**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 <lan-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).
3. Move to the directory that was just created: cd <lan-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. Run your React application: npm start​.
8. Replace the contents of App.js with the following:
9. function App(props) {
10. return <h1>Hello, {props.name}</h1>;

}

1. Update ReactDOM.render() line in index.js to pass in the name prop.
2. ReactDOM.render(
3. <React.StrictMode>
4. <App name='Manulife University' />
5. </React.StrictMode>,
6. document.getElementById('root')

);

Save the changes and show hot reloading in browser. Show React DevTools and how the App component has the props name and how you can edit that.

**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](#show-a-list-of-doctors)
2. [Creating a DoctorListItem component](#creating-a-doctorlistitem-component)
3. [Add a doctor component](#add-a-doctor)
4. [Adding a doctor to the list](#adding-a-doctor-to-the-list)
5. [Using MUX as a component library](#using-mux-as-a-component-library)
6. [Individual exercise: See more doctor details](#see-more-doctor-details)
7. [Show a list of patients](#show-a-list-of-patients)
8. [Individual Exercise: Delete a doctor](#delete-a-doctor)
9. [Individual Exercise (Optional): Page Routing](#individual-exercise-optional-page-routi)
10. [Individual Exercise (Optional): Handling errors with React Toastify](#individual-exercise-optional-handling-e)
11. [FYI Only: Sample code calling APIgee endpoint](#fyi-only-sample-code-calling-apigee-end)

**Show a list of doctors**

**Goal**: Show a list of doctor names taken from the [REST API](https://rest-example-node.apps.cac.preview.pcf.manulife.com/v1/doctors).

**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 components folder.
2. Create a class component for DoctorsList that renders the title of the list.
3. import React, { Component } from 'react';
4. export default class DoctorsList extends Component {
5. constructor(props) {
6. super(props);
7. }
8. render() {
9. return <h2>Doctors List</h2>;
10. }

}

1. 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](https://reactjs.org/docs/fragments.html) to achieve this.

import DoctorsList from './components/DoctorsList';

function App(props) {

return (

<>

<h1>Hello, {props.name}</h1>

<DoctorsList />

</>

);

}

1. 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: <https://rest-example-node.apps.cac.preview.pcf.manulife.com/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(

'https://rest-example-node.apps.cac.preview.pcf.manulife.com/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](https://reactjs.org/docs/react-component.html#componentdidmount).

1. Display the list of doctors taken from the local state.
2. export default class DoctorsList extends Component {
3. constructor(props) {
4. super(props);
5. this.state = { doctors: [] };
6. }
7. componentDidMount() {
8. fetch(
9. 'https://rest-example-node.apps.cac.preview.pcf.manulife.com/v1/doctors'
10. )
11. .then((response) => response.json())
12. .then((result) => this.setState({ doctors: result }));
13. }
14. renderDoctors() {
15. return this.state.doctors.map((doctor) => <div>{doctor.name}</div>);
16. }
17. render() {
18. return (
19. <>
20. <h2>Doctors List</h2>
21. {this.renderDoctors()}
22. </>
23. );
24. }

}

1. Go to your browser. You should see the Doctors List heading and the list of doctor names.
2. Check out the React DevTools and observe where the doctors array in state is for that component.
3. 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:
4. renderDoctors() {
5. return this.state.doctors.map(doctor => (
6. <div key={doctor.id}>{doctor.name}</div>
7. ));

}

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](https://reactjs.org/docs/lists-and-keys.html).

**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;

1. Let's add PropTypes! [PropTypes](https://reactjs.org/docs/typechecking-with-proptypes.html) helps developers catch bugs using type checking.
2. import React from 'react';
3. import PropTypes from 'prop-types';
4. function DoctorListItem({ id, name }) {
5. return <div>{name}</div>;
6. }
7. DoctorListItem.propTypes = {
8. id: PropTypes.string.isRequired,
9. name: PropTypes.string.isRequired,
10. };

export default DoctorListItem;

1. Lastly, replace the render in DoctorsList to show the new component for DoctorListItem.
2. import DoctorListItem from './DoctorListItem';
3. ...
4. renderDoctors() {
5. return this.state.doctors.map(doctor => <DoctorListItem key={doctor.id} id={doctor.id} name={doctor.name} />);

}

1. 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.

**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](https://reactjs.org/docs/uncontrolled-components.html) or [Controlled Components](https://reactjs.org/docs/forms.html#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.
3. import React from 'react';
4. const AddDoctor = (props) => {
5. return (
6. <div>
7. <input type='text' />
8. <button>Add Doctor</button>
9. </div>
10. );
11. };

export default AddDoctor;

In DoctorsList.js:

import AddDoctor from './AddDoctor';

...

render() {

return (

<>

<h2>Doctors List</h2>

<AddDoctor />

{this.renderDoctors()}

</>

);

}

1. 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](https://reactjs.org/docs/forms.html#controlled-components).

