react-exercises

Prerequisites

- Courses

  Please make sure you have finished the **REST** and **GraphQL** courses

- Provided apps find 3 provided apps in .zip format: **rest-exercise-node.zip**, **rest-doctor-info.zip** and **graphql-api.zip**, unzip them. For each app, make sure you run `npm install` and `npm start` to get them running on your local machine.

  - rest-exercise-node http://localhost:4000
  - rest-doctor-info http://localhost:5000
  - graphql-api http://localhost:3030

Local Setup

Ensure the following VS Code extensions are installed before starting this exercise:

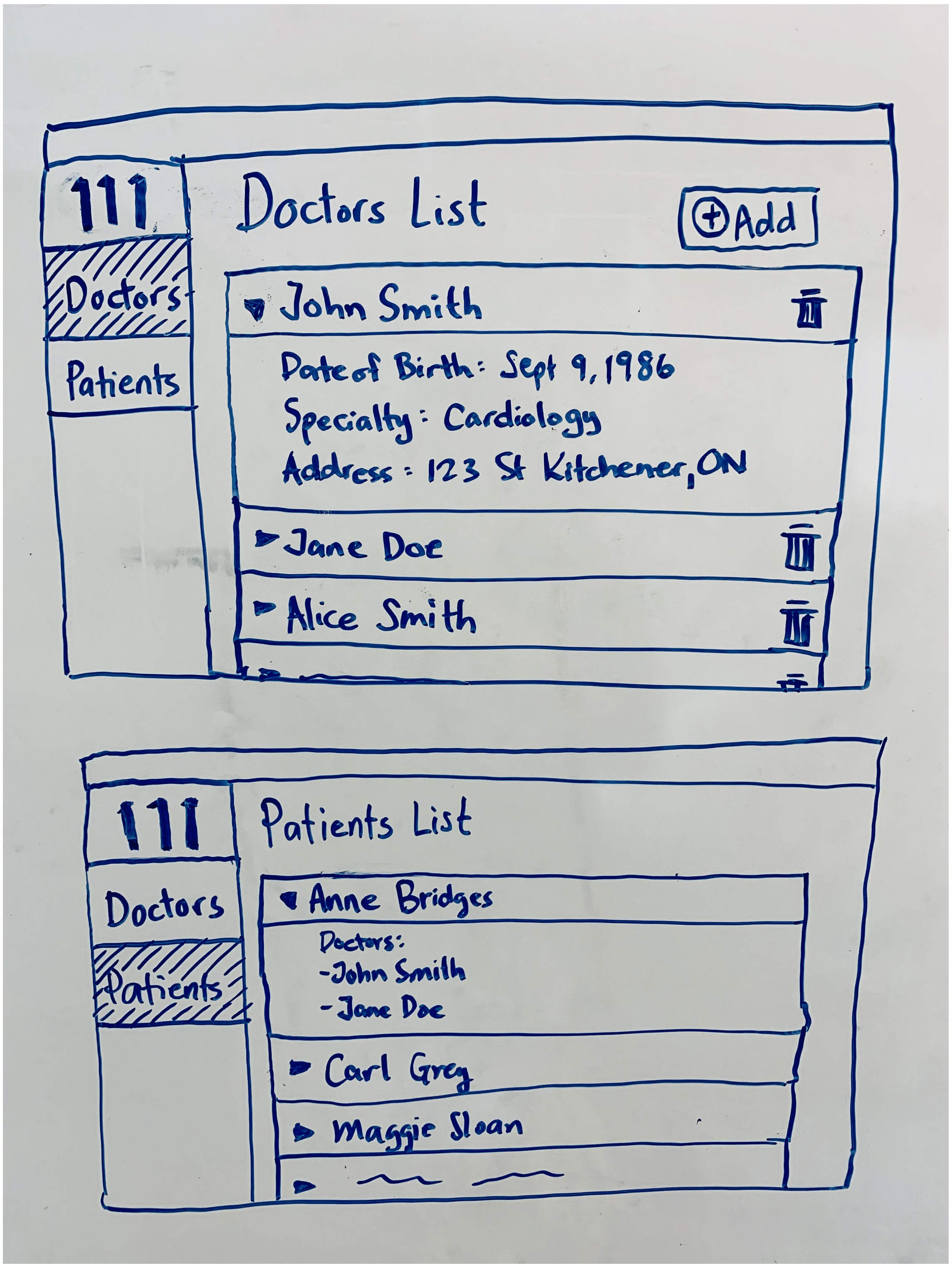
- ES7 React/Redux/GraphQL/React-Native/JS snippets

