React+Redux+Express+MongoDBでものすごくシンプルなCRUDアプリをつくる

Node.js MongoDB Express reactjs redux

177

概要

React+Redux+Express+MongoDBでCRUDアプリを作ります。

この記事の目的は、React/Reduxを触り始めた人が

- サーバーとの通信の方法（より一般的には非同期処理の方法）
- Reduxにおけるフォームの扱い
- ExpressによるAPI
- node.jsからのMongoDBの操作
- Herokuへのデプロイ

など、主にサーバー側のデータの操作に関わる基本的な事項を学ぶきっかけを作ることです。
この目的に集中するために、それ以外の点については一切気にしないことにします。
そのため、初心者以外の人（上記の内容を理解している人）がこの記事を読んでも得るものはないと思います。

この記事が書かれた背景には、少し前に自分自身がjavascriptによるフロントエンド開発からwebプログラミングを学び始めたころの経験があります。ReactやReduxの基本的な文法の理解を終えて、少しまともなwebサイトを書いてみようかと思い、サーバーと通信するチュートリアルを探してみたところ、複雑なものしかなく知りたいことを知るのに時間がかかってしまいました。
同じような状況の人に役立てば良いと思い、自分自身の理解の確認も兼ねて基本的な事項をまとめました。

CRUDアプリとは

この記事では、ブラウザからサーバーのデータベースを編集できるアプリのことをCRUDアプリ言うことにします。
「編集」というのは、以下の4つの操作のことです。

- C(reate): データを生成する （ `POST` リクエストに対応）
- R(ead): データを読み込む （ `GET` ）
- U(pdate): データを変更する （ `PUT` ）
- D(elete): データを削除する （ `DELETE` ）

方針

限りなくシンプルなCRUDアプリを作ります。
以下のような記事の目的に対して本質的ではないものについてはあまり気にしないし、使いません。慣れてきたら気にした方がいいと思うものについては、その都度言及するか、記事の最後にまとめておきます（わかる方がコメント等で補足してくれるとありがたいです）。

- 見た目（css）
- jsの便利・必須なツール類（ESLint、webpackなど）
- ReduxのMiddleware
- テスト

参考のために、以上のようなものを（部分的に）気にしているCRUDアプリのチュートリアルを貼っておきます。

- A Guide For Building A React Redux CRUD App
- Building a Simple CRUD App with React + Redux

また、Reactを使わないシンプルなCRUDアプリの非常にわかりやすいチュートリアルは以下です。

- Building a Simple CRUD Application with Express and MongoDB

前提

- node, npmがインストールされている
- MongoDBがインストールされていて、どういうものかわかっている
- React/Reduxの使い方がある程度わかっている
- ES6の基本的な記法がわかっている（とくにarrow関数、spread演算子、同名propertyの省略）

ES6以外の点については、記事中でできるだけリファレンスを貼るので、あまりわかっていなくても大丈夫だと思います。

環境

- Mac OSX 10.12.2
- node v7.3.0
- MongoDB v3.4.0

その他のライブラリ・モジュールのバージョンは記事中で示す `package.json` を見てください。

完成品

キャラクターの名前と年齢をサーバーに保存し、編集できるアプリを作りましょう。
CRUDそれぞれに対応する動作は以下のようにします。

- C(reate): キャラクターの名前と年齢を入力してサーバーのデータベースに保存
- R(ead): サーバーに保存されているキャラクターの情報をブラウザに表示
- U(pdate): キャラクターの年齢を 1 つ上げる
- D(elete): キャラクターのデータを消去

デモ（gifアニメ）

コード

https://github.com/ymr-39/simple-crud

ファイル構成

最終的なファイル構成を簡略化したものは以下です。サーバーのディレクトリの中に、クライアントのコードが入ったディレクトリを置くことにします。サーバー側・クライアント側でそれぞれ独立に `package.json` を置いて、モジュールを管理します。

```
package.json
server.js       // サーバー側のエントリーポイント
characters.js   // データベースのモデル
.babelrc
node_modules/
client/         // クライアント側のコード
    package.json
    node_modules/
    public/
    src/
    build/
```

この記事で記載するコード・コマンドについて

いろいろとコードやコマンドを示していきますが、コードはできるだけdiff形式でハイライトして全体を示したいと思います。
また、コマンドラインの操作では、あえて明示したり直前のコマンドから引き続いていない限り、`server.js`があるルートディレクトリにいるものとします。

サーバー側モジュール

必要なモジュールをnpmでインストールしていきます。
まずはサーバー側から。

```
mkdir simple-crud && cd simple-crud
npm init -y
npm i --save express
npm i --save mongoose  // nodeからMongoDBを操作
npm i --save babel-cli babel-preset-es2015  // ES6→ES5の変換
npm i --save body-parser  // クライアントからのHTTPリクエストをパース
```

次に、`package.json`を書き換えていきます。

package.json

```
{
  "name": "simple-crud",
  "version": "1.0.0",
  "description": "",
-   "main": "index.js",
+   "main": "server.js",
  "scripts": {
-     "test": "echo \"Error: no test specified\" && exit 1"
```

```
+     "start": "babel-node server.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "babel-cli": "^6.22.2",
    "babel-preset-es2015": "^6.22.0",
    "body-parser": "^1.16.0",
    "express": "^4.14.0",
    "mongoose": "^4.7.7"
  }
}
```

`"main"` の変更は、クライアント側のエントリーポイントも `index.js` という名前にするのでまぎらわしいため（まったく本質的ではないです）。ついでに `server.js` のファイルを作成しておきます。

```
touch server.js
```

また、 `"scripts"` に `"start"` を書き加えたことで、コマンドラインから `npm start` と打つだけでサーバーが起動するようにしました。この際実行される `babel-node` は、 `babel-cli` のインストールによって入るコマンドで、通常の `node` による実行と異なり（以下で説明する `.babelrc` があれば） `server.js` をES6で書くことができます。

`babel` が読む設定ファイルである `.babelrc` を書いておきます。
今回はES6(ES2015)を使いたいだけなので、以下のようにしておけば大丈夫です。

.babelrc

```
{
  "presets" : ["es2015"],
}
```

クライアント側モジュール

続いて、クライアント側です。
本来、Reactのコードをブラウザから読むためには、書いたjsのコード群をwebpackやbrowserifyなどのツールでビルドする必要があります。これは、ReactやReduxのコードの書き方自体を勉強する上では本質的でない作業だと思うので、そのような作業を省いてくれるツールであるcreate-react-appを使います（今回は使いませんが、慣れてきたらwebpackを勉強した方が良いと思います。create-react-app自体も内部でwebpackを使っています）。

以下のコマンドでcreate-react-appをインストールしたのち、create-react-appコマンドでプロジェクトを生成します。

```
npm i -g create-react-app  // グローバルにインストール
create-react-app client
```

create-react-appで生成したプロジェクトには、最初から `react` と `react-dom` のモジュールがインストールされているので、追加で必要なモジュールのみインストールします。

```
cd client
npm i --save redux
npm i --save axios  // サーバーへHTTPリクエスト
```

create-react-appによって、（今回の記事にとっては）不要なファイルがいろいろできているので、以下のコマンドで削除します。
また、srcの下にcomponentsというディレクトリを作って、Reactのcomponentはすべてそこで管理することにします。さらに、Reduxのreducerとactionを書くファイルを作っておきます。

```
rm README.md .gitignore
cd src
rm App.test.js logo.svg *.css
mkdir components && mv App.js components/  // componentsディレクトリの作成
touch reducers.js actions.js
```

ファイル構成確認

ここまでの作業で、ファイル構成は以下のようになっていると思います。

```
package.json
server.js
.babelrc
node_modules/
client/
    package.json
    node_modules/
    public/
        index.html
        favicon.ico
    src/
        index.js
        reducers.js
        actions.js
```

```
        components/
            App.js
```

アプリ作成

それでは、実際にアプリを作っていきましょう。C(reate)、R(ead)、U(pdate)、D(elete)の機能を順番に実装していきます。
C(reate)の項ではいろいろ説明したり共通のコードを書くのでかなり長いですが、それ以降の3つは短いので安心してください。

Create (`POST`)

クライアント側、サーバー側に分けてコードを書いていきます。

クライアント側

React/Reduxでアプリを作っていく方法は人それぞれあると思いますが、今回は以下の順序で書いています。

1. (`App.js` に)アプリを構成するcomponentsを書く
2. (`reducers.js` に)storeの初期状態を書く
3. (`reducers.js` に)reducer(s)の動作を思いつく限り書いてしまう
4. (`actions.js` に) `reducers.js` を書いていて必要になったactionの文字列定数とaction createrを書く
5. (`index.js` に)storeを作成してAppに渡す

6. それぞれのcomponentのビュー・イベントハンドラを作る

1.(`App.js` に)アプリを構成するcomponentsを書く

まずは、アプリの構成をつかむためにどのような部品が必要か考えてみましょう。
完成品を思い浮かべると、ブラウザ上には

- 追加したいキャラクターの情報を入力するフォーム
- サーバーにあるキャラクターのデーター一覧を表示するリスト

があるはずです。
それぞれ `<AddForm />` と `<CharacterList />` というcomponentで表すことにしましょう。
これらのコンポーネントに対応する実際のコードはこれから書きますが、すでに書いてあるものと思って `App.js` からimportしておきましょう。

client/src/components/App.js

```
import React, { Component } from 'react';
import AddForm from './AddForm'
import CharacterList from './CharacterList'

class App extends Component {
  render() {
    return (
      <div>
        <AddForm />
        <CharacterList />
      </div>
    )
  }
}
```

```
export default App
```

