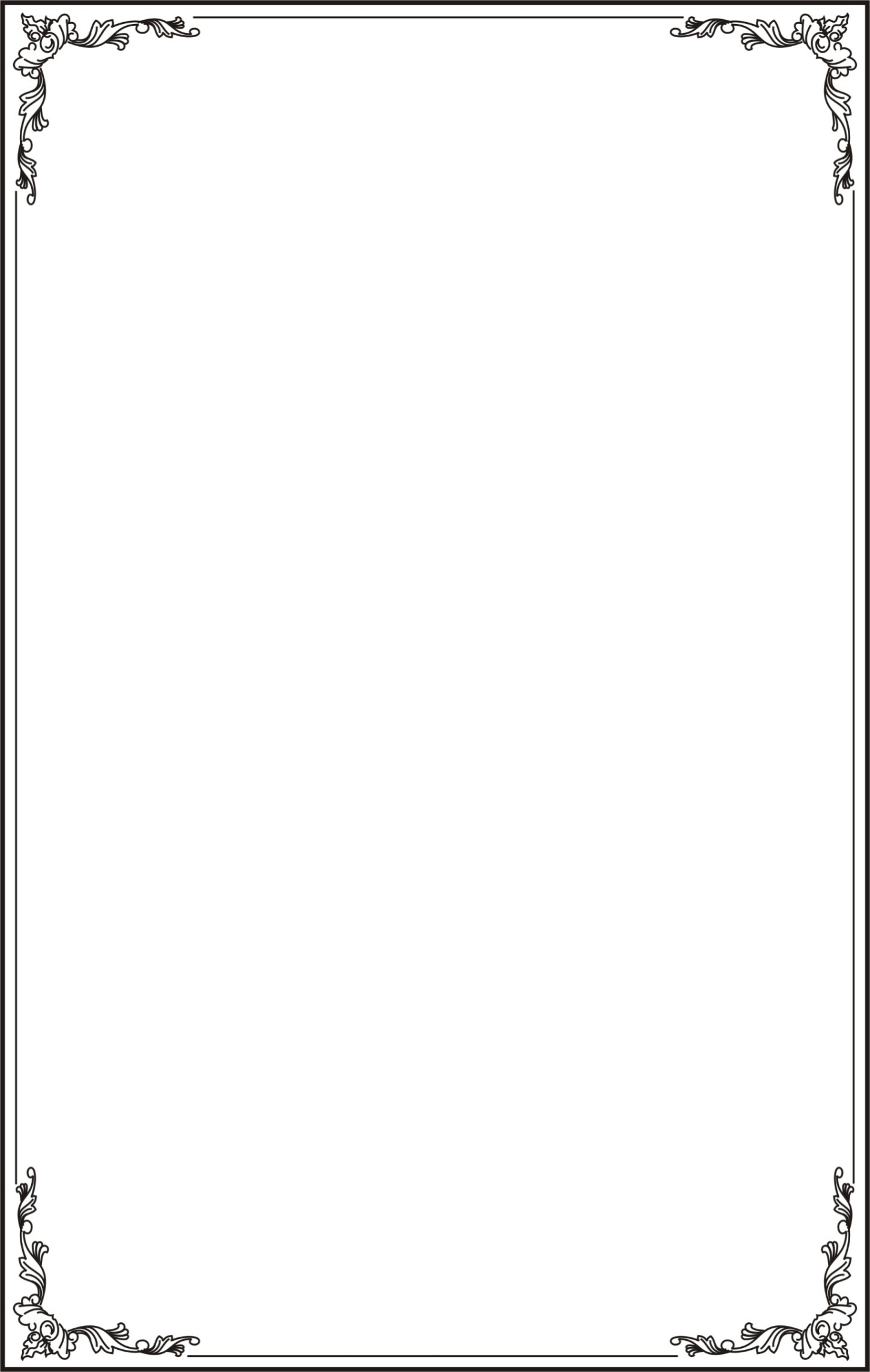
**CÔNG TY CỔ PHẦN**

**VIỄN THÔNG FPT**

****

**BÀI THU HOẠCH**

**ĐÀO TẠO TÂN BINH R&D TUẦN 3 & 4**

|  |  |
| --- | --- |
| Người hướng dẫn: | **MR. TRƯƠNG TẤN SANG** |
| Người thực hiện: | **TRẦN KHÁNH GIA UY** |
| Đơn vị công tác: | **Trung tâm giám sát và đảm bảo dịch vụ (SCC)** |
| Vị trí: | **Open Source Web Software Developer** |
| Bộ phận: | **SCC** |