Ensure the following Chrome extension is installed before starting this exercise:

- React Developer Tools

Design

We want to create a web app that looks something like this:



Features

- Sidebar to show navigation to pages for Doctors and Patients
- On the Doctors Page:
  - Show a list of doctors (names)
  - Add a doctor

    - Delete a doctor
    - Click on a doctor to see more details (date of birth, specialty, address)
    - Show an error message when doctor details can't be loaded
- On the Patients Page:
    - Show a list of patients (names)
    - Click on a patient to see more details (list of patient's doctors)

Think about:

- What components do you need to implement these features? To match the design?
- What would your component tree look like?
- What props would each component take? What state would they have internally?

Note that we will not be implementing *all* the features listed, some will be left as optional individual exercises.

Create a React App

1. Open a bash terminal in VS Code. Ensure you're in the root directory you set up for the University.
2. Run the following command to create a react app: `npx create-react-app <student-id>-react-redux-doctors`. Do not copy this command, type it yourself as copying can cause issues. We're also using this React app as a starting point for the Redux course project (up next).

    > ***Note:*** *If the* `create-react-app` *command hangs and does nothing for over a minute, your create-react-app may be out of date. Cancel the process and run the following to uninstall it:*
    >
    > `npm uninstall -g create-react-app`
    >
    > *Then run the following to install it again:*
    >
    > `npm install -g create-react-app`

3. Move to the directory that was just created: `cd <student-id>-react-redux-doctors`, and open it up in your text editor.
4. Open `./src/index.js`. and examine the file. The line `ReactDOM.render(<App />, document.getElementById('root'));` mounts the App component onto the “root” element in index.html. The application stays on index.html the entire time.
5. Open `./src/App.js`. This is your top-most application component. A React application is a tree of components. All future components we write will be children or grandchildren of this component.
6. Look at the contents of `package.json` and the `node_modules` folder. (dependencies).
7. **NEW** with Create-React-App 4 for hot reloading feature

    Fast refresh is the official React implementation of the old feature, hot reloading. It is akin to hot reloading but it is much more reliable.

    Unlike hot reloading, fast refresh is off by default. To enable it, let's navigate to package.json file and update npm script "start" to be

    ```
    "start": "react-scripts start -FAST_REFRESH=true",
    ```

8. Run your React application: `npm start`.

9. Replace the contents of `App.js` with the following:

   ```
   function App(props) {
     return <h1>Hello, {props.name}</h1>;
   }
   ```

   Save the changes and fast refresh shall update the view in browser for you automatically.

10. Update ReactDOM.render() line in `index.js` to pass in the `name` prop.

    ```
    ReactDOM.render(
      <React.StrictMode>
        <App name='University of Waterloo' />
      </React.StrictMode>,
      document.getElementById('root')
    );
    ```

    WARNING: With Fast Refresh, there is a limitation at this moment, that changes in `index.js` won't show in browser automatically. We need to manually refresh the browser. Changes for other components refresh properly.

    Take a look at React DevTools and observe the component tree.

Adding functionality to your app

1. Create a folder called `components` under the `src` folder. This is where the component files will be stored.

Throughout this course, we'll be using both class components and functional components so you can see the difference between both!

Jump to:

1. Show a list of doctors

Goal: Show a list of doctor names taken from the REST API.

Concepts:

- Class components

- Using state
- `componentDidMount` lifecycle method
- `key` prop for lists
- Using the `fetch` method for calling your REST API

1. Create a file called `DoctorsList.js` in the `src/components` folder.

2. Create a class component for DoctorsList that renders the title of the list.

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

export default class DoctorsList extends Component {
  constructor(props) {
    super(props);
  }

  render() {
    return <h2>Doctors List</h2>;
  }
}
```

3. Display the `DoctorsList` component in App.js.

   Note that React components need to return only one top-level component. The `<>` syntax is to denote a React Fragment to achieve this.