2.(`reducers.js` に)storeの初期状態を書く

Reduxのstoreはどのような情報を保持するべきでしょうか。
アプリを構成するcomponentsをもとに考えると、

- `<AddForm />` に入力されている文字列
- `<CharacterList />` に表示するキャラクターのリスト

が必要な気がします。これらをそれぞれ別に表すことにして、初期状態を表すオブジェクト `initialState` を以下のように書いておきます。

client/src/reducers.js

```
const initialState = {
  form: {  // AddFormに入力されている文字列
    name: '',
    age: '',
  },
  characters: {
    isFetching: false,  // サーバーからキャラクターのリストを取ってきている最中かどうか
    characterArray: [],  // キャラクターのデータを入れるArray
  },
}
```

`initialState.form` について

Reactでのフォームの扱い方は大きく、

- Controlled（入力された情報をReduxのstoreやReact componentが持つ）
- Uncontrolled（入力された情報をDOMが持つ）

の二通りに分けられます。

今回はフォームをControlledとして扱います。これは、`<form>`の`<input>`要素に`value`と`onChange`を設定し、`onChange`の中で`<input>`に加えられた変更を即座にReduxのstore（もしくはReact componentのstate）に反映し、`value`にはstoreから値を取ってきて設定するというアプローチです。すなわち、`<input>`要素は単なるビュー（＋入力装置）であり、真の情報はstoreが持っているという考え方です。

一方、Uncontrolled componentでは、formに入力された文字列の情報は`<input>`要素に持たせておいて、文字列が変更されるたびに、Reactの`ref` (Refs and the DOM)を使ってReact componentの`state`か、ローカル変数にその値をコピーします。このアプローチでは、`<input>`要素が情報を持っていて、ロジック側では必要になったときにはその入力の値を知るという感じです。

Uncontrolledの方が実装が簡単ですが、少し複雑なこと（リアルタイムでの入力のvalidationなど）をしたいときには、Controlledの方が柔軟に処理ができます。
双方のアプローチについてさらに詳しく知りたい場合は、以下を参考にしてください。
とくに一番目の記事は非常にわかりやすいです。

- Controlled and uncontrolled form inputs in React don't have to be complicated
- Uncontrolled Components
- Forms

`initialState.characters`について

`isFetching`は一見重要でないようにも感じますが、以下の理由で必要です。

1. `isFetching === true` のときにローディングアイコン（「Now Loading...」みたいなやつ）を出せるので、ユーザーがアプリの状況を把握できる
2. `isFetching` の値を見ることで、アプリがアプリ自身の状況を把握できる

より本質的なのは2の方です（1は2の副産物）。

「アプリの状態はすべてstoreを見ればわかる」というのがReduxの良いところだと思います。
これに対して、サーバーからデータを取ってくるというような非同期な処理では、いつデータが降ってくるのかをアプリが把握することができません。すなわち、データをリクエストした後、アプリは「自分が今データを待ち受けている状態なのか、すでにデータを受け取っているのか」がわからず、上記のReduxのメリットが失われてしまうことになります。
これに対して、 `isFetching` のような変数を作っておいて、サーバーにデータをリクエストすると同時に `isFetching = true` とし、データを受け取ると同時に `isFetching = false` とすることで、アプリが現在の状況を把握することができます。これにより、サーバーにリクエストした直後に別の処理をしたくなった場合、「データを待っているのか、すでに受け取っているのか」によって処理内容を分けることが出来るようになるなど、「アプリの状態はすべてstoreを見ればわかる」というReduxの良さが復活します。

3. (`reducers.js` に)reducer(s)の動作を思いつく限り書いてしまう

考えられるreducerの動作をすべて書きましょう。この作業をすることで、自動的に、必要なactionを定義することになります。

ここではCreateの機能だけを実装するので、とりあえず主にReadの機能に対応する `characters` のreducerは放っておいて、 `form` に対するreducerのみ書けばいいです。

`form` の内のデータを変更するのに必要そうな機能とそれに対応するaction名を挙げていきましょう。

- `form` を初期化する (`INITIALIZE_FORM`)
- `form.name` を変更する (`CHANGE_NAME`)
- `form.age` を変更する (`CHANGE_AGE`)

この段階では、「アプリの部品としてのフォーム」に必要な操作ではなく、あくまで「storeの部品としての `form` 」に必要なactionを考えていれば良いと思います。例えば、アプリの部品としてのフォームには、「入力したデータをサーバーにsubmitする」という操作が必要だと思いますが、そういった操作はcomponentsのイベントハンドラとして書くことにし、reducerは直接関知しないことにします。

上記の3つのactionに対応するreducerの処理を書きましょう。
実際にはまだ `actions.js` は書いていませんが、すでにあると想定して、必要なaction を `import` しておきます。以上３つのactionに対応する `form` のreducerを書きます。 `CHANGE_NAME` と `CHANGE_AGE` については、actionオブジェクトがそれぞれ `name` 、 `age` プロパティを持っていることにします。

client/src/reducers.js

```
import { combineReducers } from 'redux'
import { CHANGE_NAME, CHANGE_AGE, INITIALIZE_FORM } from './actions'

const initialState = {
  form: {
    name: '',
    age: '',
  },
  characters: {
    isFetching: false,
    characterArray: [],
```

```
  },
}

const formReducer = (state = initialState.form, action) => {
  switch (action.type) {
    case CHANGE_NAME:
      return {
        ...state,
        name: action.name,
      }
    case CHANGE_AGE:
      return {
        ...state,
        age: action.age,
      }
    case INITIALIZE_FORM:
      return initialState.form
    default:
      return state
  }
}
```

まだ `characters` のためのreducerは考えなくていいのですが、後で楽をするために形だけ書いておきます。2つのreducerを `combineReducers` して `export` すると、以下のようになると思います。

client/src/reducers.js

```
import { combineReducers } from 'redux'
import { CHANGE_NAME, CHANGE_AGE, INITIALIZE_FORM } from './actions'

const initialState = {
  form: {
    name: '',
    age: '',
  },
```

```
  characters: {
    isFetching: false,
    characterArray: [],
  },
}

const formReducer = (state = initialState.form, action) => {
  switch (action.type) {
    case CHANGE_NAME:
      return {
        ...state,
        name: action.name,  // actionのnameプロパティに入力された名前を入れることにする
      }
    case CHANGE_AGE:
      return {
        ...state,
        age: action.age,  // nameと同様
      }
    case INITIALIZE_FORM:
      return initialState.form  // 初期状態を返す
    default:
      return state
  }
}

+ const charactersReducer = (state = initialState.characters, action) => {
+   switch (action.type) {
+     default:
+       return state
+   }
+ }

+ const rootReducer = combineReducers({
+   form: formReducer,
+   characters: charactersReducer,
+ })
```

```
+ export default rootReducer
```

もし基本的なreducerの書き方や `combineReducers` についてわからなければ、Getting Started with Reduxを（途中までででも）見るのがわかりやすいと思います。あまりこだわらなければ、とりあえず `combineReducers` すれば複数のreducerを組み合わせることができる、という理解でこのまま進んで大丈夫です。

4. (`actions.js` に) `reducers.js` を書いていて必要になったactionの文字列定数とaction createrを書く

すでに `reducers.js` を書くときにactionについては考え終わっているので、何を書くべきかはほとんど明らかです。
以下のようになると思います。

client/src/actions.js

```
// 文字列定数
export const CHANGE_NAME     = 'CHANGE_NAME'
export const CHANGE_AGE      = 'CHANGE_AGE'
export const INITIALIZE_FORM = 'INITIALIZE_FORM'

// action creaters
export const changeName = name => ({
  type: CHANGE_NAME,
  name,
})
export const changeAge = age => ({
  type: CHANGE_AGE,
  age,
})
export const initializeForm = () => ({
  type: INITIALIZE_FORM,
})
```

5. (`index.js` に)storeを作成してAppに渡す

複雑なアプリを作りたいときは、storeとcomponentsをつなぐためにreact-reduxを使った方がいいと思いますが、今回は単純に、 `index.js` で作成したstoreを `prop` としてすべてのcomponentsに渡していくことにします。

client/src/index.js

```
import React from 'react'
import ReactDOM from 'react-dom'
import { createStore } from 'redux'
import App from './components/App'
import rootReducer from './reducers'

const store = createStore(rootReducer)

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

store.subscribe(() => {
  render()
  console.log(store.getState().form)   // 動作確認のためコンソール出力
})
render()
```

client/src/components/App.js

```
import React, { Component } from 'react'
import AddForm from './AddForm'
import CharacterList from './CharacterList'

class App extends Component {
```

```
  render() {
    return (
      <div>
-         <AddForm />
+         <AddForm store={this.props.store} />
-         <CharacterList />
+         <CharacterList store={this.props.store} />
      </div>
    )
  }
}

export default App
```

`index.js` では、後で行う動作確認のため、storeが更新されるたびに呼ばれる `store.subscribe()` の中で、 `form` の情報をコンソールに出力しています。このような目的には本来、Middlewareのredux-loggerを使った方がいいと思いますが、今回は使わないことにします。

