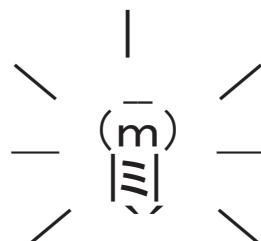


やる夫で学ぶ React、Reduxだお…

知識ゼロから環境構築するお。
Redux-toolkitまで学ぶお。



2022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～ wwwwww
キターネ～(* v 明日からはフロント担当だお…
TypeScript… ちよwww 吹いたwww
! あめでとう ハハハハ 僕も知りたいす!
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 12（2022 年）新刊
2022 年 1 月 22 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。

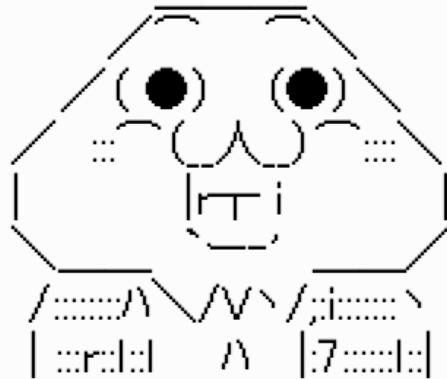
■商標

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、TM、[®]、[©]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 は簡単だよ…」を、手にしていただき誠にありがとうございます。



はじめまして、やる夫です。

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、古くなった情報は隨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、読書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 GitHub 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。GitHub 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

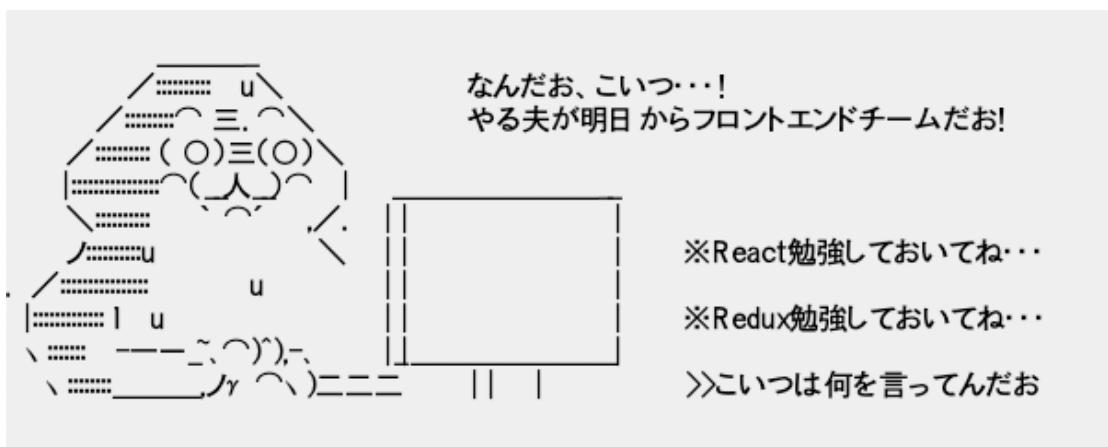
i

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	28
2.1.2 create-react-appで作成された中身	30
2.2 ゼロから構築してみる	33
2.2.1 ステップ1 Node.jsプロジェクト作成	33
2.2.2 webpackのインストールと設定	34
2.2.3 webpackの動作確認	37
2.2.4 webpackの設定ファイル	44
2.2.5 webpack設定ファイルを分割する	55
2.2.6 Babel.jsのインストールと設定	59
2.2.7 Reactのインストール	68
2.2.8 TypeScriptのインストール	74
2.3 eslint、prettierとは?	82
2.3.1 eslint、prettierのインストール	83
2.3.2 create-react-app作成のプロジェクトへ「eslint,prettier」を設定	92
2.4 eslint、prettierの指摘を修正	96

2.5	第2章のまとめ	101
第3章	日記アプリケーションの作成 (Reactのみ)	102
3.1	Reactとは?	104
3.2	表示するデータの型	107
3.3	Material Design 5の導入	109
3.3.1	MUIのインストール	110
3.3.2	データ表示画面を作る	111
3.3.3	MUI5のサイトからテンプレートを拝借	111
3.3.4	カード一覧画面の表示	120
3.3.5	リファクタリング1(サンプルデータ全表示)	123
3.3.6	リファクタリング2(カードヘッダを別コンポーネントへ)	126
3.3.7	入力フォームコンポーネントの作成	141
3.4	データの追加・編集・削除	153
3.4.1	削除・編集関数をPropsにデータと渡す	155
3.4.2	保存・キャンセル関数をPropsにデータと渡す	164
3.5	第3章のまとめ	177
第4章	日記アプリケーションの作成 (Redux使用)	178
4.1	Reduxとは?	178
4.1.1	Redux導入のメリット	179
4.2	Reduxの動作イメージ	180
4.3	Reduxの導入	182
4.4	Reduxのアプリケーションへの導入	183
4.5	コンポーネントの修正	190
4.6	第4章のまとめ	205
第5章	日記アプリケーションの作成 (Redux-toolkit使用)	206
5.1	Redux-toolkitの導入	206
5.2	Sliceの作成	207
5.3	トップコンポーネントに登録	210
5.4	コンポーネントから使用する	211
5.5	devToolsで確認	212
5.6	第5章のまとめ	214
第6章	おわりに	215

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。

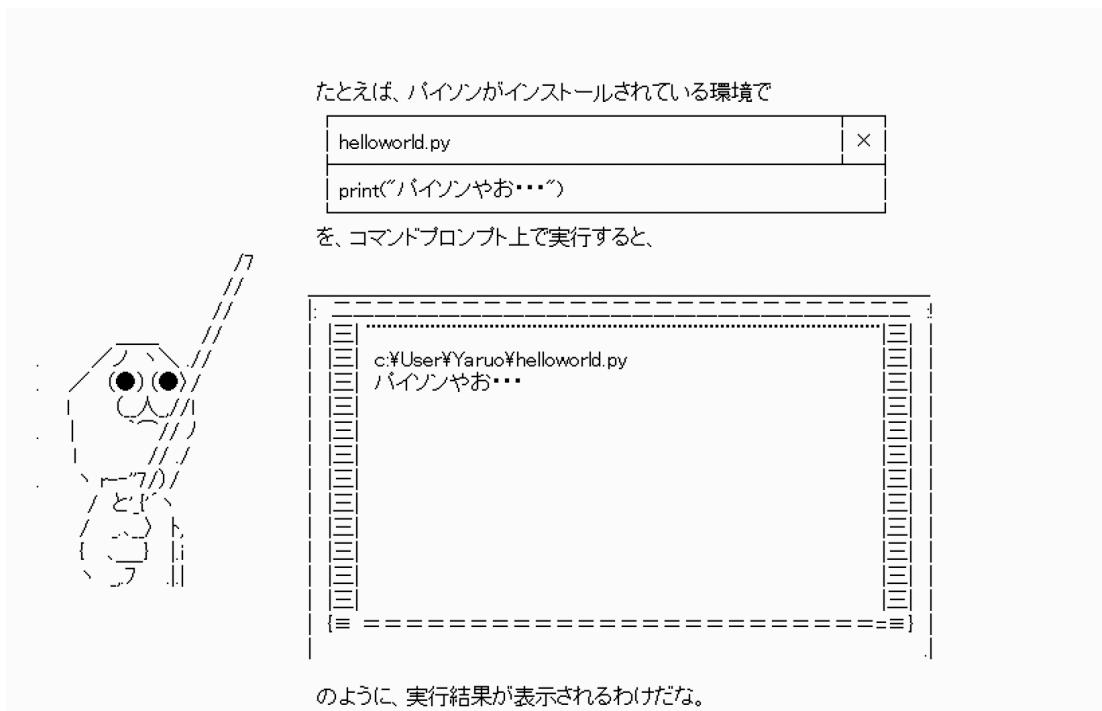
1.1

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='こまけえこたあいいんだよ';

console.log(`${name}が、こんなこと言っています。${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

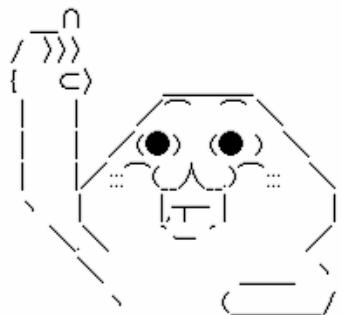
☒ node test01.js

やる夫が、こんなこと言ってます。こまけえこたあいいんだよ

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

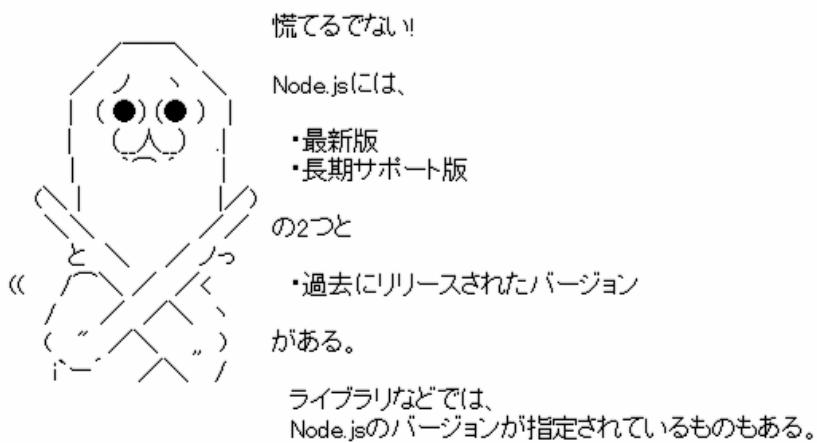


Node.jsについて



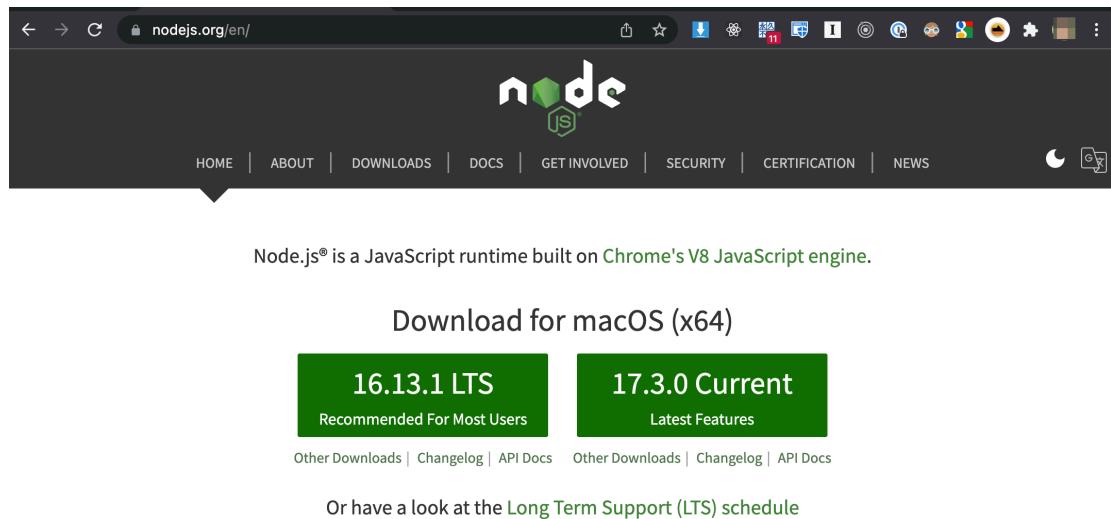
では、さっそくNode.jsを手に入れるお！

▲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。

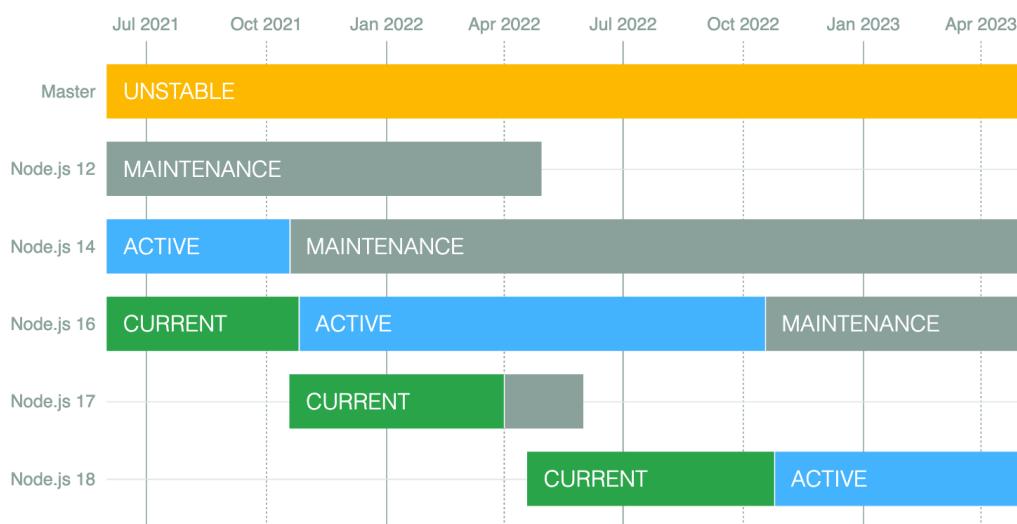
LTS版、最新版は以下のロードマップにより更新されます。



^{*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

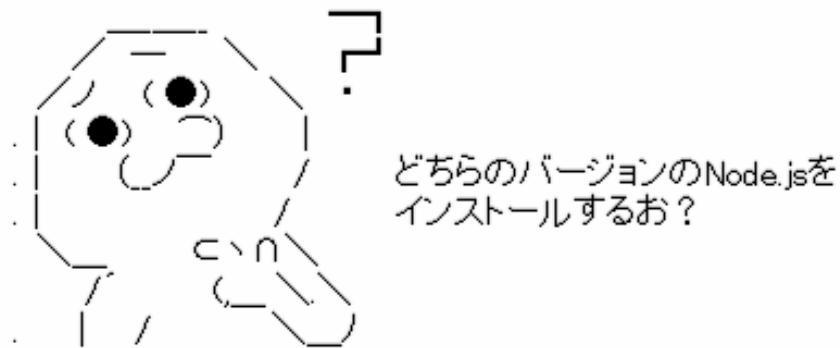
Node.js の Releases:<https://nodejs.org/ja/about/releases/> にあるように、Node.js は、各年の 4 月、10 月にリリースされ、

- 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。

*2 <https://qiita.com/heppokofrontend/items/5c4cc738c5239f4afe02>



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。

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 to
>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 node
>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 64
>bit version (defaults to system arch).
                                         Set [arch] to "all" to install 32 AND 64 bit versio
>nions.
                                         Add --insecure to the end of this command to bypass
>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 specify
>32/64bit architecture.
                                         nvm use <arch> will continue using the selected version,
>but switch to 32/64 bit mode.
```

```
nvm root [path]          : Set the directory where nvm should store different versions of Node.js.  
>t If <path> is not set, the current root will be displayed.  
>splayed.  
nvm version             : Displays the current running version of nvm for Windows. 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/release
```

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

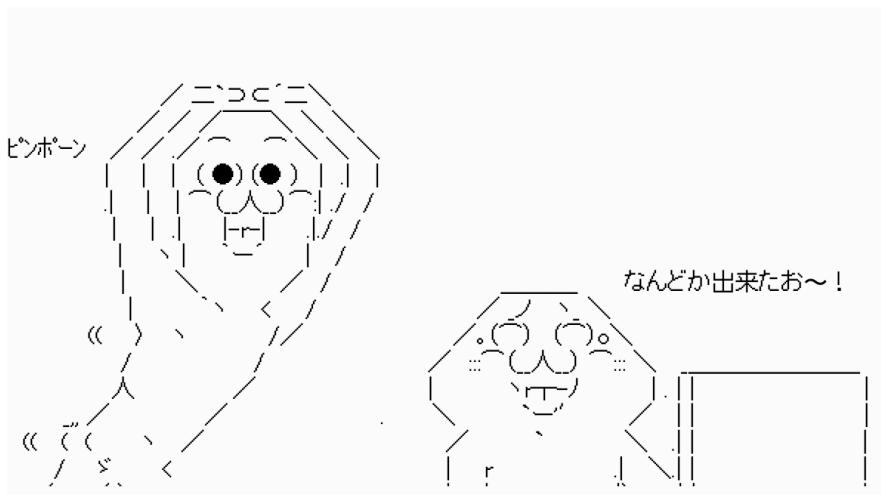
先ほどの「インストールされている Node.js のバージョン表示」で、現在使われている

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

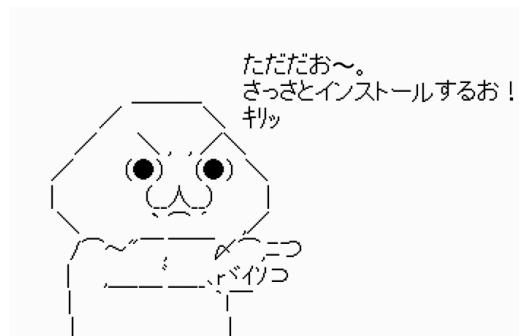
Microsoft Visual Studio Code + 拡張機能

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「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。

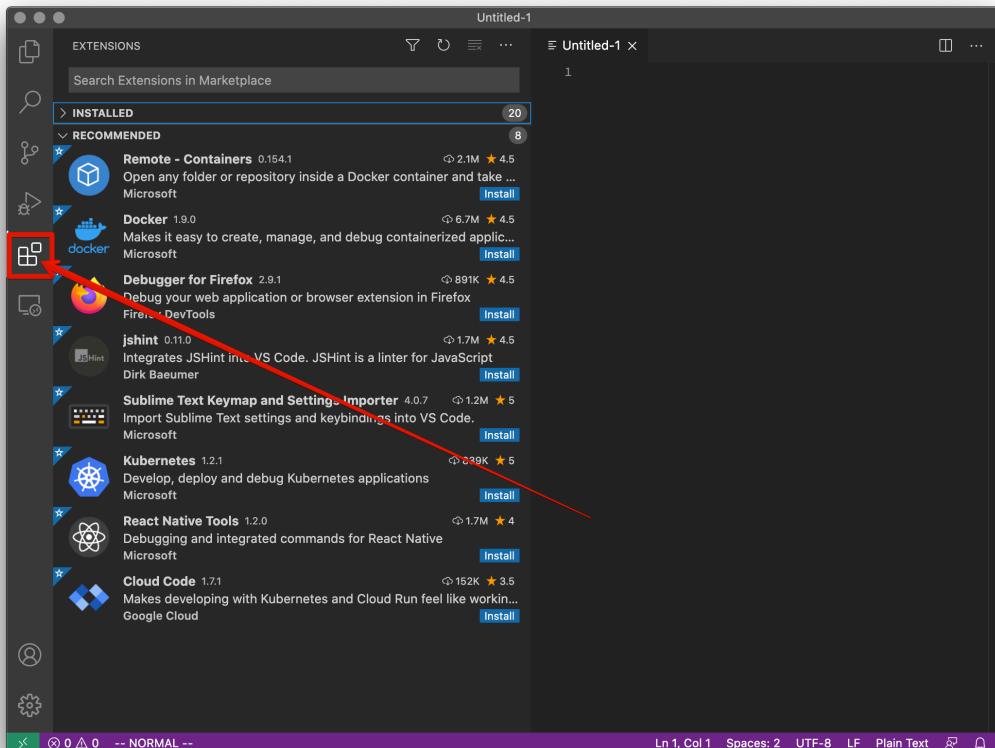
♡| 1.2.2 VSCode の拡張機能

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。

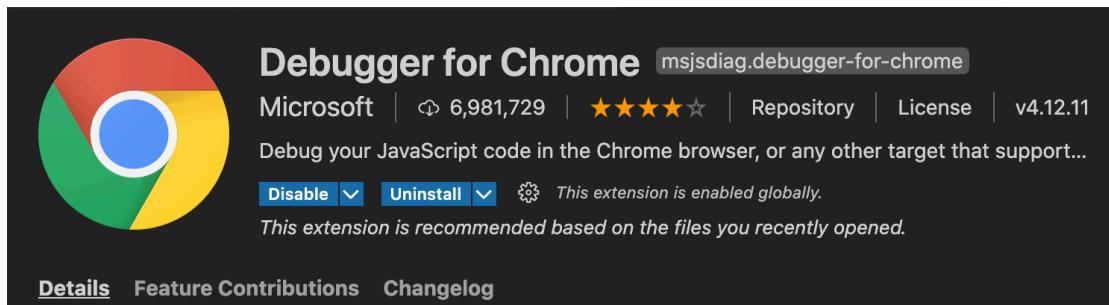


▲図 1.7: VSCode の拡張機能

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

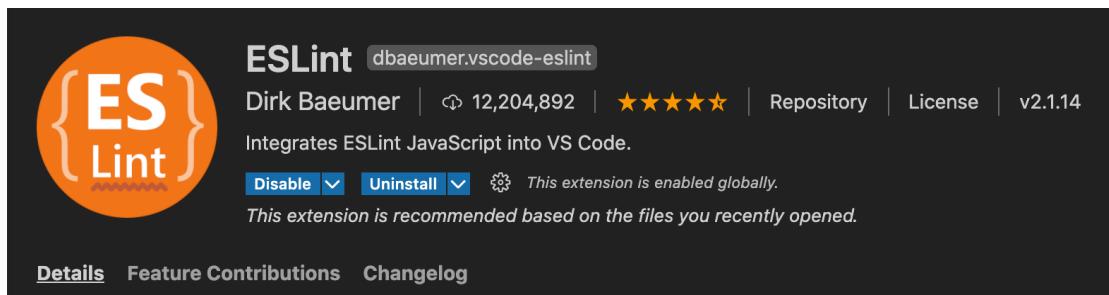
デバッグの際に、PC にインストールされている Chrome を自動で起動してくれます。Chrome の DevTools は、非常に強力です。また、Chrome にも React、Redux 用の拡張機能を追加すると更に便利になります。



▲図1.8: Beautify

♥| 1.2.3 Eslint

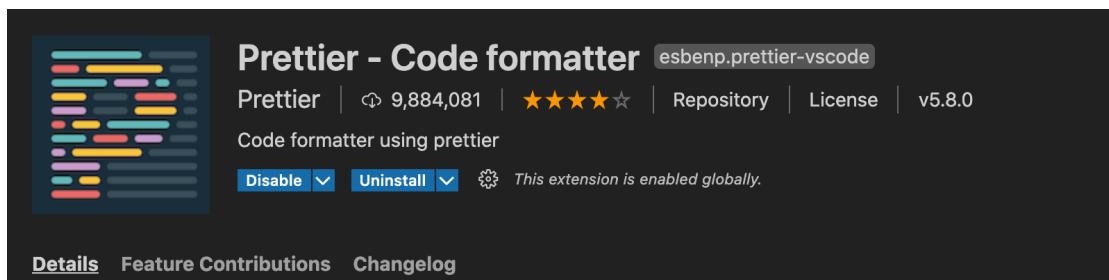
コード記法の間違いを指摘・修正してくれます。



▲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 ヘ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

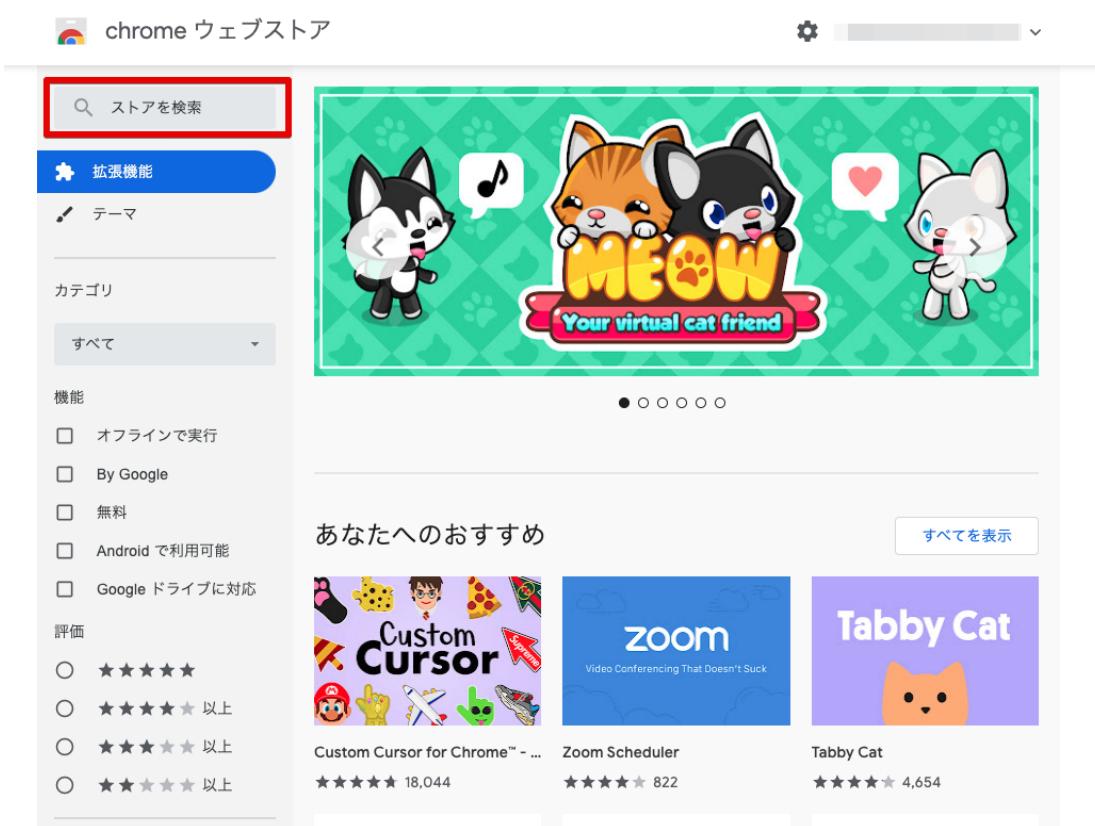
こちらも、VSCode と同様に拡張機能を追加することで、さらに便利に使うことができます。

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>

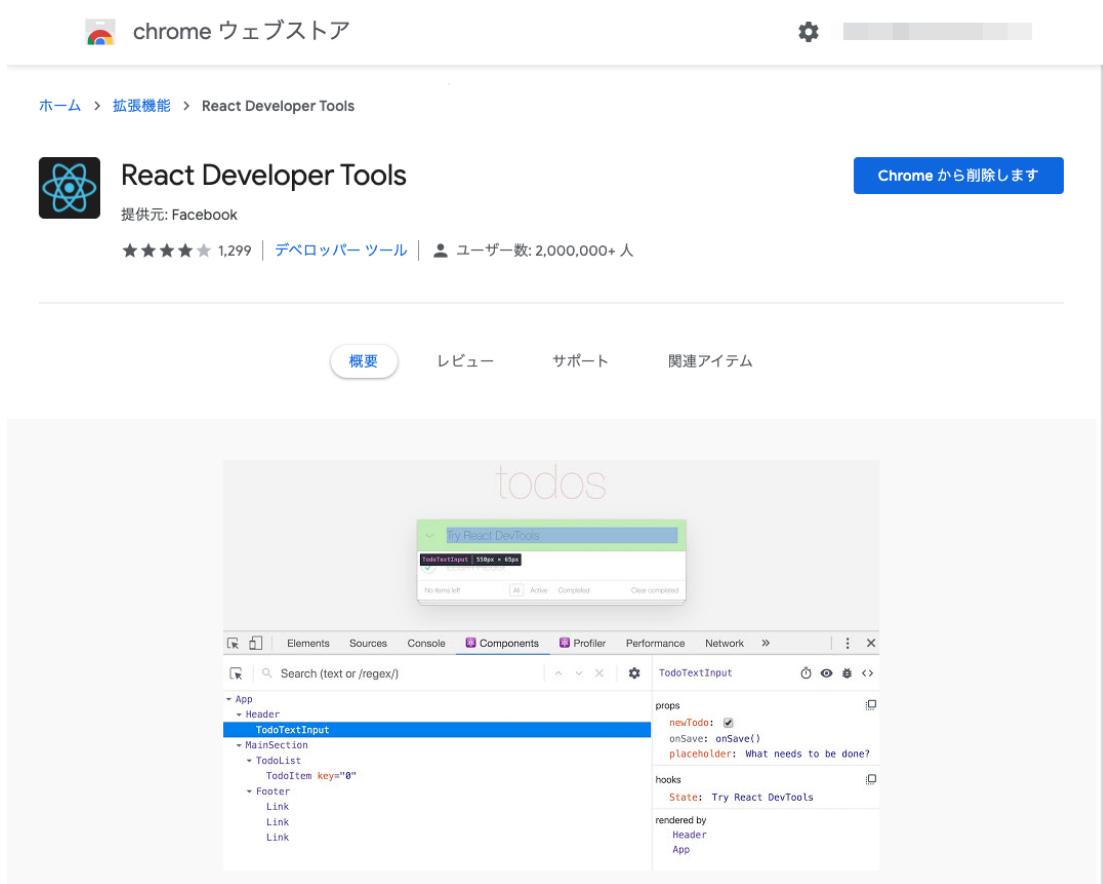
hl=ja



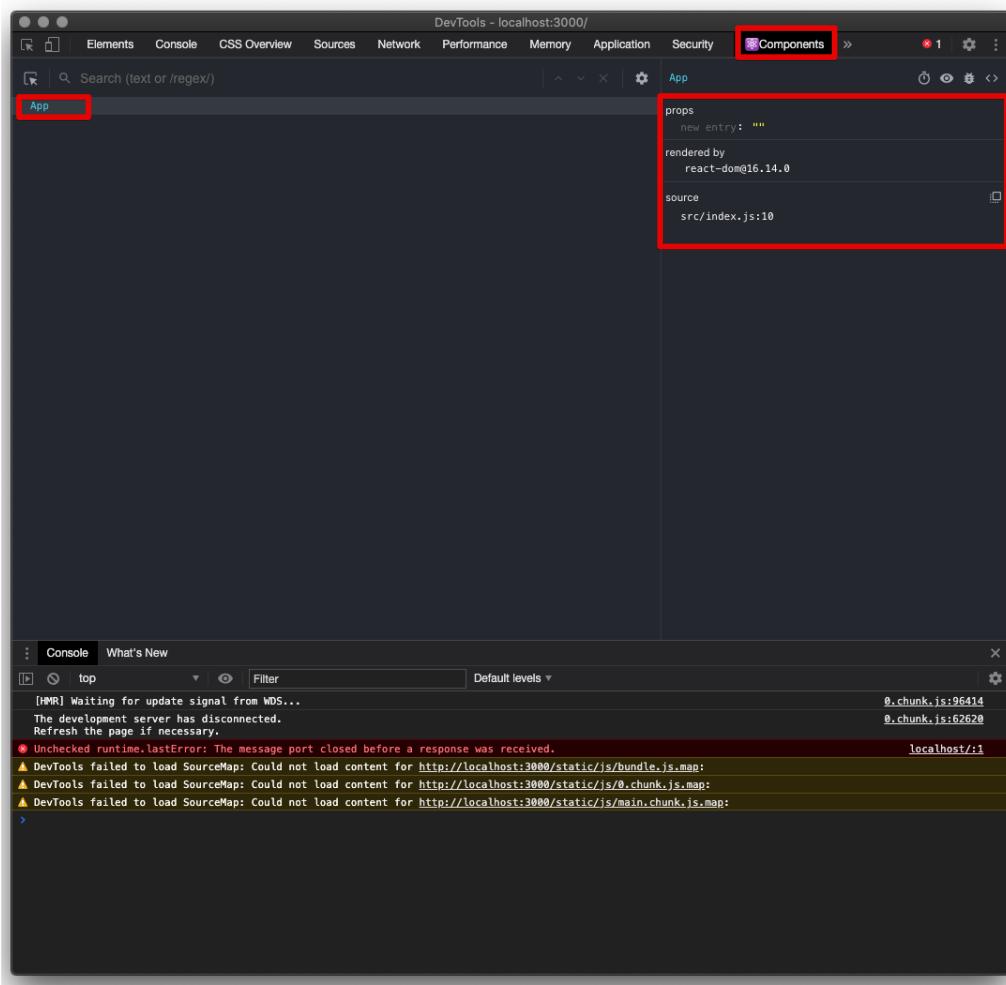
▲図 1.12: Chrome Web store

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。



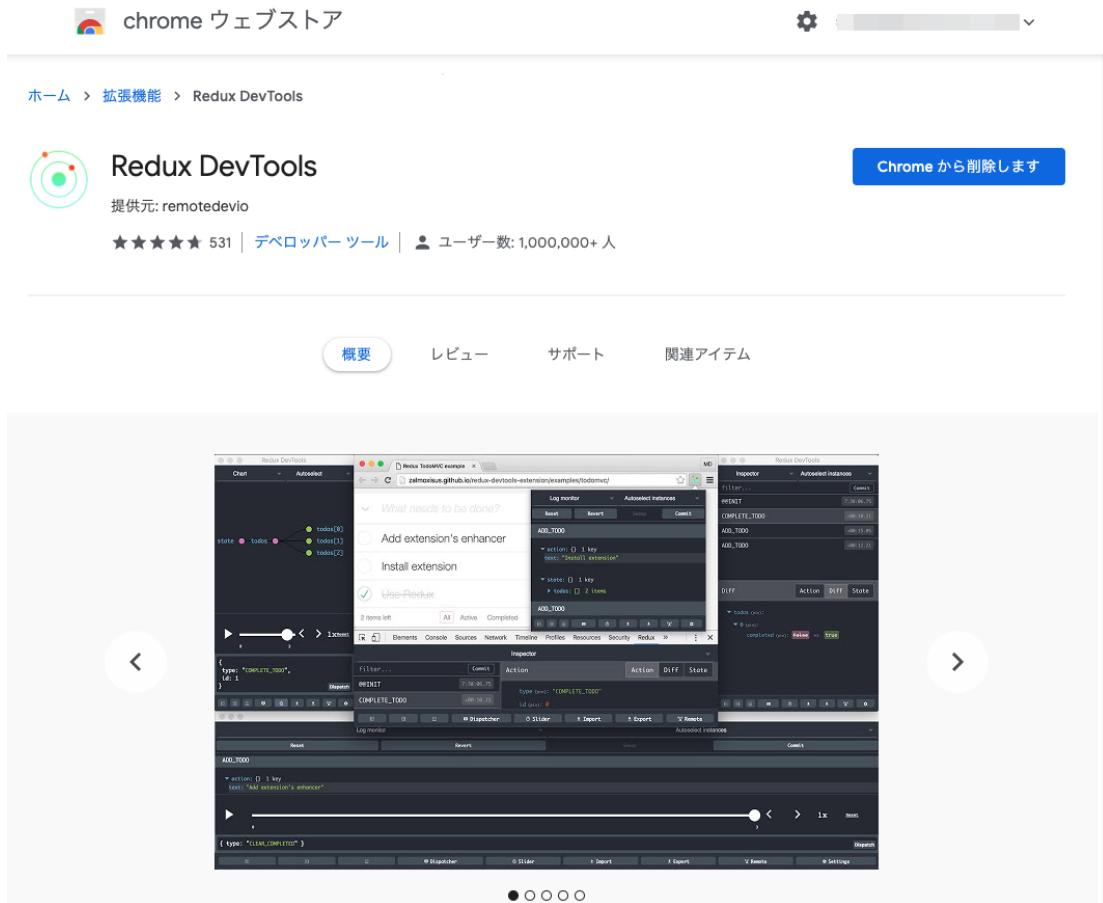
▲図 1.13: React Developer Tools



▲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 DevTools 拡張機能

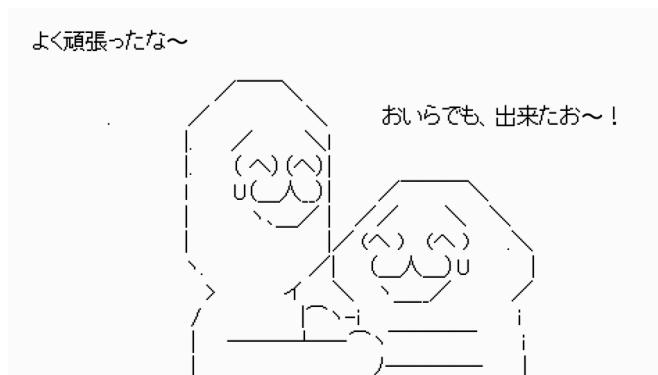
1.4

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)を中心に関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、すぐにもうコードを書き始めるここと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。



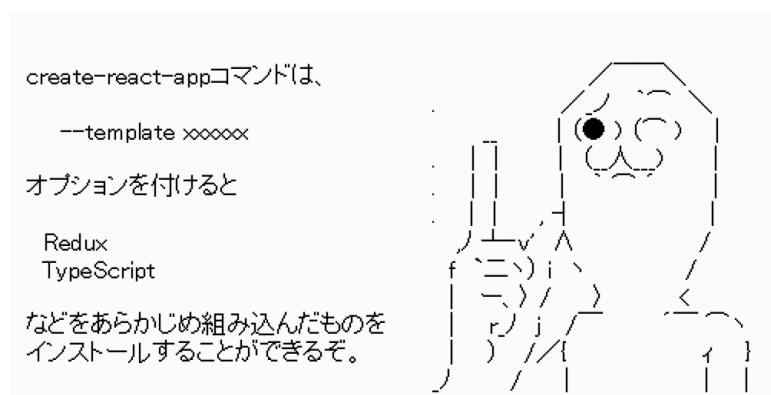
では、実際に手を動かしましょう。ターミナルを起動し、プロジェクトを作成するフォル

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/yaruo/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

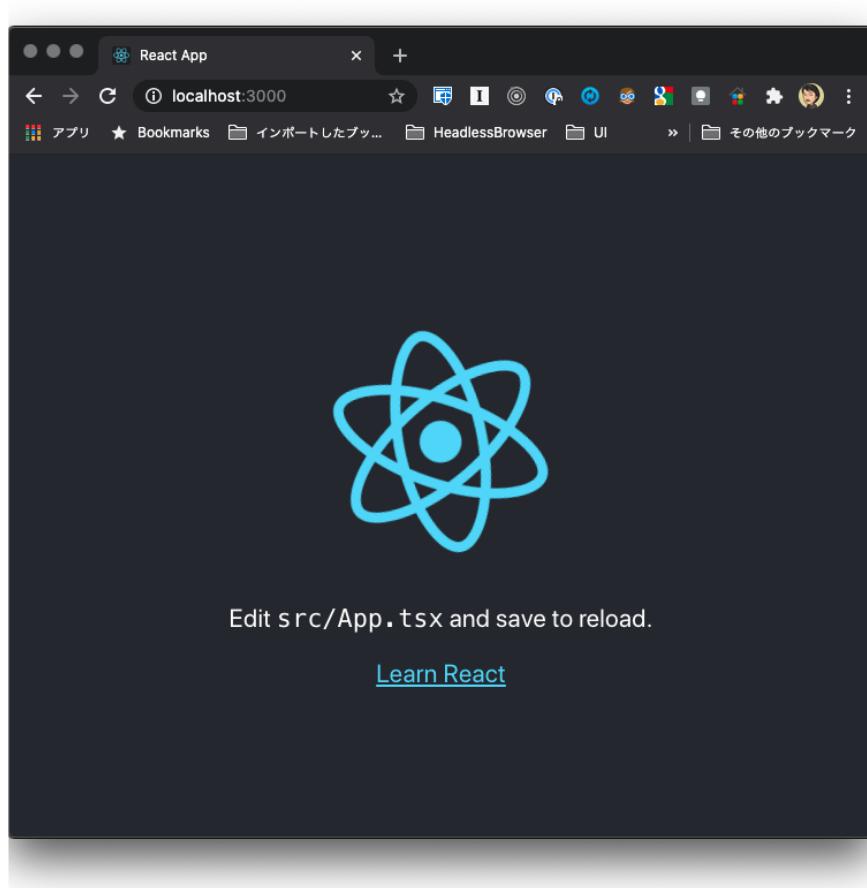
```
Compiled successfully!