```
import DoctorsList from './components/DoctorsList';

function App(props) {
  return (
    <>
      <h1>Hello, {props.name}</h1>
      <DoctorsList />
    </>
  );
}
```

4. In the DoctorsList component, use fetch to call the REST service to get a list of doctors.

   You can use the REST API endpoint here: http://localhost:4000/v1/doctors

   We are using the component's state to store the result from the REST API call.

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

export default class DoctorsList extends Component {
  constructor(props) {
    super(props);

    this.state = { doctors: [] };
  }

  componentDidMount() {
    fetch(
      'http://localhost:4000/v1/doctors'
    )
      .then((response) => response.json())
      .then((result) => this.setState({ doctors: result }));
  }

  render() {
    return <h2>Doctors List</h2>;
  }
}
```

Learn more about the `componentDidMount` lifecycle method here.

5. Display the list of doctors taken from the local state.

```
export default class DoctorsList extends Component {
  constructor(props) {
    super(props);

    this.state = { doctors: [] };
  }

  componentDidMount() {
    fetch(
      'http://localhost:4000/v1/doctors'
    )
      .then((response) => response.json())
      .then((result) => this.setState({ doctors: result }));
  }

  renderDoctors() {
    return this.state.doctors.map((doctor) => <div>{doctor.name}</div>);
  }

  render() {
    return (
      <>
        <h2>Doctors List</h2>
        {this.renderDoctors()}
      </>
    );
  }
}
```

6. Go to your browser. You should see the `Doctors List` heading and the list of doctor names.

7. Check out the React DevTools and observe where the `doctors` array in state is for that component.

8. Uh oh! Do you see a warning in the console? Let's try to fix that. In `DoctorsList`, add the `key` prop to the repeating element:

```
  renderDoctors() {
    return this.state.doctors.map(doctor => (
      <div key={doctor.id}>{doctor.name}</div>
    ));
  }
```

Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity

Learn more about keys in lists here.

2. Creating a DoctorListItem component

Goal: Nothing new changes in the browser, but we are doing some refactoring.

Concepts:

- Functional Components
- PropTypes
- Component tree and passing props

When working with React, we need to think in *components*. For example, in `DoctorsList`, right now we're only displaying the name. Next, we need to be able to click on the doctor and show details, or delete the doctor. That's a lot of functionality that is gonna clutter up the `DoctorsList` component.

To avoid that clutter, we can extract each doctor into a `DoctorListItem` component.

1. Create a file called `DoctorListItem.js`.

2. Create a `DoctorListItem` functional component.

   We can also destructure the props here to take in `id` and `name`:

   ```
   import React from 'react';

   function DoctorListItem({ id, name }) {
     return <div>{name}</div>;
   }

   export default DoctorListItem;
   ```

3. Let's add PropTypes! PropTypes helps developers catch bugs using type checking.

   ```
   import React from 'react';
   import PropTypes from 'prop-types';

   function DoctorListItem({ id, name }) {
     return <div>{name}</div>;
   }

   DoctorListItem.propTypes = {
     id: PropTypes.string.isRequired,
     name: PropTypes.string.isRequired,
   };

   export default DoctorListItem;
   ```

4. Lastly, replace the render in `DoctorsList` to show the new component for `DoctorListItem`.

   ```
   import DoctorListItem from './DoctorListItem';
   ...
   renderDoctors() {
     return this.state.doctors.map(doctor => <DoctorListItem key={doctor.id} id=
   {doctor.id} name={doctor.name} />);
   }
   ```

5. In your browser, there should be no changes to what is displayed. However if you check the React DevTools, you'll see new components for `DoctorListItem`.

3. Add a doctor

Goal: A text input and button to add a doctor. Type a name and clicking on the button will log the name in your browser console.

Concepts:

- Functional component (arrow function)
- Controlled components
- Uncontrolled components
- useState, useRef hooks

Let's add functionality for adding a doctor to our list. In this exercise we'll add just the **name** for a doctor to keep things simple.

Note that there are 2 ways to handle this in React: Uncontrolled or Controlled Components. We will look at both approaches.

First let's set up the UI components, the text input and button.

1. Create a file called `AddDoctor.js`.

2. Create a `AddDoctor` as a functional component. It should have a text input for the name and a button. Render the component in `DoctorsList`, just before displaying all of the doctors.

   ```
   import React from 'react';

   const AddDoctor = (props) => {
     return (
       <div>
         <input type='text' />
         <button>Add Doctor</button>
       </div>
     );
   };

   export default AddDoctor;
   ```

   In `DoctorsList.js`:

   ```
   import AddDoctor from './AddDoctor';
   ...
   ```

```
render() {
return (
  <>
    <h2>Doctors List</h2>
    <AddDoctor />
    {this.renderDoctors()}
  </>
);
}
```

3. You should see a text input and button in your browser.

Controlled Components

Let's first take a look at implementing this with Controlled Components.