6. それぞれのcomponentのビュー・イベントハンドラを作る

データのCreateに関わる `<AddForm />` を実装します。

フォームに入力した文字列をサーバーに送る処理は一旦置いておいて、まずは、入力した文字列をstoreに反映する処理までを行います。
名前・年齢に対応する `<input>` とsubmitのための `<button>` を実装します。

client/src/components/AddForm.js

```
import React from 'react'
import { changeName, changeAge } from '../actions'

const AddForm = ({ store }) => {
```

```
  const { name, age } = store.getState().form  // storeからフォームの内容を取得

  return (
    <div>
      <form>
        <label>
          名前:
          <input value={name} onChange={e => store.dispatch(changeName(e.target.value))}
        </label>
        <label>
          年齢:
          <input value={age} onChange={e => store.dispatch(changeAge(e.target.value))} />
        </label>
        <button type="submit">submit</button>
      </form>
    </div>
  )
}

export default AddForm
```

今回はフォームをControlledとして扱いたいので、`<input>`はstoreから取ってきた値を`value`として表示しています。文字列が変更されると、即座に`onChange`の中でactionをdispatchして、変更をstoreに反映します。
`<input>`の`value`を指定した場合、`onChange`を正しく書かない限り一切変更ができなくなるので注意してください。

また、`<CharacterList />`も形だけ実装しておきましょう。

client/src/components/CharacterList.js

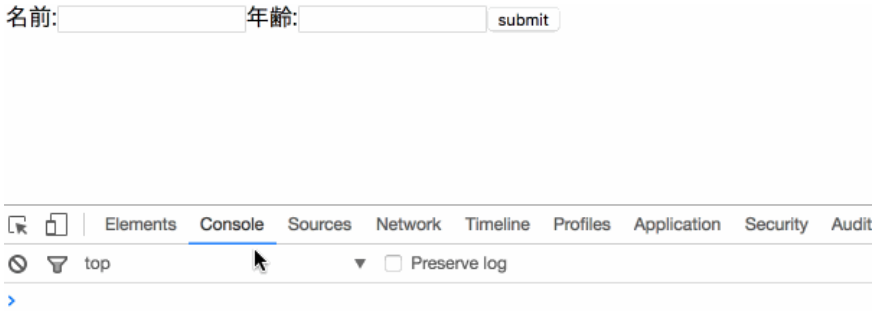
```
import React from 'react'

const CharacterList = ({ store }) => {
```

```
  return (
    <div>
    </div>
  )
}

export default CharacterList
```

フォームの変更がReduxで管理できているかどうか、ここまでのコードで動作確認をしてみましょう。

```
cd client
npm start
```

`http://localhost:3000` をブラウザで開いて、フォームに入力してみましょう。
ここで、developer toolを開いて（Chromeなら `option+command+i` ）コンソール出力を見ておきます。



フォームの変更が逐一コンソールに出力されています。
このコンソール出力は以下の流れで行われています。

- `<AddForm />` 中の `<input>` が変更されると `onChange` が発火してstoreに変更が反映される
- storeの変更のたびに呼ばれる `store.subscribe()` の中の `console.log()` がフォームの内容を出力する

つまり、うまくコンソール出力されていることから、入力文字列をstoreに反映するところまではうまく動いていそうなことがわかりました。確認が終わったので、 `store.subscribe()` の中の `console.log()` は削除しておきましょう。

それでは、いよいよstoreに入っているデータをサーバーに送りたいと思います。
javascriptでサーバーと通信したいときに使うライブラリのうち、最近よく使われているものには

- superagent
- axios
- jQuery

などがあると思いますが（superagentとaxiosの使い分け）、今回はaxiosを使います。

axiosについてはnpmのページを読めばだいたいわかると思いますが、今回には、

```
axios.get([url])
axios.post([url], [data])
axios.put([url], [data])
axios.delete([url])
```

とすれば `GET/POST/PUT/DELETE` リクエストができて、結果がPromiseで返ってくることを覚えておけば大丈夫です。

Promiseについてはわかりやすい読み物がいろいろあると思います（例えば、JavaScript Promiseの本）が、とりあえずちゃんとわからなくてもいいという人は、例えば `axios.get()` ならば、

```
axios.get([url])
.then(response => { // リクエストが成功した場合
  // do something using response
})
.catch(error => { // 失敗した場合
  // handle error
})
```

のように、処理が成功した場合は `.then` （引数はPromiseが返す値）が、失敗した場合が `.catch` （引数はPromiseが投げる `error` ）が呼ばれると考えておけば大丈夫です。

今回はサーバーにデータを送りたいので `axios.post()` を使います。 `axios.post()` では、2番目の引数にサーバーに送りたいデータをとれます。
今回は、キャラクターの名前、年齢からなるオブジェクトを送ることにします。例えば、

```
{
  name: '宮森あおい',
  age: '21'
}
```

のような感じのオブジェクトですね。
submitに対処するイベントハンドラ `handleSubmit` を加えて、 `AddForm.js` は以下のようになりました。イベントハンドラの中では、動作確認のため、リクエストが成功した場合にサーバーから送り返されてくる `response` をコンソール出力しています。

client/src/components/AddForm.js

```
import React from 'react'
+ import axios from 'axios'
import {
  changeName, changeAge,
+   initializeForm
} from '../actions'

const AddForm = ({ store }) => {
  const { name, age } = store.getState().form

+   const handleSubmit = e => {
+     e.preventDefault()    // フォームsubmit時のデフォルトの動作を抑制

+     axios.post('/api/characters', {
+       name,
+       age,
```

```
+      })  // キャラクターの名前、年齢からなるオブジェクトをサーバーにPOST
+      .then(response => {
+        console.log(response)  // 後で行う動作確認のためのコンソール出力
+        store.dispatch(initializeForm())  // submit後はフォームを初期化
+      })
+      .catch(err => {
+        console.error(new Error(err))
+      })
+    }

  return (
    <div>
-       <form>
+       <form onSubmit={e => handleSubmit(e)}>
        <label>
          名前:
          <input value={name} onChange={e => store.dispatch(changeName(e.target.value))}
        </label>
        <label>
          年齢:
          <input value={age} onChange={e => store.dispatch(changeAge(e.target.value))} />
        </label>
        <button type="submit">submit</button>
      </form>
    </div>
  )
}

export default AddForm
```

これでクライアント側の処理は終わりですが、クライアントでデータを送る処理を書いただけではうまく動きません。サーバー側でもデータを受け取る処理を書いてあげる必要があります。

サーバー側

以下の順番で書いていきます。

1. Expressでサーバーをたてる
2. `mongoose` でMongoDBのモデルを作る
3. MongoDBに接続する
4. `POST` リクエストの処理

1. Expressでサーバーをたてる

Expressはnodeでサーバーをたてるときにつかうライブラリです。とくに、いわゆるREST APIが簡単に書けるという特徴があります（ゼロからはじめるExpress + Node.jsを使ったアプリ開発）。
詳しいAPIについては公式のドキュメントを参照してもらえば良いと思いますが、今回使うのは以下です。

```
const app = express()

// GETリクエストに対処
app.get([url], (request, response) => {
  // requestをもとに処理をし、クライアントにresponseを返す
})

// POSTリクエストに対処
app.post([url], (request, response) => {
  //
})

// PUTリクエストに対処
app.put([url], (request, response) => {
```

```
  //
})

// DELETEリクエストに対処
app.delete([url], (request, response) => {
  //
})

// ポートを指定してアクセスを受け付ける
app.listen([ポート番号], callback)
```

基本的に、メソッドを指定して、コールバック関数（引数はリクエストとレスポンスを表すオブジェクト）に処理を書くという感じです。

とりあえず、アクセスを受け付けるだけの `server.js` のコードは以下の通り。

server.js

```
import express from 'express'

const app = express()
const port = 3001

app.listen(port, err => { // http://localhost:3001にサーバーがたつ
  if (err) throw new Error(err)
  else console.log(`listening on port ${port}`)
})
```

2. MongoDBに接続する

つぎに、node.jsからMongoDBを操作するためのライブラリである `mongoose` を使って、 `server.js` からMongoDBに接続します。

まず、MongoDBのプロセスを起動しておきます。
コマンドラインで

```
mongod
```

と打つことでMongoDBが起動します（環境によってコマンドが違うかも）。

`mongoose.connect()` でMongoDBに接続することができます。 `callback` 関数の引数は接続が失敗した場合の `error` になります。

```
mongoose.connect([dbのurl], callback)
```

`server.js` を以下のように書き換えます。

server.js

```
import express from 'express'
+ import mongoose from 'mongoose'

const app = express()
const port = 3001
+ const dbUrl = 'mongodb://localhost/crud' // dbの名前をcrudに指定

+ mongoose.connect(dbUrl, dbErr => {
+   if (dbErr) throw new Error(dbErr)
+   else console.log('db connected')

+   // MongoDBに接続してからサーバーを立てるために
+   // app.listen()をmongoose.connect()の中に移動
+   app.listen(port, err => {
+     if (err) throw new Error(err)

+     else console.log(`listening on port ${port}`)
```

```
+   })
+ })

- app.listen(port, err => {
-   if (err) throw new Error(err)
-   else console.log(`listening on port ${port}`)
- })
```

とりあえずここまでで、MongoDBに接続でき、かつサーバーをたてられているかを確認します。
コマンドラインで `npm start` します（先ほどは `client` ディレクトリで同じコマンドを打ちましたが、今回はサーバーのプロセスを起動したいので、その上の `server.js` があるディレクトリで行います）

```
npm start  // 'babel-node server.js'が実行される

// うまくいっていれば下記のように表示されるはず
// db connected
// listening on port 3001
```

