前端網站開發人員證書課程
(二) 進階網絡程式設計--專業React.js應用

7. Lifecycle, HTTP, and Deployment

Presented by Krystal Institute

erb 僱員再培訓局 Employees Retraining Board
SCOPE School of Continuing and Professional Education 專業進修學院 香港城市大學 City University of Hong Kong
KRYSTAL INSTITUTE DIGITAL ECONOMY CORE TECHNOLOGY

Lesson Outline

- Introduce the Lifecycle of react components
- How to make HTTP requests to external APIs in react and how to make GET, POST, and DELETE requests in react
- Discuss what is GitHub Pages, and how to publish react applications on GitHub Pages

7.1 React as Single Page Application (SPA)

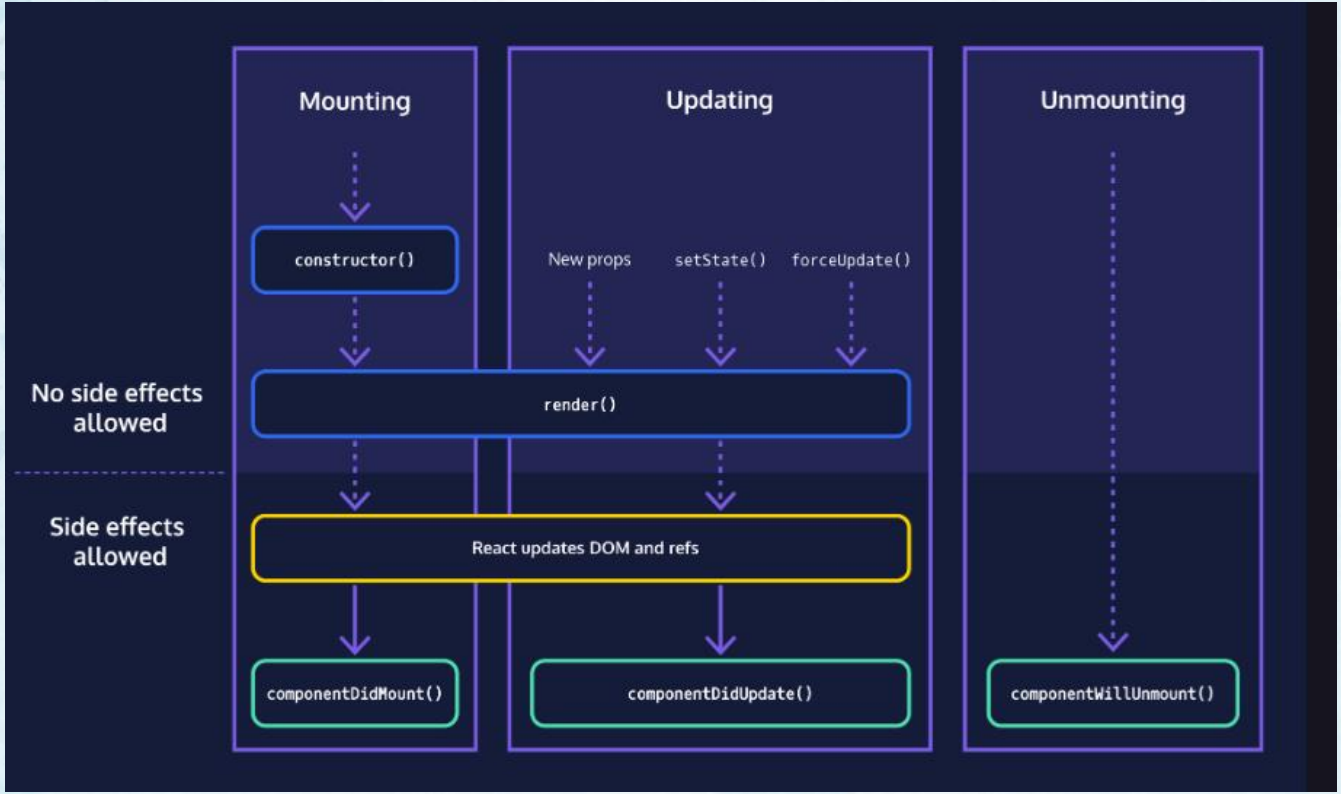
7.1 Class Component Lifecycle

- React components can be highly dynamic. They get created, rendered, added to the DOM, updated, and removed. All of these steps are part of a component's lifecycle.
- The component lifecycle has three high-level parts:
  - **Mounting:** When the component is being initialized and put into the DOM for the first time.
  - **Updating:** When the component updates as a result of a changed state or changed props.
  - **Unmounting:** When the component is being removed from the DOM.
- Every React component you have ever interacted with does the first step at a minimum. If a component was never mounted, you would never see it!
- Most interesting components are updated at some point. A purely static component, (for example, a logo)might not ever update.
- But if a component's state changes, it updates. Or if different props are passed to a component, it updates.

