように、複数ファイルに出力します。

それでは、webpack をインストールし、バンドラーの動きを確認しながら設定ファイルを作成していきます。

ターミナルに以下のコマンドを入力します。「-D(または、--save-dev)」のオプションは、開発時のみ必要で製品版には含まないライブラリをインストールするときに使います。

インストールすると、「package.json」の「devDependencies」に追記されます。

- webpack 本体
- webpack-cli コマンドライン用
- webpack-dev-server 開発用 Web サーバ

▼webpck のインストール

```
  ⌧ npm install -D webpack webpack-cli webpack-dev-server
  npm WARN deprecated querystring@0.2.0: The querystring API is considered Legacy.›
› new code should use the URLSearchParams API instead.

  added 328 packages, and audited 329 packages in 24s

  42 packages are looking for funding
    run `npm fund` for details

  found 0 vulnerabilities
```

「package.json」は、以下のようになります。「ー D」オプションを付けたため、「devDependencies」以下に追記されています。

▼package.json

```
  {
    "name": "yaruo-start-template",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
```

```
      "test": "echo \"Error: no test specified\" && exit 1"
    },
    "repository": {
      "type": "git",
      "url": "git+https://github.com/yaruo-react-redux/yaruo-start-template.git"
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "bugs": {
      "url": "https://github.com/yaruo-react-redux/yaruo-start-template/issues"
    },
    "homepage": "https://github.com/yaruo-react-redux/yaruo-start-template#readme"
,
    "devDependencies": {
      "webpack": "^5.65.0",
      "webpack-cli": "^4.9.1",
      "webpack-dev-server": "^4.6.0"
    }
  }
```

♡| 2.2.3　webpack の動作確認

インストールした webpack の動作を確認してみます。

確認方法は、便利な関数をまとめてある「lodash」ライブラリをインストールし、トップページを作成し動作確認します。

手順は、

1. src、dist フォルダを作成

ソースコードを置くフォルダ「src」とw ebpack のデフォルトの出力先フォルダ「dist」を作成します。

2. ファイルを作成

「lodash」ライブラリをインストールし、src フォルダに、下記の「index.js」ファイルを作成します。

▼lodash のインストール

```
> npm install lodash
```

▼src/index.js

```
import _ from 'lodash';

function component() {
  const element = document.createElement('div');
  // Lodash, now imported by this script
  element.innerHTML = _.join(['webpack', '動いてるお〜'], ' ');
  return element;
}

document.body.appendChild(component());
```

3. トップページを作成

webpack のデフォルトの出力先「dist」フォルダを作成し、「index.html」ファイルを作成します。

▼dist/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Getting Started</title>
  </head>
  <body>
    <script src="main.js"></script>
  </body>
</html>
```

4. 動作を確認

webpack の動作を確認するために、ターミナルで以下のコマンドを実行します。

▼webpack 実行後、ブラウザで表示

```
> npx webpack serve --open --static-directory dist --mode=development
```

コマンド解説

npx --> /node_modules/.bin フォルダにあるファイルを実行
webpack --> 今回動かすモジュール
serve --> devServer(開発用サーバ) も起動
--open --> デフォルトのブラウザで開く
--static-directory dist --> devServer の DocumentRoot を指定
--mode=development --> 出力モード

「--static-directory dist」を入力しているのは、devServer のデフォルト DocumentRoot は「public」のためです。

▼webpack をコマンドで起動

```
  ☒ npx webpack serve --open --static-directory dist --mode=development
  <i> [webpack-dev-server] Project is running at:
  <i> [webpack-dev-server] Loopback: http://localhost:8080/
  <i> [webpack-dev-server] On Your Network (IPv4): http://192.168.20.101:8080/
  <i> [webpack-dev-server] On Your Network (IPv6): http://[fe80::1]:8080/
  <i> [webpack-dev-server] Content not from webpack is served from 'dist' director›
›y
  <i> [webpack-dev-middleware] wait until bundle finished: /
  asset main.js 836 KiB [emitted] (name: main)
  runtime modules 27.2 KiB 13 modules
  modules by path ./node_modules/ 730 KiB
    modules by path ./node_modules/webpack-dev-server/client/ 52.8 KiB 12 modules
    modules by path ./node_modules/webpack/hot/*.js 4.3 KiB 4 modules
    modules by path ./node_modules/html-entities/lib/*.js 81.3 KiB 4 modules
    modules by path ./node_modules/url/ 37.4 KiB 3 modules
    modules by path ./node_modules/querystring/*.js 4.51 KiB
      ./node_modules/querystring/index.js 127 bytes [built] [code generated]
      ./node_modules/querystring/decode.js 2.34 KiB [built] [code generated]
      ./node_modules/querystring/encode.js 2.04 KiB [built] [code generated]
    ./node_modules/lodash/lodash.js 531 KiB [built] [code generated]
    ./node_modules/ansi-html-community/index.js 4.16 KiB [built] [code generated]
    ./node_modules/events/events.js 14.5 KiB [built] [code generated]
  ./src/index.js 269 bytes [built] [code generated]
  webpack 5.65.0 compiled successfully in 882 ms
```

「--open」オプションでデフォルトのブラウザが起動し、index.html が表示されます。

▲図 2.3: ブラウザで表示

devTools で「main.js」を確認すると、node_modules フォルダ以下にインストールされた JavaScript が 1 つのファイルにまとめられているのが確認できます。

▲図 2.4: devTools で main.js 内を確認

▼ビルドの実行

```
> npx webpack build
```

上記コマンドを実行すると、dist フォルダ以下に main.js ファイルが出力されます。

▲図 2.5: webpack でビルドしてみた

ここまでの内容は、GitHub 上で、以下のコマンドでクローンできます。

▼ GitHub