3. `mongoose` でMongoDBのモデルを作る

`mongoose` では、MongoDBを直接触る時と違ってスキーマ（保存するドキュメントがどのようなフィールドを持つか）を定義します。さらに、このスキーマをコンパイルすることで、保存するドキュメントのコンストラクタの働きをするモデルを作成します。
以下が、モデルを書いた `character.js` のコードです。

character.js

```
import mongoose from 'mongoose'
```

```
mongoose.Promise = global.Promise

//  スキーマの作成
//  今回保存したいドキュメントはname(String)とage(Number)の２つのフィールドを持つ
const CharacterSchema = new mongoose.Schema({
  name: String,
  age: Number,
})

// モデルの作成
// mongoose.modelの第一引数の複数形の名前（今回だと'characters'）のコレクションが生成される
const Character = mongoose.model('Character', CharacterSchema)

// モデルをexport
export default Character
```

4. `POST` リクエストの処理

ここまでで、

- サーバーとMongoDBへの接続の確立
- MongoDBに保存するデータのモデルの作成

を行うことができたので、いよいよクライアントから送られてくるデータをMongoDBに保存する処理を書きます。
`POST` リクエストに対応するためにExpressの `app.post()` を使います。

```
app.post([url], (request, response) => {
  // クライアントからのrequestを処理
  // responseをクライアントに送り返す
})
```

クライアント側では、`axios.post()`を使って、フォームに入力されたキャラクターの情報を持ったjsonをサーバーに送ったことを思い出してください。サーバー側から見ると、`app.post()`のコールバックの引数である`request`の`body`プロパティにそのjsonが入っています（`response`や`request`について詳しく知りたい場合は、Anatomy of an HTTP Transaction）。
このjsonを正しく受け取るためにはExpressに加えて、最初にインストールしておいたbody-parserが必要です。

送られてきたデータをMongoDBに保存する前に、まずはきちんとクライアントからデータが送れているか確認するために、以下のように送られてきたデータをコンソール出力してみましょう。

server.js

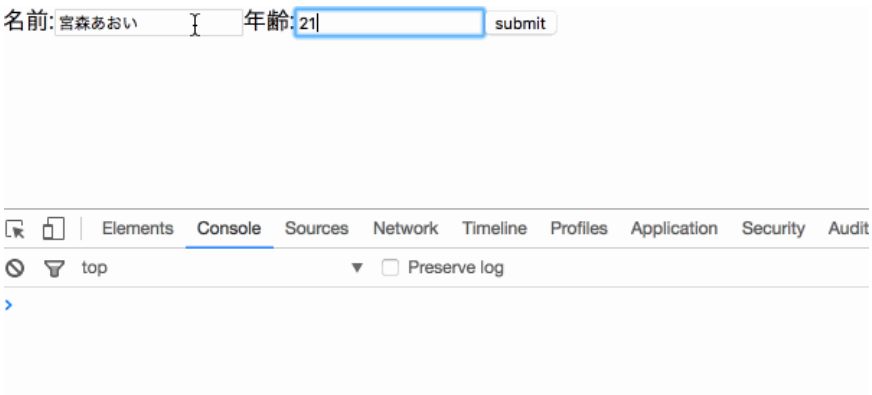
```
import express from 'express'
+ import bodyParser from 'body-parser'
import mongoose from 'mongoose'

const app = express()
const port = 3001
const dbUrl = 'mongodb://localhost/crud'

// body-parserを適用
+ app.use(bodyParser.urlencoded({ extended: true }))
+ app.use(bodyParser.json())

mongoose.connect(dbUrl, dbErr => {
  if (dbErr) throw new Error(dbErr)
  else console.log("db connected")

+   // POSTリクエストに対処
+   app.post('/api/characters', (request, response) => {
+     console.log('receive POST request')
+     console.log(request.body)  // 送られてきたデータをコンソール出力
```

```
+     response.status(200).send()  // クライアントにステータスコード(200:成功)とともにレスポン
+   })

  app.listen(port, error => {
    if (error) throw new Error(error)
    else console.log(`listening on port ${port}`)
  })
})
```

この状態でサーバー・クライアントのコードを実行しようとすると、一つ問題が生じると思います。
クライアントからサーバーにデータを送るためには、両方のプロセスを立ち上げる必要があります。立ち上げるだけならば、コンソールを2つ起動して、それぞれのディレクトリで

```
npm start
```

すれば良いのですが、実際にこの状態でブラウザのフォームからsubmitしようとすると失敗し、コンソールには以下のように表示されると思います。



さらに、クライアントから送られてきたデータを出力するはずのサーバー側のコンソールにも何も表示されていません。サーバー・クライアント双方の処理を書いて、プロセスを立ち上げているはずなのに、なぜ通信がうまくいかないのでしょうか。
この問題の原因と対処について簡単に説明します。

~~2つのプロセスが同じポートを使うことはできないため、クライアントとサーバーは異なるポートで起動させます（今回はクライアント:3000, サーバー:3001）。上記のエラーは、このような異なるオリジン間での情報のやりとりを制限する仕組みによるものです（例えば、HTTP アクセス制御 (CORS)を参照）。簡単に言うと、同じlocalhostであっても、ポート番号が違えばデータのやりとりができないということです。~~
~~この問題は、クライアントから、サーバーと同じアドレスのプロキシを通してリクエストを行うことで回避できます（詳しくは、How to get "create-react-app" to work with your API）。~~

[追記] 上記の問題の原因について勘違いしていた点を @k4h4shi さんからのコメント で教えていただきました。正しい問題の原因はリクエストのURLを間違えていたことでした。

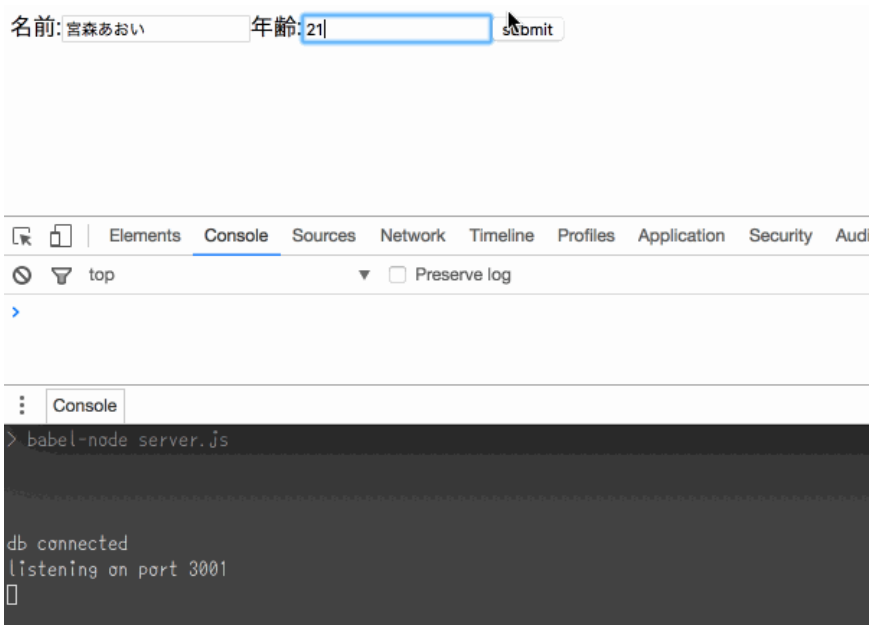
具体的な作業としては、クライアント側の `package.json` に以下のように1行を追加すれば良いです。

client/package.json

```
{
  "name": "client",
  "version": "0.1.0",
  "private": true,
  "devDependencies": {
    "react-scripts": "^0.8.5"
  },
  "dependencies": {
    "axios": "^0.15.3",
```

```
    "react": "^15.4.2",
    "react-dom": "^15.4.2",
    "redux": "^3.6.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  },
+   "proxy": "http://localhost:3001/"
}
```

それでは、サーバ・クライアント双方のプロセスを立ち上げ直して（`npm start`）、
実際にクライアントからデータを送ってみましょう。
下の黒い画面は、サーバー側の`npm start`をしたコンソールです。



submitボタンを押したあと、以下の処理がされていることがわかると思います。

- `server.js` の `app.post()` 内の `console.log()` により、送られてきたデータがターミナルに正しく出力されている
- `AddForm.js` の `axios.post()` 内の `console.log()` により、サーバー側から送り返された `response` がブラウザのコンソールに表示されている
- `AddForm.js` の `axios.post()` 内の `store.dispatch(initializeForm())` により、フォームが初期化されている

つまり、クライアントからサーバーに正しくデータを送り、サーバーからクライアントにレスポンスを送り返し、クライアント側で後処理をする、という流れが無事実装できていることになります。

それでは、サーバー側で送られてきたデータを単にコンソール出力していたのを、MongoDBにデータを保存するように書き換えます。
`mongoose` でMongoDBにドキュメントを追加（データを保存）するには、

```
new [モデル名]({
  フィールド名:値,
})
```

で１つのドキュメントを表すインスタンスを生成し、このインスタンスの `save` メソッドを使います。 `save()` の引数はコールバック関数ですので、その中でクライアントにレスポンスを送り返すようにします。
具体的には、以下のコードを見てください。

server.js

```
import express from 'express'
import bodyParser from 'body-parser'
import mongoose from 'mongoose'

+ import Character from './character' // モデルをimport
```

```
const app = express()
const port = 3001
const dbUrl = 'mongodb://localhost/crud'

app.use(bodyParser.urlencoded({ extended: true }))
app.use(bodyParser.json())

mongoose.connect(dbUrl, dbErr => {
  if (dbErr) throw new Error(dbErr)
  else console.log('db connected')

  app.post('/api/characters', (request, response) => {
-     console.log('receive POST request')
-     console.log(request.body)
-     response.status(200).send()
+     const { name, age } = request.body  // 送られてきた名前と年齢を取得

+     new Character({
+       name,
+       age,
+     }).save(err => {
+       if (err) response.status(500)
+       else response.status(200).send(`${name}(${age}) was successfully created.`)
+     })
  })

  app.listen(port, err => {
    if (err) throw new Error(err)
    else console.log(`listening on port ${port}`)
  })
})
```