***Thành phố Hồ Chí Minh, tháng 05 năm 2021***

# MỤC LỤC

[MỤC LỤC 2](#_Toc74511518)

[CHƯƠNG 1: TÌM HIỂU VỀ AXIOS, ROUTER, ANT DESIGN VÀ REDUX 2](#_Toc74511519)

[1.1 Axios 2](#_Toc74511520)

[1.2 Router 5](#_Toc74511521)

[1.3 Ant Design 7](#_Toc74511522)

[CHƯƠNG 2: TỐI ƯU TODOS VÀ TÌM HIỂU VỀ PHÂN TRANG TRONG ANT DESIGN 9](#_Toc74511523)

[2.1 Tối ưu todos 9](#_Toc74511524)

[2.3 Source code 10](#_Toc74511525)

[2.4 Demo 10](#_Toc74511526)

[CHƯƠNG 3: THỰC HIỆN DEMO REDUX 11](#_Toc74511527)

[3.1 Ý tưởng 11](#_Toc74511528)

[3.2 Cấu trúc thư mục 11](#_Toc74511529)

[3.3 Source code 13](#_Toc74511530)

[3.4 Demo 13](#_Toc74511531)

[3.5 Các vấn đề gặp phải 13](#_Toc74511532)

[3.6 Hướng đi tiếp theo 14](#_Toc74511533)

# CHƯƠNG 1: TÌM HIỂU VỀ AXIOS, ROUTER, ANT DESIGN VÀ REDUX

## Axios

**1.1.1 Get request**

* + - Đây là phương thức để lấy toàn bộ data về, chính vì thế get request thường được đặt trong life cycle componentDidMount để lấy dữ liệu :

  componentDidMount() {

    axios

      .get("https://60b0f8b91f26610017fff943.mockapi.io/api/v1/todo\_data")

      .then((res) => {

        this.setState({ todoItems: res.data });

      })

      .catch((err) => {

        console.error(err);

      });

  }

**1.1.2 Post request**

* + - Post request dùng để tạo dữ liệu mới

Axios.post("https://60b0f8b91f26610017fff943.mockapi.io/api/v1/todo\_data", {

      id: n,

      title: event.target.value,

      isEditing: false,

      isCompleted: false,

    })

    .then((res) => {

      console.log(res);

    })

    .catch((err) => {

      console.log(err);

});

**1.1.3 Put request**

* + - Put request dùng để cập nhật lại dữ liệu theo id

axios.put("https://60b0f8b91f26610017fff943.mockapi.io/api/v1/todo\_data/" + item.id,

{

title: text,

})

.then((res) => {

console.log(res.data);

})

.catch((err) => {

console.log(err);

});

**1.1.4 Delete request**

* + - Delete request dùng để xóa dữ liệu theo id

axios.delete("https://60b0f8b91f26610017fff943.mockapi.io/api/v1/todo\_data/" +

item.id

)

.then((response) => {

console.log(response);

});

**1.1.5 Vấn đề về too many request**

* + - Khi thực hiện gửi đồng loạt nhiều request trong một lúc cho server thì sẽ gặp hiện tượng lỗi too many request (error 429). Để khắc phục vấn đề này ta sẽ setTimeout cho mỗi lần lặp gửi request.

for (let i = 0; i < todoItem.length; i++) {

  setTimeout(() => {

    axios.put(

      "https://60b0f8b91f26610017fff943.mockapi.io/api/v1/todo\_data/" +

        todoItems[i].id,

      {

        isCompleted: todoItems[i].isCompleted,

      }

    )

    .then((res) => {

      console.log(res.data);

    })

    .catch((err) => {

      console.log(err);

    });

  }, 5000);

}

* + - Tuy nhiên trong JS, hàm setTimeout chỉ delay được duy nhất 1 lần trườc khi thực hiện hàm chính vì thế vẫn xuất hiện lỗi too many request
    - Để giải quyết vấn đề trên, ta sẽ tạo luôn một hàm để lặp và gọi lại hàm đó trong mỗi lần lặp.

function myLoop() {

      setTimeout(function () {

        axios

          .put(

            "https://60b0f8b91f26610017fff943.mockapi.io/api/v1/todo\_data/" +

              todoItems[i].id,

            {

              isCompleted: todoItems[i].isCompleted,

            }

          )

          .then((res) => {

            console.log(res.data);

          })

          .catch((err) => {

            console.log(err);

          });

        i++; //  increment the counter

        if (i < n) {

          myLoop();

        }

      }, 700);