```
> git clone -b 02_webpack-install https://github.com/yaruo-react-redu›
›x/yaruo-start-template.git
```

♡| 2.2.4　webpack の設定ファイル

自分で webpack、devServer を動作させてみると、webpack が何をやっているかを理解しやすくなります。

本章では、webpack の設定ファイル「webpack-config.js」を作成します。

手順は、

1. 「npx webpack-cli init」をターミナルで実行し、ひな型を作成。
2. 必要な plugin のインストールと設定ファイルへの追加
3. 不要な設定を削除し、開発時用、製品作成時用でファイルを分ける

と、なります。

ターミナルにてコマンドを実行すると、

▼ npx webpack-cli init の実行

```
> npx webpack-cli init
```

このコマンドを実行するには、「@webpack-cli/generators パッケージが必要ですが、インストールしますか？」と聞かれますので、エンターキーを押します。

@webpack-cli/generators と依存関係をもつパッケージがインストールされると、設定ファイルを作成するための質問が始まります。私が選んだ答えと括弧内は表示される選択肢です。

- どのタイプの JS を使いますか？　--> TypeScript(none, ES6)
- devServer を使いますか？　--> Yes
- バンドル用の HTML ファイルを作成しますか？　--> Yes
- PWA サポートが必要ですか？　--> No
- CSS は、どのタイプを使いますか？　--> SASS(none, CSS only, LESS, Stylus)
- SASS と一緒に CSS スタイルも使いますか？　--> Yes
- PostCSS(Node.js 製の CSS を作るためのフレームワーク) を使いますか？　--> No
- ファイル毎に CSS を別にしますか？　--> Yes
- 設定ファイルをフォーマットするのに Prettier をインストールしますか？　--> Yes
- パッケージマネージャーを選択してください。 --> npm
- package.json がすでにありますが上書きしても良いですか？　--> No
- README.md がすでにありますが上書きしても良いですか？　--> No

▼ webpack-config.js の作成

```
  > npx webpack-cli init
  [webpack-cli] For using this command you need to install: '@webpack-cli/generato›
›rs' package.
  [webpack-cli] Would you like to install '@webpack-cli/generators' package? (That›
› will run 'npm install -D @webpack-cli/generators') (Y/n)
  npm WARN deprecated urix@0.1.0: Please see https://github.com/lydell/urix#deprec›
›ated
  npm WARN deprecated resolve-url@0.2.1: https://github.com/lydell/resolve-url#dep›
›recated

  added 370 packages, and audited 699 packages in 22s

  58 packages are looking for funding
    run `npm fund` for details

  9 vulnerabilities (4 moderate, 5 high)

  To address all issues (including breaking changes), run:
    npm audit fix --force

  Run `npm audit` for details.
  ? Which of the following JS solutions do you want to use? Typescript
  ? Do you want to use webpack-dev-server? Yes
  ? Do you want to simplify the creation of HTML files for your bundle? Yes
  ? Do you want to add PWA support? No
  ? Which of the following CSS solutions do you want to use? SASS
  ? Will you be using CSS styles along with SASS in your project? Yes
  ? Will you be using PostCSS in your project? No
  ? Do you want to extract CSS for every file? Yes
  ? Do you like to install prettier to format generated configuration? Yes
  ? Pick a package manager: npm
  [webpack-cli] ☒ INFO  Initialising project...
   conflict package.json
  ? Overwrite package.json? do not overwrite
       skip package.json
     create src/index.ts
   conflict README.md
  ? Overwrite README.md? do not overwrite
       skip README.md
     create index.html
     create webpack.config.js
     create tsconfig.json

  added 65 packages, and audited 764 packages in 9s

  73 packages are looking for funding
    run `npm fund` for details
```

```
9 vulnerabilities (4 moderate, 5 high)

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
```

質問完了後に必要なパッケージがインストールされ、「webpack.config.js」が作成されます。また、TypeScript 用に「tsconfig.json」も作成されますが、後で作成しますので削除します。

作成された「webpack.config.js」を以下のように編集します。

- entry: "./src/index.ts"を"./src/index.js"へ
- output: path: path.resolve(__dirname, "public") へ
- dist フォルダを削除し、public フォルダを作成

▼webpack.config.js

```
// Generated using webpack-cli https://github.com/webpack/webpack-cli

const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

const isProduction = process.env.NODE_ENV == "production";

const stylesHandler = MiniCssExtractPlugin.loader;

const config = {
  entry: "./src/index.js", ←TypeScript 導入までは、拡張子 js で
  output: {
    path: path.resolve(__dirname, "public"), ←出力フォルダを public へ
  },
  devServer: {
    open: true,
    host: "localhost",
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: "index.html",
    }),

    new MiniCssExtractPlugin(),

    // Add your plugins here
    // Learn more about plugins from https://webpack.js.org/configuration/plugin›
```

```
›s/
    ],
    module: {
      rules: [
        {
          test: /\.(ts|tsx)$/i,
          loader: "ts-loader",
          exclude: ["/node_modules/"],
        },
        {
          test: /\.css$/i,
          use: [stylesHandler, "css-loader"],
        },
        {
          test: /\.s[ac]ss$/i,
          use: [stylesHandler, "css-loader", "sass-loader"],
        },
        {
          test: /\.(eot|svg|ttf|woff|woff2|png|jpg|gif)$/i,
          type: "asset",
        },

        // Add your rules for custom modules here
        // Learn more about loaders from https://webpack.js.org/loaders/
      ],
    },
    resolve: {
      extensions: [".tsx", ".ts", ".js"],
    },
  };

  module.exports = () => {
    if (isProduction) {
      config.mode = "production";
    } else {
      config.mode = "development";
    }
    return config;
  };
```

プラグインのインストール

追加で、以下のプラグイン、ローダも追加します。

- uglify-js プロダクション出力時に console 関数を除去
- terser-webpack-plugin 上記を webpack で使用する場合必要
- css-minimizer-webpack-plugin CSS を minimize

- webpack-merge 複数の webpack-config ファイルをマージする

ターミナルで以下のコマンドを実行します。

▼追加プラグイン、ローダのインストール