1. Set up the state for the name with the `useState` hook.

```
const [doctorName, setDoctorName] = useState('');
```

2. Make sure to import the `useState` hook at the top.

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

3. Add an event handler to the input element whenever its value changes. Set its value to be the one stored in `doctorName` state.

```
const AddDoctor = (props) => {
  const [doctorName, setDoctorName] = useState('');

  const handleChangeName = (event) => {
    setDoctorName(event.target.value);
  };

  return (
    <div>
      <input type='text' value={doctorName} onChange={handleChangeName} />
      <button>Add Doctor</button>
    </div>
  );
};
```

4. Add an event handler that will be triggered when the AddDoctor button is clicked (`onClick` prop for the `button` element). We want it to take the doctor name and pass it up to the `DoctorsList` component to add to the list of doctors. For now, we'll do a console.log to check that we have the correct value for the doctor name.

   In `AddDoctor.js`:

```
...
const handleAddDoctor = () => {
  console.log(doctorName);
};
```

```
return (
  <div>
    <input type='text' value={doctorName} onChange={handleChangeName} />
    <button onClick={handleAddDoctor}>Add Doctor</button>
  </div>
);
```

5. Test the component! You should see whatever you typed into the input show up in the browser console after you click on the button.

Uncontrolled Components

Now, let's take a look at implementing this with Uncontrolled Components.

1. Comment out the controlled component code so that you have a reference for it. Copy-paste the starting code again.

```
import React from 'react';

const AddDoctor = (props) => {
  return (
    <div>
      <input type='text' />
      <button>Add Doctor</button>
    </div>
  );
};

export default AddDoctor;
```

2. Set up the ref with the `useRef` hook and a `ref` prop on the `input` element.

```
const doctorNameInputRef = useRef(null);
...
<input type='text' ref={doctorNameInputRef} />
```

3. Make sure to import the `useRef` hook at the top.

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

4. Similar to a controlled component, we want to add an event handler that will be triggered when the AddDoctor button is clicked. To get the value from the ref, use `doctorNameInputRef.current.value`.

```
const handleAddDoctor = () => {
  console.log(doctorNameInputRef.current.value);
};

return (
  <div>
    <input type='text' ref={doctorNameInputRef} />
    <button onClick={handleAddDoctor}>Add Doctor</button>
  </div>
);
```

5. Test the component! You should see whatever you typed into the input show up in the browser console after you click on the button.

React recommends using **controlled components** in forms, so we recommend commenting the code for uncontrolled components and using the controlled component going forward.

4. Adding a doctor to the list

Goal: When the Add Doctor button is clicked, the name is added to the bottom of our list of doctors.

Concepts:

- Passing a functions as props

With our input and button set up, we can now add a doctor to the list. We do this by passing in a prop from `DoctorsList` to `AddDoctor`.

`DoctorsList` is in charge of storing the state for our list of doctors, so we need to pass the `name` value from `AddDoctor` to `DoctorsList`.

1. In `AddDoctor.js`, replace the `console.log` with the call to the prop.

```
const handleAddDoctor = () => {
  props.onAddDoctor(doctorName); // using controlled name
};
```

2. Pass the `onAddDoctor` prop to the `AddDoctor` component from the `DoctorsList`. We will be creating the `handleAddDoctor` function in the next step.

```
<AddDoctor onAddDoctor={(name) => this.handleAddDoctor(name)} />
```

3. Set up the function to add a doctor to the `doctors` list in state.

   You can try using the POST method from the REST API here, or just set it directly in state.

```
handleAddDoctor(name) {
 const newDoctor = { id: Date.now(), name: name };
 const newDoctorsList = [...this.state.doctors, newDoctor];
 this.setState({ doctors: newDoctorsList });
}
```

4. Test your changes. You should see the new name get added to the end of the list.

5. Uh oh! We have a warning in the console. This time, it's about an invalid prop being specified. What could've caused that?

   Let's look in `DoctorsList`. The new doctor `id` is being specified as a `Number`, when it's in fact expecting a `String` (as the propTypes for `DoctorListItem` tell us). Let's correct that:

```
handleAddDoctor(name) {
 const newDoctor = { id: Date.now().toString(), name: name };
 const newDoctorsList = [...this.state.doctors, newDoctor];
```

```
  this.setState({ doctors: newDoctorsList });
}
```

6. **Bonus Individual exercise:** Try clearing the name input after a doctor is added to the list.