    }

* + - Khi đó, hàm myLoop sẽ được lặp lại sau 700ms
    - Tuy nhiên cách làm này chưa được tối ưu, bởi vì khi muốn thay đổi thời gian timeout phải sửa lại toàn bộ hàm. Chính vì thế ta sẽ sửa lại hàm này với async await.

async function wait(ms) {

      return new Promise((resolve) => {

        setTimeout(resolve, ms);

      });

    }

    async function updateAPI(todoList) {

      for (const item of todoList) {

        await axios

          .put(

            "https://60b0f8b91f26610017fff943.mockapi.io/api/v1/todo\_data/" +

              item.id,

            {

              isCompleted: item.isCompleted,

            }

          )

          .then((res) => {

            console.log(res.data);

          })

          .catch((err) => {

            console.log(err);

          });

        await wait(5000);

      }

    }

* + - Với hàm wait, ta có thể tùy ý cài đặt thời gian timeout mà không cần viết lại hàm

### Router

**1.2.1 Cài đặt**

* + - Thực hiện lệnh sau trong terminal: npm install react-router-dom
    - Để điều hướng chúng ta sử dụng các thẻ Router, Switch, Route, Link. Ví dụ:

<Router>

  <div>

    <Link to="/">Home</Link>

    <Link to="/about">About</Link>

    <Link to="/topics">Topics</Link>

  </div>

  <Switch>

    <Route exact path="/about">

      <About />

    </Route>

    <Route exact path="/topics">

      <Topics />

    </Route>

    <Route exact path="/">

      <Home />

    </Route>

  </Switch>

</Router>;

* + - Tất cả các phần để điều hướng được đặt trong thẻ Router. Thẻ Switch dùng để điều hướng khi nhấn vào các thẻ Link

**1.2.2 Sử dụng trong Todo**

* + - Trong Todo có 3 loại filter là All (tất cả các công việc), Active (các công việc chưa hoàn thành), Completed (các công việc đã hoàn thành). Vậy nên ta sẽ chuyển 3 nút này thành 3 route.

**Trong app.js**

<Router>

<TodoInput onKeyUp={this.onPressEnter} chooseAll={this.chooseAll} />

   <Switch>

    <Route exact path="/">

    <TodoGeneral path="" todoState={this.state} />

    </Route>

    <Route exact path="/active">

    <TodoGeneral path="active" todoState={this.state} />

    </Route>

    <Route exact path="/completed">

    <TodoGeneral path="completed" todoState={this.state} />

    </Route>

   </Switch>

   <TodoFilter numItem={numItem} clearCompleted={this.clearCompleted} />

</Router>

**Trong TodoFilter.js**

<div className="TodoFilter">

<p className="numItem">{numItem} left</p>

   <div className="filter" onClick={routeFunc}>

    <Link to="/">All</Link>

        <Link to="/active">Active</Link>

        <Link to="/completed">Completed</Link>

   </div>

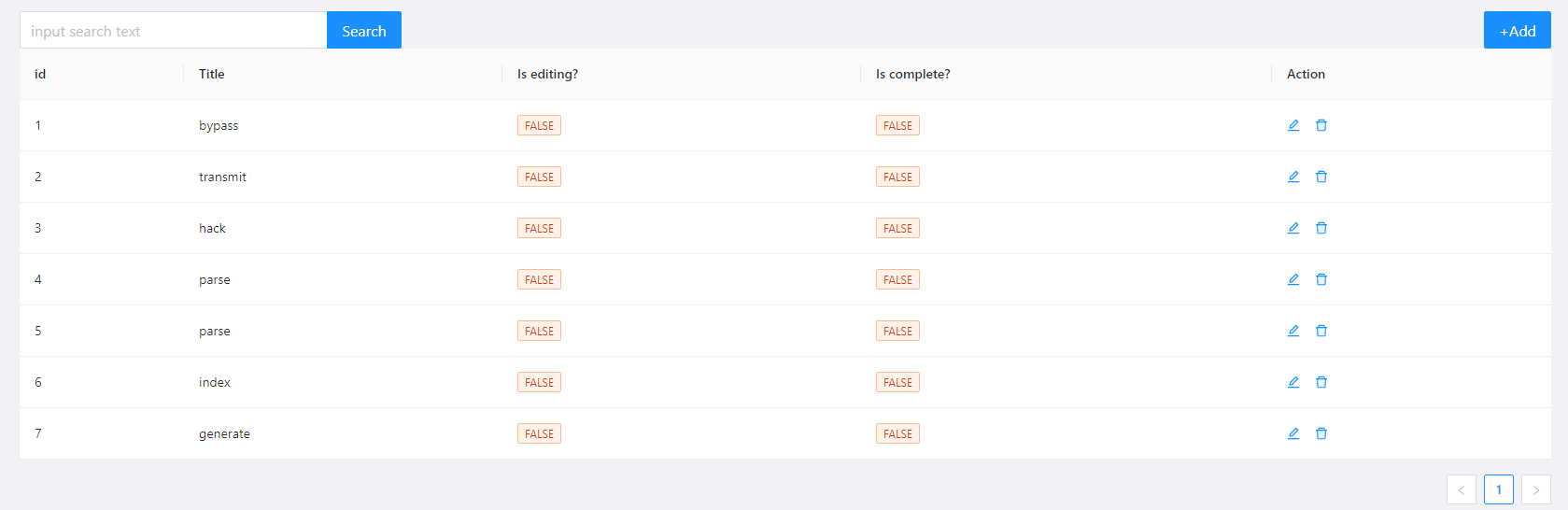
   <button className="clearCompleted" onClick={clearCompleted}>Clear completed</button>