```
 > npm install -D uglify-js terser-webpack-plugin css-minimizer-webpack-plugin web›
›pack-merge
```

追加のプラグイン、ローダの設定を追加した webpack.config.js です。devServer でページを表示する際に、デフォルトのブラウザではなく devTools の強力な「Google Chrome」を使うようにしました。

OS 毎に Chrome のアプリケーション名が違うため OS を取得し対応した「Chrome 名」に変換するための switch 文を追加しています。「create-react-app」だと、結構複雑なことをやって Google Chrome を起動しています。

興味のある方は、「create-react-app」を使ってプロジェクト作成し、react-scripts を追うとお勉強になります。

▼webpack.config.js

```
// Generated using webpack-cli https://github.com/webpack/webpack-cli
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
const TerserPlugin = require('terser-webpack-plugin');
const CssMinimizerPlugin = require('css-minimizer-webpack-plugin');
const os = require('os');

const isProduction = process.env.NODE_ENV == 'production';

const stylesHandler = MiniCssExtractPlugin.loader;

let devBrowser = 'Google Chrome';
switch (os.platform()) {
  case 'win32':
    devBrowser = 'chrome';
    break;
  case 'linux':
    devBrowser = 'google-chrome';
  default:
    break;
}

const config = {
```

```
    entry: './src/index.js',
    output: {
      path: path.resolve(__dirname, 'public'),
      assetModuleFilename: 'images/[name][ext][query]',
      clean: true,
    },
    plugins: [
      new HtmlWebpackPlugin({
        template: 'index.html',
      }),

      new MiniCssExtractPlugin(),

      // Add your plugins here
      // Learn more about plugins from https://webpack.js.org/configuration/plugin›
›s/
      new CssMinimizerPlugin(),
    ],
    module: {
      rules: [
        {
          test: /\.(ts|tsx)$/i,
          loader: 'ts-loader',
          exclude: ['/node_modules/'],
        },
        {
          test: /\.css$/i,
          use: [stylesHandler, 'css-loader'],
        },
        {
          test: /\.s[ac]ss$/i,
          use: [stylesHandler, 'css-loader', 'sass-loader'],
        },
        {
          test: /\.(eot|svg|ttf|woff|woff2|png|jpg|gif)$/i,
          type: 'asset',
        },

        // Add your rules for custom modules here
        // Learn more about loaders from https://webpack.js.org/loaders/
      ],
    },
    resolve: {
      extensions: ['.tsx', '.ts', '.js'],
    },
    optimization: {
      minimize: true,
      minimizer: [
        new TerserPlugin({
```

```
          minify: TerserPlugin.uglifyJsMinify,
          terserOptions: {
            compress: {
              drop_console: true,
            },
          },
        },
        new CssMinimizerPlugin(),
      ],
    },
    devtool: 'eval-source-map',
    devServer: {
      open: {
        app: {
          name: devBrowser,
        },
      },
      host: 'localhost',
      port: 3000,
      static: './public',
    },
  };

  module.exports = () => {
    if (isProduction) {
      config.mode = 'production';
    } else {
      config.mode = 'development';
    }
    return config;
  };
```

動作確認のために「index.js」を書き換えます。いつの間にかプロジェクトフォルダ直下に「index.html」も作成されています。

「index.js」に、動作確認用に追加するスタイル指定用の「sytle.css」、「style.scss」も追加します。

▼style.css

```
div {
  background-color: aqua;
}
```

▼ style.scss

```
$primary-color: #f00;

div {
  color: $primary-color;
}
```

適当な画像ファイルを用意し、「src/assets/images」フォルダを作成し追加します。

コマンドを「package.json」に、スクリプトとして追加します。

▼ package.json のスクリプト部分

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build": "webpack --mode=production",
  "build:dev": "webpack --mode=development",
  "build:prod": "webpack --mode=production",
  "start": "webpack serve"
},
```

まずは、ファイルを出力しないでブラウザで表示します。

▼ ブラウザで表示

```
> npm run start
```

「div」要素の背景、文字色も「style.css」、「style.scss」から作成された「main.css」から反映されています。また、「index.html」には、作成された「main.js」、「main.css」を読み込む部分はありませんが、webpack が「HtmlWebpackPlugin」で自動で読込部分が追加されています。

localhost:3000
Hello world!
Tip: Check your console
webpack 動いてるお〜

▲図 2.6: desc

次にプロダクション用にビルドしてみます。

▼ビルド

```
> npm run build
```

下図のように、

- index.html
- main.js
- main.css
- images/画像ファイル

が出力されていますので、ファイル開き内容を確認してください。

package.json — yaruo-start-template

EXPLORER

OPEN EDITORS
- index.js src M
- style.css src U
- style.scss src U
- yaruo.png src/assets/ima... U
- package.json M
- webpack.config.js U

YARUO-START-TEMPL...
- node_modules
- public
  - images
    - yaruo.png U
  - index.html U
  - main.css U
  - main.js U
- src
  - assets / images
    - yaruo.png U
  - index.js M
  - index.ts U
  - style.css U
  - style.scss U
- .gitignore
- index.html U
- package-lock.json M
- package.json M
- README.md
- webpack.config.js U

index.js M | style.css U | style.scss U | yaruo.png U | package.json M | webpack.config.js U