You can now view your project 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

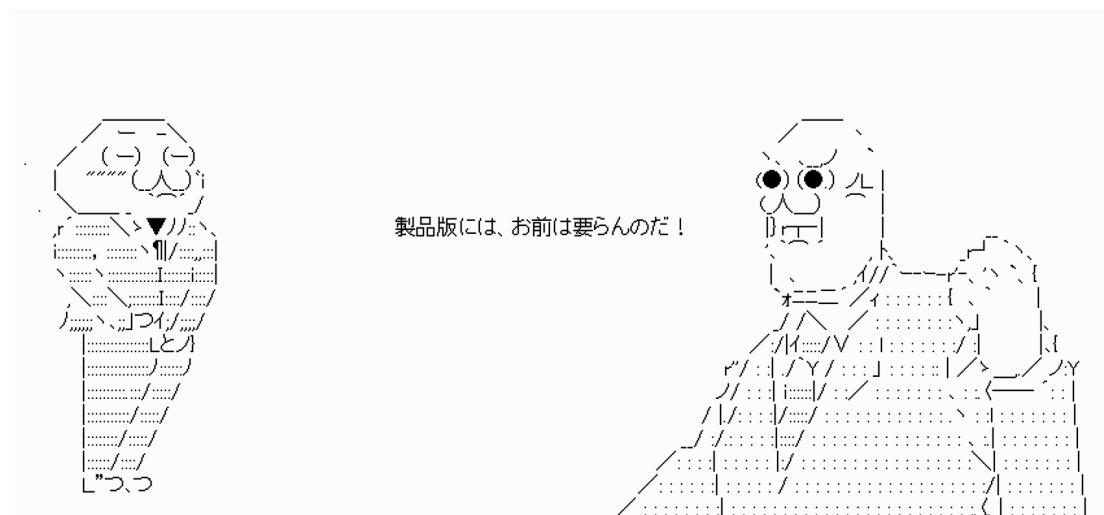
```
{  
  "name": "作成時に入力したプロジェクト名",  
  "version": "0.1.0",  
  "private": true,  
  "dependencies": {  
    "@testing-library/jest-dom": "^5.16.1",  
    "@testing-library/react": "^12.1.2",  
    "@testing-library/user-event": "^13.5.0",  
    "@types/jest": "^27.0.3",  
    "@types/node": "^16.11.14",  
    "@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。

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

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。

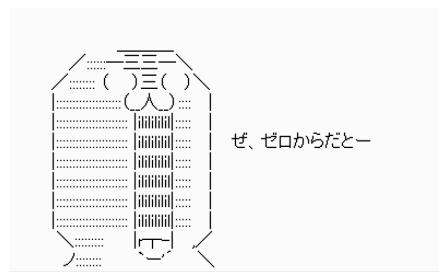


2.2

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おりますので、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"
}
```

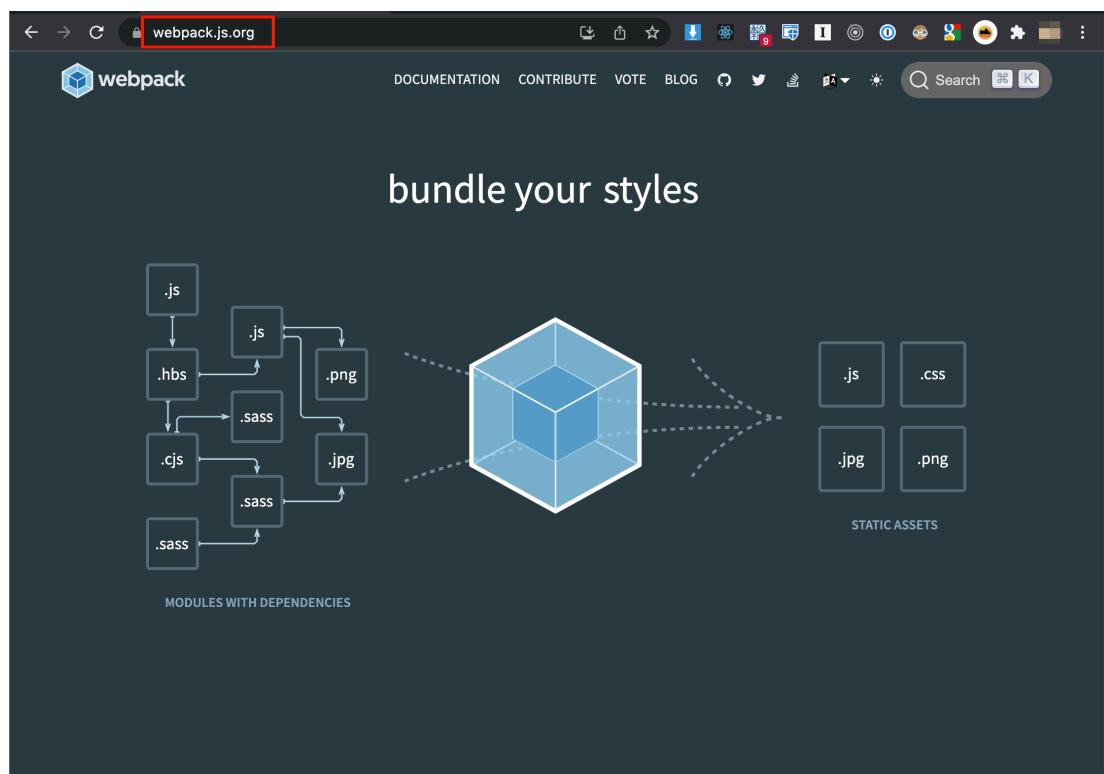
作成された「package.json」が表示されます。

ここまで的内容は、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

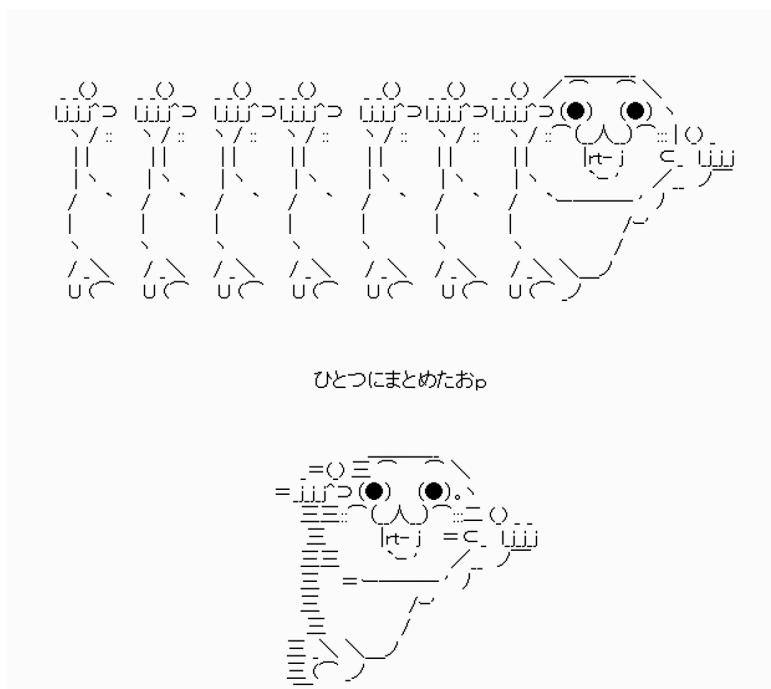


▲ 図 2.2: desc

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

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、

^{*1} <https://webpack.js.org/>

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

▼webpack 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。「-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": "^4.6.0"
  }
}
```

```
    "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webpack 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

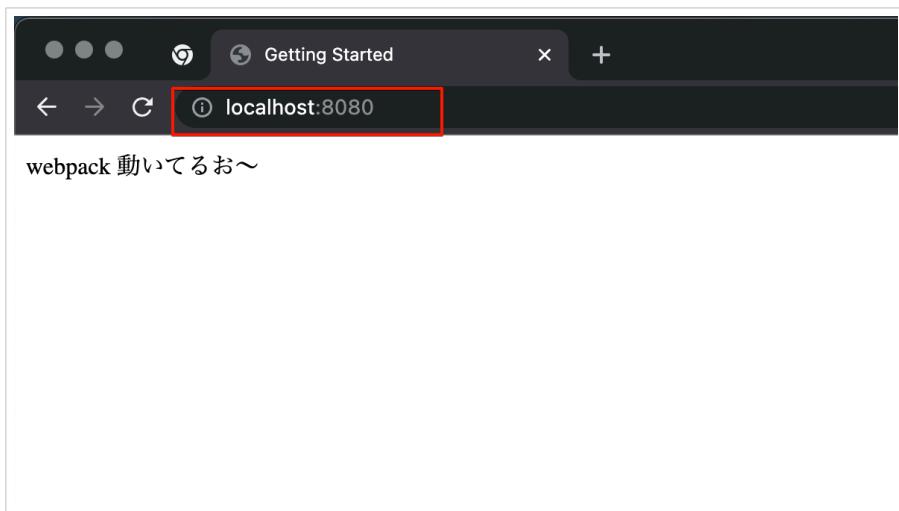
「--static-directory dist」を入力しているのは、devServer のデフォルト DocumentRoot は「public」のためです。

.....

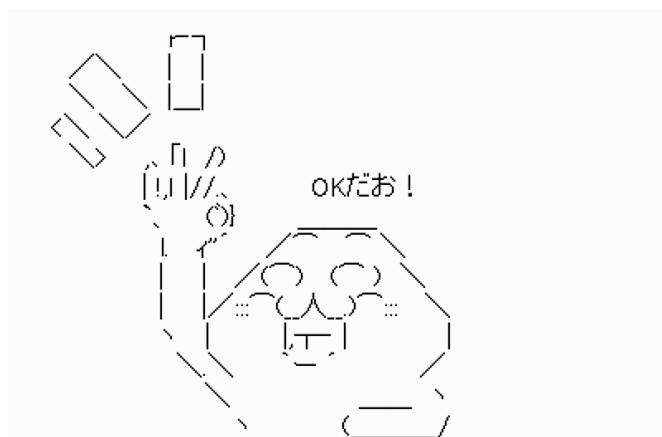
▼ webpack をコマンドで起動

```
☒ 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



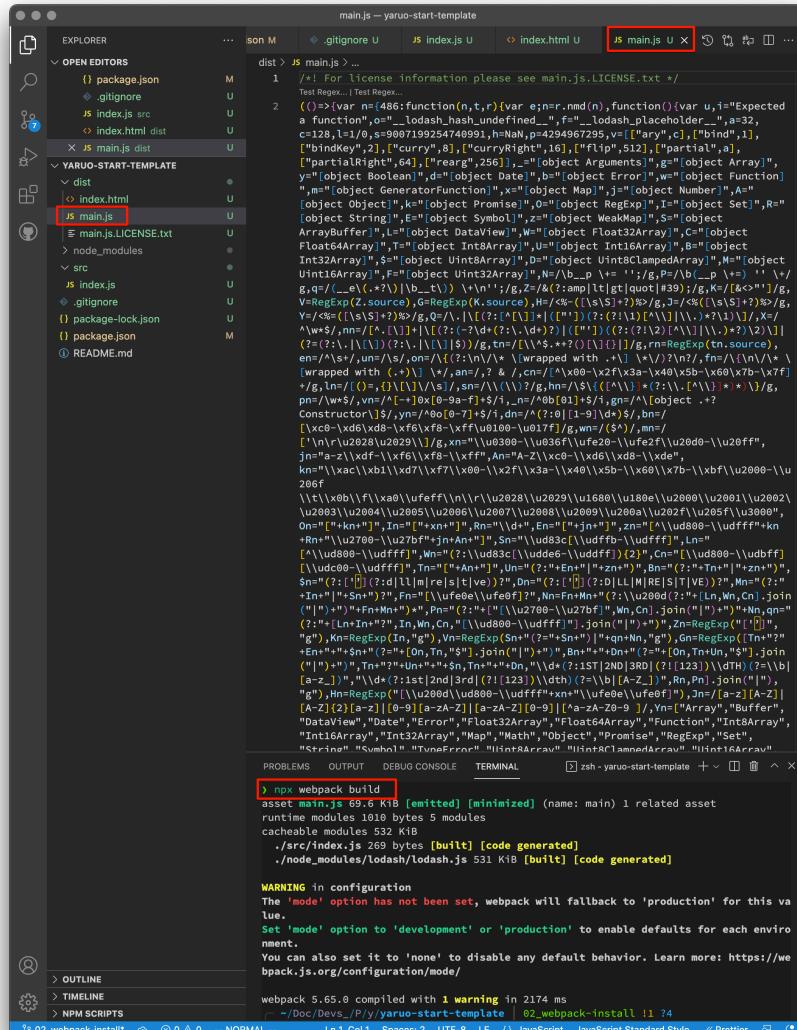
devTools で「main.js」を確認すると、node_modules フォルダ以下にインストールされた JavaScript が 1 つのファイルにまとめられているのが確認できます。

▲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³出力されます。



▲図 2.5: webpack でビルドしてみた

ここまで的内容は、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
```



gitについての知識はあるのか？

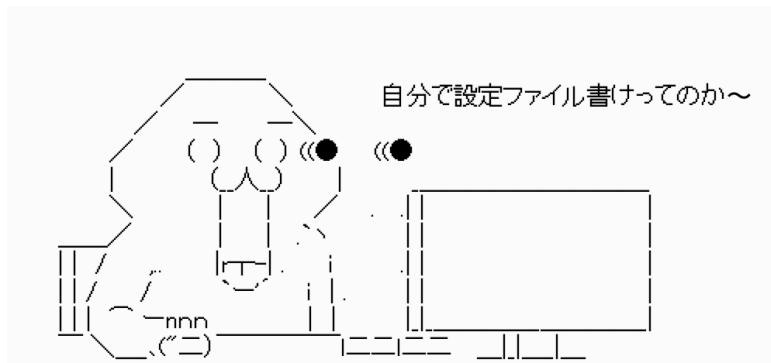
バカにしないでください。それは美味しいの？



♥| 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 がすでにありますか？上書きしても良いですか？ --> No
- README.md がすでにありますか？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/generator'
> 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/plugins/
>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",
    },
    {

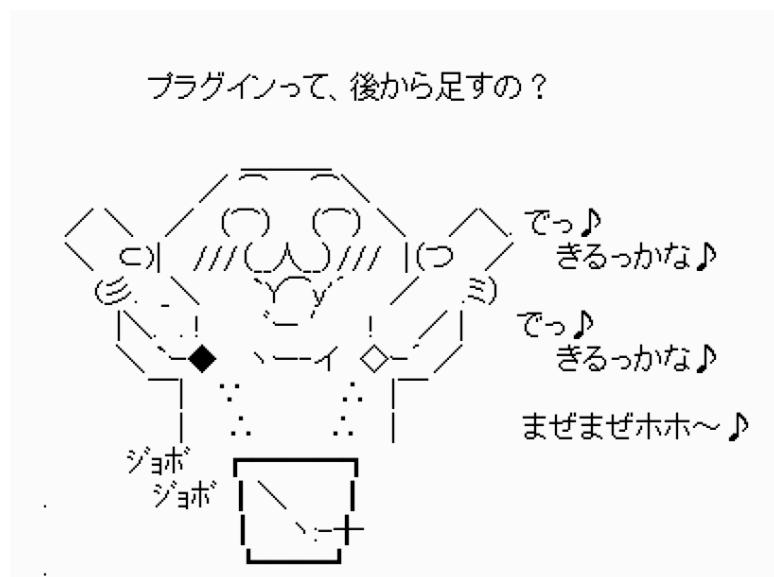
      // Add your rules for custom modules here
      // Learn more about loaders from https://webpack.js.org/loaders/
    },
  ],
  resolve: {
    extensions: [".tsx", ".ts", ".js"],
  },
};

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 webpack-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 每に 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

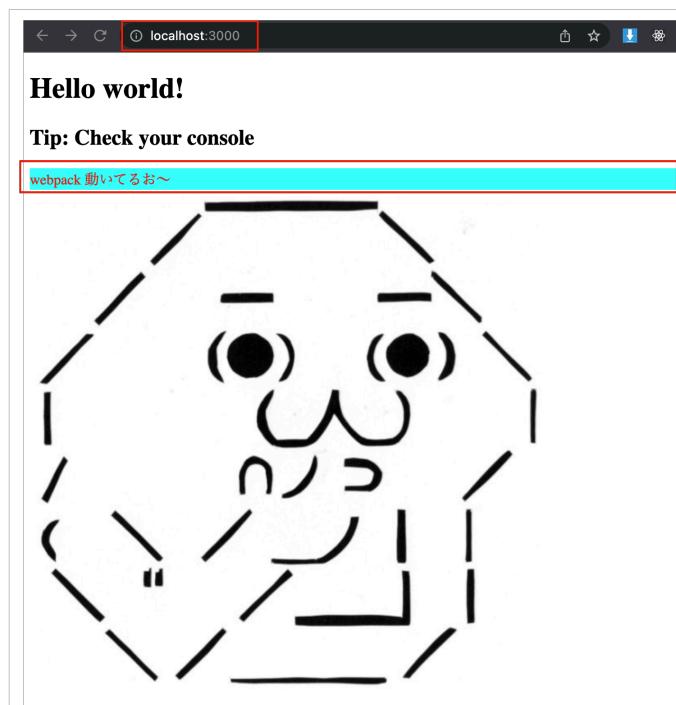
```
"scripts": {  
  "test": "echo \\\"Error: no test specified\\\" && exit 1",  
  "build": "webpack --mode=production",  
  "build:dev": "webpack --mode=development",  
  "build:prod": "webpack --mode=production",  
  "start": "webpack serve"  
},
```

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」で自動で読み込み部分が追加されています。



▲ 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。

The screenshot shows the VS Code interface with the following details:

- EXPLORER:** Shows the project structure:
 - YARUO-START-TEMPLATE
 - public
 - images
 - yaruo.png
 - index.html
 - main.css
 - main.js
- OPEN EDITORS:** package.json (highlighted with a red box) and webpack.config.js.
- package.json — yaruo-start-template** (Editor tab):

```
1 {  
2   "name": "yaruo-start-template",  
3   "version": "1.0.0",  
4   "description": "",  
5   "main": "index.js",  
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": "^6.5.3",  
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": "4.5.5",  
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  },  
43}
```
- COMMANDS:** npm run build (highlighted with a red box).
- OUTPUT:** Shows the build logs:

```
> 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/minify-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)
```
- STATUS:** Ln 10, Col 47, Spaces: 2, UTF-8, LF, JSON, Prettier.

ここまで的内容は、GitHub 上で、以下のコマンドでクローンできます。

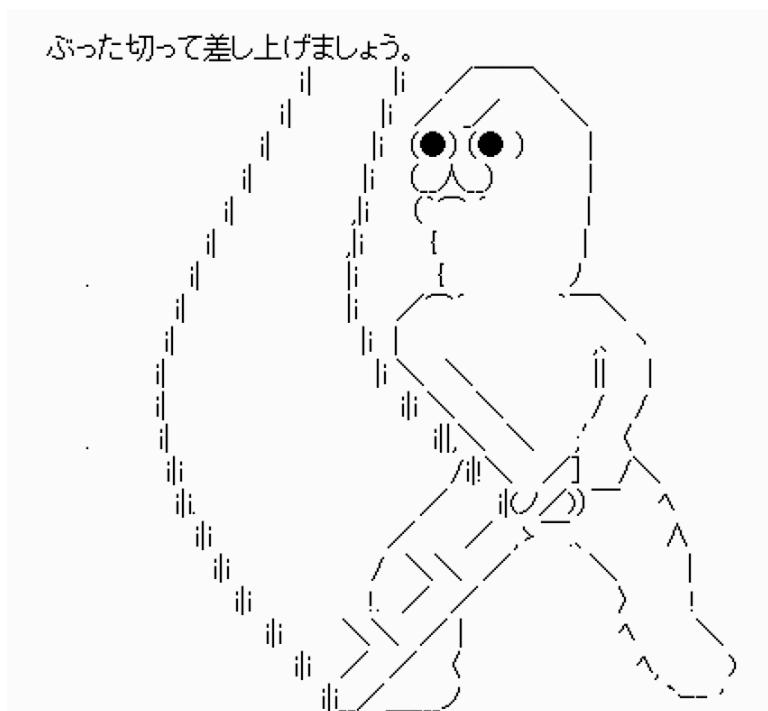
▼ GitHub

```
> 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」を以下のように分割し、共用部分は「webpack-merge」を使用して統合します。

^{*2} <https://webpack.js.org/guides/production/>

- 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';
    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: /\.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。

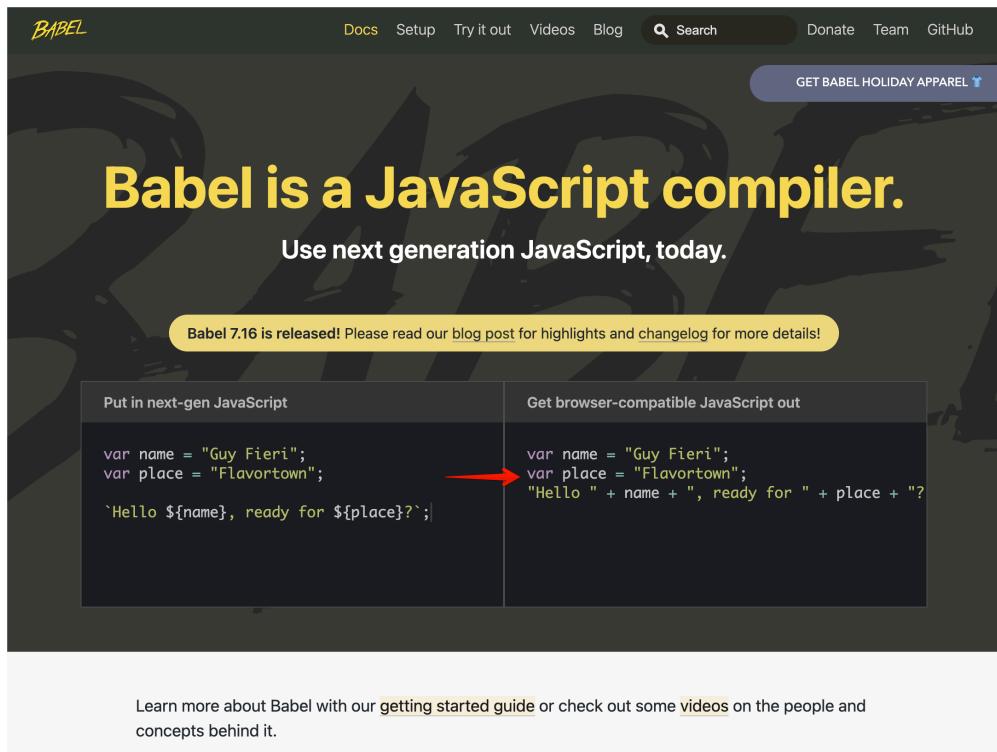
ここまで的内容は、GitHub 上で、以下のコマンドでクローンできます。

▼ GitHub

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

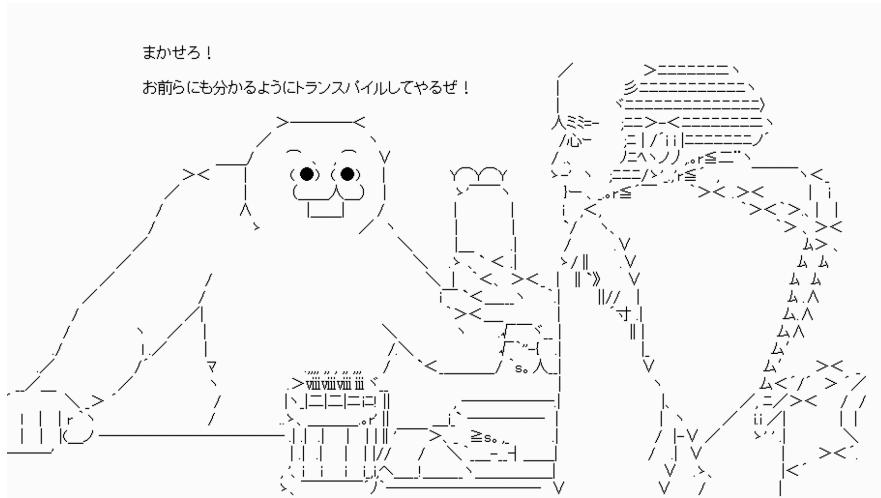
♡| 2.2.6 Babel.js のインストールと設定

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クンパイラ(変換器)です。

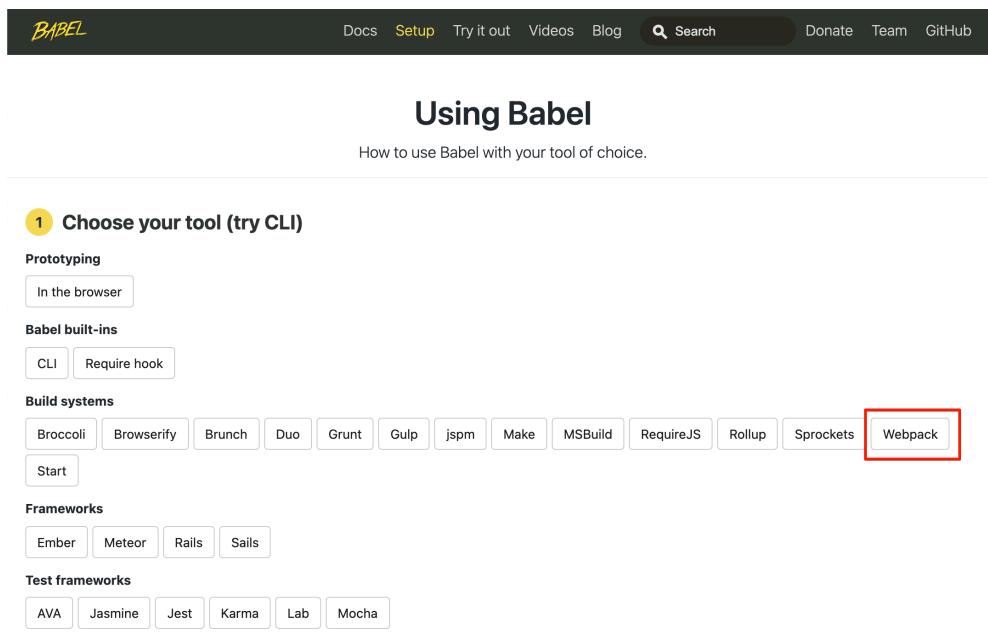


▲図 2.7: Babel.js 本家ページ

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。



▲図 2.8: 使用するツールを選択

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 が表示されましたので、順に実行していきます。

2 Installation

Shell

Copy

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

3 Usage

Via config

JavaScript

Copy

```
{
  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

```
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

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

▲図 2.9: babel のインストール手順

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。今まで npm でインストール時に「-D」としていたのは、「--save-dev」の略です。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、手順 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
```