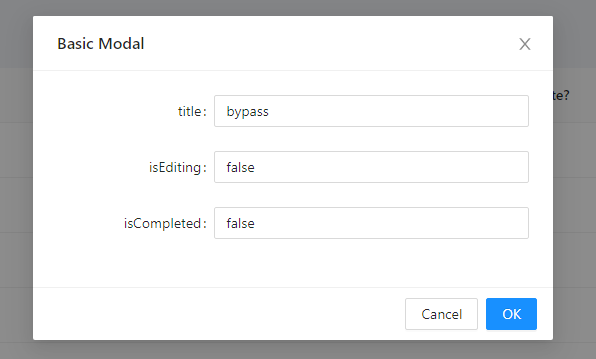
</div>

### Ant Design

**1.3.1 Cài đặt vào project**

* + - Ta sẽ sử dụng Ant Design để tạo table và navbar điều hướng cho app.
    - Table ở đây sẽ lấy data từ Mock api về và cho phép người dùng thêm xóa sửa cũng như tìm kiếm trên table
    - Trước hết ta định nghĩa table gồm các cột sau :
      * ID : mã todo
      * Title : tên todo
      * Is editing ? : trạng thái isEditing của todo (true/false)
      * Is completed ?: trạng tháng isCompleted của todo (true/false)
      * Action : sửa hay xóa todo
    - Sau đó ta tạo các nút thêm và tìm kiếm trong table
    - Cuối cùng ta tạo các modal để sửa và thêm todo