package.json > {} scripts

```
1  {
2    "name": "yaruo-start-template",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
     ▷ Debug
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1",
8      "build": "webpack --mode=production",
9      "build:dev": "webpack --mode=development",
10     "build:prod": "webpack --mode=production",
11     "start": "webpack serve"
12   },
13   "repository": {
14     "type": "git",
15     "url": "git+https://github.com/yaruo-react-redux/yaruo-start-template.git"
16   },
17   "keywords": [],
18   "author": "",
19   "license": "ISC",
20   "bugs": {
21     "url": "https://github.com/yaruo-react-redux/yaruo-start-template/issues"
22   },
23   "homepage": "https://github.com/yaruo-react-redux/yaruo-start-template#readme",
24   "devDependencies": {
25     "@webpack-cli/generators": "^2.4.1",
26     "css-loader": "^6.5.1",
27     "css-minimizer-webpack-plugin": "^3.2.0",
28     "html-webpack-plugin": "^5.5.0",
29     "mini-css-extract-plugin": "^2.4.5",
30     "prettier": "^2.5.0",
31     "sass": "^1.44.0",
32     "sass-loader": "^12.3.0",
33     "style-loader": "^3.3.1",
34     "terser-webpack-plugin": "^5.2.5",
35     "ts-loader": "^9.2.6",
36     "typescript": "^4.5.2",
37     "uglify-js": "^3.14.5",
38     "webpack": "^5.65.0",
39     "webpack-cli": "^4.9.1",
40     "webpack-dev-server": "^4.6.0",
41     "webpack-merge": "^5.8.0"
42   },
```

PROBLEMS　OUTPUT　DEBUG CONSOLE　TERMINAL　　zsh - yaruo-start-template

```
❯ npm run build

> yaruo-start-template@1.0.0 build
> webpack --mode=production

assets by chunk 1.39 MiB (name: main)
  asset main.js 1.39 MiB [emitted] [minimized] [big] (name: main)
  asset main.css 36 bytes [emitted] [minimized] (name: main)
asset images/yaruo.png 118 KiB [emitted] [from: src/assets/images/yaruo.png] (auxiliary name: main)
asset index.html 240 bytes [emitted]
Entrypoint main [big] 1.39 MiB (118 KiB) = main.css 36 bytes main.js 1.39 MiB 1 auxiliary asset
orphan modules 4.26 KiB (javascript) 1.83 KiB (runtime) [orphan] 14 modules
runtime modules 1.83 KiB 6 modules
cacheable modules 532 KiB (javascript) 118 KiB (asset) 48 bytes (css/mini-extract)
  modules by layer 532 KiB (javascript) 118 KiB (asset)
    ./src/index.js 433 bytes [built] [code generated]
    ./node_modules/lodash/lodash.js 531 KiB [built] [code generated]
    ./src/assets/images/yaruo.png 42 bytes (javascript) 118 KiB (asset) [built] [code generated]
  css modules 48 bytes
    css ./node_modules/css-loader/dist/cjs.js!./src/style.css 34 bytes [built] [code generated]
    css ./node_modules/css-loader/dist/cjs.js!./node_modules/sass-loader/dist/cjs.js!./src/style.scss 14 bytes [built
] [code generated]

WARNING in asset size limit: The following asset(s) exceed the recommended size limit (244 KiB).
This can impact web performance.
Assets:
  main.js (1.39 MiB)
```

OUTLINE
TIMELINE
NPM SCRIPTS

03_setup-webpack-config-file*　⊗ 0 ⚠ 0　-- INSERT --　Ln 10, Col 47　Spaces: 2　UTF-8　LF　{} JSON　✓ Prettier

ここまでの内容は、GitHub 上で、以下のコマンドでクローンできます。

▼ GitHub

```
> git clone -b 03_setup-webpack-config-file https://github.com/yaruo-
›react-redux/yaruo-start-template.git
```

♡| 2.2.5　webpack 設定ファイルを分割する

問題なく動作した「webpack.config.js」ですが、今後の運用を考え「開発用」、「プロダクション用」、「共通分」に切り分けます。devServer 関連はプロダクションには関係ありませんし、minimizer 関連は開発時には邪魔です。

本家でも推奨[*2]されています。

「webpack.config.js」を以下のよう分割し、共用部分は「webpack-merge」を使用して統合します。

- 共用 webpack.common.js
- 開発用 webpack.dev.js
- プロダクション用 webpack.prod.js

開発用は devServer 関係、プロダクション用は minimizer 関係、それ以外は共用として分けていきます。

「webpack.dev.js」を作成し、「webpack.config.js」全体を貼り付け devServer、debtool を残し、「module」は CSS 関係のみで「style-loader」を使うように変更します。

また、「mode:'development'」を追加します。

▼ webpack.dev.js

```
const { merge } = require('webpack-merge');
const common = require('./webpack.common');
const os = require('os');

let devBrowser = 'Google Chrome';
switch (os.platform()) {
  case 'win32':
    devBrowser = 'chrome';
```

[*2] https://webpack.js.org/guides/production/

```
      break;
    case 'linux':
      devBrowser = 'google-chrome';
      break;
    default:
      break;
  }

  module.exports = merge(common, {
    mode: 'development',
    module: {
      rules: [
        {
          test: /\.css$/i,
          use: ['style-loader', 'css-loader'],
        },
        {
          test: /\.s[ac]ss$/i,
          use: ['style-loader', 'css-loader', 'sass-loader'],
        },
      ],
    },
    devtool: 'eval-source-map',
    devServer: {
      open: {
        app: {
          name: devBrowser,
        },
      },
      host: 'localhost',
      port: 3000,
      static: './public',
    },
  });
```

プロダクション用も、「webpack.config.js」全体を貼り付け、CssMinimizer 関連を中心に「module」は CSS の抽出のままで不要な部分を削除します。

こちらは、「mode:'production'」を追加します。

▼webpack.prod.js

```
const { merge } = require('webpack-merge');
const common = require('./webpack.common');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
const TerserPlugin = require('terser-webpack-plugin');
const CssMinimizerPlugin = require('css-minimizer-webpack-plugin');
```

```
module.exports = merge(common, {
  mode: 'production',
  plugins: [new MiniCssExtractPlugin(), new CssMinimizerPlugin()],
  module: {
    rules: [
      {
        test: /\.css$/i,
        use: [MiniCssExtractPlugin.loader, 'css-loader'],
      },
      {
        test: /\.s[ac]ss$/i,
        use: [MiniCssExtractPlugin.loader, 'css-loader', 'sass-loader'],
      },
    ],
  },
  resolve: {
    extensions: ['.tsx', '.ts', '.js'],
  },
  optimization: {
    minimize: true,
    minimizer: [
      new TerserPlugin({
        minify: TerserPlugin.uglifyJsMinify,
        terserOptions: {
          compress: {
            drop_console: true,
          },
        },
      }),
      new CssMinimizerPlugin(),
    ],
  },
});
```