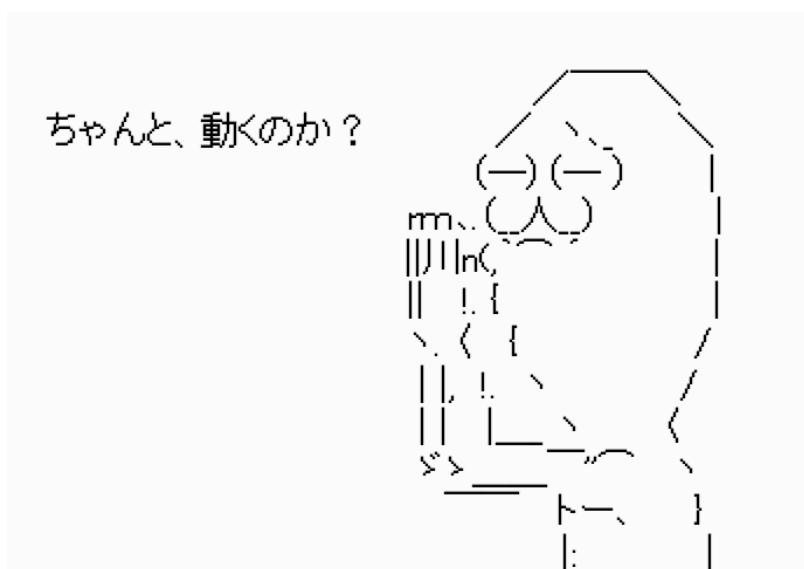
インストールが完了したら、プロジェクトフォルダ直下に「`babel.config.json`」ファイルを作成し、手順 4 の内容を貼り付けます。

▼ 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」に少し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

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 = 'こまけえこたあいいんだよ～';
  const message = `<br />${myName}のログセなのか？<br />${words}`;
  element.innerHTML = _.join(['webpack', '動いてるお～'], ' ') + message;

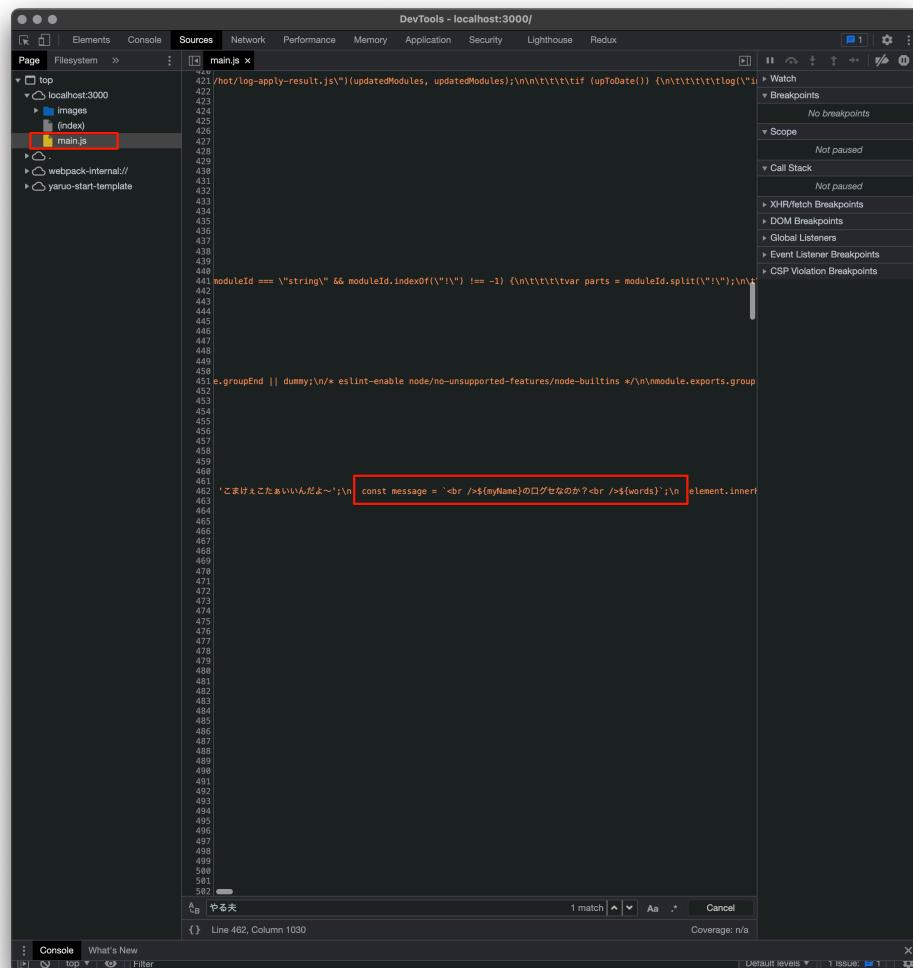
  return element;
}

document.body.appendChild(component());

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

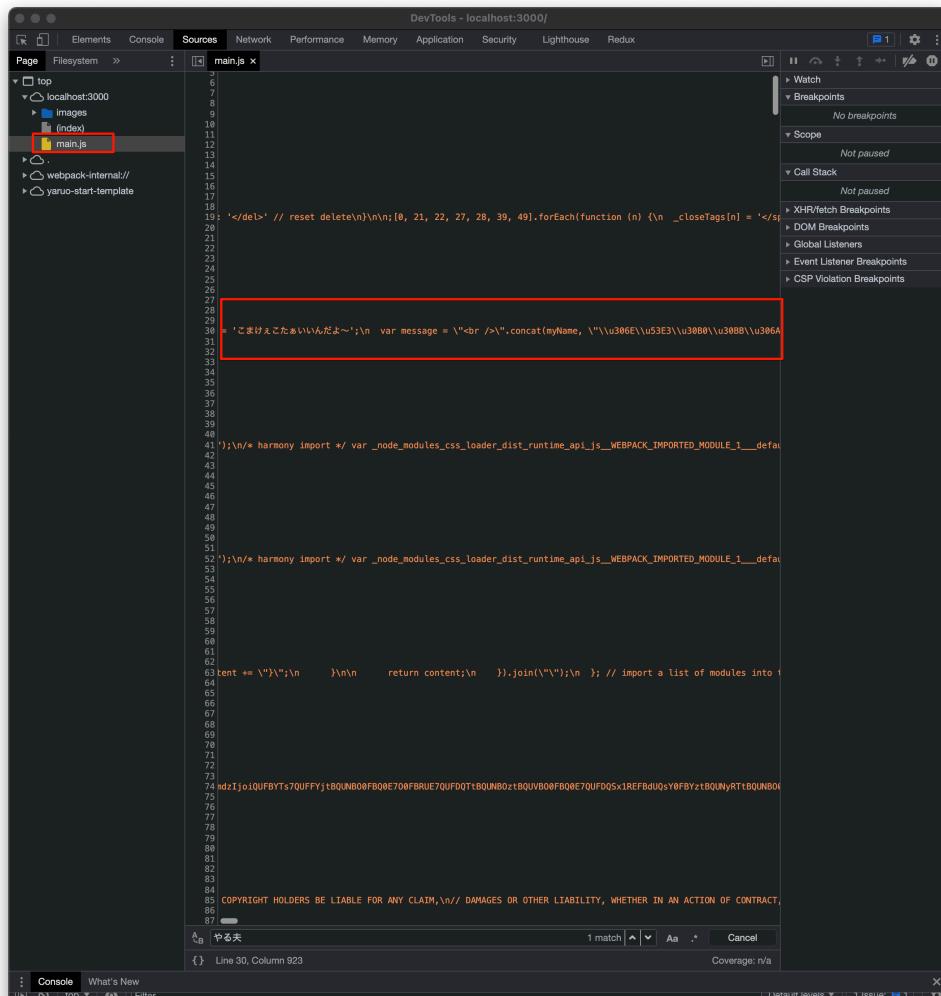
document.body.appendChild(image);
```

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。



The screenshot shows the Google Chrome DevTools interface with the Sources tab selected. The file main.js is open. A red box highlights the line of code: 'var message = "
'.concat(myName, "\u306E\u53E3\u30BB\u306A". This indicates that the Babel.js compiler has converted the template literal into a standard string concatenation with escape sequences for the Japanese characters.

▲図 2.11: concat に変換されています。

ここまで的内容は、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
```

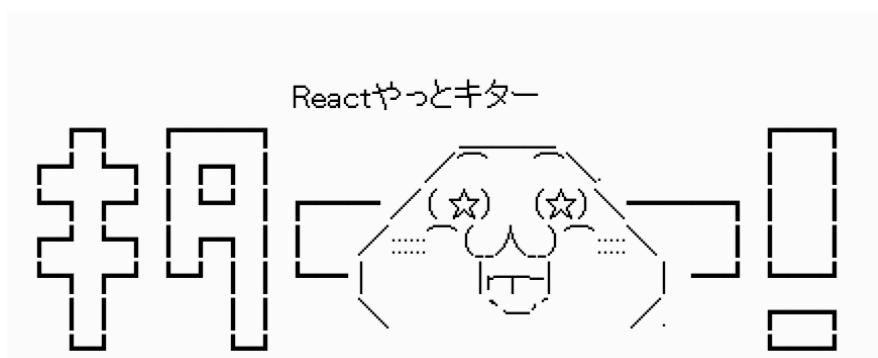
♥| 2.2.7 React のインストール

《注意》index.ts を削除してね。

ここで気が付いたのですが、「src/index.ts」は、現時点では不要なので削除してください。

そろそろ完成に近付いてきました。React をインストールします。

React は、拡張子「.jsx」を使った HTML に JavaScript を埋め込むコンポーネントがメインです。通常の JavaScript とは記法が違うため「Babel.js」に理解してもらえるように「@babel/preset-react」をインストールします。



The screenshot shows the official documentation for the `@babel/preset-react` plugin. At the top, there's a navigation bar with links to `Docs`, `Setup`, `Try it out`, `Videos`, and `Blog`. A search bar is also present. The main content area has a title `@babel/preset-react` with an `EDIT` button. To the left, there's a sidebar with sections for `Guides`, `Config Reference`, `Presets`, `Misc`, and `Integration Packages`, each listing various sub-links. The main content area starts with a note about included plugins: `@babel/plugin-syntax-jsx`, `@babel/plugin-transform-react-jsx`, and `@babel/plugin-transform-react-display-name`. It then discusses the `development` option, mentioning `@babel/plugin-transform-react-jsx-self` and `@babel/plugin-transform-react-jsx-source`. A note states that Flow syntax support is no longer enabled in v7, and users need to add the `Flow` preset. The `Installation` section includes a command: `npm install --save-dev @babel/preset-react`. The `Usage` section is divided into "With a configuration file (Recommended)" and "Without options". The configuration example is shown in JSON:

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

▲図 2.12: `@babel/preset-react`

ターミナルに以下のコマンドでインストールします。

▼ @babel/preset-react のインストール

```
> npm install -D @babel/preset-react
```

インストールが完了しましたら、「babel.config.json」へプリセットを追加します。

▼ babel.config.json

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

React を使うためにインストールするパッケージは、

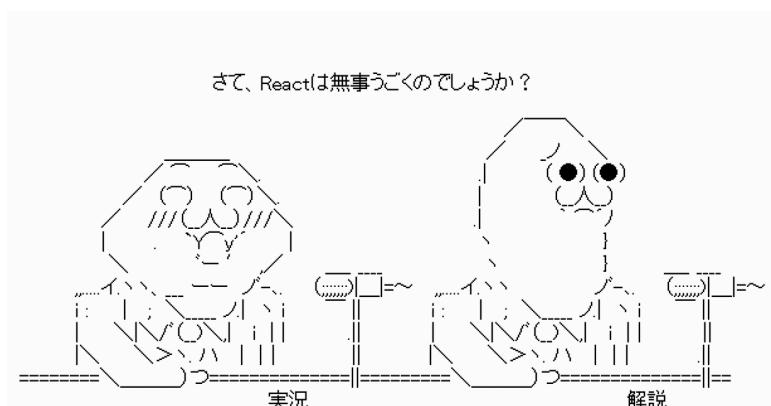
- react (react 本体)
- react-dom (DOM へのエントリポイント、React の描画を提供)

の 2 つです。

▼ React のインストール

```
> npm install react react-dom
```

以上で、React のインストールは完了しましたので動作確認を行います。



手順は、

1. src/index.html に描画する場所を指定する。
2. src/index.js で上記「index.html」へ ReactDOM で描画する。
3. App コンポーネントを src/components/App.jsx ファイルへ作成する。
4. 「webpack.common.js」へ React コンポーネント「jsx」を扱うよう変更する。

となります。

「src/index.html」は、React の描画する部分を指定します。

▼ src/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Appだお～</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

「src/index.js」にて ReactDOM を使用して「public/index.html」へ描画する。

▼ src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

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

「src/components/App.jsx」に App コンポーネントを作成します。React コンポーネントは、HTML 内に JavaScript を埋め込む記法となります。

▼ src/components/App.jsx

```
import React from 'react';
import _ from 'lodash';
import Yaruo from '../assets/images/yaruo.png';

const App = () => {
  const myName = 'やる夫';
  const words = 'こまけえこたあいいんだよ～';
  return (
    <div>
      {_.join(['webpack', '動いてるお～'], ' ')}
      <br />
      {myName}ログセなのか？
```

```
<br />
{words}
<div>
  <img src={Yaruo} />
</div>
</div>
);
};

export default App;
```

「webpack.common.js」を以下のように変更します。

- React コンポーネントの拡張子「.jsx」を扱うように変更する。
- テンプレートを「src/index.html」に変更する。

▼ webpack.common.js

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

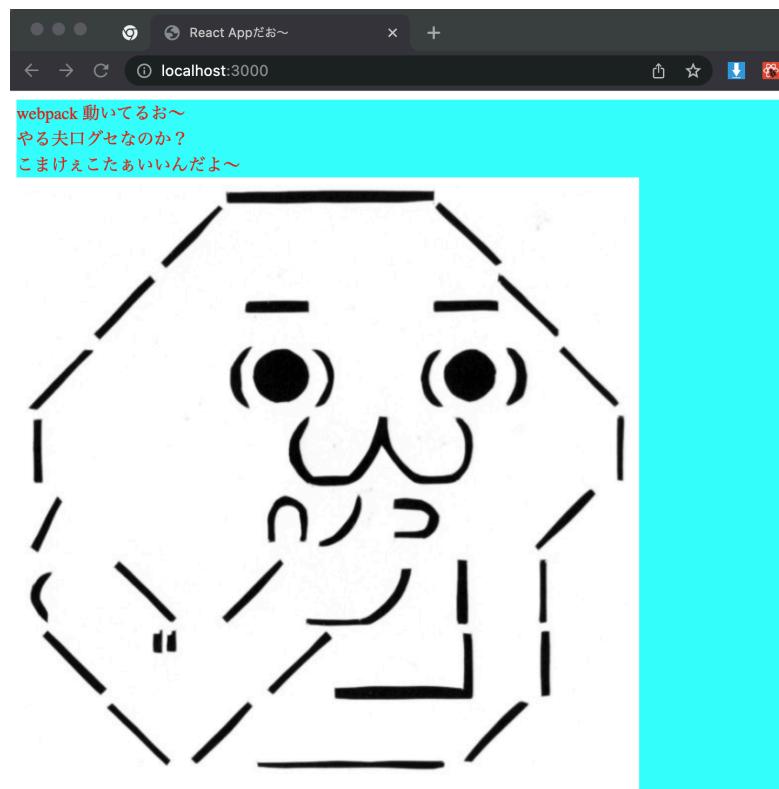
module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'public'),
    assetModuleFilename: 'images/[name][ext][query]',
    clean: true,
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: 'src/index.html',
    }),
  ],
  module: {
    rules: [
      {
        test: /\.ts|tsx$/i,
        loader: 'ts-loader',
        exclude: ['/node_modules/'],
      },
      {
        test: /\.js|jsx$/i,
        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', '.jsx'],
},
};
```

動作確認を行います。

▼ react の動作確認

```
> npm run start
```



▲ 図 2.13: react の動作確認

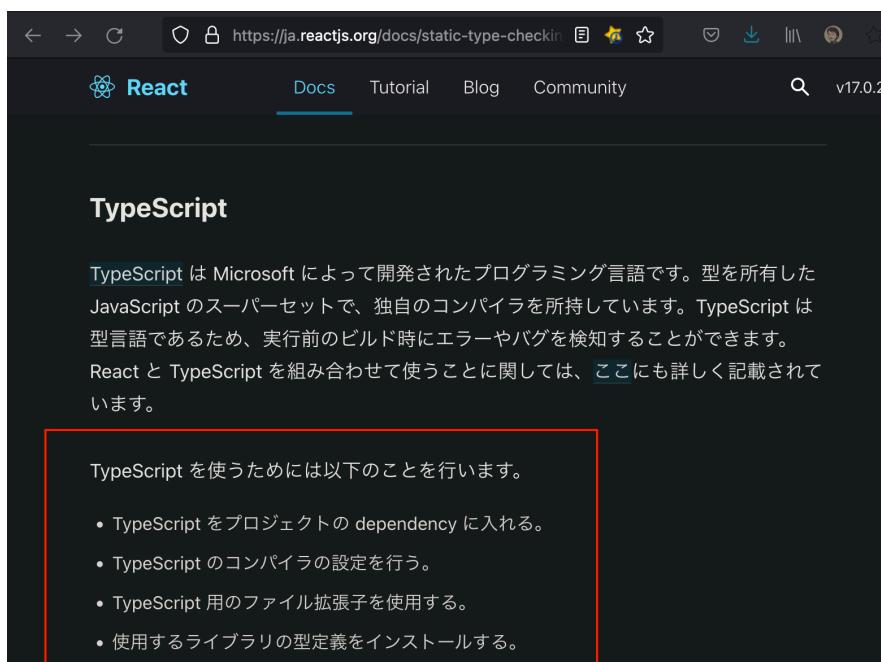
ここまで的内容は、GitHub 上で、以下のコマンドでクローンできます。

▼ GitHub

```
> git clone -b 06_react-install https://github.com/yaruo-react-redux/\
> yaruo-start-template.git
```

♥| 2.2.8 TypeScript のインストール

ここまで作成した React プロジェクトに TypeScript を導入します。本家 React では、TypeScript の導入^{*3}に関して以下のような手順を示しています。



▲ 図 2.14: TypeScript の導入

では手順に従って、TypeScript を導入していきます。

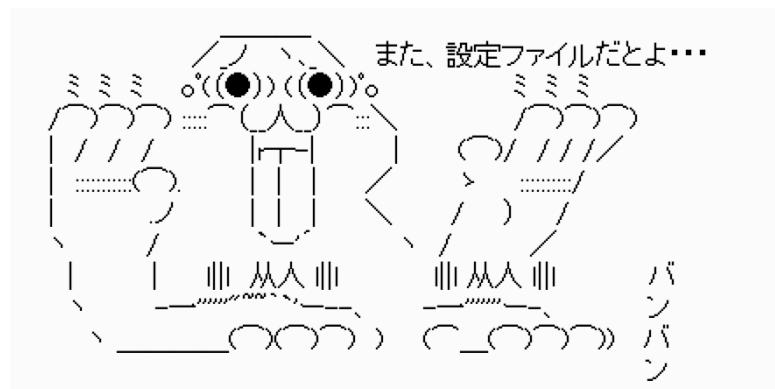
最初は、TypeScript パッケージのインストールです。

▼ TypeScript パッケージのインストール

^{*3} <https://ja.reactjs.org/docs/static-type-checking.html#typescript>

```
> npm install -D typescript
```

次に、TypeScript のコンパイラ設定ファイル「tsconfig.json」を作成します。



▼ tsconfig.json の作成

```
> npx tsc --init

Created a new tsconfig.json with:
>
          TS
>
  target: es2016
  module: commonjs
  strict: true
 esModuleInterop: true
skipLibCheck: true
forceConsistentCasingInFileNames: true

You can learn more at https://aka.ms/tsconfig.json
```

コマンドを実行すると、「tsconfig.json」ファイルが作成されます。コメントアウトされているものは、デフォルト値です。

```
tsconfig.json > ...
1  {
2    "compilerOptions": {
3      /* Visit https://aka.ms/tsconfig.json to read more about this file */
4
5      /* Projects */
6      // "incremental": true,                                /* Enable incremental compilation */
7      // "composite": true,                                 /* Enable constraints that allow a TypeScript project to be used with */
8      // "tsBuildInfoFile": "./",                           /* Specify the folder for .tsbuildinfo incremental compilation files. */
9      // "disableSourceOfProjectReferenceRedirect": true,   /* Disable preferring source files instead of declaration files when */
10     // "disableSolutionSearching": true,                 /* Opt a project out of multi-project reference checking when editing */
11     // "disableReferencedProjectLoad": true,              /* Reduce the number of projects loaded automatically by TypeScript. */
12
13     /* Language and Environment */
14     "target": "es2016",                                  /* Set the JavaScript language version for emitted JavaScript and included modules. */
15     "lib": [],                                         /* Specify a set of bundled library declaration files that describe the global environment. */
16     // "jsx": "preserve",                               /* Specify what JSX code is generated. */
17     // "experimentalDecorators": true,                  /* Enable experimental support for TC39 stage 2 draft decorators. */
18     // "emitDecoratorMetadata": true,                   /* Emit design-type metadata for decorated declarations in source files. */
19     // "jsxFactory": "",                             /* Specify the JSX factory function used when targeting React JSX emit. */
20     // "jsxFragmentFactory": "",                      /* Specify the JSX Fragment reference used for fragments when targeting React. */
21     // "jsxImportSource": "",                         /* Specify module specifier used to import the JSX factory functions when targeting React. */
22     // "reactNamespace": "",                         /* Specify the object invoked for `createElement`. This only applies when targeting React. */
23     // "noLib": true,                                  /* Disable including any library files, including the default lib.d.ts. */
24     // "useDefineForClassFields": true,                /* Emit ECMAScript-standard-compliant class fields. */
25
26     /* Modules */
27     "module": "commonjs",                            /* Specify what module code is generated. */
28     // "rootDir": "./",                             /* Specify the root folder within your source files. */
29     // "moduleResolution": "node",                  /* Specify how TypeScript looks up a file from a given module specifier. */
30     // "baseUrl": "./",                            /* Specify the base directory to resolve non-relative module names. */
31     // "paths": {},                                /* Specify a set of entries that re-map imports to additional lookup locations. */
32     // "rootDirs": [],                            /* Allow multiple folders to be treated as one when resolving modules. */
33     // "typeRoots": [],                           /* Specify multiple folders that act like './node_modules/@types'. */
34     // "types": [],                                /* Specify type package names to be included without being referenced. */
35     // "allowUmdGlobalAccess": true,                /* Allow accessing UMD globals from modules. */
36     // "resolveJsonModule": true,                  /* Enable importing .json files */
37     // "noResolve": true,                           /* Disallow 'import's, 'require's or '<reference>'s from expanding the module graph. */
38
39     /* JavaScript Support */
40     // "allowW3C": true,                           /* Allow JavaScript files to be a part of your program. Use the 'checkJs' option to enable type checking. */
41     // "checkJs": true,                            /* Enable error reporting in type-checked JavaScript files. */
42     // "maxNodeModuleJsDepth": 1,                  /* Specify the maximum folder depth used for checking JavaScript files. */
43
44     /* Emission */
45     // "declaration": true,                        /* Generate .d.ts files from TypeScript and JavaScript files in your project. */
46     // "declarationMap": true,                      /* Create sourcemaps for d.ts files. */
47     // "emitDeclarationOnly": true,                /* Only output d.ts files and not JavaScript files. */
48     // "sourceMap": true,                          /* Create source map files for emitted JavaScript files. */
49     // "outFile": "./",                           /* Specify a file that bundles all outputs into one JavaScript file. */
50     // "outDir": "./",                           /* Specify an output folder for all emitted files. */
51     // "removeComments": true,                     /* Disable emitting comments. */
52     // "noEmit": true,                            /* Disable emitting files from a compilation. */
53     // "importHelpers": true,                      /* Allow importing helper functions from tslib once per project, instead of each file. */
54     // "importSnapshotUsedValues": "remove",       /* Specify emit/checking behavior for imports that are only used for type checking. */
55     // "downlevelIteration": true,                 /* Emit more compliant, but verbose and less performant JavaScript for iterables. */
56     // "sourceRoot": "",                           /* Specify the root path for debuggers to find the reference source code. */
57     // "mapRoot": "",                            /* Specify the location where debugger should locate map files instead of inline. */
58     // "inlineSourceMap": true,                    /* Include sourcemap files inside the emitted JavaScript. */
59     // "inlineSources": true,                      /* Include source code in the sourcemaps inside the emitted JavaScript. */
60     // "emitBOM": true,                           /* Emit a UTF-8 Byte Order Mark (BOM) in the beginning of output files. */
61     // "newline": "crlf",                          /* Set the newline character for emitting files. */
62     // "stripInternal": true,                     /* Disable emitting declarations that have '@internal' in their JSDoc comments. */
63     // "noEmitHelpers": true,                      /* Disable generating custom helper functions like '__extends' in compiled code. */
64     // "noEmitOnError": true,                     /* Disable emitting files if any type checking errors are reported. */
65     // "preserveConstEnums": true,                /* Disable erasing 'const enum' declarations in generated code. */
66     // "declarationDir": "./",                  /* Specify the output directory for generated declaration files. */
67     // "preserveValueImports": true,              /* Preserve unused imported values in the JavaScript output that would otherwise be removed. */
68 }
```

▲図 2.15: 作成された tsconfig.json

TypeScript コンパイラのオプションは、こちら^{*4} で確認できます。

TypeScript 開発元の Microsoft は、React へ導入した「tsconfig.json」のお勧め^{*5} を以下のようにしています。

▼ Microsoft お勧め React 下の tsconfig.json

```
{  
  "compilerOptions": {  
    "outDir": "build/dist",  
    "module": "commonjs",  
    "target": "es5",  
    "lib": ["es6", "dom"],  
    "sourceMap": true,  
    "allowJs": true,  
    "jsx": "react",  
    "moduleResolution": "node",  
    "rootDir": "src",  
    "noImplicitReturns": true,  
    "noImplicitThis": true,  
    "noImplicitAny": true,  
    "strictNullChecks": true  
},  
  "exclude": [  
    "node_modules",  
    "build",  
    "scripts",  

```

Microsoft お勧めの設定に修正したものが、こちらです。コメントアウトされているものは削除しています。

▼ 修正後の tsconfig.json

```
{  
  "compilerOptions": {  
    "target": "es5",  
  }
```

^{*4} <https://www.typescriptlang.org/docs/handbook/tsconfig-json.html>

^{*5} <https://github.com/Microsoft/TypeScript-React-Starter/blob/master/tsconfig.json>

```
"lib": [
  "es6",
  "dom"
],
"jsx": "react",
"module": "commonjs",
"rootDir": "src",
"outDir": "public",
"esModuleInterop": true,
"forceConsistentCasingInFileNames": true,
"strict": true,
"skipLibCheck": true
},
"exclude": ["node_modules", "public", "webpack"]
}
```

TypeScript は、ファイル拡張子が、

- .js ---> .ts
- .jsx --> .tsx

となるため、「webpack.common.js」の「entry」のファイル名の拡張子を「.tsx」に変えます。

▼ webpack.common.js の一部

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

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

プロジェクト内のファイル名も変更します。

- 「src/index.js」 ---> 「src/index.tsx」
- 「src/components/App.jsx」 --> 「src/components/App.tsx」

次に、使用するライブラリの型定義をインストールします。ライブラリによっては自身が型定義を持っている場合もありますし、有志で作成された型定義ファイルは、npm リポジ

トリの「@types/」にある場合もあります。

使用する React、Node.js の型定義ファイルは、「@types/」以下にありますのでインストールします。

▼ React、Node.js の型定義のインストール

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

型定義のインストールが完了しても、「App.tsx」で、

- lodash の型定義がない
- yaruo.png の型定義がない

と、エラーが表示されています。

The screenshot shows the VS Code interface with the following details:

- Explorer View:** Shows the project structure with files like `webpack.common.js`, `tsconfig.json`, `index.html`, `babel.config.json`, `App.tsx` (which is currently open), and `yaruo.png`.
- Code Editor:** The `App.tsx` file contains the following code:

```

    1 import React from 'react';
    2 import _ from 'lodash';
    3 import Yaruo from './assets/images/yaruo.png';
    4
    5 const App = () => {
    6   const myName = 'やるう';
    7   const words = 'こまけんたあいなんだよ～';
    8   return (
    9     <div>
    10       <_._join(['webpack', '動いてるお～'], ' ')>
    11       {myName}!びやか?
    12       <br />
    13       {words}
    14       <div>
    15         <img src={Yaruo} />
    16       </div>
    17     </div>
    18   );
    19 }
    20
    21
    22 export default App;
  
```
- Terminal:** Shows the command `npm i -D @types/react` being run.
- Problems View:** Displays an error message:

```

    TS App.tsx(3,10) : Cannot find a declaration file for module 'lodash'. Try 'npm i --save-dev @types/lodash' if it exists or add a new declaration file containing 'declare module 'lodash';' in /Users/kazuya/Downloads/Dev_Jc2/Project_B/書籍2020/2/yaruo/react_simple/yar...
    TS App.tsx(3,10) : Cannot find module './assets/images/yaruo' or its corresponding type declarations. ts(2307) [3,10]
  
```
- Status Bar:** Shows the file path `07_typescript-install.tsx` and other status information.

▲図 2.16: App.tsx でエラー表示

「lodash」の型定義はインストールすればエラーが消えますが、今後「lodash」は使わないのでアンインストールし、該当コードも削除します。

▼ lodash のアンインストール

```
> npm uninstall lodash
```

「png」については、プロジェクト用の型定義ファイルを作成します。

「src/types/index.d.ts」を作成し、以下のように記入します。

▼ src/types/index.d.ts

```
declare module '*.png'
```

作成したファイルを「tsconfig.json」に型定義ファイルの位置を追加します。

▼ tsconfig.json

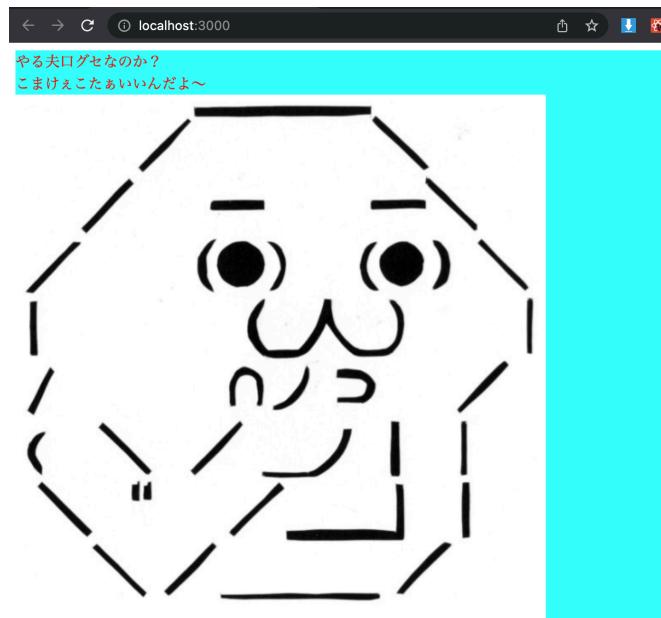
```
"compilerOptions": {  
  "typeRoots": [  
    "types"  
  ]  
}
```

動作確認を行います。

▼ 動作確認

```
> npm run start
```

lodash 部分のコードが削除されたトップページを表示します。



▲図 2.17: TypeScript 導入後動作確認

ここまで的内容は、GitHub 上で、以下のコマンドでクローンできます。

▼ GitHub

```
> git clone -b 07_typescript-install https://github.com/yaruo-react-r>
> edux/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 の方が優れています。

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 で作成したプロジェクト

「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。

追加のパッケージ、設定については、本章の後半で解説しています。

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」が未インストール状態でしたら、「eslint」をインストールするのか問われます。

▼ eslint の初期化

```
$ npx eslint --init
Need to install the following packages:
  eslint
Ok to proceed? (y) y
```

「y」を入力しエンターキーを押すと「eslint」がインストールされ、設定ファイルを作成するための質問が始まります。

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

▼ 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? · style
☒ 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
☒ How would you like to define a style for your project? · guide
☒ Which style guide do you want to follow? · airbnb
☒ What format do you want your config file to be in? · JavaScript
Checking peerDependencies of eslint-config-airbnb@latest
Local ESLint installation not found.
The config that you've selected requires the following dependencies:

eslint-plugin-react@^7.27.1 @typescript-eslint/eslint-plugin@latest eslint-config-airbnb@latest eslint@^7.32.0 || ^8.2.0 eslint-plugin-import@^2.25.3 eslint-plugin-jsx-a11y@^6.5.1 eslint-plugin-react-hooks@^4.3.0 @typescript-eslint/parser@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": "^5.8.0",
  "@typescript-eslint/parser": "^5.8.0",
  "@webpack-cli/generators": "^2.4.1",
  "eslint": "^8.5.0",
  "eslint-config-airbnb": "^19.0.2",
  "eslint-plugin-import": "^2.25.3",
  "eslint-plugin-jsx-a11y": "^6.5.1",
  "eslint-plugin-react": "^7.28.0",
  "eslint-plugin-react-hooks": "^4.3.0",
}
```

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

▼ .eslintrc.js

```
module.exports = {
  "env": {
    "browser": true,
    "es2021": true
  },
  "extends": [
    "plugin:react/recommended",
    "airbnb"
```