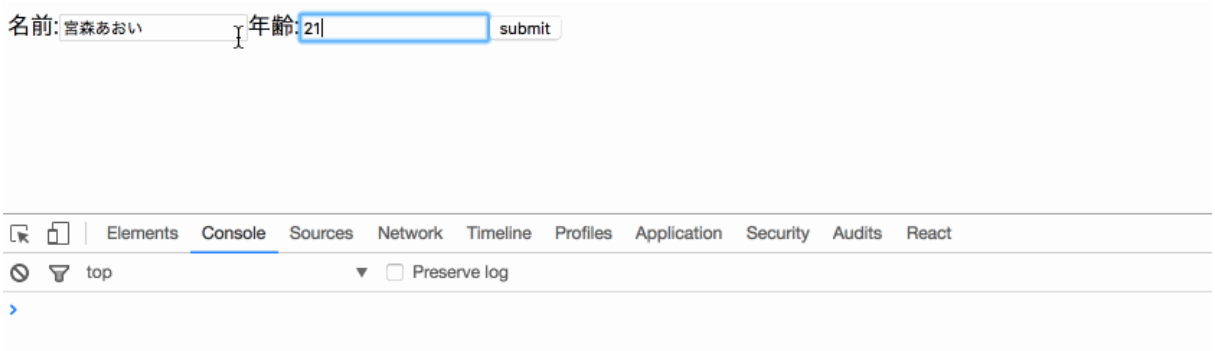
`save()` のコールバックを見るとわかるように、動作確認のため、レスポンスの内容を具体的にしてあります。

これにより、クライアント側では、例えば

`宮森あおい(21) was successfully created.`

のようなレスポンスを受け取ることができるはずです。

サーバーのプロセスを立ち上げ直し（`npm start`）、再びブラウザからフォームをsubmitしてみましょう。
サーバー側で`npm start`し直さないとファイルの変更が反映されないので注意してください。今後も動作確認がうまくいかない場合は、すべてのファイルを保存してから`npm start`し直しているかを確認して見てください（ファイルの更新のたびに自動的にサーバーを立ち上げ直してくれるライブラリとしてnodemonがあります）。



コンソールによると、データが無事追加されているようです。ただし、本当に追加されているかは結局サーバー側のデータベースを見て確認するべきなので、MongoDBのシェルに入って、データが追加されていることを確認してみます。

コンソールから

```
mongo
```

でMongoDBのシェルを起動し、以下のコマンドを打ちます。

```
use crud // 'crud'というdbに入る
db.characters.find().pretty() // 'characters'というcollectionのドキュメントをすべて表示（pre
```

以下のような結果が得られれば、無事にクライアント側からサーバーのMongoDBにデータが追加できていることになります。

ここで、自分でスキームに書いた `name` 、 `age` 以外にもフィールドができていることに注意してください。これらはそれぞれ、

- `_id` : ドキュメントにMongoDBが付加するユニークなID（mongodbのObjectIdの生成規則）
- `__v` : こちらはmongooseが付加するversion key（これについてはあまり理解していないので、とりあえず公式のドキュメントをリンクしておきます）

です。 `_id` はそのキャラクターのデータを識別するためのidとしてアプリ内で使うことにし、 `__v` は今回は気にしないことにします。

Read (`GET`)

Createと同じように進めていきます。

まず、Readの機能に関わるreducerを書きましょう。

storeの `initialState` の形を再掲します。

```
const initialState = {
  form: {    // AddFormに入力されている文字列
    name: '',
    age: '',
  },
  characters: {
    isFetching: false,    // サーバーからキャラクターのリストを取ってきている最中かどうか
    characterArray: [],    // キャラクターのリストを入れるArray
  },
}
```

Readの機能に必要なactionは

- データをサーバーにリクエストするときに `characters.isFetching` を `true` にする (`REQUEST_DATA`)
- データをサーバーから受け取った時に `characters.characterArray` に入れるとともに `characters.isFetching` を `false` にする (`RECEIVE_DATA_SUCCESS`)
- データの受け取りに失敗した時に `characters.isFetching` を単に `false` にする (`RECEIVE_DATA_FAILED`)

くらいです。以上を `reducers.js` に書き加えます。

client/src/reducers.js

```
import { combineReducers } from 'redux'
import {
  CHANGE_NAME, CHANGE_AGE, INITIALIZE_FORM,
+   REQUEST_DATA, RECEIVE_DATA_SUCCESS, RECEIVE_DATA_FAILED
} from './actions'
```

```
const initialState = {
  form: {
    name: '',
    age: '',
  },
  characters: {
    isFetching: false,
    characterArray: [],
  },
}

const formReducer = (state = initialState.form, action) => {
  switch (action.type) {
    case CHANGE_NAME:
      return {
        ...state,
        name: action.name,
      }
    case CHANGE_AGE:
      return {
        ...state,
        age: action.age,
      }
    case INITIALIZE_FORM:
      return initialState.form
    default:
      return state
  }
}

const charactersReducer = (state = initialState.characters, action) => {
  switch (action.type) {
+     case REQUEST_DATA:
+       return {
+         ...state,

+         isFetching: true,
```

```
+       }
+     case RECEIVE_DATA_SUCCESS:
+       return {
+         ...state,
+         isFetching: false,
+         characterArray: action.characterArray,
+       }
+     case RECEIVE_DATA_FAILED:
+       return {
+         ...state,
+         isFetching: false,
+       }
    default:
      return state
  }
}

const rootReducer = combineReducers({
  form: formReducer,
  characters: charactersReducer,
})

export default rootReducer
```

この変更に対応して、`actions.js` にも加筆します。

client/src/actions.js

```
export const CHANGE_NAME     = 'CHANGE_NAME'
export const CHANGE_AGE      = 'CHANGE_AGE'
export const INITIALIZE_FORM = 'INITIALIZE_FORM'
+ export const REQUEST_DATA         = 'REQUEST_DATA'
+ export const RECEIVE_DATA_SUCCESS = 'RECEIVE_DATA_SUCCESS'
+ export const RECEIVE_DATA_FAILED  = 'RECEIVE_DATA_FAILED'

export const changeName = name => ({
  type: CHANGE_NAME,
```

```
  name,
})
export const changeAge = age => ({
  type: CHANGE_AGE,
  age,
})
export const initializeForm = () => ({
  type: INITIALIZE_FORM,
})
+ export const requestData = () => ({
+   type: REQUEST_DATA,
+ })
+ export const receiveDataSuccess = characterArray => ({
+   type: RECEIVE_DATA_SUCCESS,
+   characterArray,
+ })
+ export const receiveDataFailed = () => ({
+   type: RECEIVE_DATA_FAILED,
+ })
```

最後に、component内のビューとイベントハンドラを書きます。
ビューをどのように表示するかを考えるためには、表示するべきデータである `characterArray` をどういう形をしているか決めておく必要があります。
今回は単純に、MongoDBのドキュメント

```
> use crud
switched to db crud
> db.characters.find().pretty()
{
        "_id" : ObjectId("5885a55875ffd05b30c67085"),
        "name" : "宮森あおい",
        "age" : 21,
        "__v" : 0
}
>
```

と同じ形のオブジェクトが単純にそのまま入っている `Array` にしましょう。すなわち、例えば以下のような `Array` です。

```
[
  {
    "_id" : ObjectId("5885a55875ffd05b30c67085"),
    "name" : "宮森あおい",
    "age" : 21,
    "__v" : 0
  },
  {
    "_id" : ObjectId("5885aa4852fabc5c7c832a20"),
    "name" : "安原絵麻",
    "age" : 21,
    "__v" : 0
  },
  {
    "_id" : ObjectId("5885aa5352fabc5c7c832a21"),
    "name" : "坂木しずか",
    "age" : 21,
    "__v" : 0
  }
]
```

このデータ構造を頭に入れた上で、 `<CharacterList />` は以下のように書けると思います。

client/src/components/CharacterList.js

```
import React from 'react'

const CharacterList = ({ store }) => {
+ const { isFetching, characterArray } = store.getState().characters


  return (
```

```
  return (
    <div>
+       {
+         isFetching  // isFetchingの値で分岐
+           ? <h2>Now Loading...</h2>  // データをFetch中ならばローディングアイコンを表示
+           : <div>
+               <ul>
+                 {characterArray.map(character => (
+                   <li key={character._id}>
+                     {`${character.name} (${character.age})`}
+                   </li>
+                 ))}
+               </ul>
+             </div>
+       }
    </div>
  )
}

export default CharacterList
```

三項演算子を使って、`isFetching`が`true`のときにはローディングアイコンを表示しています。
データの表示には、`characterArray`を`map()`して、名前と年齢を含んだ`<li>`要素を生成しています。`<li>`の中身は、例えば「宮森あおい (21)」のように表示するようになっていますね。

ここで、`<li>`に`key`というattributeが付いていることに注意してください、`key`がどういうもので何のために必要なのかについては公式のドキュメントがわかりやすいですが、とりあえず

- 表示するReact elementが`Array`に入っている時に要素を得意するための文字列
- ユニークでstable（`Array`の変更の影響を受けない）な文字列である必要がある