共通部分も、「webpack.config.js」全体を貼り付け、上記ファイルにあるものを削除します。

▼webpack.common.js

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

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'public'),
    assetModuleFilename: 'images/[name][ext][query]',
    clean: true,
```

```
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: 'index.html',
    }),
  ],
  module: {
    rules: [
      {
        test: /\.(ts|tsx)$/i,
        loader: 'ts-loader',
        exclude: ['/node_modules/'],
      },
      {
        test: /\.(eot|svg|ttf|woff|woff2|png|jpg|gif)$/i,
        type: 'asset',
      },
    ],
  },
  resolve: {
    extensions: ['.tsx', '.ts', '.js'],
  },
};
```

webpack の設定ファイル名がデフォルトから変更になったので、「package.json」のスクリプト部分を変更します。

▼package.json のスクリプト部分

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build": "webpack --config webpack.prod.js",
  "build:dev": "webpack --config webpack.dev.js",
  "build:prod": "webpack --config webpack.prod.js",
  "start": "webpack serve --config webpack.dev.js"
},
```

ターミナル上で、「npm run start」、「npm run build」で動作確認を行います。

ここまでの内容は、GitHub 上で、以下のコマンドでクローンできます。

▼GitHub

```
> git clone -b 04_webpack-config-split https://github.com/yaruo-react›
›-redux/yaruo-start-template.git
```

♡| 2.2.6　Babel.js のインストールと設定

Babel.js とは、Babel.js のトップページの例にあるように、モダン JS(ES2015 移行の JavaScript) を未対応の古いブラウザでも解釈できるような JavaScript に変換してくれるトランスコンパイラ (変換器) です。

▲図 2.7: Babel.js 本家ページ

それでは、プロジェクトに「Babel.js」を導入していきます。

Babel.js のトップページの上部にあるメニューの「Setup」をクリックします。すると、手順に従うように番号のついた案内ページが表示されます。

このプロジェクトでは、「webpack」を使用しますので、「webpack」ボタンをクリックします。

手順 2〜4 が憑依されましたので、順に実行していきます。

▲図 2.8: 使用するツールを選択

2 Installation

Shell　Copy

```shell
npm install --save-dev babel-loader @babel/core
```

3 Usage

Via config

JavaScript　Copy

```javascript
{
  module: {
    rules: [
      {
        test: /\.m?js$/,
        exclude: /node_modules/,
        use: {
          loader: "babel-loader",
          options: {
            presets: ['@babel/preset-env']
          }
        }
      }
    ]
  }
}
```

> For more information see the babel/babel-loader repo.

4 Create `babel.config.json` configuration file

Great! You've configured Babel but you haven't made it actually *do* anything. Create a babel.config.json config in your project root and enable some presets.

To start, you can use the env preset, which enables transforms for ES2015+

Shell　Copy

```shell
npm install @babel/preset-env --save-dev
```

In order to enable the preset you have to define it in your `babel.config.json` file, like this:

JSON　Copy

```json
{
  "presets": ["@babel/preset-env"]
}
```

▲図 2.9: パッケージの pa

最初に指示されているパッケージをインストールします。「--save-dev」は省略すると「-D」と書きます。右端にあるコピーアイコンをクリックしてターミナルに貼り付けても良いです。

▼Babel.js パッケージインストール

```
> npm install --save-dev babel-loader @babel/core
```

次に、3 のコードを「webpack.common.js」へコピーします。コピー後の「webpack.common.js」です。

▼webpack.common.js,Babel ローダの導入

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

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'public'),
    assetModuleFilename: 'images/[name][ext][query]',
    clean: true,
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: 'index.html',
    }),
  ],
  module: {
    rules: [
      {
        test: /\.(ts|tsx)$/i,
        loader: 'ts-loader',
        exclude: ['/node_modules/'],
      },
      {
        test: /\.m?js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-env'],
          },
        },
      },
      {
        test: /\.(eot|svg|ttf|woff|woff2|png|jpg|gif)$/i,
        type: 'asset',
      },
```

```
      ],
    },
    resolve: {
      extensions: ['.tsx', '.ts', '.js'],
    },
  };
```

次に、手順 4 に進みます。最初に「@babel/preset-env」パッケージをインストールします。

▼@babel/preset-env のインストール

```
> npm install --save-dev @babel/preset-env
```

インストールが完了しましたら、プロジェクトフォルダ直下に「babel.config.json」ファイルを作成し、手順 4 の内容を貼り付けます。

▼babel.config.json

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

以上で、Babel.js の導入は完了です。

動作を確認してみる。

「src/index.js」に以下のコードを追加します。ES6 で新しく使えるようになったテンプレートリテラルを使用したコードです。

▼src/index.js

```
import _ from 'lodash';
import './style.css';
import './style.scss';

import Yaruo from './assets/images/yaruo.png';

function component() {
  const element = document.createElement('div');

  // Lodash, now imported by this script
  const myName = 'やる夫';
  const words = 'こまけぇこたぁいいんだよ〜';
  const message = `<br />${myName}の口グセなのか？ <br />${words}`;
  element.innerHTML = _.join(['webpack', '動いてるお〜'], ' ') + message;
```

```
  return element;
}

document.body.appendChild(component());

const image = new Image();
image.src = Yaruo;

document.body.appendChild(image);
```

Babel.js 導入前の実行結果です。

▲図 2.10: Babel.js 導入前ですので、そのまま

Babel.js 導入後の実行結果です。

▲図 2.11: concat に変換されています。

ここまでの内容は、GitHub 上で、以下のコマンドでクローンできます。

▼GitHub