```
],
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "ecmaFeatures": {
      "jsx": true
    },
    "ecmaVersion": 13,
    "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ルールに上書きされます。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 --init」時にインストールされたルールが適用されるように「extends」に追加し

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

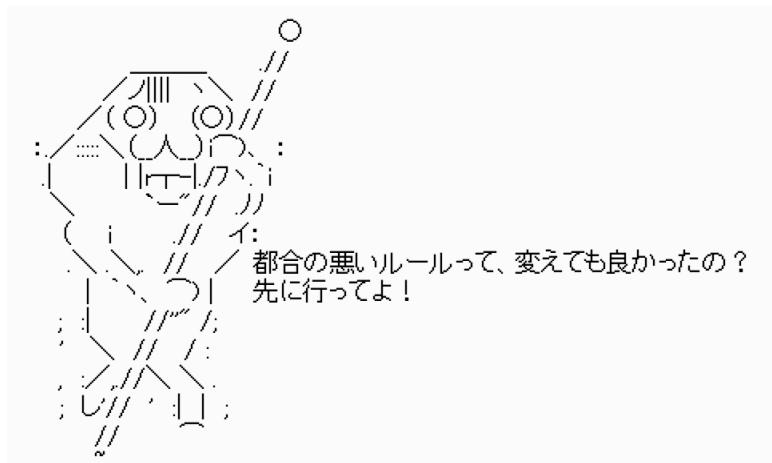
再度、ルールを出力すると適用されるルールがずいぶん増えているのが分かります。

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きをします。



ルール「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-extension 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"  
  }  
,
```

「no-void」は、void 演算子を使用すると undefined を返すため禁止してあります。「create-react-app」で作成される「reportWebVitals.ts」で void 使います。文としての使用だけを可能にします。

▼ no-void の現在

```
"no-void": [  
  "error"  
,
```

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",
        },
    ],
    'no-void': [
        'error',
        {
            allowAsStatement: true,
        },
    ],
},
}
```

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": {
    "eslint-config-prettier": "^8.3.0",
    "prettier": "^2.5.1"
}
```

Prettier 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^{*6}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。

^{*6} <https://prettier.io/docs/en/options.html>

▼ package.json

```
"scripts": {  
  "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",  
    "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  
  },  
},
```

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。

以上で環境構築は完了なのですが、「.eslintrc.js」にてエラーが表示されています。

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.

このエラーは、「parserOptions.project」で「tsconfig.json」を指定していますが、「tsconfig.json」では、「module」で「commonjs」を指定しています。

そのため、「.eslintrc.js」ファイルが、どこからも import されていないので警告が出ています。

解決策として、「eslint」の対象外とするために「.eslintignore」を作成し、「.eslintrc.js」を指定します。

▼ .eslintignore

```
.eslintrc.js
```

これでエラーが解消されます。

ここまでのお問い合わせは、GitHub 上で、以下のコマンドでクローンできます。

▼ GitHub

```
> git clone -b 07_typescript-install https://github.com/yaruo-react-r>  
> edux/yaruo-start-template.git
```

2.3.2 create-react-app 作成のプロジェクトへ「eslint,prettier」を設定

「create-react-app」で作成したプロジェクトには、下図のように eslint が組み込まれています。

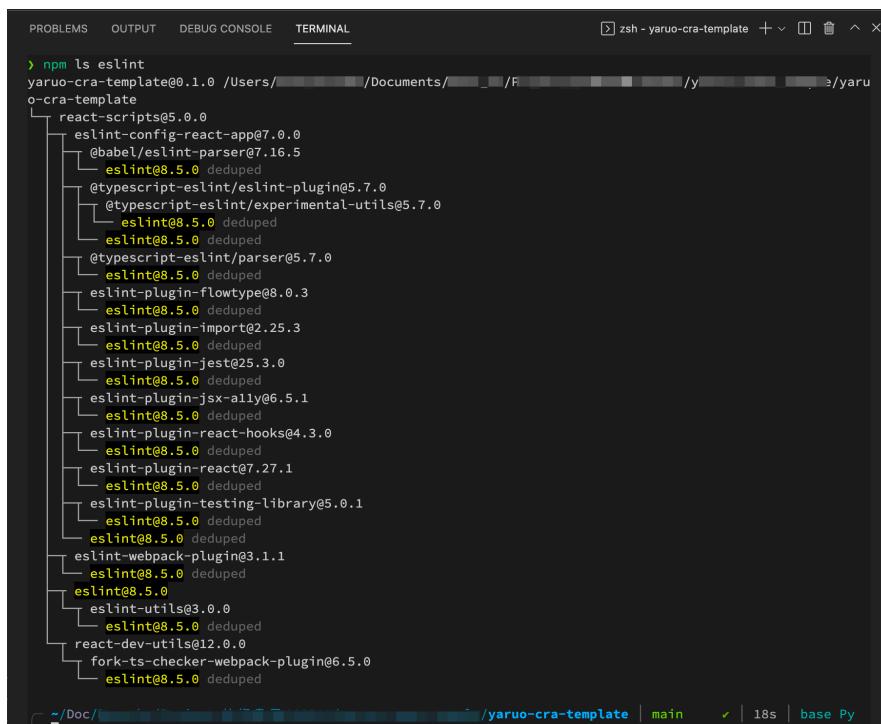


図 2.18: create-react-app の eslint

「`npx eslint --init`」でインストールした、

- eslint-plugin-react@^7.27.1
 - @typescript-eslint/eslint-plugin@latest
 - eslint-config-airbnb@latest
 - eslint@^7.32.0 || ^8.2.0

- eslint-plugin-import@^2.25.3
- eslint-plugin-jsx-a11y@^6.5.1
- eslint-plugin-react-hooks@^4.3.0
- @typescript-eslint/parser@latest

のうち、airbnb 以外はインストールされています。

そのため、「eslint-config-airbnb」だけをインストールします。

▼ create-react-app 作成プロジェクトへ eslint-config-airbnb のインストール

```
☒ npm install -D eslint-config-airbnb
```

設定ファイル「.eslintrc.js」を「ゼロからの構築」と同じものを作成します。

▼ .eslintrc.js

```
module.exports = {  
  env: {  
    browser: true,  
    es2021: true,  
  },  
  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',  
  ],  
  parser: '@typescript-eslint/parser',  
  parserOptions: {  
    ecmaFeatures: {  
      jsx: true,  
    },  
    ecmaVersion: 13,  
    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',
    },
],
},
};
```

Prettier はインストールされていないためインストールします。

▼ create-react-app 作成プロジェクトへ prettier のインストール

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

「.eslintignore」、「.prettierrc」を作成します。

▼ .eslintignore

```
.eslintrc.js
```

▼ .prettierrc

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

「package.json」にある「eslint」の設定を削除します。

▼ package.json の eslint 設定を削除

```
"eslintConfig": {
    "extends": [
        "react-app",
    ],
}
```

```
    "react-app/jest"
  ],
}
```

最後に、「package.json」にスクリプトを追加し、eslint、prettier が動作するようにします。

▼ package.json

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject",
  "lint": "eslint 'src/**/*.{js,jsx,ts,tsx}'",
  "fix": "npm run format && npm run lint:fix",
  "format": "prettier --write 'src/**/*.{js,jsx,ts,tsx}'",
  "lint:fix": "eslint --fix 'src/**/*.{js,jsx,ts,tsx}'"
},
```

以上で、「create-react-app」で作成したプロジェクトに追加で airbnb のチェックを追加できました。

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ができます。

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", "typescriptreact"],
```

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、App.tsx ファイルを修正します。

エラー内容は、「Function コンポーネントは、arrow 関数にしなさい。」とのことで、これは、rules に追加したためです。

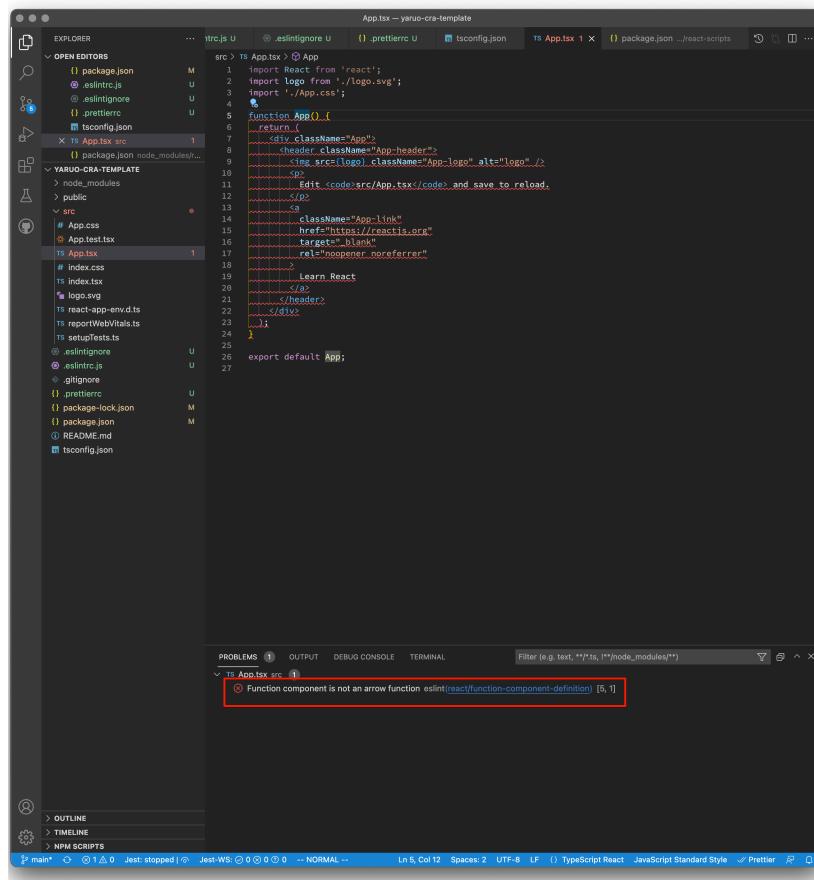
筆者が VSCode を日本語化していないのは、エラーメッセージでググる場合を考えてのことです。英語でのエラーメッセージの方が的確なページを見つけやすいと考えています。

では、指摘されている点を修正していきます。

▼ 修正後の App.tsx

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

const App = () => (
  <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.
    </header>
  </div>
)
```



▲図 2.19: App.tsx の修正

```
</p>
<a
  className='App-link'
  href='https://reactjs.org'
  target='_blank'
  rel='noopener noreferrer'
>
  Learn React
</a>
</header>
</div>
);

export default App;
```

次に、「repoWebVitals.ts」でエラーが発生しています。これは import 文に「void」を付

けることで解決します。

▼ 修正後の reportWebVitals.ts

```
import { ReportHandler } from 'web-vitals';

const reportWebVitals = (onPerfEntry?: ReportHandler) => {
  if (onPerfEntry && onPerfEntry instanceof Function) {
    void import('web-vitals').then(
      ({ getCLS, getFID, getFCP, getLCP, getTTFB }) => {
        getCLS(onPerfEntry);
        getFID(onPerfEntry);
        getFCP(onPerfEntry);
        getLCP(onPerfEntry);
        getTTFB(onPerfEntry);
      }
    );
  }
};

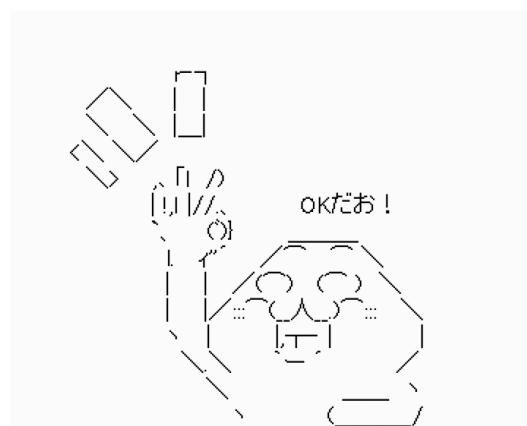
export default reportWebVitals;
```

これで現時点での指摘はすべて修正できました。動作確認を行います。

▼ create-react-app の動作確認

```
> npm run start
```

これで、トップページが表示されます。



ここまで的内容は、GitHub 上で、以下のコマンドでクローンできます。

▼ GitHub

```
$ > git clone https://github.com/yaruo-react-redux/yaruo-cra-template.git
```

2.5 第2章のまとめ

React を使用しスタートアップ用のアプリケーションの作成方法を

- 「create-react-app」コマンドで作成
- ゼロから構築

の2通りで解説しました。

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。

第3章

日記アプリケーションの作成 (React のみ)



▲図 3.1: 完成サンプルアプリケーション

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。

また、表示用のUIには、Googleが提唱するMaterial DesignのReact用UIの最新版MUI5を使用します。

サンプルアプリケーションで使用している書籍データ(表紙画像)は、版元ドットコム^{*1}の規約に基づいています。

^{*1} https://www.hanmoto.com/about_bookdata

3.1 React とは？

まずは、「React とは、いったい何なのでしょうか？」

本家「reactjs.org」のドキュメントは日本語(ja.reactjs.org)^{*2}でも読むことができます。でも、ここで言われている特徴、

- 宣言的な View
- コンポーネントベース
- 一度学習すれば、どこでも使える

って、「なに言っているのか、良く分かりません。」

フロントエンド用フレームワークが当たり前の我々は知らないのですがjQueryなどでガチのJavaScriptプログラミングされていた方は、たいへんな思いをしていたのではないでしょうか？

たとえば、テキストボックスの入力を検証をする場合には、

1. テキストボックスをdom内から取得(getElementById)
2. その要素に検証用関数の呼び出しを行うイベントの追加(addEventListener)
3. 検証しエラーがある場合には、エラー表示用の要素をdom内から取得
4. エラー表示用要素へ表示(innerHTML)

が必要です。

コードにすると、

▼テキストボックス表示部分

```
<div>
  <input id="yaruo" type="text" value="">
  <p id="yaranai"></p>
</div>
```

▼JavaScript 部分

```
<script>
  const input = document.getElementById("yaruo");
  input.addEventListener("input", validationFunc);
```

^{*2} <https://ja.reactjs.org>

```
function validationFunc(event) {
    const inputValue = event.target.value;

    let errMsg = '';
    // check value
    if (inputValue.length < 5) {
        errMsg = '5文字以上の入力が必要です。';
    }

    // display error message
    const errMsgArea = document.getElementById("yaranaio");
    errMsgArea.innerText = errMsg;
}

</script>
```

このように、テキストボックス1つでも、けっこうなコード量になり、おまけに表示部分とコード部分が分かれています。

もし、入力部分が数個あるフォームだと、入力の表示部分と対応コード部分を探すものた
いへんですし、デバッグも悪夢のようだと思いませんか？

また、使用する関数がそれぞれ別のファイルに分かれていた場合など、関数とファイル名
の対応表まで必要になります。

それが「React」を使うと、

▼ React の場合

```
import React, { useState } from 'react';

const YaruoInput = () => {
    const [inputValue, setInputValue] = useState('');
    const [errMsg, setErrMsg] = useState('');

    const handleValidate = (event: React.ChangeEvent<HTMLInputElement>) => {
        setInputValue(event.target.value);

        if (event.target.value.length < 5) {
            setErrMsg('5文字以上の入力が必要です。');
        } else {
            setErrMsg('');
        }
    };

    return (

```

```
<div>
  <input
    id='yaruo'
    type='text'
    value={inputValue}
    onChange={handleValidate}
  />
  <p id='yaranaio'>{errMsg}</p>
</div>
);
};

export default YaruoInput;
```

このように UI コンポーネントとして定義でき、再利用も簡単に行えます。

▼ React のコンポーネントを使う

```
import React from 'react';
import YaruoInput from filePath

const App = () => (
<>
  <YaruoInput />
  <別なコンポーネント />
  <HTMLタグ>
  </HTMLタグ>
);

export default App;
```

同じ機能を持つテキストボックスでも React を使うことで、UI コンポーネントとして定義し再利用・保守が格段に上がりました。

これが、「宣言的な View」、「コンポーネントベース」です。

3.2 表示するデータの型

それでは、サンプルアプリケーションについて説明します。このサンプルアプリケーションは、読書日記としていますが通常の日記でもかまいません。

保持するデータは、

diaryId(文字列)

日記の ID、ユニークな文字列とする。

title(文字列)

日記のタイトル

postDate(文字列)

投稿日を YYYYMMDD 形式で持つ

imageUrl(文字列)

アイキャッチ画像の URL

imageLabel(文字列)

画像の alt 属性

mainContent(文字列)

本文

readmore(文字列の配列)

追記。段落を配列の要素とする。

とします。

最初に表示するデータを初期値として持ちます。

実際の初期値は、こちらです（長い文字列は省略しています。GitHub 上で実物を確認してください）。

▼ 読書日記の初期値

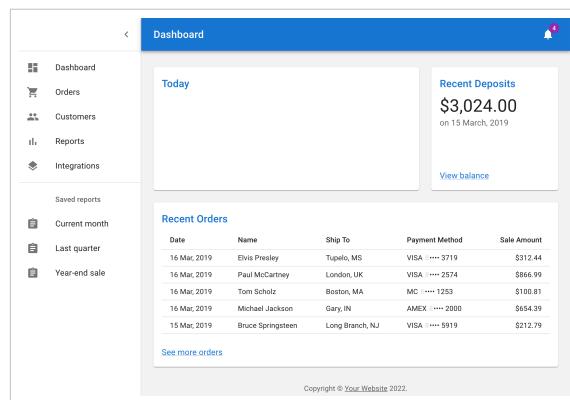
```
export type Diary = {  
  diaryId: string;  
  title: string;  
  postDate: string;  
  imageUrl: string;  
  imageLabel: string;  
  mainContent: string;  
  readmore: string[];  
};
```

```
const diaries: Diary[] = [
  {
    diaryId: '9784781611495',
    title: '「タモリ学 戸部田誠」を読んだお',
    postDate: '20210601',
    imageUrl: 'http://inazuma.xsrv.jp/book_images/9784781611495_100.jpg',
    imageLabel: '',
    mainContent:
      'デビュー時から現在までの、・・・',
    readmore: [
      'タモリにとって「アドリブとは何か？」',
      'タモリをもっと知りたくて。デビュー時から現在までの、・・・',
      '著者について',
      '78年生まれ、いわき市在住のテレビっ子。お笑い、格闘技、・・・',
    ],
  },
  . . . 中略
  {
    diaryId: '9784041047361',
    title: '「経済ヤクザ 一橋文哉」を読んだお',
    postDate: '20211101',
    imageUrl: 'http://inazuma.xsrv.jp/book_images/9784041047361_100.jpg',
    imageLabel: '',
    mainContent:
      '日本の経済はこうして動かされてきた。政界や一般企業に食い込み、・・・',
    readmore: [
      '政界や企業に食い込み、ハイエナの如くマネーを貪った「経済ヤクザ」たち。・・・',
      '彼らが復興利権やITバブルをいかにして我が物としてきたか',
    ],
  },
];
];
```

3.3 Material Design 5 の導入

ここでは、表示に使用する UI として Google 推奨の「Material Design」に従ってデザインされた React 用 UI の「MUI5(Material Design User Interface version5)^{*3}」を導入します。

MUI を使うと、こんな画面もパーツを組み合わせるだけで構築できます。



▲図 3.2: MUI サンプル

もちろんカスタマイズも自由自在で、コンポーネントの各ページにあるサンプル付属のコードをブラウザ上でフロントエンド環境を試せる CodeSandbox^{*4}をクリック一発で開き、カスタマイズを試すことができます。

Basic button

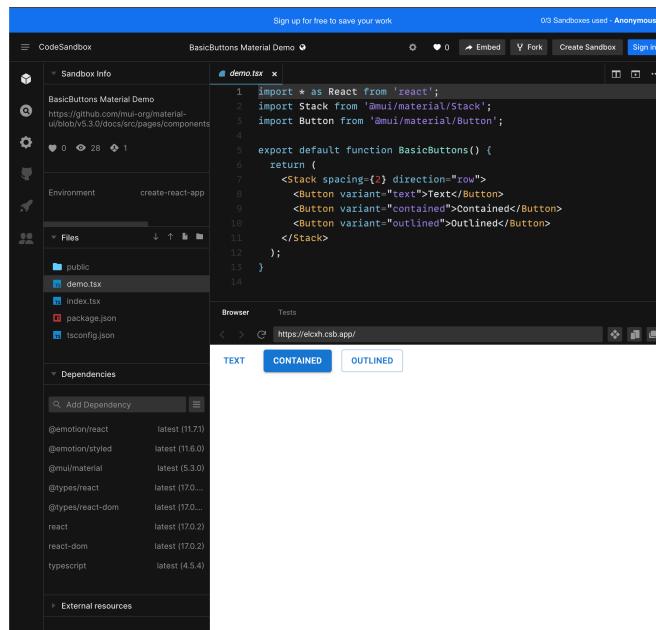
The Button comes with three variants: text (default), contained, and outlined.

```
<Button variant="text">Text</Button>
<Button variant="contained">Contained</Button>
<Button variant="outlined">Outlined</Button>
```

▲図 3.3: コンポーネントのサンプル

*3 <https://mui.com/>

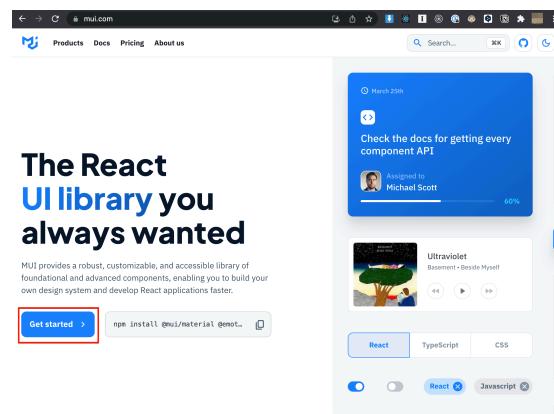
*4 <https://codesandbox.io/>



▲図 3.4: クリックで CodeSandbox を開いたところ

3.3.1 MUI のインストール

MUI のインストールは、MUI トップページから「Get started」ボタンをクリックすると表示されます。



▲図 3.5: MUI トップページ

npm を使ったインストールは、material-icon も含め以下となります。

▼ MUI のインストール

```
> npm install @mui/material @emotion/react @emotion/styled @mui/icons-material
```

.....

ここまで的内容は、GitHub 上で、以下のコマンドでクローンできます。

▼ GitHub

```
> git clone -b 01_install-MUI https://github.com/yaruo-react-redux/yaruo-diary.git
```

.....

♥| 3.3.2 データ表示画面を作る

データの一覧画面を作成します。

製品版の場合は、1画面に表示するデータ数を決め、それを超えた場合にはページネーションを作成しますが、今回のサンプルアプリケーションは、1画面とします。

1画面のデータ表示を4とかに決め、ページネーションにて前後ページに移動するように魔改造してみてください。

では、データ表示コンポーネント (DiaryBoard) を作成します。「/src/components」に「DiaryBoard.tsx」ファイルを作成します。

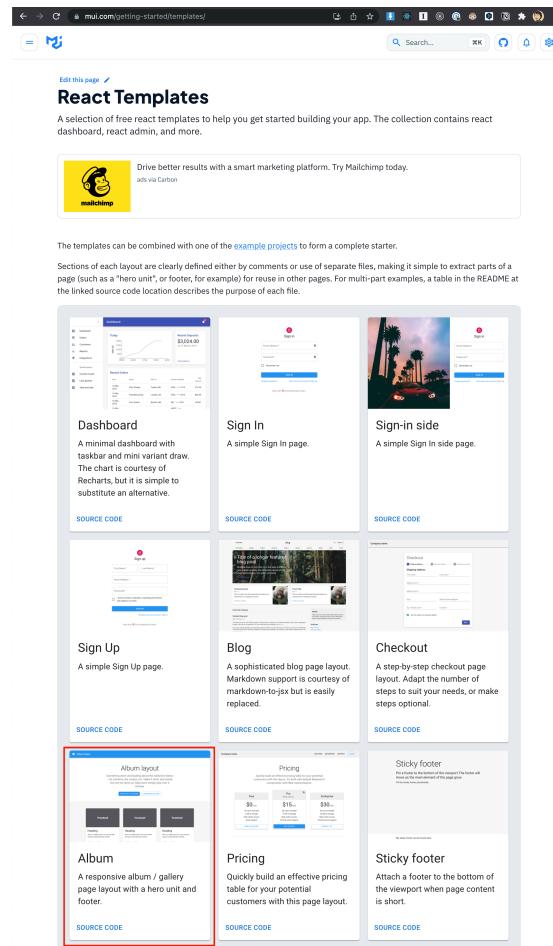
♥| 3.3.3 MUI5 のサイトからテンプレートを拝借

MUI5 のサイトの左上部のメニューには、

- Getting Started
 - Templates
- Components
 - Button などのコンポーネント
- Compotent API
 - Component の Props(引数) の詳細

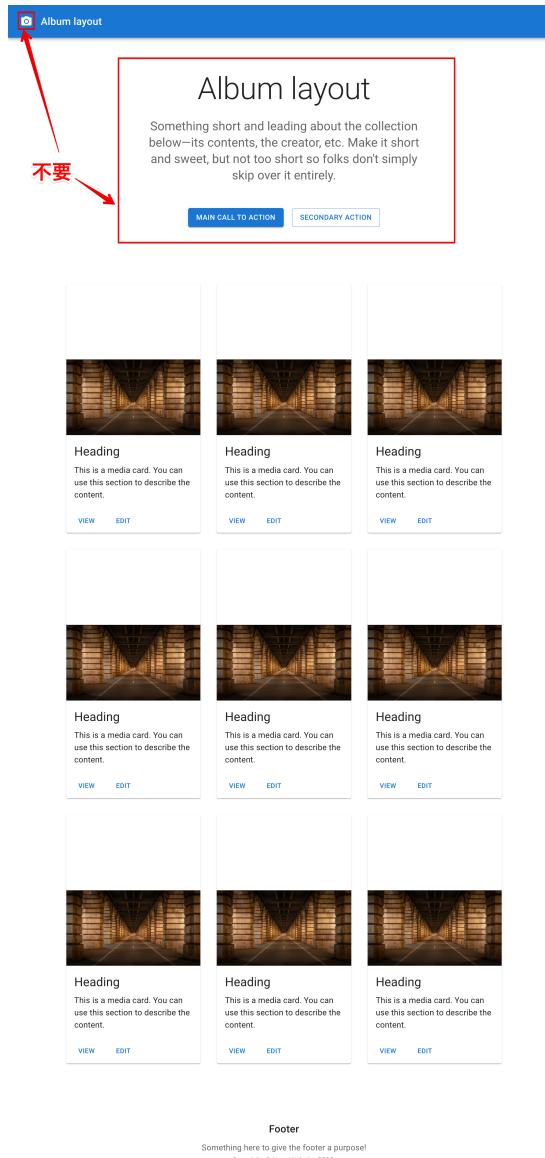
があります。

その Template を選択すると、以下のように MUI で作成されたサンプルページ (コンポーネントを組み合わせたもの) が表示されます。



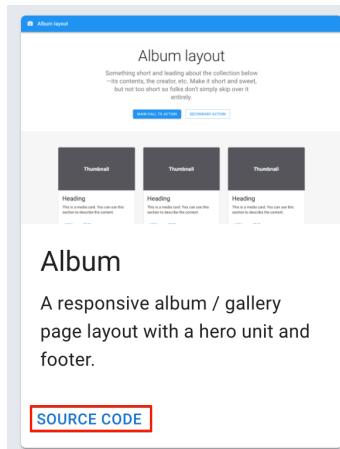
▲ 図 3.6: MUI の Template ページ

今回は、この中から「Album layout」を拝借して改造していきます。



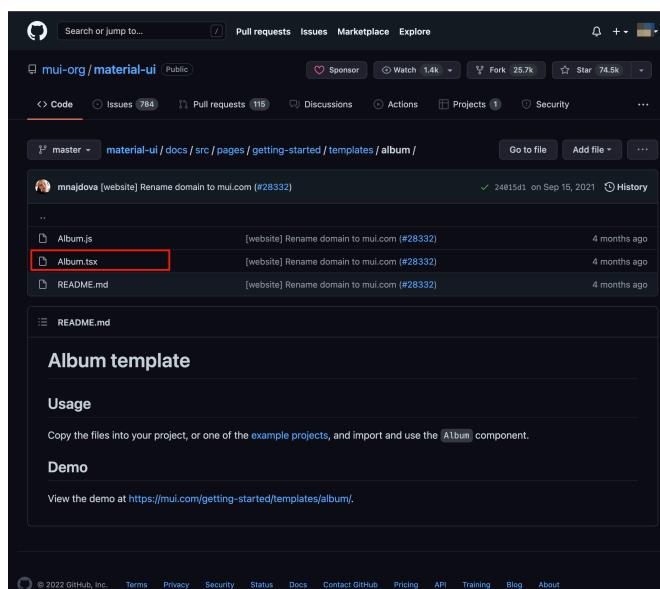
▲図 3.7: アルバムレイアウト

先ほどのテンプレートページの「Album Layout」内の「SOURCECODE」をクリックします。



▲図 3.8: ソースコードへのリンク

MUI の GitHub が開き、JavaScript、TypeScript のソースコードがあります。今回は、Album.tsx を開きます。



▲図 3.9: GitHub での Album のソースコード

このコードを作成した「DiaryBoard.tsx」ファイルへ貼り付け、以下を変更します。

- コンポーネント名(関数名)を「DiaryBoard」へ。
- ThemeProviderコンポーネント、CssBaselineコンポーネントを削除。
- ThemeProviderコンポーネントの位置に、`<></>`を置くこと。
- Hero unit、カメラアイコンの削除。
- 「`const theme = createTheme()`」を削除。
- `function`をアロー関数に変更。
- 削除したコンポーネントのimport文を削除。

「`<></>`」をトップのタグにした理由は、「JSXはひとつの要素のみ」と怒られるからです。このタグをトップに置くことで、要素は1つ(ほかは子要素)となります。また、不要なdivタグに変換されません。

変更の完了したコードがこちらです。なります。

▼ src/components/DiaryBoard.tsx

```
import * as React from 'react';
import AppBar from '@mui/material/AppBar';
import Button from '@mui/material/Button';
import Card from '@mui/material/Card';
import CardActions from '@mui/material/CardActions';
import CardContent from '@mui/material/CardContent';
import CardMedia from '@mui/material/CardMedia';
import Grid from '@mui/material/Grid';
import Box from '@mui/material/Box';
import Toolbar from '@mui/material/Toolbar';
import Typography from '@mui/material/Typography';
import Container from '@mui/material/Container';
import Link from '@mui/material/Link';

const Copyright = () => (
  <Typography variant='body2' color='text.secondary' align='center'>
    {'Copyright © '}
    <Link color='inherit' href='https://mui.com/'>
      Your Website
    </Link>
    {` ${new Date().getFullYear()}`}
  </Typography>
);

const cards = [1, 2, 3, 4, 5, 6, 7, 8, 9];

const DiaryBoard = () => (
  <>
```

```
<AppBar position='relative'>
  <Toolbar>
    <Typography variant='h6' color='inherit' noWrap>
      やる夫の読書日記
    </Typography>
  </Toolbar>
</AppBar>
<main>
  <Container sx={{ py: 8 }} maxWidth='md'>
    <Grid container spacing={4}>
      {cards.map((card) => (
        <Grid item key={card} xs={12} sm={6} md={4}>
          <Card
            sx={{
              height: '100%',
              display: 'flex',
              flexDirection: 'column',
            }}
          >
            <CardMedia
              component='img'
              sx={{
                // 16:9
                pt: '56.25%',
              }}
              image='https://source.unsplash.com/random'
              alt='random'
            />
            <CardContent sx={{ flexGrow: 1 }}>
              <Typography gutterBottom variant='h5' component='h2'>
                Heading
              </Typography>
              <Typography>
                This is a media card. You can use this section to describe
                the content.
              </Typography>
            </CardContent>
            <CardActions>
              <Button size='small'>View</Button>
              <Button size='small'>Edit</Button>
            </CardActions>
          </Card>
        </Grid>
      )))
    </Grid>
  </Container>
</main>
{/* Footer */}<Box sx={{ bgcolor: 'background.paper', p: 6 }} component='footer'>
```

```
<Typography variant='h6' align='center' gutterBottom>
  Footer
</Typography>
<Typography
  variant='subtitle1'
  align='center'
  color='text.secondary'
  component='p'
>
  Something here to give the footer a purpose!
</Typography>
<Copyright />
</Box>
{/* End footer */}
</>
);

export default DiaryBoard;
```

次に、App コンポーネントを変更し、DiaryBoard コンポーネントを表示するように変更します。また、先ほど削除した「ThemeProvider」、「CssBaseline」、「createTheme()」も追加します。

追加完了したコードが以下となります。

▼ 変更後の App コンポーネント

```
import React from 'react';
import CssBaseline from '@mui/material/CssBaseline';
import { createTheme, ThemeProvider } from '@mui/material/styles';

import DiaryBoard from './DiaryBoard';

const theme = createTheme();

const App = () => (
  <ThemeProvider theme={theme}>
    <CssBaseline />
    <DiaryBoard />
  </ThemeProvider>
);

export default App;
```

最後に、不要になった「style.css」、「style.scss」を削除し、「index.tsx」を変更します。

▼ index.tsx

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/App';