5. See more doctor details

Goal: When you click on a doctor, it shows the details for the doctor taken from your REST api.

Concepts:

- inline if and logical && operator

1. Create a file called `DoctorDetails.js`.
2. Create a `DoctorDetails` functional component (use ES6 arrow notation instead of the function notation).

   This component will display the doctor's date of birth, specialty, and address; these are passed into the component through the props. We can also destructure the props inline.

   ```
   import React from 'react';

   const DoctorDetails = ({ dob, specialty, address }) => {
     const { street, city, province, postal_code } = address;
     return (
       <>
         <h5>Date of birth: {dob}</h5>
         <h5>Specialty: {specialty}</h5>
         <h5>
           Address: {street}, {city}, {province} {postal_code}
         </h5>
       </>
     );
   };

   export default DoctorDetails;
   ```

   This is a good example of a presentational component, where it's only responsibility is to display data (props) passed to it. No logic involved!

3. Add a click handler in `DoctorListItem` to trigger an API call and store the doctor's details in its state. Since it's a functional component, we'll use the `useState` hook.

   ```
   import React, { useState } from 'react';
   ...
   function DoctorListItem({ id, name }) {
     const [details, setDetails] = useState(null);

     function handleLoadDetails() {
       fetch(
         `http://localhost:5000/v1/doctor/${id}`
       )
         .then(response => response.json())
         .then(response => setDetails(response));
   ```

```
  }

  return (
    <div>
      <a href='#' onClick={handleLoadDetails}>
        {name}
      </a>
    </div>
  );
}
```

4. Finally, display the details (`DoctorDetails` component) only when the details state is set.

   We can use conditional rendering with an inline if and logical `&&` operator.

   Or we can extract to a separate function and do an early return.

   ```
   {
     details && (
       <DoctorDetails
         dob={details.dob}
         specialty={details.specialty}
         address={details.address}
       />
     );
   }
   ```

   OR:

   ```
   renderDoctorDetails() {
     if (!details) { return; } // early return since there is nothing to display
     return <DoctorDetails dob={details.dob} specialty={details.specialty}
   address={details.address} />
   }

   ...

   render() {
     ...
     {renderDoctorDetails()}
   }
   ```

   Both approaches are valid.

5. When displaying `DoctorDetails`, we could also use the spread operator to relay the props, since `details` contains the date of birth, specialty and address properties.

   ```
   <DoctorDetails {...details} />
   ```

   What are the advantages and disadvantages of using the spread operator here?

6. Show a list of patients

Goal: Display a list of patient names. When a patient is clicked, display their list of doctors. Use the REST API and GraphQL API to get data.

Concepts:

- Functional component
- Similar pattern to DoctorsList/DoctorListItem
- useState, useEffect hooks
- Fetching data using a GraphQL API

Similar to the Doctors side of the exercise, we'll tackle Patients now. We can add the component to go below or above the `DoctorsList` for now. As an optional exercise, you can try setting up React Router to have 2 separate pages, one for Doctors and another for Patients.

1. `PatientsList` is a functional component, so we'll use hooks here (whereas the `DoctorsList` had lifecycle methods). We need to call our REST API for the list of patients when the component mounts, store it in state, and render `PatientListItem` components for each patient.

   We can use the `useState` hook to keep track of the `patients` array.

   We can create a `fetchPatients` function to do the REST API call.

   We don't have access to `componentDidMount` lifecycle method in a functional component, but we still need to ensure that the `fetchPatients` is called only once when the component mounts. Here we want to the `useEffect` hook with an empty dependency array.

   Read more about the `useEffect` hook here.