7.1 Class Component Lifecycle

- Finally, a component is unmounted when it's removed from the DOM.
- For example:
  - If you have a button that hides a component, chances are that component will be unmounted.
  - If your app has multiple screens, it's likely that each screen (and all of its child components) will be unmounted.
  - If a component is "alive" for the entire lifetime of your app (say, a top-level *<App />* component or a persistent navigation bar), it won't be unmounted. But most components can get unmounted one way or another!

7.1 Class Component Lifecycle



7.2 Lifecycle Methods

7.2 Lifecycle Methods

- React components have several methods, called lifecycle methods, that are called at different parts of a component's lifecycle.
  - *constructor()* is the first method called during the mounting phase. The *constructor()* method is only executed during the mounting phase.
  - *render()* is called later during the mounting phase, to render the component for the first time, and during the updating phase, to re-render the component. The *render()* method is executed during both the mounting and updating phase.
  - *componentDidMount()* is the final method called during the mounting phase. The *componentDidMount()* runs statements that require that the component is already placed in the DOM.
  - *componentWillUpdate()* is called just before rendering.
  - *componentDidUpdate()* is called just after rendering.
  - *componentWillUnmount()* is called in the unmounting phase, right before the component is completely destroyed. It's a useful time to clean up any of your component's mess.

7.3 Functional Component Lifecycle

7.3.1 Why React Hook?

- React introduced the **Hooks** for managing state, and using the lifecycle methods in the function-based component.
- React allows us to hook into the React library while using the functional component. This is easy and convenient to use. It reduces the code length.
- *useEffect* is a tool that lets us interact with the outside world but not affect the rendering or performance of the component that it's in.
- Basic Syntax:

```
// 1. import useEffect
import { useEffect } from 'react';

function MyComponent() {
  // 2. call it above the returned JSX
  // 3. pass two arguments to it: a function and an array
  useEffect(() => {}, []);

  // return ...
}
```

7.3.1 Why React Hook?

- Example:

```
import { useEffect } from 'react';

function User({ name }) {
  useEffect(() => {
    document.title = name;
  }, [name]);

  return <h1>{name}</h1>;
}
```

- The function passed to useEffect is a callback function. This will be called after the component renders.
- The second argument is an array, called the dependencies array. This array should include all of the values that our side effect relies upon.

7.3.2 React Functional Component

Mount:

- In the mounting phase, the react components are rendered for the first time.
- If you want to execute something when a React Function Component did mount, you can use the *useEffect* hook.

Update:

- Every time incoming props or state of the component change, the component triggers a re-render to display the latest status quo which is often derived from the props and state.
- A render executes everything within the Function Component's body.
- If you want to act upon a rerender, you can use the Effect Hook again to do something after the component did update.
- Only on certain variable changes, the second argument of the Effect Hook can be used.

7.3.2 React Functional Component

Example:

- Let's understand this better with help of the Timer example:
  - Create a react app.

```
C:\workspace\react tutorial\create react app>npx create-react-app timer

Creating a new React app in C:\workspace\react tutorial\create react app\timer.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...

[        ...........] \ idealTree:webpack-dev-server: sill fetch manifest emojis-list@^3.0.0
```

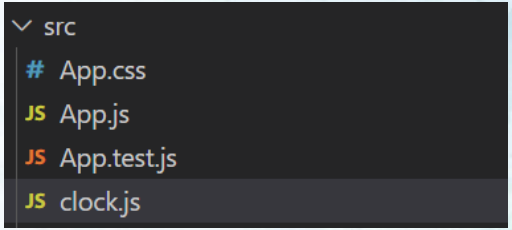
  - Run the app.

```
C:\workspace\react tutorial\create react app>cd timer

C:\workspace\react tutorial\create react app\timer>npm start

> timer@0.1.0 start
> react-scripts start
```

7.3.2 React Functional Component

Example:

- Create a new component file - clock.js
- Import useState and useEffect from react in clock.js.




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

- Create a function component that gets the current time and displays it on the screen

```
function Clock(){
    const [currentTime, setCurrentTime] = useState(new Date());
    let timerInterval = null;

    return (
        <>
            <h3>{currentTime.toLocaleTimeString()}</h3>
        </>
    );
}

export default Clock;
```

7.3.2 React Functional Component

Example:

- Switch to App.js, import useState, and the clock component.