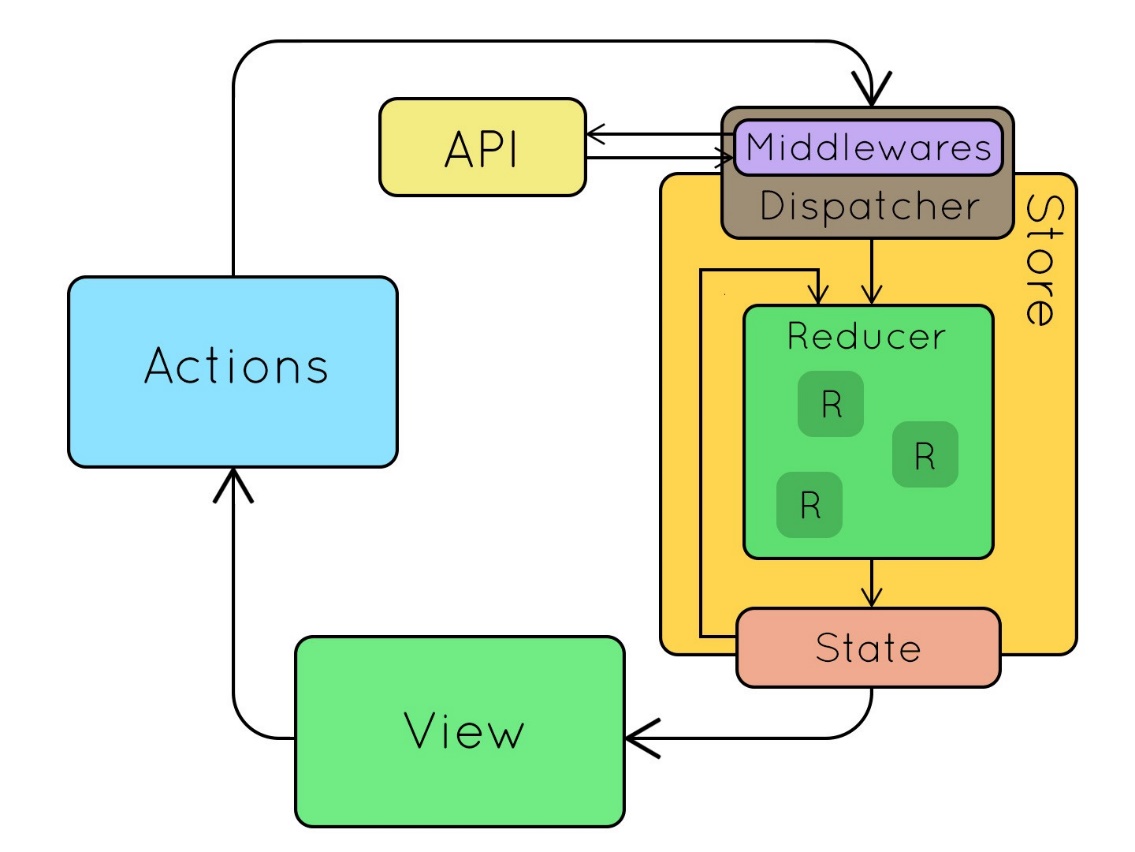
**1.3.2 Demo**

****

* 1. **Redux**

**1.4.1 Giới thiệu**

* + - Redux ra đời với mục đích quản lý state có thể dự đoán được. cung cấp một vùng chứa state duy nhất và các quy tắc nghiêm ngặt về cách có thể thay đổi state.
    - Có thể sử dụng Redux cho React, Angular, …



**1.4.2 Concept chính**

* + - Store : đây là nơi lưu trữ state và chỉ nên có 1 store trong 1 app
    - Action: một đối tượng JS đơn thuần. Action mô tả rõ ràng lý do tại sao thay đổi state và có thể thực hiện ở bất kỳ đâu trong app. Ví dụ:

{

  type: "ADD\_STUDENT",

  name: "Thu Hương"

}

* + - Reducer: là thành phần để liên kết store với action. Nó nhận state và action làm đối số và trả về state tiếp theo của ứng dụng.
    - Một reducer có thể xử lý nhiều action chính vì thế thường sử dụng cấu trúc switch trong reducer
    - Tốt nhất là cho mỗi reducer quản lý 1 phần state
    - Ví dụ:

function studentsApp(state, action) {

  // Câu lệnh switch

  // Kiểm tra type và trả về sate mới

  switch (action.type) {

    case "ADD\_STUDENT":

      return [ ...state,  action.name ]

    default:

      // Không có case nào khớp

      // thì trả về state hiện tại

      return state

}

# CHƯƠNG 2: TỐI ƯU TODOS VÀ TÌM HIỂU VỀ PHÂN TRANG TRONG ANT DESIGN

* 1. **Tối ưu todos**

**2.1.1 Vấn đề về component cha và con và thêm proptypes vào các component**

* Hiện tại các xử lý logic chính nằm ở trong component con, vì thế để tiện cho việc dò lỗi và update, nên chuyển tất cả các logic đó cho component cha và các component con chỉ nhận props từ component cha.
* Để đảm bảo cho việc mỗi props truyền vô component là chính xác, việc thêm proptypes vào các components là cần thiết. Ví dụ :

import PropTypes from "prop-types";

TodoFilter.propTypes = {

  numItem: PropTypes.number,

  clearCompleted: PropTypes.func,

  onClickAll: PropTypes.func,

  onClickActive: PropTypes.func,

  onClickCompleted: PropTypes.func,

};

**2.1.2 Vấn đề về tối ưu logic update và search**

* Mỗi lần update và search trong table đều phải get data lại để hiển thị trong màn hình. Điều này làm chương trình chạy rất chậm và không hiệu quả.
* Trong Mockapi, khi ta thực hiện một method bất kì nó sẽ trả về data của method đó, chính vì thế, việc lấy data từ response của Mockapi để hiển thị data sẽ giúp cải thiện tốc độ rất nhiều

**2.1.3 Vấn đề về validate data**

* Trong Ant Design đã cũng cấp 1 số tính năng để validate data khi nhập vào bảng. Ví dụ trong một form thì props “rules” sẽ quy định data khi nhập vô sẽ thuộc kiểu nào.

rules={[

{

   required: true,

    pattern: new RegExp("([0-9]\\s\*)+"),

    message: "please input phone num",

  },

* 1. **Phân trang trong Ant Design**

**2.2.1 Tại sao phải phân trang ?**

* Trong table của Ant Design đã tích hợp sẵn công cụ phân trang. Chỉ cần truyền dữ liệu cho table thì lập tức nó sẽ phân trang theo dữ liệu đó.
* Tuy nhiên với một dữ liệu cực lớn (bảng có 1 triệu dòng chẳng hạn) việc load toàn bộ data này sẽ rất mất thời gian khi tải trang. Chính vì thế nên phân thành những trang nhỏ để mỗi lần chuyển trang sẽ lấy đúng data mà trang đó có.