```
> git clone -b 05_babel-install https://github.com/yaruo-react-redux/
yaruo-start-template.git
```

2.3 eslint、prettier とは？

「lint」は、C 言語用のコンパイラよりも詳細で厳密なチェックを行うプログラムです。コンパイル前にコードをチェックするために使われます。

それが、いつしかコードをチェック・解析することを「lint」、lint を行うプログラムを linter と呼ぶようになったそうです。

JavaScript(ECMAScript) 用の linter が、「eslint」になります。もちろん、Java、HTML、Python などにも linter があります。

「eslint」は、設定ファイルで指定されたルールと違うコードの書き方をしている部分を指摘してくれます。その指定されたルールとは、たいていの場合には JavaScript に詳しい人達が決めたもので、良く使われるものは、かの有名な AirBnB の開発チームのものです。もちろん、ルールは改変・追加もできます。

チェックしてくれるのは、たとえば、

- const で宣言している変数への代入
- 未定義の変数やモジュールの使用
- 分割代入の使用を推奨

などがありますが、何をチェックし指摘するのかは、チーム毎、プロジェクト毎に自由に決めることができます。

「prettier」は、コードを整形 (インデント、改行など) してくれるツールです。実は、eslint でもコード整形はできるのですが、コード整形は prettier の方が優れいます。

そのために、

- コードチェックは、eslint
- コード整形は、prettier

と、得意なものに任せます。

♡| 2.3.1　eslint、prettier のインストール

eslint のパッケージ追加と設定

create-react-app を使用して作成したスタートアッププロジェクトには、eslint は導入済みですので設定し直し、必要な関連パッケージをインストールします。

ターミナルに以下のように「eslint --init」と初期化コマンドを入力します。

▼eslint の初期化

```
$ npx eslint --init
```

「?」が行頭にある質問と選択枝が表示されますので、カーソルキーで選択枝を選びエンターキーで次ぎの質問に移ります。

▼eslint の質問に答える

```
  ? How would you like to use ESLint? …
    To check syntax only
  > To check syntax and find problems     ←選択したものに　>　が表示される
    To check syntax, find problems, and enforce code style
```

最後の質問に答えると必要なパッケージをインストールするか尋ねられますので「Yes」と答えてます。

▼eslint への答え

```
  ☒ How would you like to use ESLint? ·  problems
  ☒ What type of modules does your project use? ·  esm
  ☒ Which framework does your project use? ·  react
  ☒ Does your project use TypeScript? ·  No / Yes        ← Yes を選択
  ☒ 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-e›
›slint/parser@latest eslint@latest
  ☒ Would you like to install them now with npm? ·  No / Yes  ← Yes を選択
```

▼package.json に eslint 関連のパッケージがインストールされました。

```
  "devDependencies": {
    "@typescript-eslint/eslint-plugin": "^5.4.0",
    "@typescript-eslint/parser": "^5.4.0",
    "eslint": "^8.2.0",
    "eslint-plugin-react": "^7.27.0"
  }
```

また、eslint の設定ファイル「.eslintrc.js」が作成されています。

▼ .eslint.js

```
module.exports = {
    "env": {
        "browser": true,
        "es2021": true
    },
    "extends": [
      "eslint:recommended"
      "plugin:react/recommended",
      "plugin:@typescript-eslint/recommended",
    ],
    "parser": "@typescript-eslint/parser",
    "parserOptions": {
        "ecmaFeatures": {
            "jsx": true
        },
        "ecmaVersion": 12,
        "sourceType": "module"
    },
    "plugins": [
        "react",
        "@typescript-eslint"
    ],
    "rules": {
    }
};
```

設定ファイル「.eslintrc.js」で、どのようなルールが適用されるのかを確認します。適用されるルールが、「current_rules.txt」に書き出されます。

書き出されたルールは、ルール名に適用方法{"off(適用しない)","warn(警告)","error(エラー)"}が記されています。表記は、{0,1,2}の数字で表示される場合もあります。同じルールがあった場合には、後から読み込まれたものが上書きされます。個別に上書きしたいものは「.eslintrc.js」ファイルの「rules」セクションに追加します。

▼ eslint 設定で適用されるルール

```
$ npx eslint --print-config .eslintrc.js > current_rules.txt
```

eslint で使用するルールは一般的なものをベースにしたいので、airbnb のルールをインストールします。

▼ airbnb のルーツのインストール

```
$  npx install-peerdeps --dev eslint-config-airbnb
   install-peerdeps v3.0.3
   It seems as if you are using Yarn.
   Would you like to use Yarn for the installation? (y/n) n← yarn を使っているのか聞かれるので、no である「n」を入力
```

airbnb のルールをインストールしたので、設定ファイルに追加します。

▼ .eslintrc.js へ airbnb ルールを適用

```
  "extends": [
      "eslint:recommended",
      "plugin:react/recommended",
      "airbnb",        ← airbnb のルール
      "airbnb/hooks", ← airbnb の React hooks のルール
      "plugin:@typescript-eslint/recommended",
  ],
```

再度、ルールを出力すると適用されるルールがずいぶん増えているのが分かります。

次に、TypeScript もチェックできるようにルールを追加します。「plugin:」の下 3 行を追加しました。

▼ .eslintrc.js の extends 部分

```
  "extends": [
      "eslint:recommended",
      "plugin:react/recommended",
      "airbnb",
      "airbnb/hooks",
      "plugin:@typescript-eslint/recommended",
      "plugin:@typescript-eslint/recommended-requiring-type-checking",
      "plugin:import/recommended",
      "plugin:import/typescript",
  ],
```

TypeScript 用ルールを追加しましたので、「parserOptions」を以下のように変更する。

▼ .eslintrc.js の parserOptions 部分