1. Set up the state for the name with the [useState hook](https://reactjs.org/docs/hooks-state.html).

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

1. Make sure to import the useState hook at the top.

import React, { useState } from 'react';

1. Add an event handler to the input element whenever its value changes. Set its value to be the one stored in doctorName state.
2. const AddDoctor = (props) => {
3. const [doctorName, setDoctorName] = useState('');
4. const handleChangeName = (event) => {
5. setDoctorName(event.target.value);
6. };
7. return (
8. <div>
9. <input type='text' value={doctorName} onChange={handleChangeName} />
10. <button>Add Doctor</button>
11. </div>
12. );

};

1. 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>

);

1. 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](https://reactjs.org/docs/uncontrolled-components.html).

1. Comment out the controlled component code so that you have a reference for it. Copy-paste the starting code again.
2. import React from 'react';
3. const AddDoctor = (props) => {
4. return (
5. <div>
6. <input type='text' />
7. <button>Add Doctor</button>
8. </div>
9. );
10. };

export default AddDoctor;

1. Set up the ref with the [useRef hook](https://reactjs.org/docs/hooks-reference.html#useref) and a ref prop on the input element.
2. const doctorNameInputRef = useRef(null);
3. ...

<input type='text' ref={doctorNameInputRef} />

1. Make sure to import the useRef hook at the top.

import React, { useRef } from 'react';

1. 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.
2. const handleAddDoctor = () => {
3. console.log(doctorNameInputRef.current.value);
4. };
5. return (
6. <div>
7. <input type='text' ref={doctorNameInputRef} />
8. <button onClick={handleAddDoctor}>Add Doctor</button>
9. </div>

);

1. 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.

**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.
2. const handleAddDoctor = () => {
3. props.onAddDoctor(doctorName); // using controlled name

};

1. 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)} />

1. 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 });

}

1. Test your changes. You should see the new name get added to the end of the list.
2. 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 });

}

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

**Using MUX as a component library**

**Goal**: Add MUX to your React project.

**Concepts**:

* Importing MUX
* Using a React component library

[MUX](https://mux.manulife.com) is a React component library that implements Manulife's Global Branding.

1. Install MUX by running npm install @awesomecomponents/mux styled-components. (MUX also needs the styled-components package).
2. Add the fonts in App.js (add before the App.css import).

import '@awesomecomponents/mux/core/typography/assets/fonts/fonts.css';

1. Try out the <H1> typography component by replacing in App.js with
2. import { H1 } from '@awesomecomponents/mux/core/typography';
3. ...

<H1>Hello, {props.name}</H1>;

1. Try adding the UtilityHeader component.
2. ...
3. import '@awesomecomponents/mux/core/typography/assets/fonts/fonts.css';
4. import UtilityHeader from '@awesomecomponents/mux/core/components/UtilityHeader';
5. import { H1 } from '@awesomecomponents/mux/core/typography';
6. ...
7. function App(props) {
8. return (
9. <div className='App'>
10. <UtilityHeader />
11. <H1>Hello, {props.name}</H1>
12. <DoctorsList />
13. </div>
14. );

}

1. Note that there is a create-react-app starter with MUX already installed. Run npx create-react-app my-app --template manulife-mux for new projects to use MUX.
2. **Bonus Individual exercise**: Replace the components we're using for text inputs, buttons and text with MUX components.

**See more doctor details**

**Goal**: When you click on a doctor, it shows the details for the doctor taken from a [REST API.](https://doctor-info.apps.cac.preview.pcf.manulife.com/docs)

**Concepts**:

* inline if and logical && operator

The API for getting doctor details can be found here: <https://doctor-info.apps.cac.preview.pcf.manulife.com/docs>

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!

1. 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.
2. import React, { useState } from 'react';
3. ...
4. function DoctorListItem({ id, name }) {
5. const [details, setDetails] = useState(null);
6. function handleLoadDetails() {
7. fetch(
8. `https://doctor-info.apps.cac.preview.pcf.manulife.com/v1/doctor/${id}`
9. )
10. .then(response => response.json())
11. .then(response => setDetails(response));
12. }
13. return (
14. <div>
15. <a href='#' onClick={handleLoadDetails}>
16. {name}
17. </a>
18. </div>
19. );

}

1. 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](https://reactjs.org/docs/conditional-rendering.html#inline-if-with-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.

1. 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?

**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](https://rest-example-node.apps.cac.preview.pcf.manulife.com/v1/patients) and [GraphQL API](https://graphql-doctor-patient.apps.cac.preview.pcf.manulife.com/) 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](https://rest-example-node.apps.cac.preview.pcf.manulife.com/v1/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](https://reactjs.org/docs/hooks-effect.html).