ということを覚えていれば十分だと思います。
例えば、`key`に`character.name`を指定した場合には同じ名前のキャラクターが複数いた際にユニークでなくなる可能性があり、`Array`の`index`を指定してしまうとキャラクターの削除・追加が起こった際にstableではなくなるので気を付けてください。これに対して、今回`key`として使っているMongoDBの`_id`はユニークであることが保証されており、かつデータに固有なためstableですので、非常に`key`に向いている変数です。

以上で、ビューを「どのように」表示するかという部分については完了しましたが、「いつ」表示するかがまだ決まっていません。普通に考えると、ページを開いたときに自動的に表示してほしいのですが、今回は簡単のため`<button>`要素を置いておき、これがクリックされた時にそのイベントハンドラとしてデータを`GET`リクエストし、表示するようにします（もし自動的に表示したい場合は`<CharacterList />`をクラスに書き換えてライフサイクルメソッドの`componentDidMount()`内でデータをリクエストすると良いと思います(React.jsのComponent Lifecycle)）。

`GET`リクエストをするので、`axios.get()`を使います。
リクエストが成功した場合は、受け取ったデータを引数にとった`receiveDataSuccess()`を`dispatch`することで、storeにデータを反映します。

client/src/components/CharacterList.js

```
import React from 'react'
+ import axios from 'axios'
+ import { requestData, receiveDataSuccess, receiveDataFailed } from '../actions'

const CharacterList = ({ store }) => {
  const { isFetching, characterArray } = store.getState().characters

+   const handleFetchData = () => {
+     store.dispatch(requestData())  // axios.get()を呼ぶ前にisFetchingをtrueにしておく
```

```
+     axios.get('/api/characters')
+     .then(response => {  // データ受け取りに成功した場合
+       const _characterArray = response.data
+       store.dispatch(receiveDataSuccess(_characterArray))    // データをstoreに保存すると
+     })
+     .catch(err => {  // データ受け取りに失敗した場合
+       console.error(new Error(err))
+       store.dispatch(receiveDataFailed())  // isFetchingをfalseに
+     })
+   }

  return (
    <div>
      {
        isFetching  // isFetchingの値で分岐
          ? <h2>Now Loading...</h2>  // データをFetch中ならばローディングアイコンを表示
          : <div>
+             <button onClick={() => handleFetchData()}>fetch data</button>
            <ul>
              {characterArray.map(character => (
                <li key={character._id}>
                  {`${character.name} (${character.age})`}
                </li>
              ))}
            </ul>
          </div>
      }
    </div>
  )
}

export default CharacterList
```

storeのデータを `<li>` 要素の中に表示するためのロジックはすでに書いてあるので、これで、 `axios.get()` によるstoreの更新後、自動的にキャラクターのデータが表示され

ることになります。

以上でクライアント側は完成です。
次に、サーバー側で `GET` リクエストに対処しましょう。

Expressの文法は、 `POST` リクエストとほとんど同じです。 `app.get()` で `GET` リクエストを受け付けた後、MongoDBからデータを取得し、クライアントに送り返します。
MongoDBからのデータの取得にはmongooseの `find()` メソッドを使います。

```
[モデル名].find(query, (error, documents) => {
  // documentsを使って処理する
})
```

今回は保存されているすべてのドキュメントを取得したいので、 `query` は必要ないです。 `server.js` は以下のようになります。

server.js

```
import express from 'express'
import bodyParser from 'body-parser'
import mongoose from 'mongoose'
import Character from './character'

const app = express()
const port = 3001
const dbUrl = 'mongodb://localhost/crud'

app.use(bodyParser.urlencoded({ extended: true }))
app.use(bodyParser.json())

mongoose.connect(dbUrl, dbErr => {
  if (dbErr) throw new Error(dbErr)
  else console.log('db connected')
```

```
  app.post('/api/characters', (request, response) => {
    const { name, age } = request.body

    new Character({
      name,
      age,
    }).save(err => {
      if (err) response.status(500)
      else response.status(200).send(`${name}(${age}) was successfully created.`)
    })
  })

+   app.get('/api/characters', (request, response) => {
+     Character.find({}, (err, characterArray) => {  // 取得したドキュメントをクライアント側
+       if (err) response.status(500).send()
+       else response.status(200).send(characterArray)  // characterArrayをレスポンスとして
+     })
+   })

  app.listen(port, err => {
    if (err) throw new Error(err)
    else console.log(`listening on port ${port}`)
  })
})
```

ここまでで、Readの機能が実装できたので、サーバーのプロセスを立ち上げ直してからブラウザを開いて確認してみましょう。

「fetch data」ボタンを押すことでサーバーのデータを表示できています。さらに、別のデータをCreateしたのちにそのデータをReadすることにも成功しています。

Update (`PUT`)

続いて、Updateの機能を実装します。
Updateでは、キャラクターの年齢を一つ上げます。

具体的には、クライアント側にUpdate機能に対応する `<button>` 要素を置いておき、そのイベントハンドラで、

- どの人物の年齢を上げたいのかを指定してサーバーにお願いする
- サーバー側でMongoDBにアクセスして年齢を上げる
- 成功した場合、あらためてすべてのデータをクライアントにレスポンスとして送る
- クライアント側で受け取ったデータをstoreに反映する

という処理をします。
サーバーからクライアントにレスポンスを返すとき、更新したデータのみを送り返せば通信データ量が減る上に、よりUpdateっぽい動作になるので本来はそうすべきかもしれませんが、今回は素朴に行くことにします（余裕がある人は方法を考えてみてください）。

Updateしたい人物を指定するのには、キャラクターを表すオブジェクトの `_id` を使いましょう。 `_id` はMongoDB内のドキュメントの `_id` でもあるので、サーバー側の処理

にもそのまま使えます。

クライアント側で書き換えるのは `CharacterList.js` のみです。Updateのための `<button>` とそのイベントハンドラを追加します。

client/src/components/CharacterList.js

```
import React from 'react'
import axios from 'axios'
import { requestData, receiveDataSuccess, receiveDataFailed } from '../actions'

const CharacterList = ({ store }) => {
  const { isFetching, characterArray } = store.getState().characters

  const handleFetchData = () => {
    store.dispatch(requestData())
    axios.get('/api/characters')
    .then(response => {
      const _characterArray = response.data
      store.dispatch(receiveDataSuccess(_characterArray))
    })
    .catch(err => {
      console.error(new Error(err))
      store.dispatch(receiveDataFailed())
    })
  }

+   const handleUpdateCharacter = id => {
+     store.dispatch(requestData())
+     axios.put('/api/characters', {
+       id,
+     })
+     .then(response => {
+       const _characterArray = response.data
+       store.dispatch(receiveDataSuccess(_characterArray))
+     })
+     .catch(err => {
```

```
+       console.error(new Error(err))
+       store.dispatch(receiveDataFailed())
+     })
+   }

  return (
    <div>
      {
        isFetching
          ? <h2>Now Loading...</h2>
          : <div>
              <button onClick={() => handleFetchData()}>fetch data</button>
              <ul>
                {characterArray.map(character => (
                  <li key={character._id}>
                    {`${character.name} (${character.age})`}
+                     <button onClick={() => handleUpdateCharacter(character._id)}>+1</bu
                  </li>
                ))}
              </ul>
            </div>
      }
    </div>
  )
}

export default CharacterList
```

追加したイベントハンドラは、Readのための `handleFetchData()` とほとんど変わらないことがわかります。

違いとしては、 `axios.get()` ではなく、Updateに対応する `PUT` リクエストのための `axios.put()` を呼んでいることのみです。

`axios.put()` の中では、 `axios.post()` と同様に第2引数にとったデータをサーバー

に送ることができます。今回は、Updateしたい人物の `_id` が入ったオブジェクトを送っています。

次に、サーバー側の処理を書きます。
サーバー側で年齢を更新するには、 `mongoose` の `findOneByIdAndUpdate()` メソッドを使います（本当はさらに引数に `option` をつけられますが今回は必要ないです。詳しい使い方は公式のドキュメントを見てください）。

```
[モデル名].findByIdAndUpdate(id, query, callback)
```

`server.js` に以下のように書き足します。 `PUT` リクエストに対応するので `app.put()` を使っています。中身は、 `GET` 、 `POST` に対する処理が複合したような形になっています。

server.js

```
import express from 'express'
import bodyParser from 'body-parser'
import mongoose from 'mongoose'
import Character from './character'

const app = express()
const port = 3001
const dbUrl = 'mongodb://localhost/crud'

app.use(bodyParser.urlencoded({ extended: true }))
app.use(bodyParser.json())

mongoose.connect(dbUrl, dbErr => {
  if (dbErr) throw new Error(dbErr)
  else console.log('db connected')

  app.post('/api/characters', (request, response) => {
```

```
    const { name, age } = request.body

    new Character({
      name,
      age,
    }).save(err => {
      if (err) response.status(500)
      else response.status(200).send(`${name}(${age}) was successfully created.`)
    })
  })

  app.get('/api/characters', (request, response) => {
    Character.find({}, (err, characterArray) => {
      if (err) response.status(500).send()
      else response.status(200).send(characterArray)
    })
  })

+   app.put('/api/characters', (request, response) => {
+     const { id } = request.body  // updateするキャラクターのidをリクエストから取得
+     Character.findByIdAndUpdate(id, { $inc: {"age": 1} }, err => {
+       if (err) response.status(500).send()
+       else {  // updateに成功した場合、すべてのデータをあらためてfindしてクライアントに送る
+         Character.find({}, (findErr, characterArray) => {
+           if (findErr) response.status(500).send()
+           else response.status(200).send(characterArray)
+         })
+       }
+     })
+   })

  app.listen(port, err => {
    if (err) throw new Error(err)
    else console.log(`listening on port ${port}`)
  })
})
```