```
  "parserOptions": {
    "ecmaFeatures": {
        "jsx": true
    },
    "ecmaVersion": 12,
    "sourceType": "module",
    "tsconfigRootDir": __dirname,
    "project": ["./tsconfig.json"],
  },
```

これでルールの適用は完了しましたが、都合の悪いルールには設定ファイルでルールの上書をします。

「import/extensions」は、インポート宣言で node_modules 以下にあるパッケージからは拡張子が不要 (import aaa from 'aaa') で、相対パスからの import は、拡張子が必要と言うルールです。

現在はすべてがエラー、node_modules 下のパッケージ内の指定された拡張子は除外となっていますが、node_modules 下以外でも{js,jsx,ts,tsx}は除外したいのでルールを追加します。

▼ import/extensions の現時点

```
  "import/extensions": [
    "error",
    "ignorePackages",
    {
      "js": "never",
      "mjs": "never",
      "jsx": "never"
    }
  ],
```

「react/jsx-filename-extension」は、JSX を含むファイルの拡張子を制限するルールです。

現時点では、拡張子「.jsx」に制限されていますが、拡張子「.tsx」も追加したいのでルールに追加します。

▼ react/jsx-filename-extention の現時点

```
  "react/jsx-filename-extension": [
    "error",
    {
      "extensions": [
        ".jsx"
      ]
    }
```

```
  ],
```

「react/react-in-jsx-scope」は、JSX ファイルに「import React from 'react'」がない場合にはエラーにしてくれるのですが React17 からは、「import React from 'react'」を書かなくてもよくなりました。そのため、このルールを OFF にします。

▼react/react-in-jsx-scope

```
  "react/react-in-jsx-scope": [
    "error"
  ],
```

「react/function-component-definition」は、関数コンポーネントに特定の関数タイプを強制します。現時点では、function の使用を強制されるので、アロー関数強制に変更します。

▼react/function-component-definition の現在

```
  "react/function-component-definition": [
    "error",
    {
      "namedComponents": "function-expression",
      "unnamedComponents": "function-expression"
    }
  ],
```

上書きしたいルールを、「.eslintrc.js」へ追加します。

▼.eslintrc.js の rules へ追加

```
  "rules": {
      "import/extensions": [
          "error",
          {
            js: "never",
            jsx: "never",
            ts: "never",
            tsx: "never",
          },
        ],
        "react/jsx-filename-extension": [
          "error",
          {
            extensions: [".jsx", ".tsx"],
          },
        ],
        "react/react-in-jsx-scope": "off",
        "react/function-component-definition": [
          "error",
```

```
        {
          namedComponents: "arrow-function",
          unnamedComponents: "arrow-function",
        },
      ],
}
```

Prettier のインストールと設定

ここからは、Prettier のインストールと設定をします。

▼ Prettier のインストール

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

インストールが完了すると、package.json に追加されます。

▼ package.json

```
"devDependencies": {
  "@typescript-eslint/eslint-plugin": "^5.4.0",
  "@typescript-eslint/parser": "^5.4.0",
  "eslint": "^8.2.0",
  "eslint-config-airbnb": "^19.0.0",
  "eslint-config-prettier": "^8.3.0",
  "eslint-plugin-import": "^2.25.3",
  "eslint-plugin-jsx-a11y": "^6.5.1",
  "eslint-plugin-react": "^7.27.0",
  "eslint-plugin-react-hooks": "^4.3.0",
  "prettier": "^2.4.1"
}
```

Pretter のチェックを「.eslintrc.js」へ追加します。

▼ .eslintrc.js

```
"extends": [
    "plugin:react/recommended",
    "airbnb",
    "airbnb/hooks",
    "plugin:@typescript-eslint/recommended",
    "plugin:@typescript-eslint/recommended-requiring-type-checking",
    "plugin:import/recommended",
    "plugin:import/typescript",
    "prettier",      ← prettier を追加
],
```

pritter の設定ファイル「.prettierrc」を追加します。設定可能なオプションは、Prettier オ

プション[*3]で確認できます。ほぼすべてがデフォルトでも良いのですが、create-react-app がシングルクオートなので設定します。

▼ .prettierrc

```
{
  "singleQuote": true,
  "jsxSingleQuote": true
}
```

eslint と prettier が衝突すると検出・修正ループに入りますので、チェックします。

▼ eslint、prettier の衝突検出

```
$ npx eslint-config-prettier 'src/**/*.{js,jsx,ts,tsx}'
  No rules that are unnecessary or conflict with Prettier were found.
```

無事に衝突なしとなりました。

package.json にスクリプトコマンドを追加します。

▼ package.json

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "lint": "eslint 'src/**/*.{js,jsx,ts,tsx}'", ← lint:チェック
  "fix": "npm run format && npm run lint:fix", ← fix:整形してチェックして自動修復
  "format": "prettier --write 'src/**/*.{js,jsx,ts,tsx}'", ← format:整形
  "lint:fix": "eslint --fix 'src/**/*.{js,jsx,ts,tsx}'",  ← lint:fix チェック後修復

  "eject": "react-scripts eject"
},
```

Eslint、Prettier の設定が完了しましたので、src フォルダにある「App.tsx」を開いてみると、ルールから外れるものは指摘されています。

[*3] https://prettier.io/docs/en/options.html

2.4 eslint、prettier の指摘を修正

ESlint、Prettier は指摘するだけではなく、修正案の提示・修正 (できるものだけですが...) までしてくれます。

VSCode に Prettier 拡張機能を追加してあれば、以下のように、VSCode 側で設定すると、ファイルを保存する度に自動で修正をいれることもできます。

私は、修正を自分のタイミングで行いたいので VSCode 側の設定は行っていません。

もし、VSCode 側の設定をする場合には、VSCode で [File]->[Preferences]->[Settings] にて、以下の各項目を検索して設定するか、settings.json へ追加するか、このプロジェクトのみ適用の場合は、プロジェクトフォルダ直下に「.vscode」フォルダを作成し、「settings.json」ファイルへ書き込みます。
ユーザー設定ファイルの内容が、この設定で上書きされます。