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

import PatientListItem from './PatientListItem';

const PatientsList = () => {
  const [patients, setPatients] = useState(null);

  const fetchPatients = () => {
    fetch(
      'http://localhost:4000/v1/patients'
    )
      .then((response) => response.json())
      .then((result) => setPatients(result));
  };

  // only runs once on component mount
  useEffect(() => {
    fetchPatients();
  }, []);

  return (
    <>
      <h2>Patients List</h2>
      {patients &&
        patients.map((patient) => (
          <PatientListItem key={patient.id} patient={patient} />
        ))}
      </>
  );
};

export default PatientsList;
```

2. See patient details when a patient is clicked. Create mock data to display for the patient's doctors for now. The next step will go over how to get real data from a GraphQL API. Refer to `DoctorListItem` for an example pattern on how to structure this code.

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

const PatientListItem = ({ patient }) => {
  const [doctorsList, setDoctorsList] = useState(null);

  const handleLoadDoctors = () => {
    console.log('// TODO: Load doctors with GraphQL API...');
    setDoctorsList([
      { id: 1, name: 'Dr. Pepper' },
      { id: 2, name: 'Doctor Who' },
    ]);
  };

  return (
    <div>
      <a href='#' onClick={handleLoadDoctors}>
        {patient.name}
      </a>
      {doctorsList &&
        doctorsList.map((doctor) => (
          <div key={doctor.id}>{doctor.name}</div>
        ))}
    </div>
  );
};

export default PatientListItem;
```

3. Instead of using a REST API, we'll be using a GraphQL endpoint to get a list of doctors for a patient: https://localhost:3030/.

   To craft the query, test it out in the GraphQL Playground. We will end up with:

```
query($id: ID!) {
  patient(id: $id) {
    doctors {
      id
      name
    }
  }
}
```

In the Query Variables section, add:

```
{ "id": "1" }
```

4. In `PatientListItem.js`, store the GraphQL API URL and the query in variables inside the `PatientListItem`.

```
const GQL_API = `https://localhost:3030/`;
const GQL_QUERY = `
 query($id: ID!) {
   patient(id: $id) {
     doctors {
       id
       name
     }
   }
 }`;

 ...

 const handleLoadDoctors = () => { ... };
```

5. In `handleLoadDoctors`, where we'll be doing the call, declare the variables we'll be passing on to the query. This matches the Query Variables section in the GraphQL Playground.

```
const handleLoadDoctors = () => {
  const variables = { id: patient.id };
};
```

6. To get the data from GraphQL, we need to make a POST request, where the body of the request contains a `query` and `variables` properties and is in JSON format. More info here.

```
const handleLoadDoctors = () => {
  const variables = { id: patient.id };
  fetch(GQL_API, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      query: GQL_QUERY,
      variables,
    }),
  })
    .then((response) => response.json())
    .then((result) => setDoctorsList(result.data.patient.doctors));
};
```

7. Delete a doctor

Goal: Each doctor has a delete icon beside their name. When the delete icon is clicked, the doctor is removed from the list.

Concepts:

- Nothing new, this will be review!
- Similar pattern to adding a doctor to the list

1. Add the button to delete the doctor and the handler placeholder for when the button is clicked.

```
import React from 'react';

function DoctorListItem({ id, name, onDeleteDoctor }) {
  function handleDeleteDoctor() {
    console.log('TODO: delete doctor...');
  }

  return (
    <div>
      {name}
      <button onClick={handleDeleteDoctor}>X</button>
    </div>
  );
}

export default DoctorListItem;
```

2. The `DoctorsList` component is responsible for storing the state of the full list of doctors. So when a `DoctorListItem`'s delete button is clicked, the `DoctorsList` needs to know about that event and delete the doctor from its state.

   We will pass that deleteDoctor handler from the parent component (`DoctorsList`) to the child component (`DoctorsListItem`) as a prop.

   In `DoctorListItem.js` :

```
function handleDeleteDoctor() {
  onDeleteDoctor(id); // this is coming from props, it was destructured inline
in the line above
}
```

   In `DoctorList.js` , add the prop and the corresponding function.

```
handleDeleteDoctor(id) {
  console.log(`TODO: Delete doctor with id ${id}`)
}

renderDoctors() {
  return this.state.doctors.map(doctor => (
    <DoctorListItem key={doctor.id}
                    id={doctor.id}
                    name={doctor.name}
                    onDeleteDoctor={(id) => this.handleDeleteDoctor(id)}
    />
  ));
}
```

3. Test in the browser! You should be seeing a console log of `TODO: Delete doctor with id: ?` with the appropriate doctor id.

4. The REST API doesn't have a DELETE endpoint, so we'll fake it for our own purposes. Note that the data deletion will not be persistent.

```
handleDeleteDoctor(id) {
  const newDoctorsList = this.state.doctors.filter(doctor => doctor.id !== id);
  this.setState({doctors: newDoctorsList});
}
```