`query`の書き方については今回は説明しないので、公式のドキュメント(MongoDBの$incについて)を見てください。とりあえず、`{ $inc: {"age": 1} }`が年齢を1つincrementしろという命令を表しています。

以上で完成です。
それでは、ブラウザを開いて試してみましょう。

Delete (`DELETE`)

最後に、Deleteの機能を実装します。
Deleteでは、指定したキャラクターの情報を削除します。

少し考えてみると、Updateの場合とほとんど同じ処理をすればいいことがわかると思います。すなわち、

- どの人物の情報を削除したいのかを`_id`で指定してサーバーにお願いする
- サーバー側でMongoDBにアクセスしてドキュメントを削除
- 成功した場合、あらためてすべてのデータをクライアントにレスポンスとして送る

という感じです。
以上の方針に従い、Updateの場合と同様に、Deleteするための`<button>`とそのイベン

トハンドラを実装しましょう。

ここで一点だけ注意があります。これまでの例から考えても、`axios` によるサーバーへのリクエストでは、

```
このコードは動きません

axios.delete('/api/characters', {
  id,
})
```

としたくなると思います。
しかし、公式のAPIの部分を見るとわかるように、`axios.delete()` では、`axios.post()` や `axios.put()` のように引数に送りたいデータを含めることができません。
これを回避するために、やりたいことはまったく同じですが、

```
axios({
  method: 'delete',
  url: '/api/characters',
  data: {
    id,
  }
})
```

とします。
ちなみに、`POST` や `GET` 、 `PUT` についても上記の例と同じように

```
axios([config])
```

という形でリクエストができますが、`axios.post()`などとした方がわかりやすいし簡単だと思うので今回はそうしています。

以上をふまえて、`CharacterList.js`に書き加えます。

client/src/components/CharacterList.js

```
import React from 'react'
import axios from 'axios'
import { requestData, receiveDataSuccess, receiveDataFailed } from '../actions'

const CharacterList = ({ store }) => {
  const { isFetching, characterArray } = store.getState().characters

  const handleFetchData = () => {
    store.dispatch(requestData())
    axios.get('/api/characters')
    .then(response => {
      const _characterArray = response.data
      store.dispatch(receiveDataSuccess(_characterArray))
    })
    .catch(err => {
      console.error(new Error(err))
      store.dispatch(receiveDataFailed())
    })
  }

  const handleUpdateCharacter = id => {
    store.dispatch(requestData())
    axios.put('/api/characters', {
      id,
    })
    .then(response => {
      const _characterArray = response.data
      store.dispatch(receiveDataSuccess(_characterArray))
    })
    .catch(err => {
```

```
      console.error(new Error(err))
      store.dispatch(receiveDataFailed())
    })
  }

+   const handleDeleteCharacter = id => {
+     store.dispatch(requestData())
+     // 気持ちとしては、axios.delete('/api/characters', { id })
+     axios({
+       method: 'delete',
+       url: '/api/characters',
+       data: {
+         id,
+       }
+     })
+     .then(response => {
+       const _characterArray = response.data
+       store.dispatch(receiveDataSuccess(_characterArray))
+     })
+     .catch(err => {
+       console.error(new Error(err))
+       store.dispatch(receiveDataFailed())
+     })
+   }

  return (
    <div>
      {
        isFetching
          ? <h2>Now Loading...</h2>
          : <div>
              <button onClick={() => handleFetchData()}>fetch data</button>
              <ul>
                {characterArray.map(character => (
                  <li key={character._id}>
                    {`${character.name} (${character.age})`}
                    <button onClick={() => handleUpdateCharacter(character._id)}>+1</butt
```

```
+                    <button onClick={() => handleDeleteCharacter(character._id)}>delete
                   </li>
                 ))}
               </ul>
             </div>
       }
     </div>
   )
}

export default CharacterList
```

次に、サーバー側です。

サーバー側の処理も、`PUT`に対する処理と同じような流れです。
MongoDBのドキュメントを削除するために、mongooseの`findByIdAndRemove()`を使います（詳しい使い方は公式のドキュメントを参照）。

```
[モデル名].findByIdAndRemove(id, callback)
```

その他の前後の処理は`PUT`と同じものを書いておくと、`server.js`は以下のようになります。

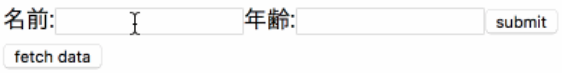
server.js

```
import express from 'express'
import bodyParser from 'body-parser'
import mongoose from 'mongoose'
import Character from './character'

const app = express()

const port = 3001
```

```
const dbUrl = 'mongodb://localhost/crud'

app.use(bodyParser.urlencoded({ extended: true }))
app.use(bodyParser.json())

mongoose.connect(dbUrl, dbErr => {
  if (dbErr) throw new Error(dbErr)
  else console.log('db connected')

  app.post('/api/characters', (request, response) => {
    const { name, age } = request.body

    new Character({
      name,
      age,
    }).save(err => {
      if (err) response.status(500)
      else response.status(200).send(`${name}(${age}) was successfully created.`)
    })
  })

  app.get('/api/characters', (request, response) => {
    Character.find({}, (err, characterArray) => {
      if (err) response.status(500).send()
      else response.status(200).send(characterArray)
    })
  })

  app.put('/api/characters', (request, response) => {
    const { id } = request.body
    Character.findByIdAndUpdate(id, { $inc: {"age": 1} }, err => {
      if (err) response.status(500).send()
      else {
        Character.find({}, (findErr, characterArray) => {
          if (findErr) response.status(500).send()
          else response.status(200).send(characterArray)

        })
      }
```

```
      }
    })
  })

+   app.delete('/api/characters', (request, response) => {
+     const { id } = request.body
+     Character.findByIdAndRemove(id, err => {
+       if (err) response.status(500).send()
+       else {
+         Character.find({}, (findErr, characterArray) => {
+           if (findErr) response.status(500).send()
+           else response.status(200).send(characterArray)
+         })
+       }
+     })
+   })

  app.listen(port, err => {
    if (err) throw new Error(err)
    else console.log(`listening on port ${port}`)
  })
})
```

動作確認をしてみましょう。



ふたたびCreate (`POST`)

ようやくCRUDアプリが完成しました。
しかし、実際に使って見ると少し違和感があると思います。

UpdateやDeleteの後は自動的にデータが更新されるのに、Createの後だけそうなっておらず、自分でfetch dataをクリックする必要があります。

この点を改善するには、`axios.post()`においても、`axios.put()`などと同様にすべてのデータをサーバーからのレスポンスとして送ってもらい、これをstoreに反映すれば良いです。
ここまでくればやるべきことは明らかだと思うので、コードの変更点だけ載せておきます。

client/src/components/AddForm.js

```
import React from 'react'
import axios from 'axios'
import {
  changeName, changeAge, initializeForm,
+   requestData, receiveDataSuccess, receiveDataFailed
} from '../actions'


const AddForm = ({ store }) => {
  const { name, age } = store.getState().form

  const handleSubmit = e => {
    e.preventDefault()
```

```
+     store.dispatch(requestData())
    axios.post('/api/characters', {
      name,
      age,
    })
    .then(response => {
-       console.log(response)
      store.dispatch(initializeForm())
+       const characterArray = response.data
+       store.dispatch(receiveDataSuccess(characterArray))
    })
    .catch(err => {
      console.error(new Error(err))
+       store.dispatch(receiveDataFailed())
    })
  }

  return (
    <div>
      <form onSubmit={e => handleSubmit(e)}>
        <label>
          名前:
          <input value={name} onChange={e => store.dispatch(changeName(e.target.value))}
        </label>
        <label>
          年齢:
          <input value={age} onChange={e => store.dispatch(changeAge(e.target.value))} />
        </label>
        <button type="submit">submit</button>
      </form>
    </div>
  )
}

export default AddForm
```

server.js

```
import express from 'express'
import bodyParser from 'body-parser'
import mongoose from 'mongoose'
import Character from './character'

const app = express()
const port = 3001
const dbUrl = 'mongodb://localhost/crud'

app.use(bodyParser.urlencoded({ extended: true }))
app.use(bodyParser.json())

mongoose.connect(dbUrl, dbErr => {
  if (dbErr) throw new Error(dbErr)
  else console.log('db connected')

  app.post('/api/characters', (request, response) => {
    const { name, age } = request.body

    new Character({
      name,
      age,
    }).save(err => {
      if (err) response.status(500)
-       else response.status(200).send(`${name}(${age}) was successfully created.`)
+       else {
+         Character.find({}, (findErr, characterArray) => {
+           if (findErr) response.status(500).send()
+           else response.status(200).send(characterArray)
+         })
+       }
    })
  })


  app.get('/api/characters', (request, response) => {
```

```
    Character.find({}, (err, characterArray) => {
      if (err) response.status(500).send()
      else response.status(200).send(characterArray)
    })
  })

  app.put('/api/characters', (request, response) => {
    const { id } = request.body
    Character.findByIdAndUpdate(id, { $inc: {"age": 1} }, err => {
      if (err) response.status(500).send()
      else {
        Character.find({}, (findErr, characterArray) => {
          if (findErr) response.status(500).send()
          else response.status(200).send(characterArray)
        })
      }
    })
  })

  app.delete('/api/characters', (request, response) => {
    const { id } = request.body
    Character.findByIdAndRemove(id, err => {
      if (err) response.status(500).send()
      else {
        Character.find({}, (findErr, characterArray) => {
          if (findErr) response.status(500).send()
          else response.status(200).send(characterArray)
        })
      }
    })
  })

  app.listen(port, err => {
    if (err) throw new Error(err)
    else console.log(`listening on port ${port}`)



  })
})
```

```
});
```