▼ VSCode の設定

```
"editor.formatOnSave": true,
"[JavaScript]": {
  "editor.formatOnSave": false
},
"[JavaScriptreact]": {
  "editor.formatOnSave": false
},
"[typescript]": {
  "editor.formatOnSave": false
},
"[typescriptreact]": {
  "editor.formatOnSave": false
},
"editor.codeActionsOnSave": {
    "source.fixAll": true,
    "source.fixAll.eslint": false
},
"prettier.disableLanguages": ["JavaScript", "JavaScriptreact", "typescript", "type›
›scriptreact"],
```

VSCode 上で、

- 赤波線で指摘されている

- 問題タブに表示されている

ものを修正します。

まずは、.eslintrc.js 自体に問題があるようです。

赤波線の上にマウスポンタを置くと eslint のコード、この場合は「no-use-before-define」が表示されます。、さらに、「コマンドキー (Windows では、ctrl)　＋ ピリオド」を押すと、修正方法が提示されます。

.eslintrc.js ファイルでの指摘は、「es6 モジュールの書き方へ移行しろ！」とのことです。以下のように、.eslintrc.js を変更します。

▼.eslintrc.js

```
const config = {
    "env": {
        "browser": true,
        "es2021": true
    },
    "extends": [
        "eslint:recommended",
        "plugin:react/recommended",
        "airbnb",
        "airbnb/hooks",
        "plugin:@typescript-eslint/recommended",
        "plugin:@typescript-eslint/recommended-requiring-type-checking",
        "plugin:import/recommended",
        "plugin:import/typescript",
        "prettier",
    ],
    "parser": "@typescript-eslint/parser",
    "parserOptions": {
        "ecmaFeatures": {
            "jsx": true
        },
        "ecmaVersion": 12,
        "sourceType": "module",
        "tsconfigRootDir": __dirname,
        "project": ["./tsconfig.json"],
    },
    "plugins": [
        "react",
        "@typescript-eslint"
    ],
    "rules": {
        "import/extensions": [
```

```
            "error",
            {
              js: "never",
              jsx: "never",
              ts: "never",
              tsx: "never",
            },
          ],
          "react/jsx-filename-extension": [
            "error",
            {
              extensions: [".jsx", ".tsx"],
            },
          ],
          "react/react-in-jsx-scope": "off",
          "react/function-component-definition": [
            "error",
            {
              namedComponents: "arrow-function",
              unnamedComponents: "arrow-function",
            },
          ],
    }
};

export default config
```

このように修正して保存すると、次の指摘がきます。

> Parsing error: "parserOptions.project" has been set for @typescript-eslint/parser. The file does not match your project config: .eslintrc.js. The file must be included in at least one of the projects provided.

これは、ファイルが「どこからも import されていない」場合に表示されるエラーです。「.eslintrc.js」は、ESLint の設定ファイルですので、どこからもインポートされていません。

解消法は、「npx elsint --init」でファイルを作成した際に「.eslitrc」ファイルを json 形式、または、yaml(yml) 形式で作成を選択するか、.eslintrc.js ファイル自体をチェックの対象から除外します。

今回は、JavaScript 形式で作成したのでチェック除外のための、「.eslintignore」ファイルをプロジェクトフォルダ直下に作成し、lint.js や config.js のパターンが含まれるファイ

ル、パッケージがインストールされる node_modules フォルダなどを除外するように指定します。

▼ .eslintignore

```
build/
public/
**/node_modules/
*.config.js
.*lintrc.js
```

これで、.eslintrc.js については怒られなくなりました。

次に、App.tsx ファイルを修正します。

筆者が VSCode を日本語化していないのは、エラーメッセージでググる場合を考えてのことです。英語でのエラーメッセージの方が的確なページをみつけやすいと考えています。

では、指摘されている点を修正していきます。

「react/function-component-definition」は、関数コンポーネントに一貫した関数タイプを適用しなさいと怒られています。

関数をアロー関数に直し、関数型の宣言も追加します。

▼ App.tsx

```
// React17からは、JSXでReactのインポートが不要になりましたので、以下の行を削除します。
import React from 'react';
```

▼ App.tsx

```
import { VFC } from 'react';
import logo from './logo.svg';
import './App.css';

const App: VFC = () => (
  <div className="App">
    <header className="App-header">
      <img src={logo} className="App-logo" alt="logo" />
      <p>
        Edit <code>src/App.tsx</code> and save to reload.
      </p>
      <a
        className="App-link"
```

```
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );

  export default App;
```

これで現時点での指摘はすべて修正できました。

2.5 第 2 章のまとめ

React を使用したアプリケーションは、スタートアップ用のアプリケーションがコマンド一発でインストールできます。

バグの混入を防いだりより良いコーディングをするためにも、ESlint、Prettier を導入しましょう。

ここまでの内容は、GitHub 上で、以下のコマンドでクローンできます。

▼ GitHub

```
  $ > git clone -b 02_eslint_prettier https://github.com/yaruo-react-›
›redux/yaruo-diary.git
```

第 3 章

日記アプリケーションの作成 (React のみ)

本章では、第 2 章で作成したスタートアップ用のアプリケーションを魔改造し、日記アプリケーションを作成します。

3.1 React とは？

3.2　表示するデータの型を決める

3.3 データ表示画面

3.4 React hooks を使用して、データの追加・編集・削除

やる夫で学ぶ「react-redux」

redux-toolkit で、簡単・完璧理解だお･･･

2021 年 6 月 25 日　ver 1.0

著　者　気分はもう

© 2020 気分はもう

(powered by Re:VIEW Starter)

少しは分かったのか？

ヒャヒャヒャ！
何とかなるお･･･

ほんとうは、非同期でのデータの取得や
UI関連の説明もしたかったのじゃが
紙面の関係で次回にする。

―― エッ！

なんてこった･･･！

まだ、まだ学ぶことが
沢山あるのかお！

―― というわけで
ここまで読んでくれて
ありがとうございますお！