**2.2.2 Cách phân trang khi lấy dữ liệu từ api**

* Phân trang trong Ant Design cần thuộc tính “pagination”, ngoài ra, cần thêm thuộc tính “onChange” để có thể xử lý logic khi chuyển trang.

<Table

        columns={columns}

        dataSource={data}

        pagination={tablePagination}

        loading={loading}

        onChange={handleTableChange}

      />

* Đây là hàm xử lý khi thay đổi một trang. Hàm fetch() có nhiệm vụ lấy dữ liệu theo trang được truyền vô

const handleTableChange = (page, pageSize) => {

    let currentPage = {

      current: (page.current - 1) \* page.pageSize,

      pageSize: pageSize,

    };

    console.log(page);

    fetch(currentPage);

  };

const tablePagination = {

    pageSize: paramsContact.limit,

    page: paramsContact.offset / paramsContact.limit + 1,

    total: paramsContact.total,

    pageSizeOptions: ["5", "10", "15", "20", "25"],

    showSizeChanger: true,

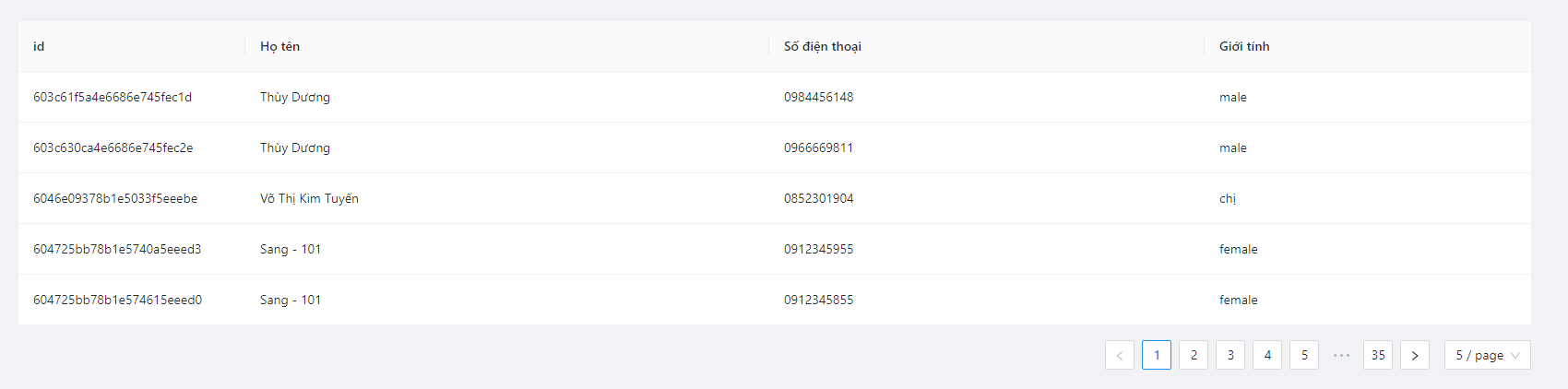
  };

* 1. **Source code**

Tất cả source code nằm trong link github sau:

[internSCC/React/todo-list-optimize at main · uytran36/internSCC (github.com)](https://github.com/uytran36/internSCC/tree/main/React/todo-list-optimize)

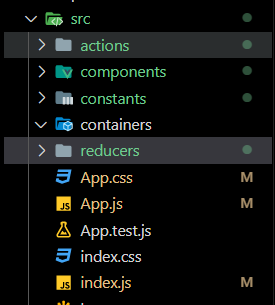
* 1. **Demo**

****

# CHƯƠNG 3: THỰC HIỆN DEMO REDUX

* 1. **Ý tưởng**
* Em sẽ xây dựng app cho người dùng đăng ký, đăng nhập. Nếu đăng nhập thành công thì sẽ chuyển đến trang chính để có thể thêm, xóa, sửa dữ liệu.
* Phần đăng ký em sẽ sử dụng thư viện “bcrypt” để mã hóa mật khẩu và lưu vô database. Đồng thời chuyển cặp giá trị username, password thành một token bằng jwt và lưu lại trong database.
* Khi người dùng đăng nhập, mật khẩu sẽ được so sánh với mật khẩu đã mã hóa. Nếu đúng sẽ cho người dùng quyền truy cập vào trang chính đồng thời đưa cho người dùng token và lưu vào trong local storage.
* Khi đăng xuất thì app sẽ xóa token đó trong local storage và chuyển về trang đăng nhập
* Trang chính sẽ hiển thị thông tin người dùng bao gồm họ tên, số điện thoại, giới tính