無事、submit後に自動的にデータが更新されるようになりました。

（おまけ）Herokuにデプロイしてみる

作ったアプリを公開したくなったとしましょう。Herokuを使います。
今回のアプリはとくに公開するようなものではありませんが、同じような構成でアプリを書いてとりあえず公開したいというときのための簡単なやり方を書いておきます。

Herokuでは、自分でサーバーを立てる際の面倒な作業が必要ありません。gitのリモートブランチとしてHerokuを登録しておいて、`git push`すれば自動的にデプロイされるという非常にありがたいサービスです。Heroku自体を使ったことがない人は、Getting Started on Heroku with Node.jsを一度やってみるとよいと思います。
Heroku CLIがインストールされていない場合はSet upを見てインストールしておいてください。

今回のアプリを公開するためにやることは以下です。

- クライアントのコードをビルドする
- ビルドしたコードをサーバーが提供するようにする
- MongoLabの設定

- Herokuにデプロイ

まずはクライアント側のコードをビルドしましょう。

```
cd client
npm run build
```

ビルドが終わると、`client/build`というディレクトリができていると思います。この中の`static/js`ディレクトリには、クライアント側の（React componentsも含めたすべての）javascriptコードをビルドした`main.*.js`というファイルが入っていると思います。この`.js`ファイルは依存しているライブラリも含んでいるので、この`.js`ファイルだけ置いておけば動作してくれます。
`client/build`の`index.html`を見ると、この`.js`ファイルが読み込まれていることがわかります。

サーバー側ではこの`build`ディレクトリをブラウザに返すようなコードを書けば良いことになります。`express.static()`で、ブラウザに静的ファイルを返すことができます（express.static()）。

server.js

```
import express from 'express'
import bodyParser from 'body-parser'
+ import path from 'path'
import mongoose from 'mongoose'
import Character from './character'

const app = express()
- const port = 3001
+ const port = process.env.PORT || 3001 // herokuの環境変数で指定されるportを使う

const dbUrl = 'mongodb://localhost/crud'
```

```
+ app.use(express.static(path.join(__dirname, 'client/build')))
app.use(bodyParser.urlencoded({ extended: true }))
app.use(bodyParser.json())

mongoose.connect(dbUrl, dbErr => {
  if (dbErr) throw new Error(dbErr)
  else console.log('db connected')

  app.post('/api/characters', (request, response) => {
    const { name, age } = request.body

    new Character({
      name,
      age,
    }).save(err => {
      if (err) response.status(500)
      else {
        Character.find({}, (findErr, characterArray) => {
          if (findErr) response.status(500).send()
          else response.status(200).send(characterArray)
        })
      }
    })
  })

  app.get('/api/characters', (request, response) => {
    Character.find({}, (err, characterArray) => {
      if (err) response.status(500).send()
      else response.status(200).send(characterArray)
    })
  })

  app.put('/api/characters', (request, response) => {
    const { id } = request.body
    Character.findByIdAndUpdate(id, { $inc: {"age": 1} }, err => {

      if (err) response.status(500).send()
```

```
      else {
        Character.find({}, (findErr, characterArray) => {
          if (findErr) response.status(500).send()
          else response.status(200).send(characterArray)
        })
      }
    })
  })

  app.delete('/api/characters', (request, response) => {
    const { id } = request.body
    Character.findByIdAndRemove(id, err => {
      if (err) response.status(500).send()
      else {
        Character.find({}, (findErr, characterArray) => {
          if (findErr) response.status(500).send()
          else response.status(200).send(characterArray)
        })
      }
    })
  })

  app.listen(port, err => {
    if (err) throw new Error(err)
    else console.log(`listening on port ${port}`)
  })
})
```

ここで、herokuでは環境変数として自動的に割り当てられるポート番号を使わなければいけないので、そのように書き換えてあります（node.jsでは `process.env.[変数名]` で環境変数を読んでくれます）。
ローカルでは（とくに何もしていなければ） `PORT` という環境変数は定義されていないと思うので、今まで通りポート番号として `3001` が使われます。

今までは、クライアント側とサーバー側両方のプロセスを同時に立ち上げていましたが、上記の変更で、サーバー側だけ立ち上げておけば、いままでクライアント側で提供していたものをブラウザに表示してくれるようになります。試してみましょう。

```
npm start // サーバー側のみ
```

`http://localhost:3001` を開くとアプリが表示されていると思います。これで、サーバー側からビルドしたのクライアント側のコードを提供できています。

Herokuの設定をしていきます。
先ほども述べましたが、Herokuを使うためにはgitが必要です。管理したくないファイルを `.gitignore` を書いておきます。

.gitignore

```
node_modules
npm-debug.log
.DS_Store
// その他管理したくないファイル
```

`git init` しましょう。

```
git init
git add .
git commit -m "init"
```

次に、Herokuから使うMongoDBの設定をしましょう。開発段階では `localhost` のMongoDBを使っていましたが、当然Herokuではそうはいきません。
HerokuでMongoDBを使うためには、 `heroku create` でアプリを作成してから、mongolabのアドオンを追加します。

```
heroku create // heroku appを作成
heroku addons:create mongolab // addonとしてMongoLabを追加。
heroku config | grep MONGODB_URI // heroku configにMONGODB_URIが表示されているのでgrep
// ここで'mongodb://'から始まるMONGODB_URIが表示されるはず
```

ここで、`server.js`の中の、MongoDBのURLを表す変数である`dbUrl`を、現在の`localhost`のものから、表示された'mongodb://'から始まる`MONGODB_URI`に書き換えてください。もしくは直接環境変数の`MONGODB_URI`を使っても良いです（この場合そのままだとローカル環境から動かせなくなりますが、dotenvというライブラリを使えば対処できます）。

```
- const dbUrl = 'mongodb://localhost/crud'
// いずれかに変更
+ const dbUrl = 'mongodb://heroku------'  // grepで表示されたurl
+ const dbUrl = process.env.MONGODB_URI
```

それでは、Herokuにデプロイしましょう。まず、Herokuで実行されるべきコマンドを書く`Procfile`というファイルを作成する必要があります。以下の一行のみ書いておきます。

```
web: npm start
```

詳しい書き方（「web:」とはなんなのかなど）はProcess Types and the Procfileを参照してください。

herokuの設定をしたときに、リモートレポジトリにherokuが登録されているはずなので、コミットしてから`push`します。
以上で完了です。実際にサイトを開いて動かしてみましょう。

```
git add .
git commit -m "modify MONGODB_URI"
git push heroku master
heroku open
```

まとめ

とりあえず動くCRUDアプリが作成できました。
ただし、最初にも述べたように今回はCRUDアプリにとって最小限のことしか気にしておらず、一般的なベストプラクティスからはほど遠いコードになっています。以下で、次のステップとして気にした方が良さそうなものを挙げていきます。

今回は気にしていないけど気にした方が良さそうなもの

Presentational / Container componentsへの分割

今回作ったReact components（`<AddForm />`、`<CharacterList />`）は、

- ロジック（データの処理・イベントハンドリング）
- ビュー（データの表示）

を両方担当しています。

Reduxの考え方では、このようなcomponentsは、ロジックを担当するContainer components とビューを担当するPresentational componentsに分けるべきだとされています。今回のような小さなアプリではこの分割をするとかえってわかりづらくなってしまう気がしますが、より複雑なアプリを作りたいときはその方針に従った方がよいと思います。

以上の点に関して、下記のページが参考になると思います。とくに、reduxの作者による Getting Started with Reduxのvideoを見ると、この点に限らずReduxの基本的な考え方がわかるので、もし見ていない人は見ると良いと思います（英語のリスニングが厳しくても、コードを見ていればだいたいやりたいことがわかると思います）。

- Getting Started with Redux
- Presentational and Container Components
- React+Redux入門

非同期処理について

今回、非同期であるサーバーとの通信の処理を、すべてReact componentの中のイベントハンドラ中で書いています。このような書き方は、アプリが複雑になっていくにつれてコードの見通しが悪くなってしまうのと、重複したコードを書くことになる可能性があるので、（たぶん）あまりよくないと思います。

この問題については、redux-thunkやredux-sagaといった非同期処理を扱うReduxのMiddlewareを使うことで解決します。

まずMiddlewareとはなんなのか知りたいという場合は、公式の

- Middleware

が、redux-thunkについては公式のドキュメントが、redux-sagaについては

- redux-sagaで非同期処理と戦う

が参考になると思います。

デバッグ

今回は、storeの状態遷移を知るために、`store.subscribe()`の中でのコンソール出力を使っていました。同様の目的で、ReduxのMiddlewareであるredux-loggerを使うと良いです。
使い方は簡単なので、ドキュメントを軽く読めば十分だと思います。

その他

- Reactのライフサイクルメソッドを使ってページを開いた時点でサーバーからデータを取ってくるようにする
- フォーム入力中にリアルタイムでvalidationをする（たとえば、年齢のフォームに数字以外の文字が入力されているときはsubmitボタンを`disable`するなど）
- webpackを使ってコードをビルドする
- サーバー側のファイルを更新した際にいちいち`npm start`し直さなくてもいいようにwebpack-dev-serverやnodemonを使う

他にもいろいろあると思いますが、とりあえずこの辺で...。