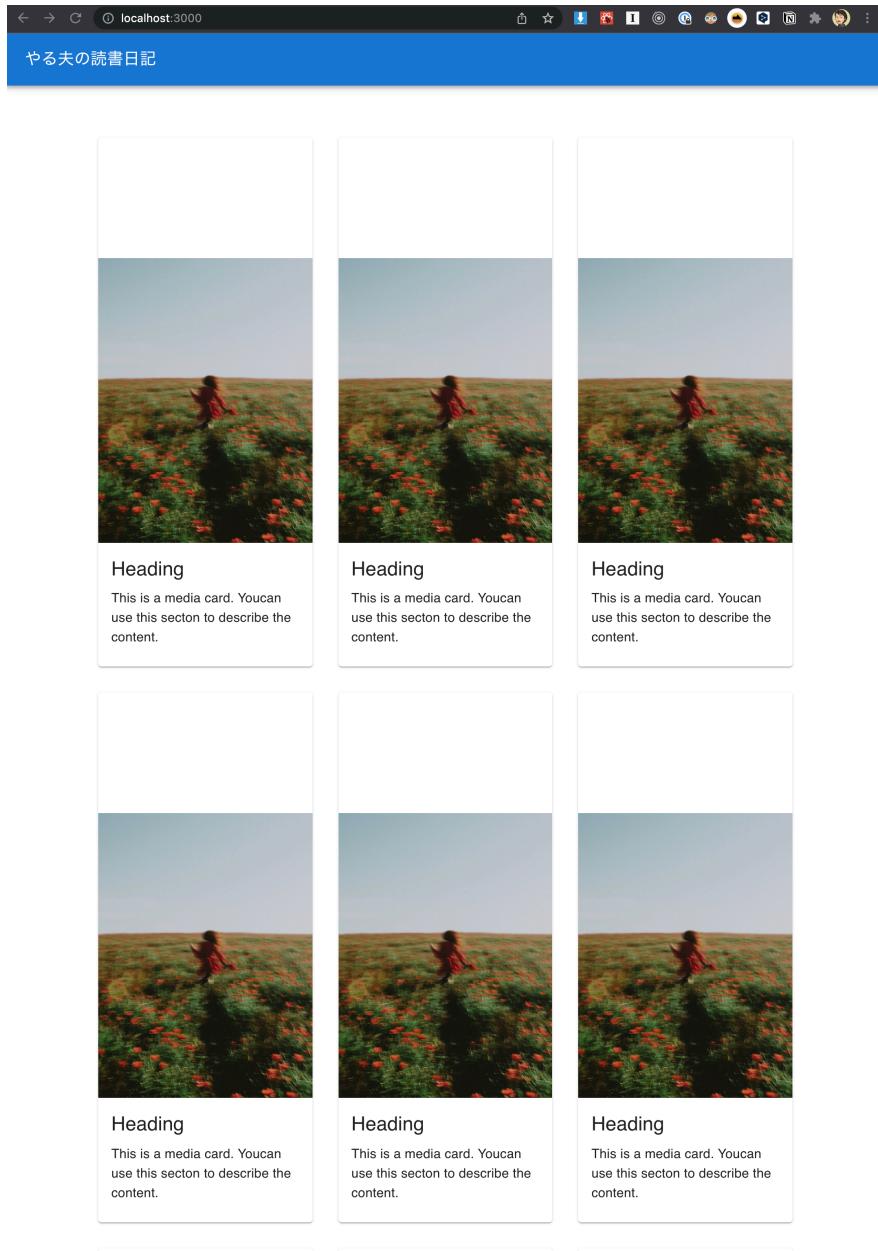
ReactDOM.render(
  <div>
    <App />
  </div>,
  document.getElementById('root')
);
```

以上の変更が完了したら、動作確認します。

▼ サンプルアプリケーションの動作確認

```
> npm run start
```

この画面が表示されれば正常です。カードの画像はランダムですので、違っていてもかまいません。



▲図 3.10: Album Layout の挙動

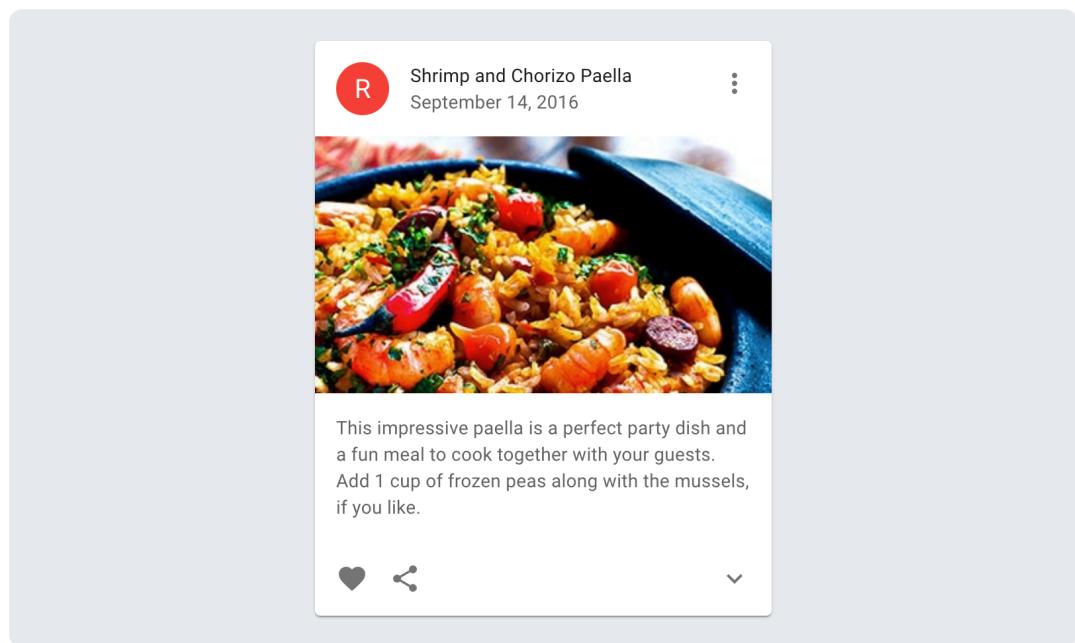
3.3.4 カード一覧画面の表示

カード表示用のボードができましたので、表示するカードは別なものにしましょう。

MUI のメニューから「Components > Card」をクリックすると、たくさんのサンプルがあります。少し下へスクロールすると、こちらが見つかりました。「<>」アイコンをクリックすると、JS/TS 別にソースコードが表示されます。

Complex Interaction

On desktop, card content can expand. (Click the downward chevron to view the recipe.)



▲図 3.11: MUI カードサンプル

「src/components/」フォルダに「DiaryCard.tsx」ファイルを作成しソースコードをkopipeします。

以下を変更します。

- 1行目「import * as React」を「import React」へ(理由はのちほど)
- function 関数をアロー関数へ
- 関数名の変更「DiaryCard」へ
- 表示するデータを Props(引数)として受け取る

- 受け取ったオブジェクトを表示するように tsx に埋め込む

import * as React 命令

「import」は、ES6 仕様のモジュール読み込み方法です。

import * as React from 'react' の場合

from にある「react」が export しているものすべてをインポート

import React from 'react' の場合

from にある「react」が「default」で export しているものだけをインポート

結果としての違いは、webpack でバンドルした場合に作成されるビルドファイルはインポートされたものを含むため、使用しないものまでインポートするとファイルサイズが肥大化する恐れがあります。

function 関数をアロー関数へ

ファンクション関数とアロー関数の違いについては、こちらの記事^{*5}が参考になります。



▲図 3.12: ファンクション関数とアロー関数の違い

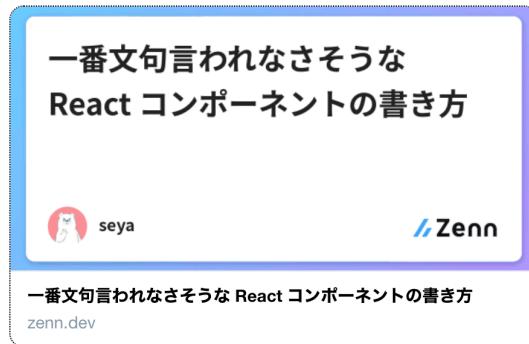
また、React でアロー関数を使うことについては、こちらの記事^{*6}が参考になります。

関数名の変更

関数名を「RecipeReviewCard」から「DiaryCard」へ変更します。export も忘れずに。

*5 <https://qiita.com/suin/items/a44825d253d023e31e4d>

*6 <https://zenn.dev/seya/articles/0317b7a61ee781>



▲図 3.13: React コンポーネントの書き方

表示するデータを受け取る

React コンポーネントの `jsx(tsx)` では、HTML 内に JavaScript のオブジェクトを「`{}`」で埋め込み表示できます。

TypeScript では、受け取るオブジェクトの型を定義します。

コンポーネントは、表示するためのデータ(表示する子要素も含む)を「Props(プロパティの意味)」として受け取ります。コンポーネント関数からしてみると引数にあたります。

受け取る Props(引数)の型を、「`src/diaryData.ts`」ファイルを作成し定義します。このファイルは、のちほど初期値も追加します。

受け取ったオブジェクトを埋め込む

受け取ったオブジェクトを変数に展開し、表示位置に埋め込みます。

DiaryCard を表示する

初期値を設定し、App コンポーネントに「DiaryBoard」の代わりに表示してみます。こちらが表示されれば完成です。もちろん、「下向き」をクリックすると詳細部分が表示されます。readmore 配列の各要素が段落になってています。

ここまで的内容は、GitHub 上で、以下のコマンドでクローンできます。

▼ GitHub

```
> git clone -b 02_Component-Cardboard https://github.com/yaruo-react->
> redux/yaruo-diary.git
```



▲図 3.14: カードの単体表示

3.3.5 リファクタリング1 (サンプルデータ全表示)

用意してあるサンプルデータを初期値として使用し、カード一覧へ表示しましょう。

先ほどデータ型を記入したデータファイル「src/diaryData.ts」へサンプル用データをコピペします。サンプルアプリケーションの GitHub サイトにありますので使ってください。

次に、「App.tsx」を変更します。

1. `initialData` は不要になったので削除。
2. 作成したサンプルデータをインポート。
3. `DiaryBoard` コンポーネントを `props` を受け付けるよう変更

変更後の「App コンポーネント」は、こちらになります。

▼ 変更後の App コンポーネント

```
import React from 'react';
import CssBaseline from '@mui/material/CssBaseline';
import { createTheme, ThemeProvider } from '@mui/material/styles';

import diaries from '../diaryData';
import DiaryBoard from './DiaryBoard';
```

```
const theme = createTheme();

const App = () => (
  <ThemeProvider theme={theme}>
    <CssBaseline />
    <DiaryBoard diaries={diaries} />
  </ThemeProvider>
);

export default App;
```

「DiaryBoard コンポーネント」を変更します。

1. props で Diary 型の配列を受け付ける
2. DiaryCard コンポーネントを使って各データを表示する
3. Footer のべた書き文字列をアプリケーション用に変更する。

変更した「DiaryBoard コンポーネント」が、こちらになります。

▼ 変更後の DiaryBoard コンポーネント

```
import React from 'react';
import AppBar from '@mui/material/AppBar';
import Grid from '@mui/material/Grid';
import Box from '@mui/material/Box';
import Toolbar from '@mui/material/Toolbar';
import Typography from '@mui/material/Typography';
import Container from '@mui/material/Container';
import Link from '@mui/material/Link';

import { Diary } from '../diaryData';
import DiaryCard from './DiaryCard';

export type DiaryBoardProps = {
  diaries: Diary[];
};

const Copyright = () => (
  <Typography variant='body2' color='text.secondary' align='center'>
    {'Copyright © '}
    <Link color='inherit' href='https://mui.com/'>
      やる夫が読書します。
    </Link>
    {` ${new Date().getFullYear()}.`}
  </Typography>
);

const DiaryBoard = (props: DiaryBoardProps) => {
```

```
const { diaries } = props;

return (
  <>
  <AppBar position='relative'>
    <Toolbar>
      <Typography variant='h6' color='inherit' noWrap>
        やる夫の読書日記
      </Typography>
    </Toolbar>
  </AppBar>
  <main>
    <Container sx={{ py: 8 }} maxWidth='md'>
      <Grid container spacing={4}>
        {diaries.map((diary) => (
          <Grid item key={diary.diaryId} xs={12} sm={6} md={4}>
            <DiaryCard diary={diary} />
          </Grid>
        ))}
      </Grid>
    </Container>
  </main>
  {/* Footer */}

  <Box sx={{ bgcolor: 'background.paper', p: 6 }} component='footer'>
    <Typography variant='h6' align='center' gutterBottom>
      やる夫の読書日記
    </Typography>
    <Typography
      variant='subtitle1'
      align='center'
      color='text.secondary'
      component='p'
    >
      お前らも本読んだ良いお～！
    </Typography>
    <Copyright />
  </Box>
  {/* End footer */}
</>
);
};

export default DiaryBoard;
```

変更が完了しましたら、動作確認をします。以下のように表示されると成功です。

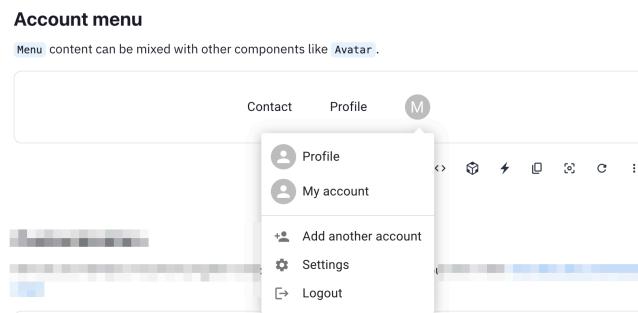


▲図 3.15: 全データ表示

3.3.6 リファクタリング 2(カードヘッダを別コンポーネントへ)

「DiaryCard コンポーネント」のタイトルの右側にある「縦の 3 点」アイコンをクリックしても、今は何も起こりません。このアイコンボタンを利用して、「編集・削除」の機能を持たせます。

MUI サイトのコンポーネント例の「Menu」にあるものを拝借します。



▲図 3.16: アイコンボタンでポップアップメニュー

「CardHeader」を再利用することはないでしょうが、管理・メンテナンスを考えて別コンポーネントにします。

1. 「DiaryCardHeader.tsx」ファイルを作成しコンポーネントのテンプレを書く
2. 「DiaryCard.tsx」から、CardHeader 部分を「DiaryCardHeader.tsx」切り出し
3. 「DiaryCard コンポーネント」に「DiaryCardHeader コンポーネント」をインポートして使用

変更が完了したファイルは、このようになります。

▼ DiaryCardHeader コンポーネント

```
import React from 'react';
import CardHeader from '@mui/material/CardHeader';
import Avatar from '@mui/material/Avatar';
import { red } from '@mui/material/colors';
import MoreVertIcon from '@mui/icons-material/MoreVert';
import IconButton from '@mui/material/IconButton';

export type DiaryCardHeaderProps = {
  diaryId: string;
  title: string;
  postDate: string;
};

const DiaryCardHeader = (props: DiaryCardHeaderProps) => {
  const { diaryId, title, postDate } = props;
  return (
    <CardHeader
```

```
avatar={
  <Avatar sx={{ bgcolor: red[500] }} aria-label='recipe'>
    R
  </Avatar>
}
action={
  <IconButton aria-label='settings'>
    <MoreVertIcon />
  </IconButton>
}
title={title}
subheader={`postID:${diaryId}-${postDate}`}
/>
);
};

export default DiaryCardHeader;
```

「DiaryCard コンポーネント」は、このようになります。

▼ DiaryCard コンポーネントの CardHeader があった部分

```
return (
<Card sx={{ maxWidth: 345 }}>
  <DiaryCardHeader diaryId={diaryId} title={title} postDate={postDate} />
  <CardMedia
    component='img'
    height='194'
    image={imageUrl}
    alt={imageLabel}
  />
  <CardContent>
    <Typography variant='body2' color='text.secondary'>
      {mainContent}
    </Typography>
  </CardContent>
</Card>
```

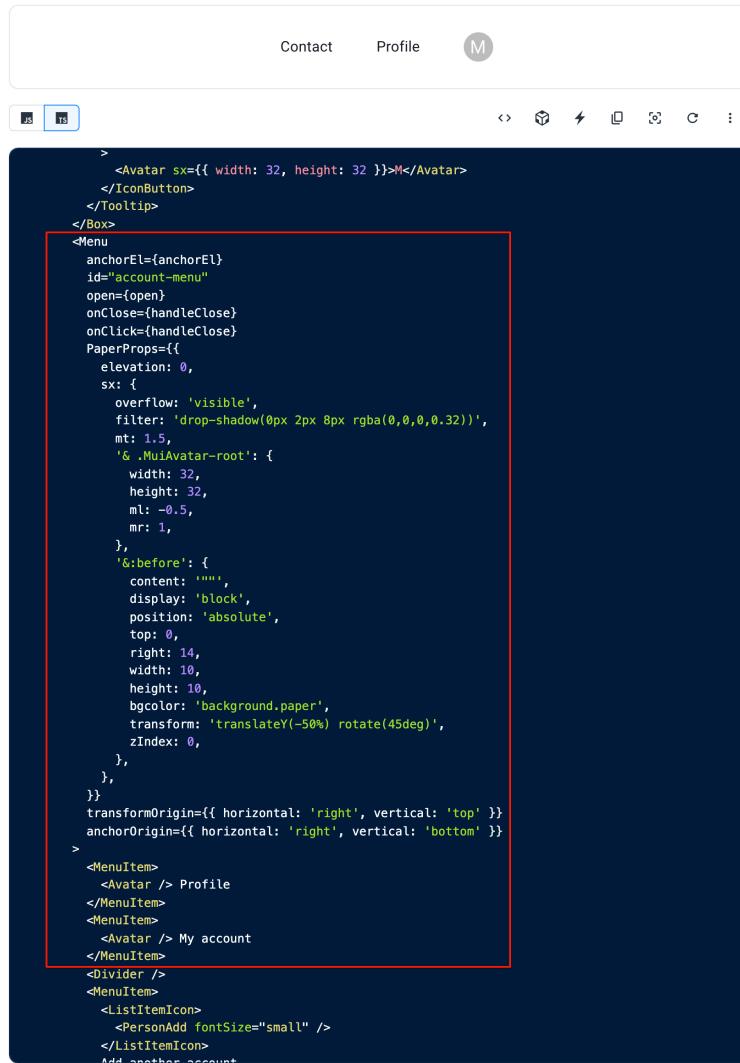
この時点で動作確認を行います。無事に表示されていれば良いです。

DiaryCarHeader コンポーネントにメニューを組み込む

MUI のサイトからメニュー部分のコードを拝借して「DiaryCardHeader コンポーネント」に追加しましょう。やりたいことは「編集、削除」ですので、メニュー項目 (MenuItem) は2つでかまいません。

Account menu

Menu content can be mixed with other components like `Avatar`.



▲ 図 3.17: MUI の Menu サンプル

拝借するコードは、「CardHeader」と同じ階層に貼り付けますが、その場合「`Jsx` はひとつの要素」と怒られますので、トップ階層に「`<></>`」を追加します。

必要な MUI のコンポーネント、アイコンもインポートするのですが、アイコンは、

編集アイコン

```
import EditIcon from '@mui/icons-material/Edit';
```

削除アイコン

```
import DeleteForeverIcon from '@mui/icons-material/DeleteForever';
```

を使い、Avatar コンポーネントではなく「ListItemIcon コンポーネント」を使います。サンプルメニューの区切り線の下の部分で使われています。

最後は、「AccountMenu コンポーネント」の関数(HTML 内に埋め込まれています)も忘れずにコピペしてください。

handleClick 関数は、MoreVertIcon の親要素の IconButton の「onClick」に追加します。

ここまで変更が完了すると「DiaryCardHeader コンポーネント」は、このようになります。

▼ 変更完了の DiaryCardHeader コンポーネント

```
import React from 'react';
import CardHeader from '@mui/material/CardHeader';
import Avatar from '@mui/material/Avatar';
import { red } from '@mui/material/colors';
import Menu from '@mui/material/Menu';
import MenuItem from '@mui/material/MenuItem';
import ListItemIcon from '@mui/material/ListItemIcon';
import MoreVertIcon from '@mui/icons-material/MoreVert';
import EditIcon from '@mui/icons-material/Edit';
import DeleteForeverIcon from '@mui/icons-material/DeleteForever';
import IconButton from '@mui/material/IconButton';

export type DiaryCardHeaderProps = {
  diaryId: string;
  title: string;
  postDate: string;
};

const DiaryCardHeader = (props: DiaryCardHeaderProps) => {
  const { diaryId, title, postDate } = props;
  const [anchorEl, setAnchorEl] = React.useState<null | HTMLElement>(null);
  const open = Boolean(anchorEl);
  const handleClick = (event: React.MouseEvent<HTMLElement>) => {
    setAnchorEl(event.currentTarget);
  };
  const handleClose = () => {
    setAnchorEl(null);
  };

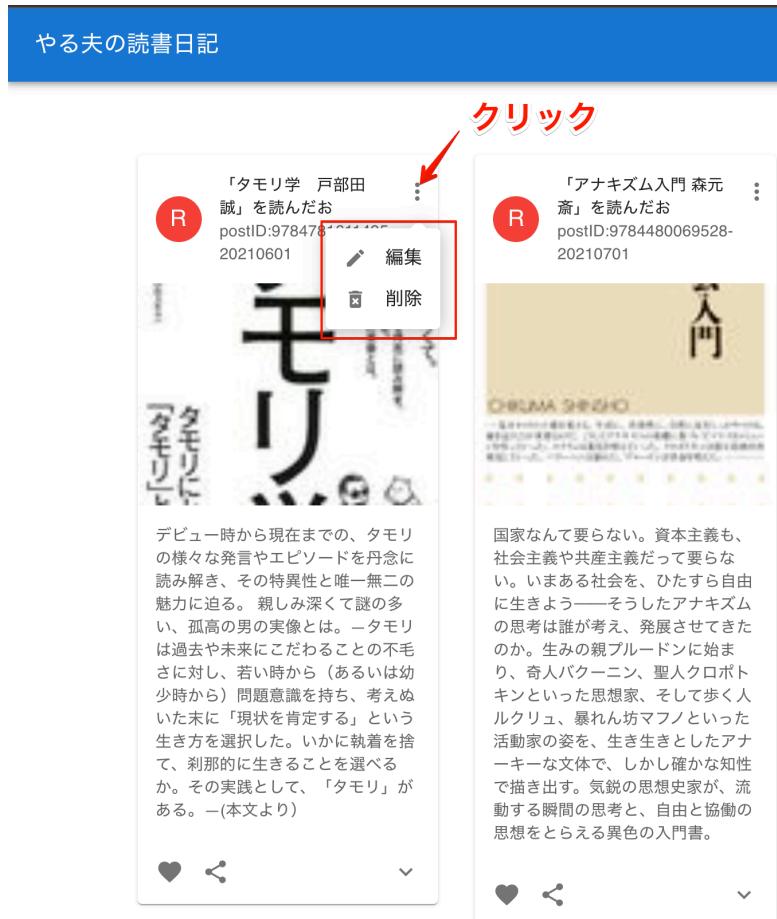
  return (
    <>
      <CardHeader
        avatar={(
          <Avatar sx={{ bgcolor: red[500] }} aria-label='recipe'>
```

```
R
  </Avatar>
}
action={
  <IconButton aria-label='settings' onClick={handleClick}>
    <MoreVertIcon />
  </IconButton>
}
title={title}
subheader={`${postID:${diaryId}-${postDate}}`}
/>
<Menu
  anchorEl={anchorEl}
  id='account-menu'
  open={open}
  onClose={handleClose}
  onClick={handleClose}
  PaperProps={{
    elevation: 0,
    sx: {
      overflow: 'visible',
      filter: 'drop-shadow(0px 2px 8px rgba(0,0,0,0.32))',
      mt: 1.5,
      '& .MuiAvatar-root': {
        width: 32,
        height: 32,
        ml: -0.5,
        mr: 1,
      },
      '&:before': {
        content: '',
        display: 'block',
        position: 'absolute',
        top: 0,
        right: 14,
        width: 10,
        height: 10,
        bgcolor: 'background.paper',
        transform: 'translateY(-50%) rotate(45deg)',
        zIndex: 0,
      },
    },
  }}
  transformOrigin={{ horizontal: 'right', vertical: 'top' }}
  anchorOrigin={{ horizontal: 'right', vertical: 'bottom' }}
>
  <MenuItem>
    <ListItemIcon>
      <EditIcon fontSize='small' />
```

```
</ListItemIcon>
編集
</MenuItem>
<MenuItem>
  <ListItemIcon>
    <DeleteForeverIcon fontSize='small' />
  </ListItemIcon>
  削除
</MenuItem>
</Menu>
</>
);
};

export default DiaryCardHeader;
```

ここまで変更を動作確認します。「縦の3点アイコン」をクリックするとメニューが表示されますか？



▲図 3.18: クリックするとメニューが表示される

メニューが無事表示されたので、のちほど実際の関数に置き換えるとして、テストとしてアラートを出してみます。「編集・削除」の MenuItem コンポーネントに「onClick」を追加し対応する関数を書きます。

ただし、削除がクリックされたときには「本当に削除しますか？」と確認のダイアログを出すようにします。ダイアログは、MUI サイトの「Dialog」から「Transitions」を拝借します。



▲図 3.19: MUI サイトの Transitions ダイアログ

「Dialog コンポーネント」のソースコードを表示し、`<Dialog>` ~ `</Dialog>`を`</Menu>`の下へコピペします。不要なものは削除し、「キャンセル」、「削除」のボタンを作成します。もちろん、必要な関数もコピペします。

コピペすると、メニューの開閉状態の「open」とダイアログ表示状態の「open」が重複しますので、それぞれ「openMenu」、「openDialog」に名前を変えます。また閉じる関数「handleClose」も重複しますので名前を変えます。

Menu の「削除」をクリック -> 確認ダイアログ表示 -> 削除 -> アラート表示になるように関数呼び出します。

ここまで変更が完了しましたら動作確認します。

- 編集メニューをクリックしたときにアラートはでましたか？
- 削除メニューをクリックしたときに確認ダイアログが表示しましたか？
- 確認ダイアログの削除をクリックしたときにアラートはでましたか？



▲ 図 3.20: 削除確認ダイアログの表示

DiaryBoardHeader の見栄え

もう少し見栄え良くしたいので、DiaryCardHeader コンポーネントの、

1. アバターの代わりに投稿月の画像を表示
2. サブタイトルに投稿日を「YYYY 年 M 月 D 日」で表示

を実装します。

アバターの代わりに投稿月画像を表示

1月～12月までのSVG画像は、GitHubサイトにあります。「src/assets/images/month-icons」フォルダを作成してコピペしてください。

読書日記データの投稿日は「YYYYMMDD」形式の文字列ですので、この文字列からJavaScriptのDateオブジェクトを返す関数を作成します。また、のちほど作成する読書日記の新規追加・編集時のため日付から「YYYYMMDD」の文字列を作成する関数も合わせて作成します。

どの場所からも使えるように「src/utilities/helper.ts」ファイルを作成し、ここに関数を作成します。

▼ src/utilities/helper.ts

```
// DateオブジェクトからYYYYMMDDの文字列へ変換
export const convertDateToString = (date: Date): string => {
  const monthString = `0${date.getMonth() + 1}`.slice(-2);
  const dayString = `0${date.getDate()}`.slice(-2);

  return `${date.getFullYear()}${monthString}${dayString}`;
};

// YYYYMMDD文字列からDateオブジェクトへ変換
export const convertStringToDate = (dateString: string): Date => {
  let date;
  if (dateString.length !== 8) {
    date = new Date();
  } else {
    date = new Date(
      +dateString.slice(0, 4),
      +dateString.slice(4, 6) - 1,
      +dateString.slice(6)
    );
  }
  return date;
};
```

convertStringToDate関数をインポートし、読書日記の「YYYYMMDD」から月を取得し、画像のソースを指定します。DiaryCardHeaderコンポーネントに追加する関数は、こちらとなります。

▼ Date と文字列の変換関数

```
// DateオブジェクトからYYYYMMDDの文字列へ変換
export const convertDateToString = (date: Date): string => {
  const monthString = `0${date.getMonth() + 1}`.slice(-2);
  const dayString = `0${date.getDate()}`.slice(-2);

  return `${date.getFullYear()}${monthString}${dayString}`;
};

// YYYYMMDD文字列からDateオブジェクトへ変換
export const convertStringToDate = (dateString: string): Date => {
  let date;
  if (dateString.length !== 8) {
    date = new Date();
  } else {
    date = new Date(
      +dateString.slice(0, 4),
      +dateString.slice(4, 6) - 1,
      +dateString.slice(6)
    );
  }
  return date;
};
```

DiaryCardHeader コンポーネントにアバターのソースを切り替える関数を作成し、「CardHeader コンポーネント内 Avatar コンポーネント」の src 要素にします。ついでに、四角形に変えサイズも大きめにします。

追加したコードです。

▼ 画像の切替と Avatar への指定

```
// postDate:YYYYMMDDから月を取得し
const datePosted: Date = convertStringToDate(postDate);
let avatarSrc;
// useEffect(() => {
switch (datePosted.getMonth()) {
  case 0:
    avatarSrc = January;
    break;
  case 1:
    avatarSrc = February;
    break;
  case 2:
    avatarSrc = March;
    break;
  case 3:
```

```
        avatarSrc = April;
        break;
    case 4:
        avatarSrc = May;
        break;
    case 5:
        avatarSrc = Jun;
        break;
    case 6:
        avatarSrc = July;
        break;
    case 7:
        avatarSrc = August;
        break;
    case 8:
        avatarSrc = September;
        break;
    case 9:
        avatarSrc = October;
        break;
    case 10:
        avatarSrc = November;
        break;
    case 11:
        avatarSrc = December;
        break;
    default:
        break;
    }

    return (
      <>
      <CardHeader
        avatar={
          <Avatar
            sx={{ width: 58, height: 58 }}
            variant='square'
            aria-label='recipe'
            src={avatarSrc}
          />
        }
      >
```

サブタイトルに投稿日を「YYYY年M月D日」で表示

「src/utilities/helper.ts」に、「YYYYMMDD」文字列から「YYYY年M月D日」に変換する関数を作成します。

▼ YYYY 年 M 月 D 日文字列の作成関数

```
// YYYYMMDDからYYYY年M月Dに変換
export const convertToLongDateString = (dateString: string) => {
  const date = convertStringToDate(dateString);
  return `${date.getFullYear()}年${date.getMonth() + 1}月${date.getDate()}日`;
};
```

DiaryCardHeader コンポーネントへ「convertToLongDateString」をインポートし、投稿日データを表示します。

▼ 投稿日を表示

```
const posted = convertToLongDateString(postDate);

<CardHeader
  avatar={
    <Avatar
      sx={{ width: 58, height: 58 }}
      variant='square'
      aria-label={`投稿日:${posted}`} ←ラベルにも表示
      src={avatarSrc}
    />
  }
  action={
    <IconButton aria-label='settings' onClick={handleClick}>
      <MoreVertIcon />
    </IconButton>
  }
  title={title}
  subheader={posted} ←サブタイトルに投稿日
/>
```

さて、ここまで変更が完了したら、動作確認を行います。



▲ 図 3.21: desc

ここまで的内容は、GitHub 上で、以下のコマンドでクローンできます。

▼ GitHub

```
> git clone -b 04_refactoring_DiaryHeader https://github.com/yaruo-re>
> act-redux/yaruo-diary.git
```

3.3.7 入力フォームコンポーネントの作成

読書データの新規追加・編集のためのフォームを作成します。

フォームは、Dialog として DiaryBoard コンポーネントに追加し、DiaryBoard コンポーネントに新規追加ボタンで表示するようにします。作成するフォームは別コンポーネントとするために「src/components/DiaryForm.tsx」ファイルを作成します。

MUI サイトの Components 内の Text field コンポーネントには、フォームとして使えるそうなサンプルがあります。

The screenshot shows the MUI website at mui.com/components/text-fields/. The page title is "Basic TextField". It features three examples: "Outlined", "Filled", and "Standard". Below the examples is a code snippet:

```
<TextField id="outlined-basic" label="Outlined" variant="outlined" />
<TextField id="filled-basic" label="Filled" variant="filled" />
<TextField id="standard-basic" label="Standard" variant="standard" />
```

Note: The standard variant of the `TextField` is no longer documented in the [Material Design guidelines](#) (here's [why](#)), but MUI will continue to support it.

Form props

Standard form attributes are supported e.g. `required`, `disabled`, `type`, etc. as well as a `helperText` which is used to give context about a field's input, such as how the input will be used.

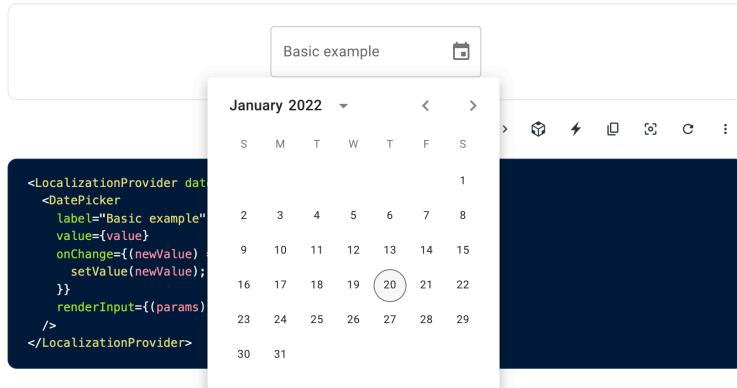
The page displays several examples of `TextField` components with various props applied, such as `Required`, `Disabled`, `Read Only`, `Helper text`, `Default Value`, and `Some important text`.

▲ 図 3.22: MUI サイトのテキストフィールド

これらを参考にしてフォームを作成します。また、投稿日の入力用に MUI の DatePicker を使うのですが、このコンポーネントはラボ (実験中) に分類されています。そのため MUI の Lab と日付を扱うためのライブラリ 「date-fns」 をインストールします。

Basic usage

The date picker is rendered as a modal dialog on mobile, and a textbox with a popup on desktop.



▲図 3.23: MUI の DatePicker

▼MUI のラボ、date-fns のインストール