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

import { H2 } from '@awesomecomponents/mux/core/typography';

import PatientListItem from './PatientListItem';

const PatientsList = () => {

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

const fetchPatients = () => {

fetch(

'https://rest-example-node.apps.cac.preview.pcf.manulife.com/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;

1. 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.
2. import React, { useState } from 'react';
3. const PatientListItem = ({ patient }) => {
4. const [doctorsList, setDoctorsList] = useState(null);
5. const handleLoadDoctors = () => {
6. console.log('// TODO: Load doctors with GraphQL API...');
7. setDoctorsList([
8. { id: 1, name: 'Dr. Pepper' },
9. { id: 2, name: 'Doctor Who' },
10. ]);
11. };
12. return (
13. <div>
14. <a href='#' onClick={handleLoadDoctors}>
15. {patient.name}
16. </a>
17. {doctorsList &&
18. doctorsList.map((doctor) => (
19. <div key={doctor.id}>{doctor.name}</div>
20. ))}
21. </div>
22. );
23. };

export default PatientListItem;

1. Instead of using a REST API, we'll be using a GraphQL endpoint to get a list of doctors for a patient: <https://graphql-doctor-patient.apps.cac.preview.pcf.manulife.com/>.

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

id

}

}

}

In the Query Variables section, add:

{ "id": "1" }

1. In PatientListItem.js, store the GraphQL API URL and the query in variables inside the PatientListItem.
2. const GQL\_API = `https://graphql-doctor-patient.apps.cac.preview.pcf.manulife.com/`;
3. const GQL\_QUERY = `
4. query($id: ID!) {
5. patient(id: $id) {
6. doctors {
7. id
8. name
9. }
10. }
11. }`;
12. ...

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

1. 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.
2. const handleLoadDoctors = () => {
3. const variables = { id: patient.id };

};

1. 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](https://graphql.org/learn/serving-over-http/).
2. const handleLoadDoctors = () => {
3. const variables = { id: patient.id };
4. fetch(GQL\_API, {
5. method: 'POST',
6. headers: {
7. 'Content-Type': 'application/json',
8. },
9. body: JSON.stringify({
10. query: GQL\_QUERY,
11. variables,
12. }),
13. })
14. .then((response) => response.json())
15. .then((result) => setDoctorsList(result.data.patient.doctors));

};

**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.
2. import React from 'react';
3. function DoctorListItem({ id, name, onDeleteDoctor }) {
4. function handleDeleteDoctor() {
5. console.log('TODO: delete doctor...');
6. }
7. return (
8. <div>
9. {name}
10. <button onClick={handleDeleteDoctor}>X</button>
11. </div>
12. );
13. }

export default DoctorListItem;

1. 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)}

/>

));

}

1. Test in the browser! You should be seeing a console log of TODO: Delete doctor with id: ? with the appropriate doctor id.
2. 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.
3. handleDeleteDoctor(id) {
4. const newDoctorsList = this.state.doctors.filter(doctor => doctor.id !== id);
5. this.setState({doctors: newDoctorsList});

}

**Individual Exercise (Optional): Page Routing**

Try using [React Router](https://reacttraining.com/react-router/web/guides/quick-start) to implement page routing so that you have 2 separate pages: 1 for Doctors and 1 for Patients. Try using the MUX SideNav component to jump between the 2 pages.

**Individual Exercise (Optional): Handling errors with React Toastify**

Try using [React Toastify](https://www.npmjs.com/package/react-toastify) to implement:

* Showing error messages to the user when a doctor's information cannot be loaded (ie newly added doctors)
* Showing success messages after the doctor's information is added

**FYI Only: Sample code calling APIgee endpoint**

For those consuming APIgee endpoints, here's a sample method to call APIgee to get a token and then call the proxy service. Remember to set your env vars for this method to work.

process.env.APIGEE\_OAUTH = 'https://manulife-development-dev.apigee.net/v1/mg/oauth2/token';

process.env.CLIENT\_ID = 'Add Your Client Id Here';

process.env.CLIENT\_SECRET = 'Add Your Client Secret Here';

process.env.GRANT\_TYPE = 'client\_credentials';

process.env.PATIENT\_SERVICE = 'https://rest-example-node.apps.cac.preview.pcf.manulife.com/v1/patients';

fetchWithProxy() {

fetch(process.env.APIGEE\_OAUTH, {

method: 'POST',

body: `{"client\_id":"${process.env.CLIENT\_ID}","client\_secret":"${process.env.CLIENT\_SECRET}", "grant\_type":"${process.env.GRANT\_TYPE}"}`,

headers: { 'Content-Type': 'application/json' }

})

.then(oauth => oauth.json())

.then(bearer => {

fetch(process.env.PATIENT\_SERVICE, {

headers: { 'Authorization': `Bearer ${bearer.access\_token}` }

})

.then(res => res.json())

.then(json => {

this.setState({ patients: json });

})

})

}