## 3.2 Cấu trúc thư mục

****

* Thông thường sẽ có 2 cách chia thư mục: Function-first và Feature-first. Nhưng ở đây em làm theo Function-first.
* Trong *index.js*, em sẽ tạo store đồng thời apply middle *Thunk* để xử lý async await function vào project.

**3.2.1 Actions**

* Đây là nơi đặt tất cả các actions, xử lý logic của app. Em chia ra 2 file action chính là *auth.js* để xử lý đăng ký, đăng nhập và *contacts.js* để xử lý logic các chức năng thêm xóa sửa dữ liệu.
* Với action đăng ký em sẽ trả về tên của action này và state của nó sau khi thực hiện action

export const register = (user) => {

  return {

    type: types.REGISTER,

    user,

  };

};

* Đồng thời, em sẽ khai báo một hàm xử lý logic và gọi action này mỗi khi nhấn đăng ký

export const registerRequest = (user) => {

  return (dispatch) => {

    const salt = bcrypt.genSaltSync(10);

    const hassPassword = bcrypt.hashSync(user.password,salt);

    const token = jwt.sign(

      {

        username: user.username,

        password: hassPassword,

      },

      "SECRET\_TOKEN"

    );

    const data = {

      username: user.username,

      password: hassPassword,

      token: token,

    };

    axios

      .post("https://60b0f8b91f26610017fff943.mockapi.io/api/v1/users", data)

      .then((response) => {

        dispatch(register(response.data));

      })

      .catch((err) => {

        console.log(err);

      });

  };

};

* Tương tự như vậy với đăng nhập và các actions khác.

**3.2.2 Components**

* Đây sẽ là nơi khai báo tất cả các component và nếu muốn gọi một action thì em sẽ gọi thông qua props. Để làm được điều này em sẽ thêm một hàm là *mapDispatchToProps* và sử hàm *connect()* để nối dispatch props với lại component. Ví dụ cho form đăng nhập:

const mapDispatchToProps = (dispatch) => {

  return {

    loginRequest: (user) => {

      dispatch(loginRequest(user));

    },

  };

};

export default connect(null, mapDispatchToProps)(FormLogin);

**3.2.3 Constants**

* Đây là nơi khai báo tên của những actions có trong app, mục đích là để mỗi lần gọi tên thì sẽ tránh trường hợp gọi nhầm tên và dễ kiểm soát lỗi hơn.

export const SET\_CURRENT\_USER = "SET\_CURRENT\_USER";

export const LOGIN = "LOGIN";

export const REGISTER = "REGISTER";

export const FETCH\_CONTACT = "FETCH\_CONTACT";

export const ADD\_CONTACT = "ADD\_CONTACT";

export const EDIT\_CONTACT = "EDIT\_CONTACT";

export const DELETE\_CONTACT = "DELETE\_CONTACT";

export const SEARCH\_CONTACT = "SEARCH\_CONTACT";

**3.2.4 Reducers**

* Đây sẽ là nơi quản lý state đầu vào, đầu ra. Bởi vì không nên để một file quản lý tất cả các state nên em sẽ chia thành nhiều file nhỏ quản lý những nhóm state. Cuối cùng em sẽ gộp lại bằng hàm *combineReducer()*

export default combineReducers({

  loginReducer,

  registerReducer,

  contactReducer,

  searchContactReducer,

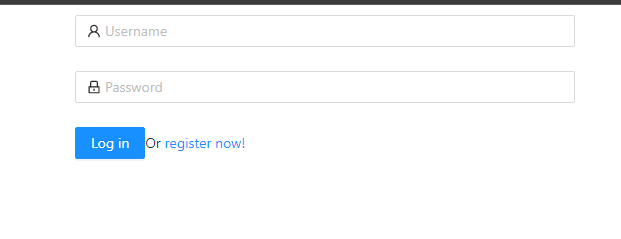
  auth

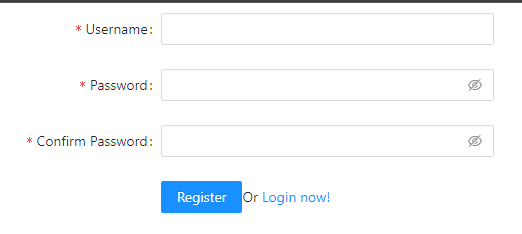
});