```
☒ npm install @mui/lab date-fns
```

DiaryForm コンポーネントは、props として読書日記データを受け取ります。

- 新規入力の場合には、すべてが空欄の読書日記データ
- 編集時には、指定された読書日記データ

となります。

DiaryForm コンポーネントにあるすべてのテキストフィールドは、React Hooks のひとつの `useState` を使用して入力されたデータを保持します。

テキストフィールドに入力されるデータは、タイトルと本文のみは 4 文字以上の入力を必須とします。

作成したフォームは、こちらになります。

▼DiaryForm

```
import React, { useState, useRef } from 'react';
import Grid from '@mui/material/Grid';
import Paper from '@mui/material/Paper';
import TextField from '@mui/material/TextField';
import AdapterDateFns from '@mui/lab/AdapterDateFns';
import LocalizationProvider from '@mui/lab/LocalizationProvider';
import DatePicker from '@mui/lab/DatePicker';
```

```
import Button from '@mui/material/Button';
import CancelIcon from '@mui/icons-material/Cancel';
import DataSaverOnIcon from '@mui/icons-material/DataSaverOn';
import Stack from '@mui/material/Stack';

import { convertStringToDate } from '../utilities/helper';
import { Diary } from '../diaryData';

interface DiaryFormProps {
  diary: Diary;
}

const DiaryForm = (props: DiaryFormProps) => {
  const { diary } = props;
  const {
    diaryId,
    title,
    postDate,
    imageUrl,
    imageLabel,
    mainContent,
    readmore,
  } = diary;

  // hooksでフォームデータ保持
  // タイトル
  const [onEditTitle, setOnEditTitle] = useState(title);
  const [diaryTitleErr, setDiaryTitleErr] = useState(false);
  const [diaryTitleErrMsg, setDiaryTitleErrMsg] = useState('');
  // 投稿日
  const [onEditPostDate, setOnEditPostDate] = useState<Date | null>(
    convertStringToDate(postDate)
  );
  // 画像URL
  const [onEditImageUrl, setOnEditImageUrl] = useState(imageUrl);
  // 画像ALT
  const [onEditImageLabel, setOnEditImageLabel] = useState(imageLabel);
  // 本文
  const [onEditMainContent, setOnEditMainContent] = useState(mainContent);
  const [mainContentErr, setMainContentErr] = useState(false);
  const [mainContentErrMsg, setMainContentErrMsg] = useState('');
  // 追記
  const [onEditReadMore, setOnEditReadMore] = useState(readmore.join('\n\n'));

  const titleRef = useRef<HTMLInputElement>(null);
  const mainContentRef = useRef<HTMLInputElement>(null);

  // 入力データ検証
  const validateFieldData = (event: React.ChangeEvent<HTMLInputElement>) => {
```

```
switch (event.target.name) {
  case 'diaryTitle':
    setOnEditTitle(event.target.value);
    if (event.target.value.length < 5) {
      setDiaryTitleErr(true);
      setDiaryTitleErrMsg('4文字以上入力してください。');
    } else {
      setDiaryTitleErr(false);
      setDiaryTitleErrMsg('');
    }
    break;
  case 'diaryMainContent':
    setOnEditMainContent(event.target.value);
    if (event.target.value.length < 5) {
      setMainContentErr(true);
      setMainContentErrMsg('4文字以上入力してください。');
    } else {
      setMainContentErr(false);
      setMainContentErrMsg('');
    }
    break;
  case 'diaryReadMore':
    setOnEditReadMore(event.target.value);
    break;
  case 'diaryImageUrl':
    setOnEditImageUrl(event.target.value);
    break;
  case 'diaryImageLabel':
    setOnEditImageLabel(event.target.value);
    break;
  default:
    break;
}
};

// 保存ボタン
const saveData = () => {};

// キャンセルボタン
const cancelForm = () => {};

return (
  <Paper variant='outlined' sx={{ m: 1, py: 2 }}>
    <Grid container spacing={2} sx={{ pl: 5 }}>
      <Grid item xs={10} sm={10} md={10} lg={10}>
        <TextField
          required
          id='diary-title'
          label='タイトル'
```

```
        error={diaryTitleErr}
        fullWidth
        name='diaryTitle'
        value={onEditTitle}
        helperText={diaryTitleErrMsg}
        onChange={validateFieldData}
        inputRef={titleRef}
    />
</Grid>
<Grid item xs={10} sm={8} md={6} lg={4}>
    {/* eslint-disable-next-line @typescript-eslint/no-unsafe-assignment */}
>/}
    <LocalizationProvider dateAdapter={AdapterDateFns}>
        <DatePicker
            label='日付'
            value={onEditPostDate}
            onChange={(newValue: Date | null) => {
                setOnEditPostDate(newValue);
            }}
            // eslint-disable-next-line react/jsx-props-no-spreading
            renderInput={(params) => <TextField {...params} />}
        />
    </LocalizationProvider>
</Grid>
<Grid item xs={10} sm={10} md={10} lg={10}>
    <TextField
        required
        multiline
        id='diary-mainContent'
        label='本文'
        error={mainContentErr}
        fullWidth
        name='diaryMainContent'
        value={onEditMainContent}
        helperText={mainContentErrMsg}
        onChange={validateFieldData}
        inputRef={mainContentRef}
    />
</Grid>
<Grid item xs={10} sm={10} md={10} lg={10}>
    <TextField
        multiline
        minRows='4'
        maxRows='6'
        id='diary-readmore'
        label='追記'
        fullWidth
        name='diaryReadMore'
        value={onEditReadMore}
```

```
        onChange={validateFieldData}
        helperText='空行を入れると段落表示されます。'
      />
    </Grid>
    <Grid item xs={10} sm={10} md={10} lg={10}>
      <TextField
        id='diary-imageUrl'
        label='画像URL'
        fullWidth
        name='diaryImageUrl'
        onChange={validateFieldData}
        value={onEditImageUrl}
        helperText='画像のURL'
      />
    </Grid>
    <Grid item xs={10} sm={10} md={10} lg={10}>
      <TextField
        id='diary-imageLabel'
        label='画像の代替テキスト'
        fullWidth
        name='diaryImageLabel'
        onChange={validateFieldData}
        value={onEditImageLabel}
        helperText='画像が表示されない場合のテキスト'
      />
    </Grid>
    <Grid item xs={10} sm={10} md={10} lg={10}>
      <Stack direction='row' spacing={2}>
        <Button
          variant='contained'
          startIcon={<CancelIcon />}
          onClick={cancelForm}
        >
          キャンセル
        </Button>
        <Button
          variant='contained'
          endIcon={<DataSaverOnIcon />}
          onClick={saveData}
        >
          保存
        </Button>
      </Stack>
    </Grid>
  </Grid>
</Paper>
);
};
```

```
export default DiaryForm;
```

作成したフォームをダイアログとして表示するコンポーネントを作成します。

「src/components/DiaryFormDialog.tsx」ファイルを作成し、MUI のダイアログの表示・非表示を実装します。

▼ DiaryFormDialog

```
import React from 'react';
import Dialog from '@mui/material/Dialog';
importDialogTitle from '@mui/material/DialogTitle';

import { Diary } from '../diaryData';
import DiaryForm from './DiaryForm';

export interface DialogDiaryFormProps {
  open: boolean;
  diary: Diary;
}

const DialogDiaryForm = (props: DialogDiaryFormProps) => {
  const { open, diary } = props;

  return (
    <Dialog open={open}>
      <DialogTitle>日記編集</DialogTitle>
      <DiaryForm diary={diary} />
    </Dialog>
  );
};

export default DialogDiaryForm;
```

DiaryBoard コンポーネントに、新規追加ボタンを追加し、フォームが開くよう実装します。

1. 空欄の初期化データ initialDiary を作成
 2. DiaryFormDialog コンポーネントをインポートし追加
 3. 新規追加アイコンをツールバーに追加し、クリックすると DiaryFormDialog が開く
- 以上の変更を加えた「DiaryBoard コンポーネント」は、こちらになります。

▼ 新規追加ボタンを追加した DiaryBoard コンポーネント

```
import React, { useState } from 'react';
import AppBar from '@mui/material/AppBar';
```

```
import Grid from '@mui/material/Grid';
import Box from '@mui/material/Box';
import Toolbar from '@mui/material/Toolbar';
import Typography from '@mui/material/Typography';
import Container from '@mui/material/Container';
import Link from '@mui/material/Link';
import IconButton from '@mui/material/IconButton';
import AddCircleOutlineIcon from '@mui/icons-material/AddCircleOutline';

import { Diary } from '../diaryData';
import DiaryCard from './DiaryCard';
import DiaryFormDialog from './DiaryFormDialog';

export type DiaryBoardProps = {
  diaries: Diary[];
};

// diary初期化データ
const initialDiary: Diary = {
  diaryId: '',
  title: '',
  postDate: '',
  imageUrl: '',
  imageLabel: '',
  mainContent: '',
  readmore: [],
};

const Copyright = () => (
  <Typography variant='body2' color='text.secondary' align='center'>
    {'Copyright © '}
    <Link color='inherit' href='https://mui.com/'>
      やる夫が読書します。
    </Link>
    {` ${new Date().getFullYear()}.`}
  </Typography>
);

const DiaryBoard = (props: DiaryBoardProps) => {
  const { diaries } = props;

  // DiaryFormDialogの開閉状態
  const [openDialog, setOpenDialog] = useState(false);

  // 新規データでDiaryFormDialogを開く
  const openForm = () => setOpenDialog(true);

  return (
    <>
```

```
<AppBar position='relative'>
  <Toolbar>
    <Box sx={{ flexGrow: 1 }}>
      <Typography variant='h6' color='inherit' noWrap>
        やる夫の読書日記
      </Typography>
    </Box>
    <Box>
      <IconButton
        size='large'
        edge='end'
        aria-label='新規読書日記'
        onClick={openForm}
        color='inherit'
      >
        <AddCircleOutlineIcon />
      </IconButton>
    </Box>
  </Toolbar>
</AppBar>
<main>
  <Container sx={{ py: 8 }} maxWidth='md'>
    <Grid container spacing={4}>
      {diaries.map((diary) => (
        <Grid item key={diary.diaryId} xs={12} sm={6} md={4}>
          <DiaryCard diary={diary} />
        </Grid>
      ))}
    </Grid>
  </Container>
</main>
{/* Footer */}
<Box sx={{ bgcolor: 'background.paper', p: 6 }} component='footer'>
  <Typography variant='h6' align='center' gutterBottom>
    やる夫の読書日記
  </Typography>
  <Typography
    variant='subtitle1'
    align='center'
    color='text.secondary'
    component='p'
  >
    お前らも本読んだ良いお～！
  </Typography>
  <Copyright />
</Box>
{/* End footer */}
<DiaryFormDialog open={openDialog} diary={initialDiary} />
</>
```

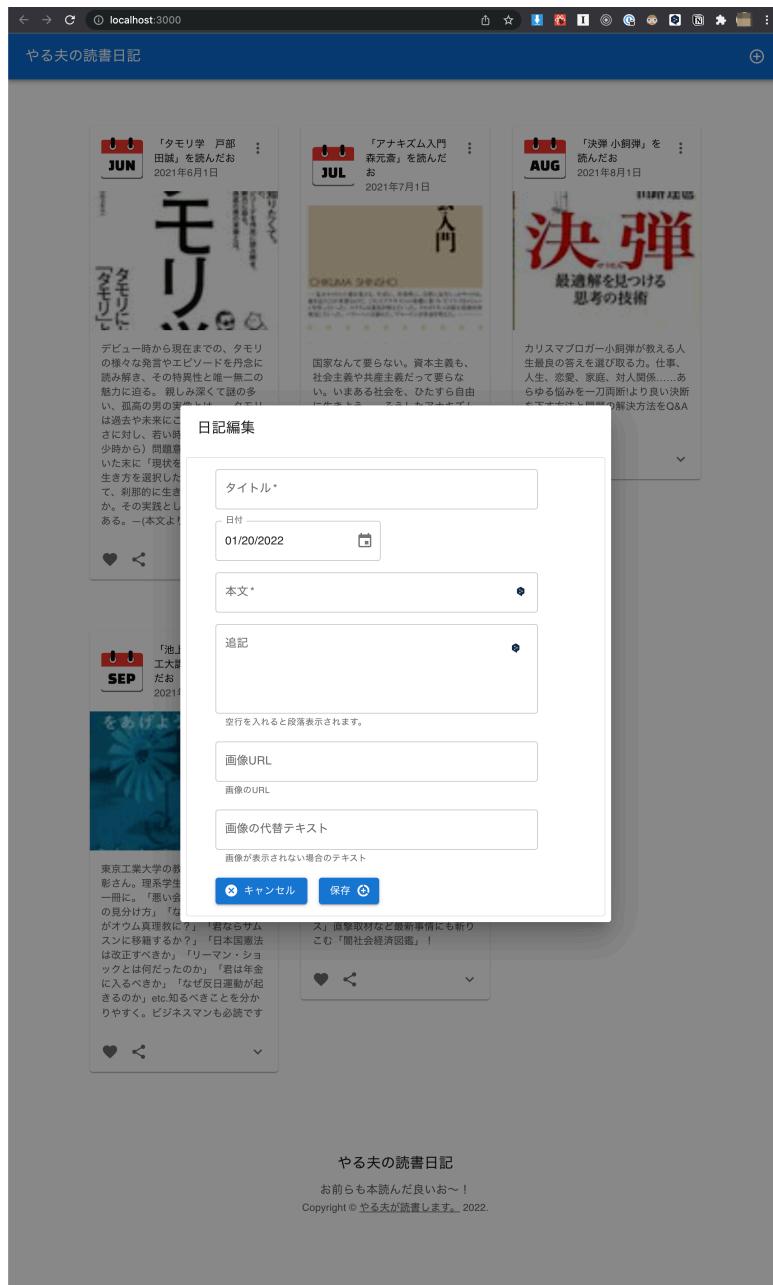
```
    );
};

export default DiaryBoard;
```

変更が完了したら動作確認をします。ツールバーに「+」のアイコンボタンが表示され、クリックすると入力フォームが開きます。

- それぞれのテキストフィールドに入力はできますか？
- タイトルと本文が4文字以下だとエラーがでますか？
- DatePickerは、動作しますか？

開いた入力フォームですが、保存・キャンセルとも機能実装していません。そのため一度開いたフォームを閉じることができません。



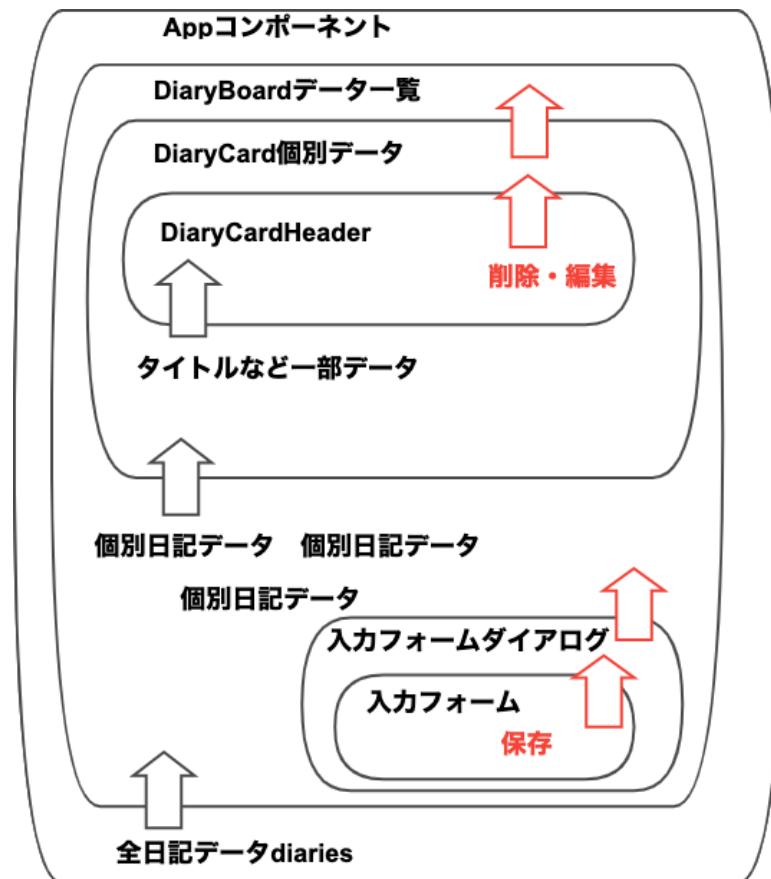
▲ 図 3.24: 入力フォームを開いた状態

3.4 データの追加・編集・削除

表示用のコンポーネントが完成したので、データの追加・編集削除ができるように機能実装を行います。

分かりづらい図ですが、コンポーネントの構成を図にしました。React コンポーネントは表示するためのデータを Props を介して受け取るため、孫コンポーネントに表示するデータも祖父母コンポーネントから親コンポーネントを経由して渡します。

これが「React あるある」のひとつ「Props バケツリレー」です。



▲図 3.25: コンポーネントの構成図

今回のように孫コンポーネントにあたる「DiaryCardHeader コンポーネント」の編集・削除の要求があった場合には全データを管理する「DiaryBoard コンポーネント」に伝えな

ければなりません。

表示は、親コンポーネントからバケツリレーで伝え、孫コンポーネントで受けたユーザーからの要求は逆のルートをたどって親まで伝えることになります。孫で発生した要求を親に伝える方法として一般的なのが、関数を Props としてデータと一緒に子・孫へ渡す方法です。

今回のアプリケーションでは、DiaryBoard コンポーネントが、

- 削除要求は、diaryId を知らせてね。
- 編集要求は、全データ持っているので編集対象の diaryId を知らせてね。フォームにセットするから。
- 入力フォームでの保存要求は、保存するデータをください。diaryId が既存のものは上書き、ほかは追加。

の関数をそれぞれのコンポーネントにバケツリレーし、孫で関数を実行すれば親に要求が伝わることになります。

データの変化が起こると、ブラウザが再描画されます。React コンポーネントの場合、再描画されるためのデータの変化とは、props とコンポーネント自身で管理しているものです。

DiaryBoard コンポーネントでは、管理するデータを useState で状態管理し、ブラウザの描画にもその状態を使う必要があります。そのため DiaryBoard コンポーネントに useState を追加します。

▼ DiaryBoard.tsx

```
// 全データ Appコンポーネントから受け取ったデータで初期化する
const [diaryData, setDiaryData] = useState(diaries);

• • • 中略

// 管理しているデータを描画に使用する
<Grid container spacing={4}>
  {diaryData.map((diary) => (
    <Grid item key={diary.diaryId} xs={12} sm={6} md={4}>
      <DiaryCard
        diary={diary}
        onClickCardHeaderAction={onClickCardHeaderAction}
      />
    </Grid>
  )))
</Grid>
```

3.4.1 削除・編集関数を Props にデータと渡す

DiaryCardHeader コンポーネントの Props に「diaryId、編集・削除フラグ」を引数とする関数を追加します。

▼ DiaryCardHeader.tsx の変更

```
export type DiaryCardHeaderProps = {
  diaryId: string;
  title: string;
  postDate: string;
  onClickCardHeaderAction: (diaryId: string, mode: string) => void;
};
```

DiaryHeader コンポーネントでは、受けた Props から関数を取り出し、メニューがクリックされたときに実行する関数を実装します。

▼ DiaryCardHeader.tsx

```
//propsからの取り出し
const { onClickCardHeaderAction, diaryId, title, postDate } = props;

// 編集メニュークリック
const handleEdit = () => {
  handleCloseMenu();
  return onClickCardHeaderAction(diaryId, 'EDIT');
};

// ダイアログの削除ボタンクリック
const handleConfirmDelete = () => {
  setOpenDialog(false);
  return onClickCardHeaderAction(diaryId, 'DELETE');
};
```

完成了した DiaryCardHeader コンポーネントは、こちらです。

▼ DiaryCardHeader.tsx

```
/* eslint-disable @typescript-eslint/no-unsafe-assignment */
import React from 'react';
import CardHeader from '@mui/material/CardHeader';
import Avatar from '@mui/material/Avatar';
import Menu from '@mui/material/Menu';
import MenuItem from '@mui/material/MenuItem';
import ListItemIcon from '@mui/material/ListItemIcon';
import MoreVertIcon from '@mui/icons-material/MoreVert';
import EditIcon from '@mui/icons-material/Edit';
import DeleteForeverIcon from '@mui/icons-material/DeleteForever';
```

```
import IconButton from '@mui/material/IconButton';
import Button from '@mui/material/Button';
import Dialog from '@mui/material/Dialog';
import DialogActions from '@mui/material/DialogActions';
import DialogContent from '@mui/material/DialogContent';
import DialogContentText from '@mui/material/DialogContentText';
import DialogTitle from '@mui/material/DialogTitle';

import January from '../assets/images/month-icons/january.svg';
import February from '../assets/images/month-icons/february.svg';
import March from '../assets/images/month-icons/march.svg';
import April from '../assets/images/month-icons/april.svg';
import May from '../assets/images/month-icons/may.svg';
import Jun from '../assets/images/month-icons/june.svg';
import July from '../assets/images/month-icons/july.svg';
import August from '../assets/images/month-icons/august.svg';
import September from '../assets/images/month-icons/september.svg';
import October from '../assets/images/month-icons/october.svg';
import November from '../assets/images/month-icons/november.svg';
import December from '../assets/images/month-icons/december.svg';

import {
  convertToLongDateString,
  convertStringToDate,
} from '../utilities/helper';

export type DiaryCardHeaderProps = {
  diaryId: string;
  title: string;
  postDate: string;
  onClickCardHeaderAction: (diaryId: string, mode: string) => void;
};

const DiaryCardHeader = (props: DiaryCardHeaderProps) => {
  const { onClickCardHeaderAction, diaryId, title, postDate } = props;

  // menuの開閉状態を管理
  const [anchorEl, setAnchorEl] = React.useState<null | HTMLElement>(null);
  // ダイアログの表示・非表示
  const [openDialog, setOpenDialog] = React.useState(false);

  // menuの開閉
  const openMenu = Boolean(anchorEl);
  // MoreVerIconクリック
  const handleClick = (event: React.MouseEvent<HTMLElement>) => {
    setAnchorEl(event.currentTarget);
  };

  // menuを閉じる
}
```

```
const handleCloseMenu = () => {
  setAnchorEl(null);
};

// ダイアログ表示
const handleClickOpenDialog = () => {
  handleCloseMenu();
  setOpenDialog(true);
};

// 編集メニュークリック
const handleEdit = () => {
  handleCloseMenu();
  return onClickCardHeaderAction(diaryId, 'EDIT');
};

// 削除メニュークリック
const handleDelete = () => {
  handleClickOpenDialog();
};

// ダイアログ非表示
const handleCloseDialog = () => {
  setOpenDialog(false);
};

// ダイアログの削除ボタンクリック
const handleConfirmDelete = () => {
  setOpenDialog(false);
  return onClickCardHeaderAction(diaryId, 'DELETE');
};

// postDate:YYYYMMDDから月を取得し
const datePosted: Date = convertStringToDate(postDate);
let avatarSrc;
// useEffect(() => {
switch (datePosted.getMonth()) {
  case 0:
    avatarSrc = January;
    break;
  case 1:
    avatarSrc = February;
    break;
  case 2:
    avatarSrc = March;
    break;
  case 3:
    avatarSrc = April;
    break;
}
```

```
case 4:
    avatarSrc = May;
    break;
case 5:
    avatarSrc = Jun;
    break;
case 6:
    avatarSrc = July;
    break;
case 7:
    avatarSrc = August;
    break;
case 8:
    avatarSrc = September;
    break;
case 9:
    avatarSrc = October;
    break;
case 10:
    avatarSrc = November;
    break;
case 11:
    avatarSrc = December;
    break;
default:
    break;
}

const posted = convertToLongDateString(postDate);

return (
  <>
  <CardHeader
    avatar={
      <Avatar
        sx={{ width: 58, height: 58 }}
        variant='square'
        aria-label={`投稿日:${posted}`}
        src={avatarSrc}
      />
    }
    action={
      <IconButton aria-label='settings' onClick={handleClick}>
        <MoreVertIcon />
      </IconButton>
    }
    title={title}
    subheader={posted}
  />
```

```
<Menu
  anchorEl={anchorEl}
  id='account-menu'
  open={openMenu}
  onClose={handleCloseMenu}
  onClick={handleCloseMenu}
  PaperProps={{
    elevation: 0,
    sx: {
      overflow: 'visible',
      filter: 'drop-shadow(0px 2px 8px rgba(0,0,0,0.32))',
      mt: 1.5,
      '&.MuiAvatar-root': {
        width: 32,
        height: 32,
        ml: -0.5,
        mr: 1,
      },
      '&:before': {
        content: '',
        display: 'block',
        position: 'absolute',
        top: 0,
        right: 14,
        width: 10,
        height: 10,
        bgcolor: 'background.paper',
        transform: 'translateY(-50%) rotate(45deg)',
        zIndex: 0,
      },
    },
  }}
  transformOrigin={{ horizontal: 'right', vertical: 'top' }}
  anchorOrigin={{ horizontal: 'right', vertical: 'bottom' }}
>
  <MenuItem onClick={handleEdit}>
    <ListItemIcon>
      <EditIcon fontSize='small' />
    </ListItemIcon>
    編集
  </MenuItem>
  <MenuItem onClick={handleDelete}>
    <ListItemIcon>
      <DeleteForeverIcon fontSize='small' />
    </ListItemIcon>
    削除
  </MenuItem>
</Menu>
<Dialog open={openDialog} onClose={handleCloseDialog}>
```

```
<DialogTitle>削除の確認だお～</DialogTitle>
<DialogContent>
  <DialogContentText>
    削除すると、基には戻せないお～！ それでも削除するのかお～？
  </DialogContentText>
</DialogContent>
<DialogActions>
  <Button onClick={handleCloseDialog}>キャンセル</Button>
  <Button onClick={handleConfirmDelete}>削除</Button>
</DialogActions>
</Dialog>
</>
);
};

export default DiaryCardHeader;
```

次に、DiaryCardHeader コンポーネントの親の DiaryCard コンポーネントも親から受け取り、子に渡します。

▼ DiaryCard.tsx

```
// propsの型定義
export type DiaryCardProps = {
  diary: Diary;
  onClickCardHeaderAction: (diaryId: string, mode: string) => void;
};

// DiaryCardコンポーネント
const DiaryCard = (props: DiaryCardProps) => {
  // 詳細表示コンポーネントの表示・非表示の状態
  const [expanded, setExpanded] = React.useState(false);

  // 詳細コンポーネントの表示・非表示を切り替える関数
  const handleExpandClick = () => {
    setExpanded(!expanded);
  };

  // 受け取ったオブジェクトを変数に展開
  const { diary, onClickCardHeaderAction } = props;

  return (
    <Card sx={{ maxWidth: 345 }}>
      <DiaryCardHeader
        diaryId={diaryId}
        title={title}
        postDate={postDate}
        onClickCardHeaderAction={onClickCardHeaderAction} ←子に送る
    
```

```
/>
```

バケツリレーですので、受け取って渡すだけです。変更後の DiaryCardHeader コンポーネントは、こうなります。

▼ DiaryCard.tsx

```
import React from 'react';
import { styled } from '@mui/material/styles';
import Card from '@mui/material/Card';
import CardMedia from '@mui/material/CardMedia';
import CardContent from '@mui/material/CardContent';
import CardActions from '@mui/material/CardActions';
import Collapse from '@mui/material/Collapse';
import IconButton, { IconButtonProps } from '@mui/material/IconButton';
import Typography from '@mui/material/Typography';
import FavoriteIcon from '@mui/icons-material/Favorite';
import ShareIcon from '@mui/icons-material/Share';
import ExpandMoreIcon from '@mui/icons-material/ExpandMore';

import { Diary } from '../diaryData';
import DiaryCardHeader from './DiaryHeader';

export type DiaryCardProps = {
  diary: Diary;
  onClickCardHeaderAction: (diaryId: string, mode: string) => void;
};

// アイコンクリックで詳細部分を表示するためのコンポーネントのprops型
interface ExpandMoreProps extends IconButtonProps {
  expand: boolean;
}

// アイコンクリックで詳細部分を表示するコンポーネント
const ExpandMore = styled((props: ExpandMoreProps) => {
  const { expand, ...other } = props;
  return <IconButton {...other} />;
})(({ theme, expand }) => ({
  transform: !expand ? 'rotate(0deg)' : 'rotate(180deg)',
  marginLeft: 'auto',
  transition: theme.transitions.create('transform', {
    duration: theme.transitions.duration.shortest,
  }),
}));


// DiaryCardコンポーネント
const DiaryCard = (props: DiaryCardProps) => {
  // 詳細表示コンポーネントの表示・非表示の状態
  const [expanded, setExpanded] = React.useState(false);
```

```
// 詳細コンポーネントの表示・非表示を切り替える関数
const handleExpandClick = () => {
  setExpanded(!expanded);
};

// 受け取ったオブジェクトを変数に展開
const { diary, onClickCardHeaderAction } = props;
const {
  diaryId,
  title,
  postDate,
  imageUrl,
  imageLabel,
  mainContent,
  readmore,
} = diary;

return (
  <Card sx={{ maxWidth: 345 }}>
    <DiaryCardHeader
      diaryId={diaryId}
      title={title}
      postDate={postDate}
      onClickCardHeaderAction={onClickCardHeaderAction}
    />
    <CardMedia
      component='img'
      height='194'
      image={imageUrl}
      alt={imageLabel}
    />
    <CardContent>
      <Typography variant='body2' color='text.secondary'>
        {mainContent}
      </Typography>
    </CardContent>
    <CardActions disableSpacing>
      <IconButton aria-label='add to favorites'>
        <FavoriteIcon />
      </IconButton>
      <IconButton aria-label='share'>
        <ShareIcon />
      </IconButton>
      <ExpandMore
        expand={expanded}
        onClick={handleExpandClick}
        aria-expanded={expanded}
        aria-label='show more'
      />
    </CardActions>
  </Card>
);
```

```
>
    <ExpandMoreIcon />
</ExpandMore>
</CardActions>
<Collapse in={expanded} timeout='auto' unmountOnExit>
    <CardContent>
        {readmore.map((parag, index) => (
            <Typography paragraph key={`${diaryId}${index.toString()}`}>
                {parag}
            </Typography>
        ))}
    </CardContent>
</Collapse>
</Card>
);
};

export default DiaryCard;
```

最後に親の DiaryBoard コンポーネントに、子・孫に送った関数を定義します。

▼ DiaryBoard.tsx

```
// 編集・削除ボタンクリック
const onClickCardHeaderAction = (diaryId: string, mode: string): void => {
    switch (mode) {
        case 'EDIT':
            setDataToForm(diaryId);
            break;
        case 'DELETE':
            deleteDiary(diaryId);
            break;
        default:
            break;
    }
};
```

削除は DiaryBoard コンポーネントが useState を使って全データを管理していますので、該当の diaryId を持つオブジェクトを削除するだけです。

編集は DiaryBoard コンポーネントに保存対象のデータを useState で定義し、全データから該当の diaryId を持つオブジェクトを targetDiaryData 変数に格納します。

新規追加は初期化データを targetDiaryData へ格納します。編集フォームへ targetDiaryData を渡して表示します。

3.4.2 保存・キャンセル関数を Props にデータと渡す

同じように、編集フォームの DiaryForm コンポーネントも、DiaryBoard コンポーネントからは孫にあたります。保存・キャンセルの関数をバケツリレーで渡します。

DiaryForm コンポーネントの Props に保存・キャンセルの関数を追加します。DiaryBoard コンポーネントからは保存・キャンセルのどちらもフォームを閉じるため「closeForm 関数」としました。

保存の場合には読書日記データ、キャンセルの場合は null を渡します。

▼ DiaryForm.tsx

```
interface DiaryFormProps {  
  diary: Diary;  
  closeForm: (diaryData: Diary | null) => void;  
}
```

入力フォームの DatePicker は Date オブジェクトを返しますので、Date オブジェクトから「YYYYMMDD」に変換する関数を「src/utilities/helper.ts」に作成してあるので DiaryForm コンポーネントにインポートします。

保存ボタンがクリックされた場合には、データ検証し問題があれば該当のテキストフィールドにフォーカスを当てます。そのため useRef を使用しています。

空関数としていた「saveData」、「cancelData」を実装します。

▼ DiaryForm.tsx

```
// 保存ボタン  
const saveData = () => {  
  const postDateValue: Date =  
    onEditPostDate !== null ? onEditPostDate : new Date();  
  
  if (onEditTitle.length < 5 && titleRef.current !== null) {  
    titleRef.current.focus();  
    return;  
  }  
  
  if (onEditMainContent.length === 0 && mainContentRef.current !== null) {  
    mainContentRef.current.focus();  
    return;  
  }  
  
  const diaryData = {
```

```
diaryId,
title: onEditTitle,
postDate: convertDateToString(postDateValue),
mainContent: onEditMainContent,
readmore: onEditReadMore.split('\n\n'),
imageUrl: onEditImageUrl,
imageLabel: onEditImageLabel,
};

closeForm(diaryData);
};

// キャンセルボタン
const cancelForm = () => {
  closeForm(null);
};
```

実装の完了した DiaryForm コンポーネントは、こちらになります。

▼ DiaryForm コンポーネント

```
import React, { useState, useRef } from 'react';
import Grid from '@mui/material/Grid';
import Paper from '@mui/material/Paper';
import TextField from '@mui/material/TextField';
import AdapterDateFns from '@mui/lab/AdapterDateFns';
import LocalizationProvider from '@mui/lab/LocalizationProvider';
import Datepicker from '@mui/lab/Datepicker';
import Button from '@mui/material/Button';
import CancelIcon from '@mui/icons-material/Cancel';
import DataSaverOnIcon from '@mui/icons-material/DataSaverOn';
import Stack from '@mui/material/Stack';

import { convertStringToDate, convertDateToString } from '../utilities/helper';
import { Diary } from '../diaryData';

interface DiaryFormProps {
  diary: Diary;
  closeForm: (diaryData: Diary | null) => void;
}

const DiaryForm = (props: DiaryFormProps) => {
  const { diary, closeForm } = props;
  const {
    diaryId,
    title,
    postDate,
    imageUrl,
    imageLabel,
    mainContent,
```

```
readmore,
} = diary;

// hooksでフォームデータ保持
// タイトル
const [onEditTitle, setOnEditTitle] = useState(title);
const [diaryTitleErr, setDiaryTitleErr] = useState(false);
const [diaryTitleErrMsg, setDiaryTitleErrMsg] = useState('');
// 投稿日
const [onEditPostDate, setOnEditPostDate] = useState<Date | null>(
  convertStringToDate(postDate)
);
// 画像URL
const [onEditImageUrl, setOnEditImageUrl] = useState(imageUrl);
// 画像ALT
const [onEditImageLabel, setOnEditImageLabel] = useState(imageLabel);
// 本文
const [onEditMainContent, setOnEditMainContent] = useState(mainContent);
const [mainContentErr, setMainContentErr] = useState(false);
const [mainContentErrMsg, setMainContentErrMsg] = useState('');
// 追記
const [onEditReadMore, setOnEditReadMore] = useState(readmore.join('\n\n'));

const titleRef = useRef<HTMLInputElement>(null);
const mainContentRef = useRef<HTMLInputElement>(null);

// 入力データ検証
const validateFieldData = (event: React.ChangeEvent<HTMLInputElement>) => {
  switch (event.target.name) {
    case 'diaryTitle':
      setOnEditTitle(event.target.value);
      if (event.target.value.length < 5) {
        setDiaryTitleErr(true);
        setDiaryTitleErrMsg('4文字以上入力してください。');
      } else {
        setDiaryTitleErr(false);
        setDiaryTitleErrMsg('');
      }
      break;
    case 'diaryMainContent':
      setOnEditMainContent(event.target.value);
      if (event.target.value.length < 5) {
        setMainContentErr(true);
        setMainContentErrMsg('4文字以上入力してください。');
      } else {
        setMainContentErr(false);
        setMainContentErrMsg('');
      }
      break;
  }
}
```

```
case 'diaryReadMore':
    setOnEditReadMore(event.target.value);
    break;
case 'diaryImageUrl':
    setOnEditImageUrl(event.target.value);
    break;
case 'diaryImageLabel':
    setOnEditImageLabel(event.target.value);
    break;
default:
    break;
}
};

// 保存ボタン
const saveData = () => {
    const postDateValue: Date =
        onEditPostDate !== null ? onEditPostDate : new Date();

    if (onEditTitle.length < 5 && titleRef.current !== null) {
        titleRef.current.focus();
        return;
    }

    if (onEditMainContent.length === 0 && mainContentRef.current !== null) {
        mainContentRef.current.focus();
        return;
    }

    const diaryData = {
        diaryId,
        title: onEditTitle,
        postDate: convertDateToString(postDateValue),
        mainContent: onEditMainContent,
        readmore: onEditReadMore.split('\n\n'),
        imageUrl: onEditImageUrl,
        imageLabel: onEditImageLabel,
    };
    closeForm(diaryData);
};

// キャンセルボタン
const cancelForm = () => {
    closeForm(null);
};

return (
    <Paper variant='outlined' sx={{ m: 1, py: 2 }}>
        <Grid container spacing={2} sx={{ pl: 5 }}>
```

```
<Grid item xs={10} sm={10} md={10} lg={10}>
  <TextField
    required
    id='diary-title'
    label='タイトル'
    error={diaryTitleErr}
    fullWidth
    name='diaryTitle'
    value={onEditTitle}
    helperText={diaryTitleErrMsg}
    onChange={validateFieldData}
    inputRef={titleRef}
  />
</Grid>
<Grid item xs={10} sm={8} md={6} lg={4}>
  /* eslint-disable-next-line @typescript-eslint/no-unsafe-assignment */
/>
<LocalizationProvider dateAdapter={AdapterDateFns}>
  <DatePicker
    label='日付'
    value={onEditPostDate}
    onChange={(newValue: Date | null) => {
      setOnEditPostDate(newValue);
    }}
    // eslint-disable-next-line react/jsx-props-no-spreading
    renderInput={(params) => <TextField {...params} />}
  />
</LocalizationProvider>
</Grid>
<Grid item xs={10} sm={10} md={10} lg={10}>
  <TextField
    required
    multiline
    id='diary-mainContent'
    label='本文'
    error={mainContentErr}
    fullWidth
    name='diaryMainContent'
    value={onEditMainContent}
    helperText={mainContentErrMsg}
    onChange={validateFieldData}
    inputRef={mainContentRef}
  />
</Grid>
<Grid item xs={10} sm={10} md={10} lg={10}>
  <TextField
    multiline
    minRows='4'
    maxRows='6'
```

```
        id='diary-readmore'
        label='追記'
        fullWidth
        name='diaryReadMore'
        value={onEditReadMore}
        onChange={validateFieldData}
        helperText='空行を入れると段落表示されます。'
    />
</Grid>
<Grid item xs={10} sm={10} md={10} lg={10}>
    <TextField
        id='diary-imageUrl'
        label='画像URL'
        fullWidth
        name='diaryImageUrl'
        onChange={validateFieldData}
        value={onEditImageUrl}
        helperText='画像のURL'
    />
</Grid>
<Grid item xs={10} sm={10} md={10} lg={10}>
    <TextField
        id='diary-imageLabel'
        label='画像の代替テキスト'
        fullWidth
        name='diaryImageLabel'
        onChange={validateFieldData}
        value={onEditImageLabel}
        helperText='画像が表示されない場合のテキスト'
    />
</Grid>
<Grid item xs={10} sm={10} md={10} lg={10}>
    <Stack direction='row' spacing={2}>
        <Button
            variant='contained'
            startIcon={<CancelIcon />}
            onClick={cancelForm}
        >
            キャンセル
        </Button>
        <Button
            variant='contained'
            endIcon={<DataSaverOnIcon />}
            onClick={saveData}
        >
            保存
        </Button>
    </Stack>
</Grid>
```

```
    </Grid>
  </Paper>
);
};

export default DiaryForm;
```

DiaryForm コンポーネントの Props は、DiaryFormDialog コンポーネントから受け取ります。DiaryFormDialog コンポーネントにも DiaryBoard コンポーネントから受け取る Props に関数を追加し DiaryForm コンポーネントへ送ります。

▼ DiaryFormDialog.tsx

```
import React from 'react';
import Dialog from '@mui/material/Dialog';
importDialogTitle from '@mui/material/DialogTitle';

import { Diary } from '../diaryData';
import DiaryForm from './DiaryForm';

export type DialogDiaryFormProps = {
  open: boolean;
  diary: Diary;
  closeForm: (diaryData: Diary | null) => void;
};

const DialogDiaryForm = (props: DialogDiaryFormProps) => {
  const { open, diary, closeForm } = props;

  return (
    <Dialog open={open}>
      <DialogTitle>日記編集</DialogTitle>
      <DiaryForm diary={diary} closeForm={closeForm} />
    </Dialog>
  );
};

export default DialogDiaryForm;
```

最後に DiaryBoard コンポーネントへ、保存・キャンセルの実装します。保存は、

- 新規は diaryId 空欄なので diaryId を作成
- 編集は diaryId が既存

となります。保存は、全データから保存対象の diaryId 以外の読書日記データ配列に追加するだけです。

diaryId を重複なく作成するためにユニークな ID を作成してくれるライブラリをインストールします。TypeScript 用の型情報もインストールします。

▼uuid のインストール

```
> npm install uuid
> npm install -D @types/uuid
```

▼DiaryBoard.tsx

```
import { v4 as uuidv4 } from 'uuid';

const saveDiary = (diary: Diary) => {
  const newDiaryData = diaryData.filter((d) => d.diaryId !== diary.diaryId);
  newDiaryData.push(diary);
  setDiaryData(newDiaryData);
};

const closeForm = (diary: Diary | null) => {
  // Dialogを閉じる
  setOpenDialog(false);
  if (diary !== null) {
    // 保存
    let newDiary: Diary;
    if (diary.diaryId === '') {
      const newDiaryId: string = uuidv4();
      newDiary = { ...diary, diaryId: newDiaryId };
    } else {
      newDiary = { ...diary };
    }
    saveDiary(newDiary);
  }
};
```

ここまで変更を動作確認します。

- 削除する
- 編集で各テキストフィールドの値を変える
- 新規データを追加する

不具合はみつかりましたか？

編集保存した場合、以前の日付で新規追加した場合とも保存したデータが最後尾に表示されます。編集後の全データをセットする前にソートするようにしましょう。

▼ DiaryBoard.tsx

```
// postDateを数値に変換して比較し並べ替え
const diarySort = (a: Diary, b: Diary) => {
  if (+a.postDate < +b.postDate) return -1;
  if (+a.postDate > +b.postDate) return 1;
  return 0;
};

const saveDiary = (diary: Diary) => {
  const newDiaryData = diaryData.filter((d) => d.diaryId !== diary.diaryId);
  newDiaryData.push(diary);
  newDiaryData.sort(diarySort);    ←ソートを追加
  setDiaryData(newDiaryData);
};
```

ソートを追加した DiaryBoard コンポーネントです。

▼ DiaryBoard.tsx

```
import React, { useState } from 'react';
import AppBar from '@mui/material/AppBar';
import Grid from '@mui/material/Grid';
import Box from '@mui/material/Box';
import Toolbar from '@mui/material/Toolbar';
import Typography from '@mui/material/Typography';
import Container from '@mui/material/Container';
import Link from '@mui/material/Link';
import IconButton from '@mui/material/IconButton';
import AddCircleOutlineIcon from '@mui/icons-material/AddCircleOutline';
import { v4 as uuidv4 } from 'uuid';

import { Diary } from '../diaryData';
import DiaryCard from './DiaryCard';
import DiaryFormDialog from './DiaryFormDialog';

export type DiaryBoardProps = {
  diaries: Diary[];
};

// diary初期化データ
const initialDiary: Diary = {
  diaryId: '',
  title: '',
  postDate: '',
  imageUrl: '',
  imageLabel: '',
  mainContent: '',
  readmore: [],
};
```

```
const Copyright = () => (
  <Typography variant='body2' color='text.secondary' align='center'>
    {'Copyright © '}
    <Link color='inherit' href='https://mui.com/'>
      やる夫が読書します。
    </Link>
    {` ${new Date().getFullYear()} `}
  </Typography>
);

const DiaryBoard = (props: DiaryBoardProps) => {
  const { diaries } = props;

  // diary data
  const [diaryData, setDiaryData] = useState(diaries);
  // DiaryFormDialogの開閉状態
  const [openDialog, setOpenDialog] = useState(false);
  // 編集対象の読書日記データ
  const [targetDiaryData, setTargetDiaryData] = useState(initialDiary);

  // postDateを数値に変換して比較し並べ替え
  const diarySort = (a: Diary, b: Diary) => {
    if (+a.postDate < +b.postDate) return -1;
    if (+a.postDate > +b.postDate) return 1;
    return 0;
  };

  // Formにデータを渡して表示する
  const setDataToForm = (diaryId: string) => {
    if (diaryId === '') {
      setTargetDiaryData(initialDiary);
      setOpenDialog(true);
    } else {
      const editDiary = diaryData.filter((d) => d.diaryId === diaryId);
      if (editDiary.length === 1) {
        setTargetDiaryData(editDiary[0]);
        setOpenDialog(true);
      }
    }
  };

  // 編集対象データでDiaryFormDialogを開く
  const openForm = () => setDataToForm('');

  // 日記データの削除
  const deleteDiary = (diaryId: string) => {
    console.log(`delete:${diaryId}`);
    const newDiaryData = diaryData.filter((diary) => diary.diaryId !== diaryId);
  };
}
```

```
        setDiaryData(newDiaryData);
    };

    // 編集・削除ボタンクリック
    const onClickCardHeaderAction = (diaryId: string, mode: string): void => {
        switch (mode) {
            case 'EDIT':
                setDataToForm(diaryId);
                break;
            case 'DELETE':
                deleteDiary(diaryId);
                break;
            default:
                break;
        }
    };

    // データを保存
    const saveDiary = (diary: Diary) => {
        const newDiaryData = diaryData.filter((d) => d.diaryId !== diary.diaryId);
        newDiaryData.push(diary);
        newDiaryData.sort(diarySort);
        setDiaryData(newDiaryData);
    };

    // 編集フォームから呼ばれる
    const closeForm = (diary: Diary | null) => {
        // Dialogを閉じる
        setOpenDialog(false);
        if (diary !== null) {
            // 保存
            let newDiary: Diary;
            if (diary.diaryId === '') {
                const newDiaryId: string = uuidv4();
                newDiary = { ...diary, diaryId: newDiaryId };
            } else {
                newDiary = { ...diary };
            }
            saveDiary(newDiary);
        }
    };
}

return (
    <>
    <AppBar position='relative'>
        <Toolbar>
            <Box sx={{ flexGrow: 1 }}>
                <Typography variant='h6' color='inherit' noWrap>
                    やる夫の読書日記

```

```
</Typography>
</Box>
<Box>
  <IconButton
    size='large'
    edge='end'
    aria-label='新規読書日記'
    onClick={openForm}
    color='inherit'
  >
    <AddCircleOutlineIcon />
  </IconButton>
</Box>
</Toolbar>
</AppBar>
<main>
  <Container sx={{ py: 8 }} maxWidth='md'>
    <Grid container spacing={4}>
      {diaryData.map((diary) => (
        <Grid item key={diary.diaryId} xs={12} sm={6} md={4}>
          <DiaryCard
            diary={diary}
            onClickCardHeaderAction={onClickCardHeaderAction}
          />
        </Grid>
      )))
    </Grid>
  </Container>
</main>
{/* Footer */}
<Box sx={{ bgcolor: 'background.paper', p: 6 }} component='footer'>
  <Typography variant='h6' align='center' gutterBottom>
    やる夫の読書日記
  </Typography>
  <Typography
    variant='subtitle1'
    align='center'
    color='text.secondary'
    component='p'
  >
    お前らも本読んだ良いお～！
  </Typography>
  <Copyright />
</Box>
{/* End footer */}
<DiaryFormDialog
  open={openDialog}
  diary={targetDiaryData}
  closeForm={closeForm}
```

```
    />
  </>
);
};

export default DiaryBoard;
```

以上でサンプルアプリケーションは完成です。見栄えを少し修正したものを GitHub へアップロードしてあります。

ここまで的内容は、GitHub 上で、以下のコマンドでクローンできます。

▼ GitHub

```
$ > git clone -b 06_implements_data_save https://github.com/yaruo-react-redux/yaruo-cra-template.git
```

3.5 第3章のまとめ

- React のスタートアッププロジェクトがあれば簡単に始めることができる。
- MUI を使えばサンプルのアレンジで画面が作成できる。
- コンポーネント間のバケツリレーはたいへん。

第 4 章

日記アプリケーションの作成 (Redux 使用)

本章では、React で作成したサンプルアプリケーションが管理しているデータを Redux で管理するように変更します。

まずは、Redux とは何か？ から始め、Redux で使われる定型の書き方を学びます。

4.1 Redux とは？

Redux とは状態管理をするためのライブラリのひとつです。状態管理とは、アプリケーションで使用するデータの状態（初期値への追加・変更・削除）を正しく保持することです。

React を使用したアプリケーションで状態管理をする方法は複数ありますが、2022 年時点ではデフィアクト・スタンダードと言われるほど使われています。

React での状態管理の方法は、こちら^{*1} の記事が参考になります。



▲ 図 4.1: React ステート管理比較

^{*1} <https://blog.uhy.ooo/entry/2021-07-24/react-state-management/>

♥| 4.1.1 Redux 導入のメリット

Redux を導入するメリットは何があるのでしょうか？

データ管理の一元化

一番のメリットは、データ管理の一元化ではないでしょうか？

リレーションナル・データベースで言えば、テーブルデータの集合体の State があり画面に必要なデータは、親コンポーネントに頼らず「useSelect」を使えば簡単に取得できます。

また、データの追加・更新・削除も親コンポーネントに頼らず「ActionCreator」にデータを渡し作成した「Action」を「dispatch 関数」に渡すだけで完了します。

バケツリレーからの脱却

サンプルアプリケーションでの「React あるあるのバケツリレー」もコンポーネントから自由にアクセスできるのでデータの取得・変更に関しては脱却できます。

メンテナンス性の向上

データを変更する箇所が「Reducer」に集約されるため、メンテナンス性が高くなります。また、middleware を組み込むことによりデバッグも簡単になります。

devTools-extention を導入すると、発行された Action をさかのぼってデータの変化を確認できます。

4.2 Redux の動作イメージ

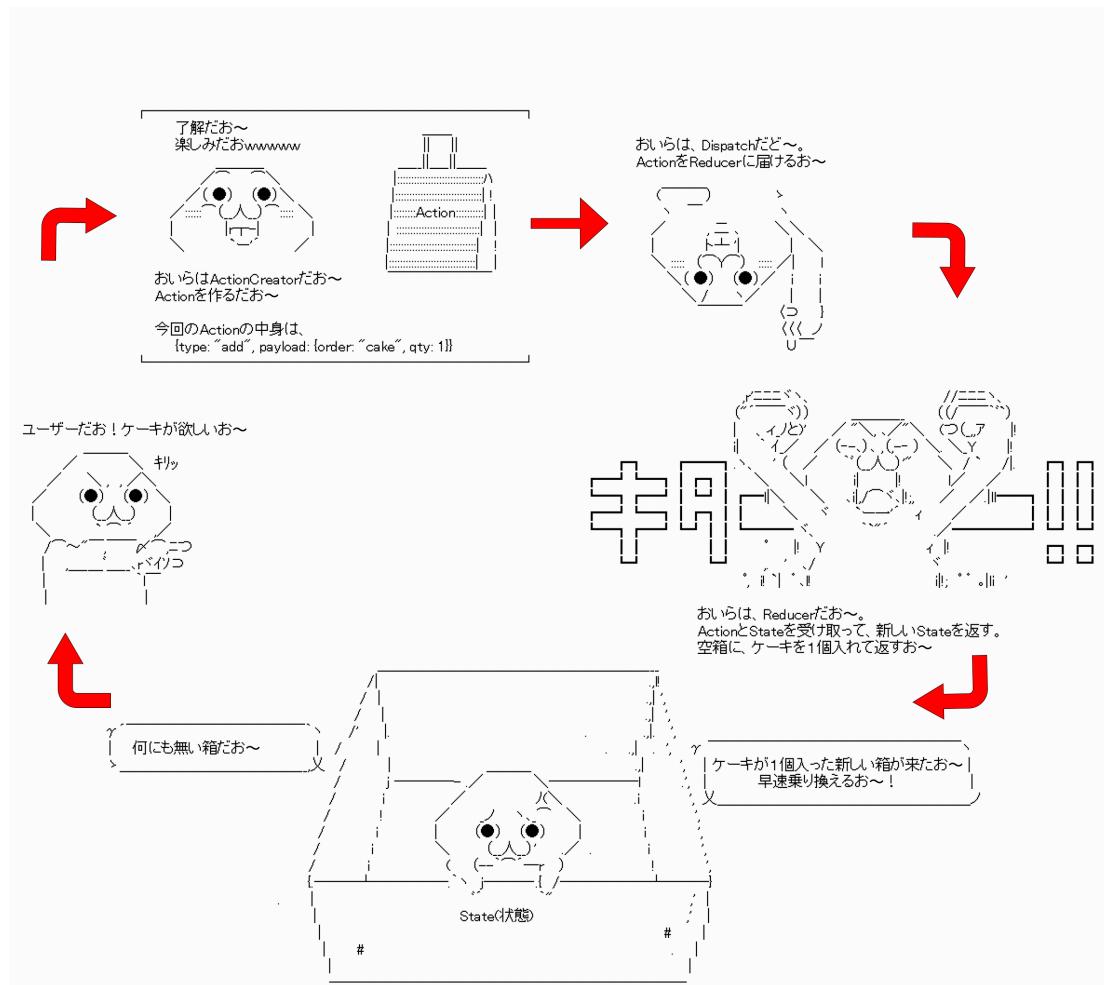
Redux の動作としては、

1. アプリケーション起動
2. 初期値を使用して最初の各 State が作成され Store を作る
3. 表示画面のコンポーネントは Store の一部 State を props として参照
4. イベントが発生し ActionCreator へ Action オブジェクトを作成してもらう。
5. 作成した Action を Dispatch 関数で Reducer へ渡す
6. Reducer は、現在の State をもとに新しく State を作成し、Action オブジェクトにより State を変更
7. props が参照していた State が変化したため再描画が発生する。

となります。

例として、Cafe の各テーブルの状態を考えます。

1. お店オープン。最初はお客様がいなためすべてのテーブル上は何もなし。
 2. あるテーブルへお客様が入り、ケーキをオーダー。
 3. actioncreator であるウェイトレスが注文伝票を作成
 4. 注文伝票 (Action) が店長により厨房へ運ばれる (dispatch)
 5. 調理担当が注文したお客様の現在のテーブル状態を別なテーブルで作成。
 6. 調理担当が作成したテーブルに注文の品をのせ、現在のお客のテーブルと入れ換える
- こんなイメージです。



▲図 4.2: Redux の動作イメージ

4.3 Redux の導入

それでは、サンプルアプリケーションに Redux を導入します。現在は Redux-toolkit があるのでですが Redux の動きを知るために、素の Redux を導入します。

Redux に必要なパッケージは、Redux 関連と Middleware とデバッグ用です。

- redux
- react-redux
- @redux-devtools/extension
- redux-logger

TypeScript を導入しているので型定義も導入します。これらは「devDependencies」にインストールしますので分けて導入します。

- @types/react-redex
- @types/redux-logger

となります。

インストールは、個別に、

▼ redux のインストール

```
> npm install redux react-redux @redux-devtools/extension redux-logger
```

▼ 型パッケージのインストール

```
> npm install -D @types/react-redux @types/redux-logger
```

4.4

Redux のアプリケーションへの導入

Redux のインストールが完了しましたので、サンプルアプリケーションへ導入していきます。

手順は、

1. Action Type を定数として定義
2. Actioncreator の作成
3. Reducer の作成
4. Reducer を組み合わせる
5. Store の作成
6. トップコンポーネントに Store を登録

の手順となります。

サンプルアプリケーションでは、管理するデータは、

- 読書日記データ
- 編集対象・新規追加の diaryId

の2つだけで Action も少ないのですが、管理するデータカテゴリ (RDB でのテーブル) が増えると同じようなファイルをたくさん作成します。

この苦行のおかげで Redux の導入はメンドウだと言われましたが、Redux-toolkit の登場で楽になりました。

このサンプルアプリケーションでは、Redux 関連のファイルをひとつにします。のちほど Redux-toolkit も導入し切り替えますので、ファイルサイズの違いを楽しみに。

それでは、「src/redux/redux-og.ts」ファイルを作成します。

Action Type の定数を作成

最初に、Action Type の定数を作成します。

Action に直接文字列を書き込んでも良いのですが、定数として定義すると、

- エディタで保管が効く

- タイポしてもエディタが指摘してくれる
ので、手間以上のメリットがあります。

▼ アクション定数

```
// Action定数
const CREATE_DIARY = 'CREATE_DIARY';
const EDIT_DIARY = 'EDIT_DIARY';
const DELETE_DIARY = 'DELETE_DIARY';
const SELECT_DIARY = 'SELECT_DIARY';
```

以上で完成です。

ActionCreator の作成

次に ActionCreator を作成します。ActionCreator は、受け取った引数を設定した Action を返すだけのものです。

▼ ActionCreator の作成

```
// Actions & Action Type
type CreateDiaryActionType = {
  type: typeof CREATE_DIARY;
  payload: Diary;
};

export const createDiaryActionCreator = (
  data: Diary
): CreateDiaryActionType => ({
  type: CREATE_DIARY,
  payload: data,
});

type EditDiaryActionType = {
  type: typeof EDIT_DIARY;
  payload: Diary;
};

export const editDiaryActionCreator = (data: Diary): EditDiaryActionType => ({
  type: EDIT_DIARY,
  payload: data,
});

type DeleteDiaryActionType = {
  type: typeof DELETE_DIARY;
  payload: { diaryId: string };
};

export const deleteDiaryActionCreator = ({
```

```
diaryId,
}): {
  diaryId: string;
}): DeleteDiaryActionType => ({
  type: DELETE_DIARY,
  payload: { diaryId },
});

type SelectDiaryActionType = {
  type: typeof SELECT_DIARY;
  payload: { diaryId: string };
};

export const selectDiaryActionCreator = ({
  diaryId,
}): {
  diaryId: string;
}): SelectDiaryActionType => ({
  type: SELECT_DIARY,
  payload: { diaryId },
});
```

Reducer の作成

次に Reducer を作成します。Redux の一番肝心な部分です。

Reducer は、Action と State を受け取って新しい State を作成し変更を加えます。元の State は変更しないため「不变 (英語では、immutable)」と言われます。

Reducer は自分の管理する State に対する Action をすべて受け付け、Action の type で switch 文で処理を分けます。

また、Reducer の受け取る State に初期値を設定することで初期化できます。

▼ Reducer の作成

```
import diaries, { Diary } from '../diaryData';

// Reducers
type DiaryActionTypes =
  | CreateDiaryActionType
  | EditDiaryActionType
  | DeleteDiaryActionType;

const diariesInitialState = diaries;
```

```
const diarySort = (a: Diary, b: Diary) => {
  if (+a.postDate < +b.postDate) return -1;
  if (+a.postDate > +b.postDate) return 1;
  return 0;
};

const diariesReducer = (
  // eslint-disable-next-line default-param-last
  state: Diary[] = diariesInitialState,
  action: DiaryActionTypes
) => {
  switch (action.type) {
    case CREATE_DIARY: {
      const { payload } = action;
      return [...state, payload].sort(diarySort);
    }
    case EDIT_DIARY: {
      const { payload } = action;
      return state
        .map((diary) => (diary.diaryId === payload.diaryId ? payload : diary))
        .sort(diarySort);
    }
    case DELETE_DIARY: {
      const { payload } = action;
      return state
        .filter((diary) => diary.diaryId !== payload.diaryId)
        .sort(diarySort);
    }
    default:
      return state;
  }
};

type SelectedDiaryActionTypes = SelectDiaryActionType;
const selectedDiaryReducer = (
  // eslint-disable-next-line default-param-last
  state: string | null = null,
  action: SelectedDiaryActionTypes
) => {
  switch (action.type) {
    case SELECT_DIARY: {
      const { payload } = action;
      const selectedDiaryId =
        payload.diaryId.length > 0 ? payload.diaryId : null;
      return selectedDiaryId;
    }
    default: {
      return state;
    }
  }
};
```

```
    }  
};
```

Reducer を組み合わせる

アプリケーション内のすべての Reducer を「combineReducers 関数」でまとめます。「combineReducers 関数」は redux パッケージからインポートします。

▼ combineReducers

```
const reducers = combineReducers({  
  diaries: diariesReducer,  
  targetDiaryId: selectedDiaryReducer,  
});
```

ここで、「key: value」形式で Reducer を登録しますが、Reducer は先ほど作成したもので key は、State 内の参照名になります。コンポーネントでデータを参照する際には、

▼ State データの参照方法

```
const 参照したいデータ = useSelector((state => state.key名));
```

で行いますので、分かり key 名にします。

Store の作成

最後にデータの集合体、Middleware の集合体の Store を作成します。

Store を作成するための「createStore 関数」も redux パッケージからインポートします。

Middleware を登録するのは、redux パッケージの「compose 関数」でも良いのですが、今回登録するのは「logger」で開発時のみの使用を想定していますので、@redux-devTools/extension の「composeWithDevTools 関数」を使用します。

ただ、Middleware を登録する「applyMiddleware 関数」は redux パッケージからインポートします。「logger」のインポートも忘れないように。

全体に関わるものですが、Store の型をここへ書いておきます。

▼ Store の作成

```
import { combineReducers, createStore, applyMiddleware } from 'redux';  
import { composeWithDevTools } from '@redux-devtools/extension';
```

```
import logger from 'redux-logger';

export default createStore(
  reducers,
  composeWithDevTools(applyMiddleware(logger))
);

export type State = {
  diaries: Diary[];
  targetDiaryId: string;
}
```

トップコンポーネントに Store を登録

トップコンポーネントに Store を登録します。「src/index.ts」を編集します。「Provider 関数」を react-redux パッケージからインストールします。

また、作成した Store もインポートします。

▼ src/index.ts

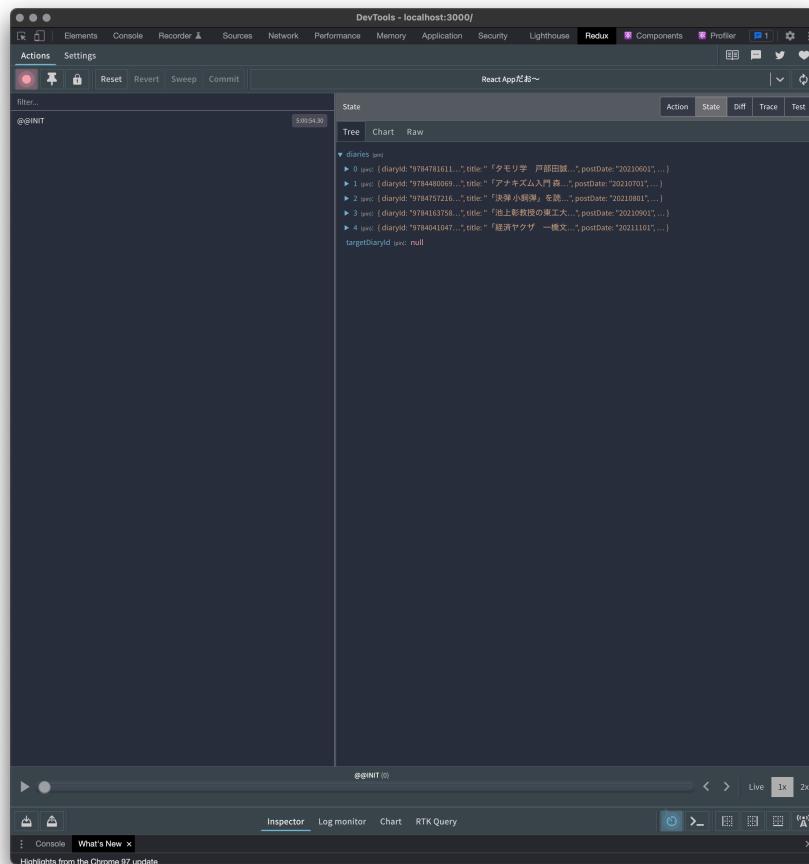
```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import App from './components/App';

import store from './redux/redux-og';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

以上が完了しましたら動作確認します。

ブラウザが起動したら devTools を確認します。Redux タブを開くと State の内容を確認できます。



▲図 4.3: State の確認

4.5 コンポーネントの修正

State が無事に作成できましたので、各コンポーネントでデータの参照は State から、データの変更は ActionCreator に引数を渡し dispatch します。

サンプルアプリケーションでの変更点は、

1. DiaryBoard コンポーネントで State から読書日記データを取得する。
2. DiaryCardHeader コンポーネントで直接データを削除する。

の2点が主ですが、それに伴う細かなコードの変更があります。

DiaryBoard コンポーネントでデータ取得

App コンポーネントで取得した「読書日記データ diaries」を DiaryBoard コンポーネントにて State から取得します。

▼ App コンポーネント

```
import React from 'react';
import CssBaseline from '@mui/material/CssBaseline';
import { createTheme, ThemeProvider } from '@mui/material/styles';

import DiaryBoard from './DiaryBoard';

const theme = createTheme();

const App = () => (
  <ThemeProvider theme={theme}>
    <CssBaseline />
    <DiaryBoard />
  </ThemeProvider>
);

export default App;
```

DiaryBoard コンポーネントで読書日記データを取得しますので、props が不要になりました。useSelector でデータを取得します。今までのコードはコメントアウトしました。

▼ DiaryBoard コンポーネントでデータ取得

```
const DiaryBoard = () => {
  // const { diaries } = props;
  const diaryData = useSelector((state: State) => state.diaries);
```