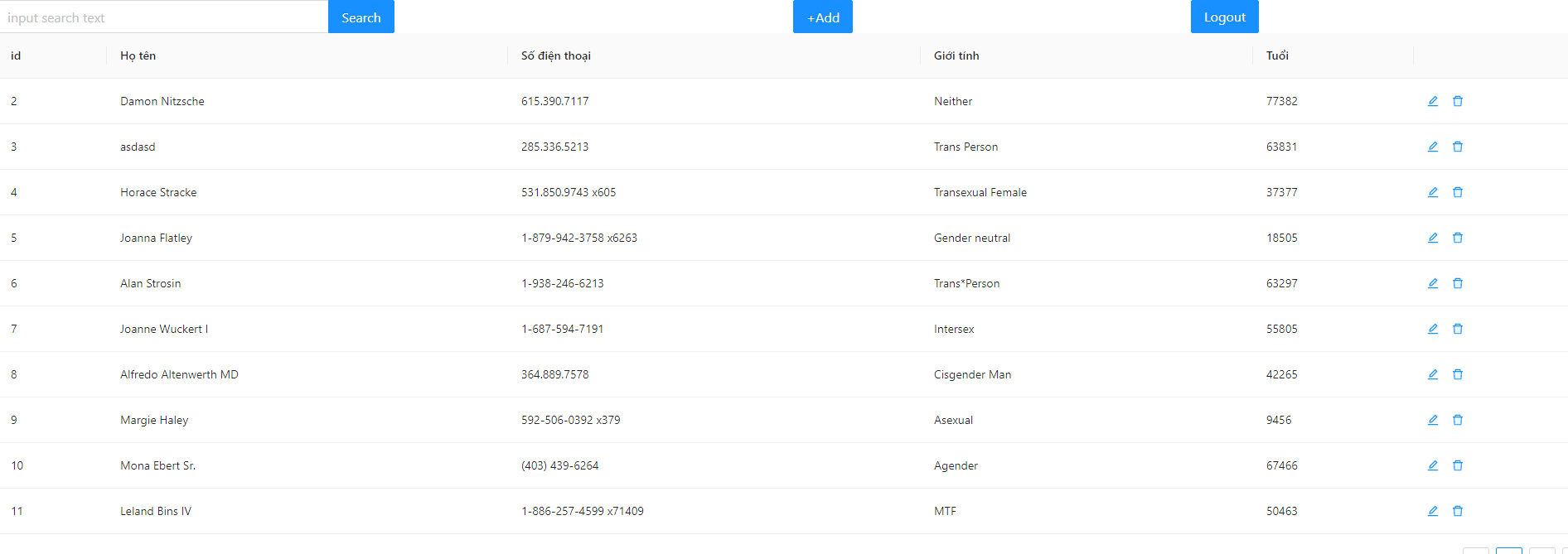
* 1. **Source code**

Tất cả source code nằm trong link github sau:

[internSCC/React/redux-app at main · uytran36/internSCC (github.com)](https://github.com/uytran36/internSCC/tree/main/React/redux-app)

* 1. **Demo**

****

****

* 1. **Các vấn đề gặp phải**

**3.5.1 Vấn đề về chuyển trang**

* **­**Hiện tại em muốn bỏ trang đăng nhập và trang chủ thành 1 đường dẫn. Chính vì thế việc em vẫn chưa biết sử dụng phương pháp gì để chuyển trang thông qua *react router dom.*
* Cách giải quyết hiện tại là em sẽ đưa một đường dẫn khác cho trang đăng nhập, nếu người dùng vô trang chính mà chưa đăng nhập, nó sẽ chuyển địa chỉ qua đăng nhập.

**3.5.2 Vấn đề về trễ state**

* Khi vào trang chính thì lập tức data sẽ được lấy về thông qua action *fetchContact* tuy nhiên state của component lại bị trễ state nên không nhận được kết quả sau khi fetch, vì thế không có dữ liệu được hiển thị trên màn hình.
  1. **Hướng đi tiếp theo**
* Tìm hiểu Redux-saga để ứng dụng vào project
* Tìm hiểu về webpack và Jest
* Đưa project về cấu trúc feature-first