ここで動作確認します。

DiaryCardHeader コンポーネントで削除

孫コンポーネントの DiaryCardHeader コンポーネントが、親コンポーネント (DiaryCard) を経由してじじコンポーネント (DiaryBoard) へ「削除の要求と diaryId」を伝えていました。

Redux を導入したので、孫から直接データを削除します。

▼ DiaryCardHeader コンポーネントでデータ削除

```
import { useDispatch } from 'react-redux';

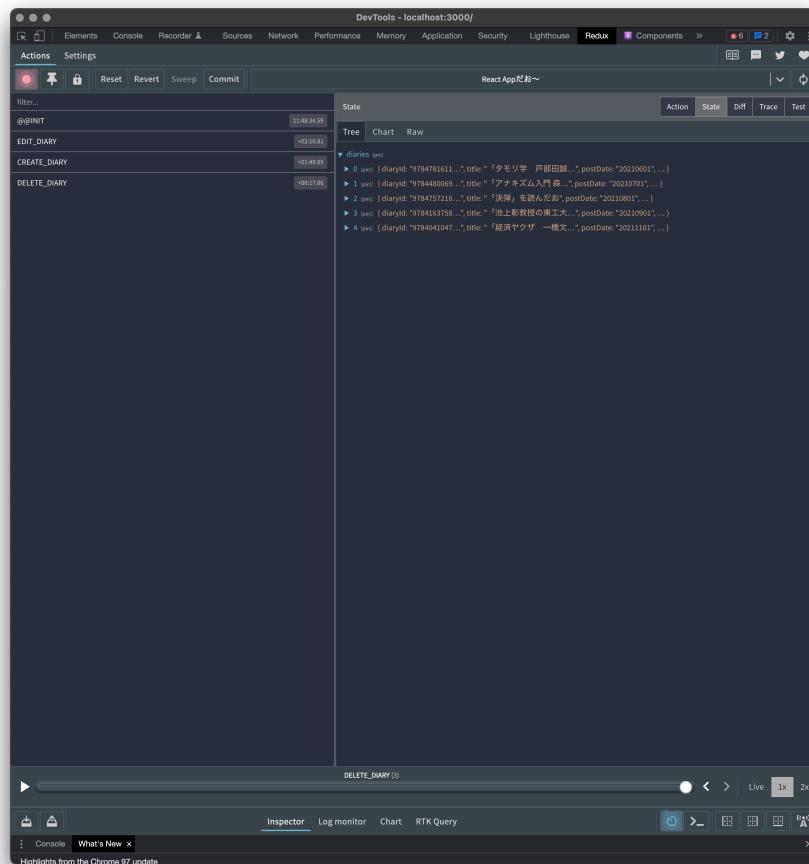
import { deleteDiaryActionCreator } from '../redux/redux-og';

const DiaryCardHeader = (props: DiaryCardHeaderProps) => {
  const dispatch = useDispatch();

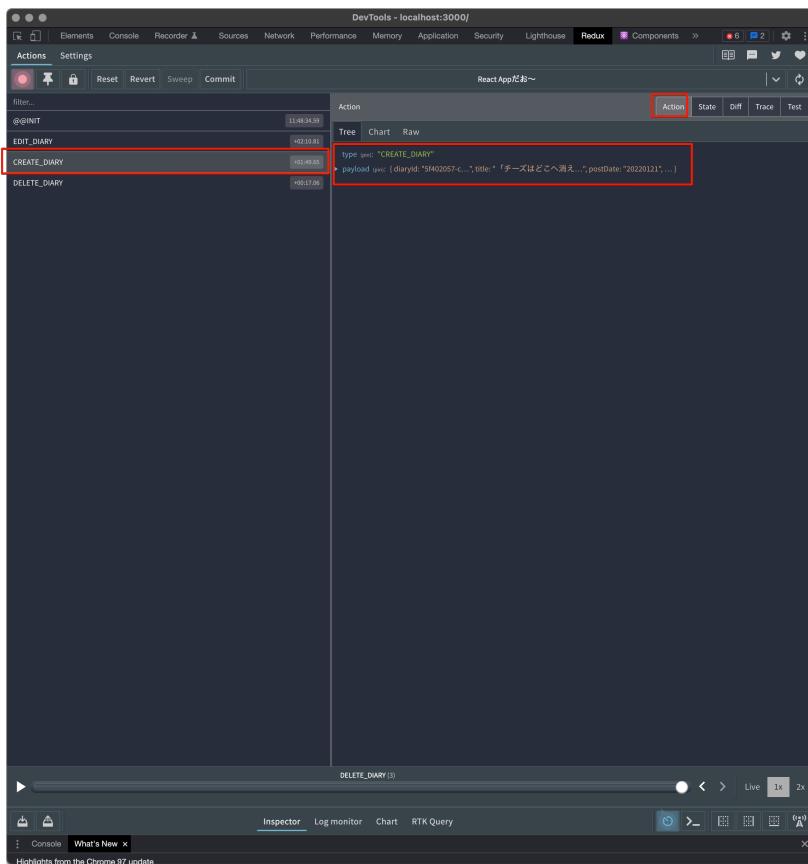
  // ダイアログの削除ボタンクリック
  const handleConfirmDelete = () => {
    setOpenDialog(false);
    // return onClickCardHeaderAction(diaryId, 'DELETE');
    dispatch(deleteDiaryActionCreator({ diaryId }));
  };
}
```

devTools での表示

Google chrome の devTools を使えば、ディスパッチされた Action の内容を確認できますし、変更された State の内容も変更前・変更後と確認できます。



▲ 図 4.4: 実行された Action の type



▲図 4.5: 実行された Action の内容

Redux 導入でコード修正したファイル

Redux の導入で修正したコードは、

- App コンポーネント 読書日記データのバケツリレーをやめた
- DiaryBoard コンポーネント State からデータ取得、削除を DiaryCardHeader コンポーネントへ委譲
- DiaryCard コンポーネント 親コンポーネントから子コンポーネントへ渡す関数の引数変更
- DiaryCardHeader コンポーネント 削除を自身で行う 編集要求の引数変更

です。

変更後のコードは、App コンポーネントは上記です。ほかのコンポーネントは、こちらになります。

▼ DiaryBoard コンポーネント

```
import React, { useState } from 'react';
import { useDispatch, useSelector } from 'react-redux';
import AppBar from '@mui/material/AppBar';
import Grid from '@mui/material/Grid';
import Box from '@mui/material/Box';
import Toolbar from '@mui/material/Toolbar';
import Typography from '@mui/material/Typography';
import Container from '@mui/material/Container';
import Link from '@mui/material/Link';
import IconButton from '@mui/material/IconButton';
import AddCircleOutlineIcon from '@mui/icons-material/AddCircleOutline';
import { v4 as uuidv4 } from 'uuid';

import {
  createDiaryActionCreator,
  editDiaryActionCreator,
  State,
} from '../redux/redux-og';

import { Diary } from '../diaryData';
import DiaryCard from './DiaryCard';
import DiaryFormDialog from './DiaryFormDialog';

// diary初期化データ
const initialDiary: Diary = {
  diaryId: '',
  title: '',
  postDate: '',
  imageUrl: '',
  imageLabel: '',
  mainContent: '',
  readmore: [],
};

const Copyright = () => (
  <Typography variant='body2' color='text.secondary' align='center'>
    {'Copyright © '}
    <Link color='inherit' href='https://mui.com/'>
      やる夫が読書します。
    </Link>
    {` ${new Date().getFullYear()}.`}
  </Typography>
);

const DiaryBoard = () => {
  // ActionCreatorから返ってきたActionをReducerへ送る
  const dispatch = useDispatch();
  // 読書日記データ
```

```
const diaryData = useSelector((state: State) => state.diaries);
// DiaryFormDialogの開閉状態
const [openDialog, setOpenDialog] = useState(false);
// 編集対象の読書日記データ
const [targetDiaryData, setTargetDiaryData] = useState(initialDiary);

// Formにデータを渡して表示する
const setDataToForm = (diaryId: string) => {
  if (diaryId.length === 0) {
    setTargetDiaryData(initialDiary);
    setOpenDialog(true);
  } else {
    const editDiary = diaryData.filter((d) => d.diaryId === diaryId);
    if (editDiary.length === 1) {
      setTargetDiaryData(editDiary[0]);
      setOpenDialog(true);
    }
  }
};

// 編集対象データでDiaryFormDialogを開く
const openForm = () => {
  setDataToForm('');
};

// 編集・削除ボタンクリック
const onClickCardHeaderAction = (diaryId: string): void => {
  setDataToForm(diaryId);
};

// 編集フォームから呼ばれる
const closeForm = (diary: Diary | null) => {
  // Dialogを閉じる
  setOpenDialog(false);
  if (diary !== null) {
    // 保存
    let newDiary: Diary;
    if (diary.diaryId === '') {
      const newDiaryId: string = uuidv4();
      newDiary = { ...diary, diaryId: newDiaryId };
      dispatch(createDiaryActionCreator(newDiary));
    } else {
      newDiary = { ...diary };
      dispatch(editDiaryActionCreator(newDiary));
    }
  }
};

return (

```

```
<>
<AppBar position='relative'>
  <Toolbar>
    <Box sx={{ flexGrow: 1 }}>
      <Typography variant='h6' color='inherit' noWrap>
        やる夫の読書日記
      </Typography>
    </Box>
    <Box>
      <IconButton
        size='large'
        edge='end'
        aria-label='新規読書日記'
        onClick={openForm}
        color='inherit'
      >
        <AddCircleOutlineIcon />
      </IconButton>
    </Box>
  </Toolbar>
</AppBar>
<main>
  <Container sx={{ py: 8 }} maxWidth='md'>
    <Grid container spacing={4}>
      {diaryData.map((diary) => (
        <Grid item key={diary.diaryId} xs={12} sm={6} md={4}>
          <DiaryCard
            diary={diary}
            onClickCardHeaderAction={onClickCardHeaderAction}
          />
        </Grid>
      )))
    </Grid>
  </Container>
</main>
{/* Footer */}
<Box sx={{ bgcolor: 'background.paper', p: 6 }} component='footer'>
  <Typography variant='h6' align='center' gutterBottom>
    やる夫の読書日記
  </Typography>
  <Typography
    variant='subtitle1'
    align='center'
    color='text.secondary'
    component='p'
  >
    お前らも本読んだ良いお～！
  </Typography>
  <Copyright />
```

```
</Box>
 {/* End footer */}
<DiaryFormDialog
  open={openDialog}
  diary={targetDiaryData}
  closeForm={closeForm}
/>
</>
);
};

export default DiaryBoard;
```

DiaryCard コンポーネントは、Props は、親コンポーネントからコンポーネントへ渡すメニューがクリックされたときの関数の引数が変わっているだけです。

▼ DiaryCard コンポーネント

```
import React from 'react';
import { styled } from '@mui/material/styles';
import Card from '@mui/material/Card';
import CardMedia from '@mui/material/CardMedia';
import CardContent from '@mui/material/CardContent';
import CardActions from '@mui/material/CardActions';
import Collapse from '@mui/material/Collapse';
import IconButton, { IconButtonProps } from '@mui/material/IconButton';
import Typography from '@mui/material/Typography';
import FavoriteIcon from '@mui/icons-material/Favorite';
import ShareIcon from '@mui/icons-material/Share';
import ExpandMoreIcon from '@mui/icons-material/ExpandMore';
import Box from '@mui/material/Box';

import { Diary } from '../diaryData';
import DiaryCardHeader from './DiaryHeader';

export type DiaryCardProps = {
  diary: Diary;
  onClickCardHeaderAction: (diaryId: string) => void;
};

// アイコンクリックで詳細部分を表示するためのコンポーネントのprops型
interface ExpandMoreProps extends IconButtonProps {
  expand: boolean;
}

// アイコンクリックで詳細部分を表示するコンポーネント
const ExpandMore = styled((props: ExpandMoreProps) => {
  // eslint-disable-next-line @typescript-eslint/no-unused-vars
```

```
const { expand, ...other } = props;
// eslint-disable-next-line react/jsx-props-no-spreading
return <IconButton {...other} />;
})(({ theme, expand }) => ({
  transform: !expand ? 'rotate(0deg)' : 'rotate(180deg)',
  marginLeft: 'auto',
  transition: theme.transitions.create('transform', {
    duration: theme.transitions.duration.shortest,
  }),
}));
}

// DiaryCardコンポーネント
const DiaryCard = (props: DiaryCardProps) => {
  // 詳細表示コンポーネントの表示・非表示の状態
  const [expanded, setExpanded] = React.useState(false);

  // 詳細コンポーネントの表示・非表示を切り替える関数
  const handleExpandClick = () => {
    setExpanded(!expanded);
  };

  // 受け取ったオブジェクトを変数に展開
  const { diary, onClickCardHeaderAction } = props;
  const {
    diaryId,
    title,
    postDate,
    imageUrl,
    imageLabel,
    mainContent,
    readmore,
  } = diary;

  return (
    <Card sx={{ maxWidth: 345 }}>
      <DiaryCardHeader
        diaryId={diaryId}
        title={title}
        postDate={postDate}
        onClickCardHeaderAction={onClickCardHeaderAction}>
      />
      <Box
        sx={{
          display: 'flex',
          alignItems: 'center',
          justifyContent: 'center',
        }}>
        >
        <CardMedia
```

```
sx={{ width: '100px' }}
component='img'
image={imageUrl}
alt={imageLabel}
/>
</Box>
<CardContent>
  <Typography variant='body2' color='text.secondary'>
    {mainContent}
  </Typography>
</CardContent>
<CardActions disableSpacing>
  <IconButton aria-label='add to favorites'>
    <FavoriteIcon />
  </IconButton>
  <IconButton aria-label='share'>
    <ShareIcon />
  </IconButton>
  <ExpandMore
    expand={expanded}
    onClick={handleExpandClick}
    aria-expanded={expanded}
    aria-label='show more'
  >
    <ExpandMoreIcon />
  </ExpandMore>
</CardActions>
<Collapse in={expanded} timeout='auto' unmountOnExit>
  <CardContent>
    {readmore.map((parag, index) => (
      <Typography paragraph key={`${diaryId}${index.toString()}`}>
        {parag}
      </Typography>
    ))}
  </CardContent>
</Collapse>
</Card>
);
};

export default DiaryCard;
```

DiaryCardHeader コンポーネントは、編集メニューがクリックされたときの引数の変更と削除をコンポーネント内で行うように変更しています。

▼ DiaryCardHeader コンポーネント

```
/* eslint-disable @typescript-eslint/no-unsafe-assignment */
import React from 'react';
import { useDispatch } from 'react-redux';
import CardHeader from '@mui/material/CardHeader';
import Avatar from '@mui/material/Avatar';
import Menu from '@mui/material/Menu';
import MenuItem from '@mui/material/MenuItem';
import ListItemIcon from '@mui/material/ListItemIcon';
import MoreVertIcon from '@mui/icons-material/MoreVert';
import EditIcon from '@mui/icons-material/Edit';
import DeleteForeverIcon from '@mui/icons-material/DeleteForever';
import IconButton from '@mui/material/IconButton';
import Button from '@mui/material/Button';
import Dialog from '@mui/material/Dialog';
import DialogActions from '@mui/material/DialogActions';
import DialogContent from '@mui/material/DialogContent';
import DialogContentText from '@mui/material/DialogContentText';
importDialogTitle from '@mui/material/DialogTitle';

import { deleteDiaryActionCreator } from '../redux/redux-og';

import January from '../assets/images/month-icons/january.svg';
import February from '../assets/images/month-icons/february.svg';
import March from '../assets/images/month-icons/march.svg';
import April from '../assets/images/month-icons/april.svg';
import May from '../assets/images/month-icons/may.svg';
import Jun from '../assets/images/month-icons/june.svg';
import July from '../assets/images/month-icons/july.svg';
import August from '../assets/images/month-icons/august.svg';
import September from '../assets/images/month-icons/september.svg';
import October from '../assets/images/month-icons/october.svg';
import November from '../assets/images/month-icons/november.svg';
import December from '../assets/images/month-icons/december.svg';

import {
  convertToLongDateString,
  convertStringToDate,
} from '../utilities/helper';

export type DiaryCardHeaderProps = {
  diaryId: string;
  title: string;
  postDate: string;
  onClickCardHeaderAction: (diaryId: string) => void;
};

const DiaryCardHeader = (props: DiaryCardHeaderProps) => {
  const dispatch = useDispatch();
```

```
const { onClickCardHeaderAction, diaryId, title, postDate } = props;

// menuの開閉状態を管理
const [anchorEl, setAnchorEl] = React.useState<null | HTMLElement>(null);
// ダイアログの表示・非表示
const [openDialog, setOpenDialog] = React.useState(false);

// menuの開閉
const openMenu = Boolean(anchorEl);
// MoreVerIconクリック
const handleClick = (event: React.MouseEvent<HTMLElement>) => {
  setAnchorEl(event.currentTarget);
};

// menuを閉じる
const handleCloseMenu = () => {
  setAnchorEl(null);
};

// ダイアログ表示
const handleClickOpenDialog = () => {
  handleCloseMenu();
  setOpenDialog(true);
};

// 編集メニュークリック
const handleEdit = () => {
  handleCloseMenu();
  return onClickCardHeaderAction(diaryId);
};

// 削除メニュークリック
const handleDelete = () => {
  handleClickOpenDialog();
};

// ダイアログ非表示
const handleCloseDialog = () => {
  setOpenDialog(false);
};

// ダイアログの削除ボタンクリック
const handleConfirmDelete = () => {
  setOpenDialog(false);
  // return onClickCardHeaderAction(diaryId, 'DELETE');
  dispatch(deleteDiaryActionCreator({ diaryId }));
};

// postDate:YYYYMMDDから月を取得し
```

```
const datePosted: Date = convertStringToDate(postDate);
let avatarSrc;
// useEffect(() => {
switch (datePosted.getMonth()) {
  case 0:
    avatarSrc = January;
    break;
  case 1:
    avatarSrc = February;
    break;
  case 2:
    avatarSrc = March;
    break;
  case 3:
    avatarSrc = April;
    break;
  case 4:
    avatarSrc = May;
    break;
  case 5:
    avatarSrc = Jun;
    break;
  case 6:
    avatarSrc = July;
    break;
  case 7:
    avatarSrc = August;
    break;
  case 8:
    avatarSrc = September;
    break;
  case 9:
    avatarSrc = October;
    break;
  case 10:
    avatarSrc = November;
    break;
  case 11:
    avatarSrc = December;
    break;
  default:
    break;
}
}

const posted = convertToLongDateString(postDate);

return (
  <>
  <CardHeader
```

```
avatar={
  <Avatar
    sx={{ width: 58, height: 58 }}
    variant='square'
    aria-label={`投稿日:${posted}`}
    src={avatarSrc}
  />
}
action={
  <IconButton aria-label='settings' onClick={handleClick}>
    <MoreVertIcon />
  </IconButton>
}
title={title}
subheader={posted}
/>
<Menu
  anchorEl={anchorEl}
  id='account-menu'
  open={openMenu}
  onClose={handleCloseMenu}
  onClick={handleCloseMenu}
  PaperProps={{
    elevation: 0,
    sx: {
      overflow: 'visible',
      filter: 'drop-shadow(0px 2px 8px rgba(0,0,0,0.32))',
      mt: 1.5,
      '& .MuiAvatar-root': {
        width: 32,
        height: 32,
        ml: -0.5,
        mr: 1,
      },
      '&:before': {
        content: '',
        display: 'block',
        position: 'absolute',
        top: 0,
        right: 14,
        width: 10,
        height: 10,
        bgcolor: 'background.paper',
        transform: 'translateY(-50%) rotate(45deg)',
        zIndex: 0,
      },
    },
  }}>
  transformOrigin={{ horizontal: 'right', vertical: 'top' }}
```

```
    anchorOrigin={{ horizontal: 'right', vertical: 'bottom' }}
```

```
>
  <MenuItem onClick={handleEdit}>
    <ListItemIcon>
      <EditIcon fontSize='small' />
    </ListItemIcon>
    編集
  </MenuItem>
  <MenuItem onClick={handleDelete}>
    <ListItemIcon>
      <DeleteForeverIcon fontSize='small' />
    </ListItemIcon>
    削除
  </MenuItem>
</Menu>
<Dialog open={openDialog} onClose={handleCloseDialog}>
  <DialogTitle>削除の確認だお～</DialogTitle>
  <DialogContent>
    <DialogContentText>
      削除すると、基には戻せないお～！ それでも削除するのかお～？
    </DialogContentText>
  </DialogContent>
  <DialogActions>
    <Button onClick={handleCloseDialog}>キャンセル</Button>
    <Button onClick={handleConfirmDelete}>削除</Button>
  </DialogActions>
</Dialog>
</>
);
};

export default DiaryCardHeader;
```

.....
ここまでの中身は、GitHub 上で、以下のコマンドでクローンできます。

▼ GitHub

```
$ > git clone -b 07_install-redux https://github.com/yaruo-react-re>
dux/yaruo-cra-template.git
.....
```

4.6 第4章のまとめ

Redux を状態管理に使う

- Redux の動作イメージ
- Redux の導入
- Action Type の定義 ~ Store の作成まで
- コンポーネントでの使用

を解説しました。

第 5 章

日記アプリケーションの作成 (Redux-toolkit 使用)

本章では、前章で導入した Redux の動作しているサンプルアプリケーションを Redux-toolkit を使いどれくらいコード量が減り苦行から解放されるのかを確認します。

5.1 Redux-toolkit の導入

それでは、さっそく redux-toolkit パッケージをインストールします。

▼ redux-toolkit のインストール

```
> npm install @reduxjs/toolkit
```

以上でインストールは完了です。

redux-toolkit では、Slice を作成します。Slice に reduce を登録すると自動で ActionCreator を作成してくれます。

それでは、Slice を作成するファイルを「src/redux/redux-tk.ts」追加してください。

5.2 Slice の作成

Slice は、「createSlice 関数」を使用して作成します。

「createSlice 関数」は、以下のような引数を受け取ります。

▼ createSlice 関数

```
function createSlice({
  // 自動生成されるActionのtypeに使用される名前
  name: string,
  // stateを初期化する初期値
  initialState: any,
  // reducerのswitch文のcase項目で使用していたkey名
  reducers: Object<string, ReducerFunction | ReducerAndPrepareObject>
  // reducerを追加するためのbuilderへのコールバック、または、追加するreducer
  | Object<string, ReducerFunction>
  | ((builder: ActionReducerMapBuilder<State>) => void)
})
```

自動生成される Action オブジェクトの type 要素は、「createSlice 関数」の name/reducer の key 名となります。のちほどサンプルアプリケーションの動作を devTools で確認しますので、どのような Action が実行されたのかが分かります。

サンプルアプリケーションの Slice です。

reduce での key 名は、

- create
- edit
- remove

にしています。「delete」は予約語ですので使えません。

Redux の特徴として、「State のコピーを用意し操作する。State を直接いじらない。」とありましたが、Redux-toolkit では immer ライブラリが自動で変換してくれるため、State を直接変更してもかまいません。

▼ サンプルアプリケーションの Slice

```
// diaries stateを初期化する値
const diariesInitialState: Diary[] = diaries;

// Sliceを作成
const diariesSlice = createSlice({
  name: 'diaries',
  initialState: diariesInitialState,
  reducers: {
```

```
create: (state, { payload }: PayloadAction<Diary>) => {
  state.push(payload);
  state.sort(diarySort);
},
edit: (state, { payload }: PayloadAction<Diary>) => {
  const index = state.findIndex(
    (diary: Diary) => diary.diaryId === payload.diaryId
  );
  if (index !== -1) {
    state.splice(index, 1, payload).sort(diarySort);
  }
},
remove: (state, { payload }: PayloadAction<{ diaryId: string }>) => {
  const index = state.findIndex(
    (diary: Diary) => diary.diaryId === payload.diaryId
  );
  if (index !== -1) {
    state.splice(index, 1).sort(diarySort);
  }
},
},
}),
});
```

各コンポーネントで使用する ActionCreator をエクスポートします。作成した Slice の「actions」が ActionCreator になります。

key を前章で作成した ActionCreator 名と同じ名前にしています。同じ名前なので各コンポーネント側では ActionCreator のインポートファイルのパスを変更するだけで、コードの変更はありません。

▼ ActionCreator のエクスポート

```
export const {
  create: createDiaryActionCreator,
  edit: editDiaryActionCreator,
  remove: deleteDiaryActionCreator,
} = diariesSlice.actions;
```

最後に、Store を作成しエクスポートします。Store は、「createStore 関数」ではなく「configureStore 関数」となります。

▼ Store の作成

```
export default configureStore({
  reducer: reducers,
```

```
  middleware: [logger],  
});
```

必要なインポート、ソート用の関数、State の型情報をいれても 58 行です。前章のファイルが 101 行ですのでこの最小のサンプルアプリケーションでも 40% の削減になります。

規模が大きくなれば、さらに削減量は増えます。

5.3 トップコンポーネントに登録

トップコンポーネントに登録するのですが、前章で登録してあるので Store のインポートパスを変えるだけです。

▼ トップコンポーネントへ登録

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import App from './components/App';

import store from './redux/redux-tk';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

5.4 コンポーネントから使用する

ActionCreator を使用するコンポーネントもインポートパスを変えるだけです。

▼ DiaryBoard コンポーネント

```
import {  
  createDiaryActionCreator,  
  editDiaryActionCreator,  
  State,  
} from '../redux/redux-tk';
```

▼ DiaryCardHeader コンポーネント

```
import { deleteDiaryActionCreator } from '../redux/redux-tk';
```

以上で Redux から Redux-toolkit へ切替が完了しました。動作確認を行います。

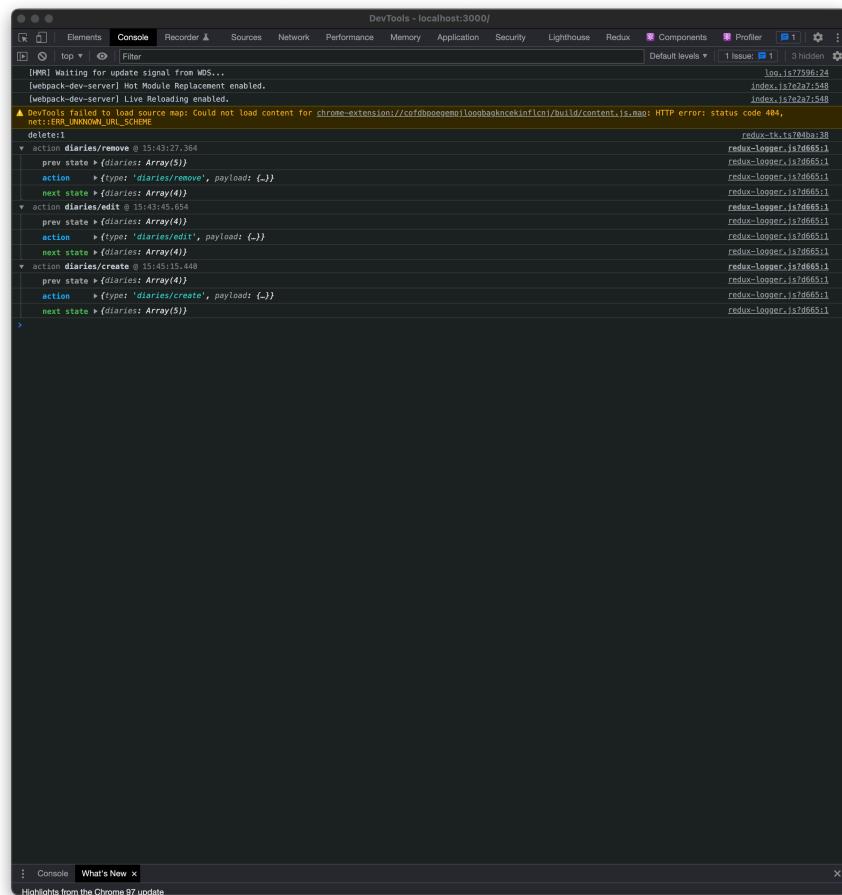
5.5 devTools で確認

動作確認で、削除・編集・追加をしました。

redux-logger を Middleware として導入していますので、devTools のコンソールに、

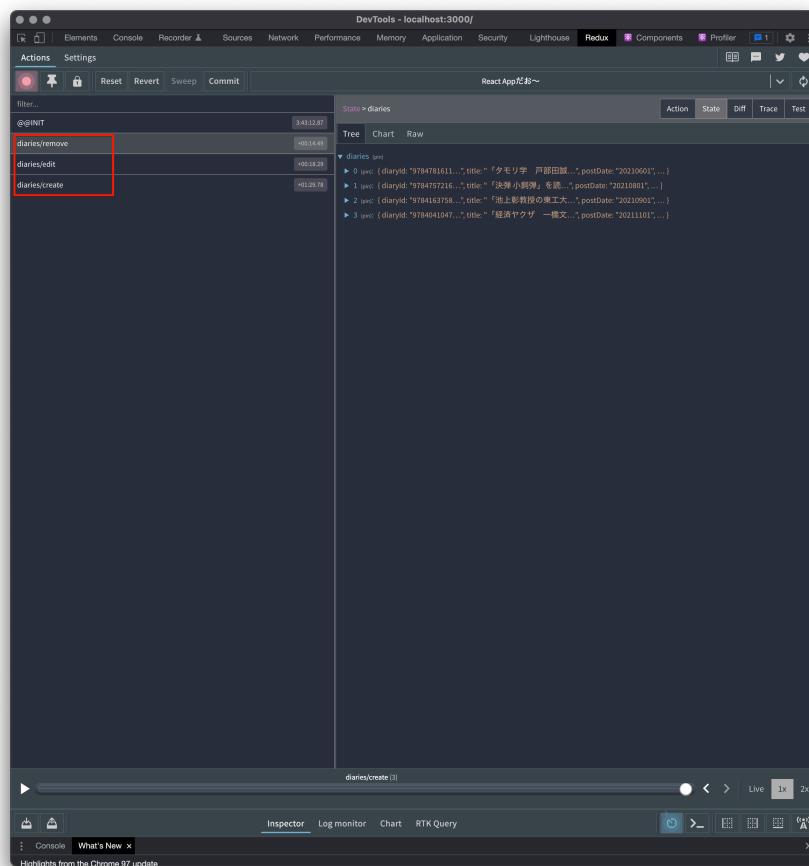
- 変更前の State
 - Action の中身
 - 変更後の State

が表示されています。



▲図 5.1: コンソールに表示されたログ

redux タブを開くと Action Type が「Slice の name/reducer のキー名」で自動生成されているのが確認できます。



▲ 図 5.2: ActionCreator の自動生成の確認

5.6 第5章のまとめ

Redux-toolkit を使うとコード量を削減できます。コード量が削減できることは、不具合の入り込む可能性を低くすることを意味します。

また、Redux ではコードが分散しがちでしたが、Redux-toolkit では Slice にまとめることができます。見通しの良いコードはさらに不具合の入り込みを減らすことでしょう。

第 6 章

おわりに

ここまで読んでいただきて、ありがとうございます。

初の技術書でしたが、いかがでしたでしょうか？

もし、この本を通じて、

- React
- Redux
- Redux-toolkit

に興味をもっていただければ、うれしいです。

もし、お時間を作って GitHub サイトにあるコードを使って試していただき、不具合がありましたら Issue を送っていただけると助かります。

また、間違いがあればプル・リクエストを送っていただけると、心より感謝します。

この本を手に取っていただき、本当にありがとうございます。

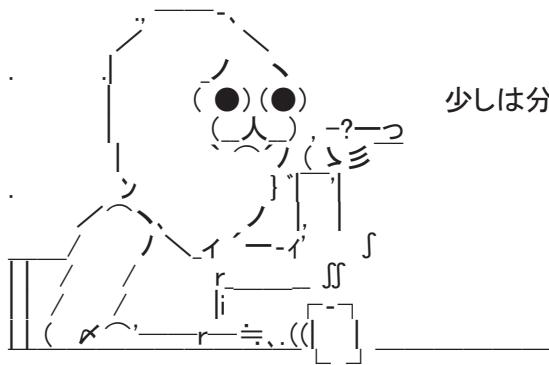
やる夫で学ぶ「react-redux」

redux-toolkit で、簡単・完璧理解だお…

2022年1月22日 ver 1.0

著 者 気分はもう

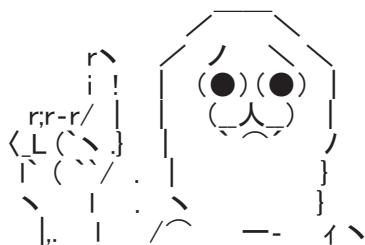
© 2022 気分はもう



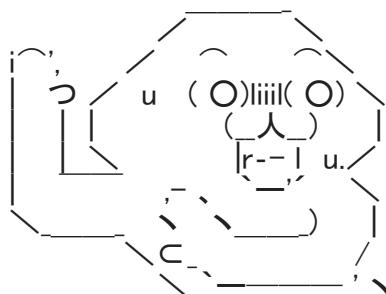
少しは分かったのか？



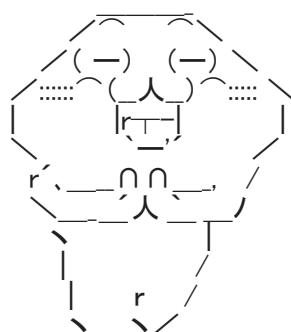
ヒヤヒヤヒヤ！
何とかなるお…



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！