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

import Clock from "./Clock";
```

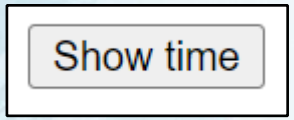
- Modify the existing App component, add useState to show the time, and display the time with the button(the Toggle button to show and hide time).

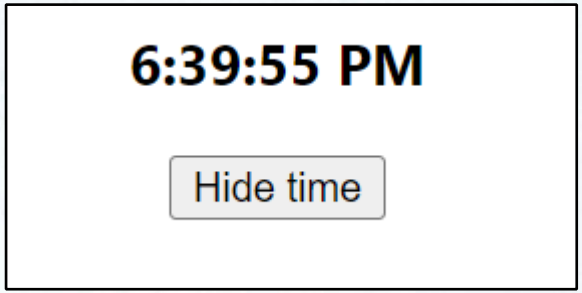
```
function App() {
  const [showTime, setShowTime] = useState(false);
  return (
    <div style={{display: 'flex', flexDirection: 'column', justifyContent: 'center', alignItems: 'center' }} >
      {showTime && <Clock />}
      <button onClick={() => setShowTime(previous => !previous)} >{showTime ? 'Hide time' : 'Show time'}</button>
    </div>
  );
}
```

7.3.2 React Functional Component

Example:

- View the Output:





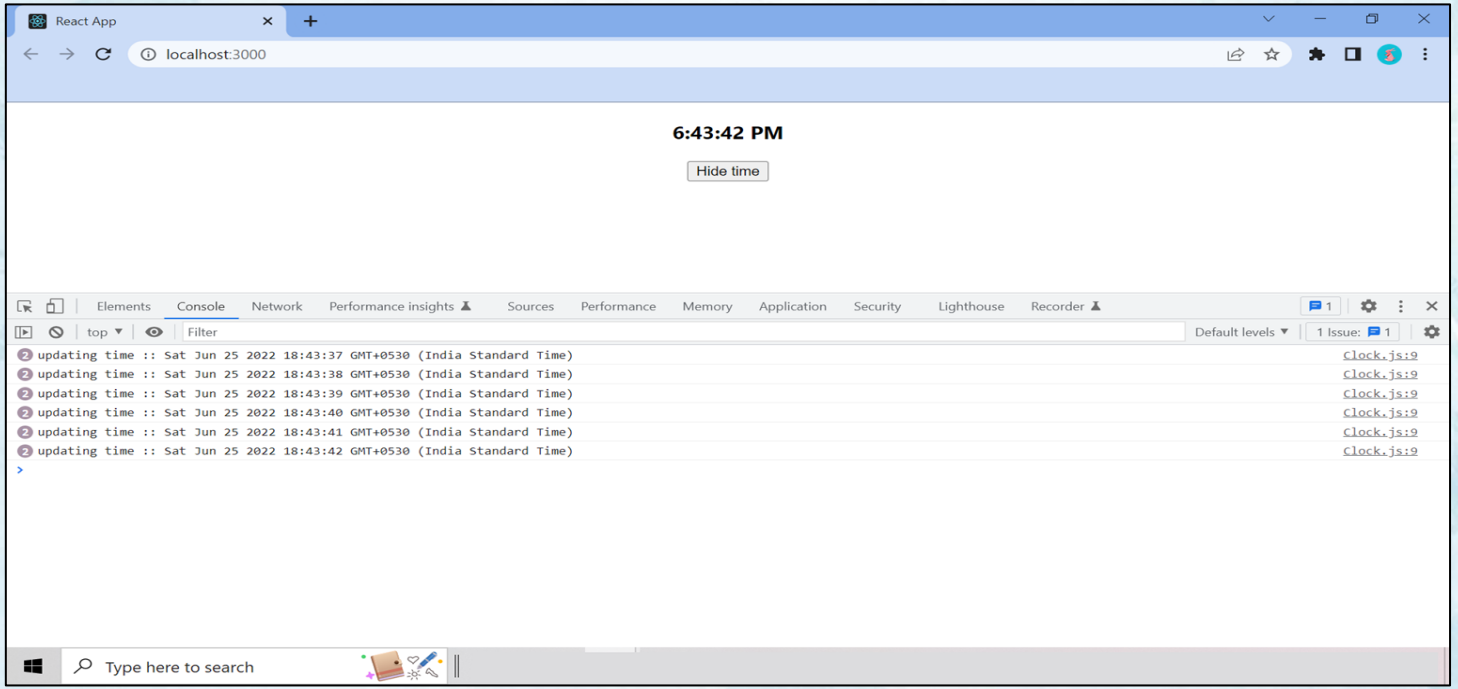
- Switch to the Clock component and add the useEffect hook to handle the timer.

```
useEffect(() => {
    timerInterval = setInterval(() => {
        console.log(`updating time :: ${new Date()}`);
        setCurrentTime(new Date());
    }, 1000);

}, []);
```

7.3.2 React Functional Component

Example:

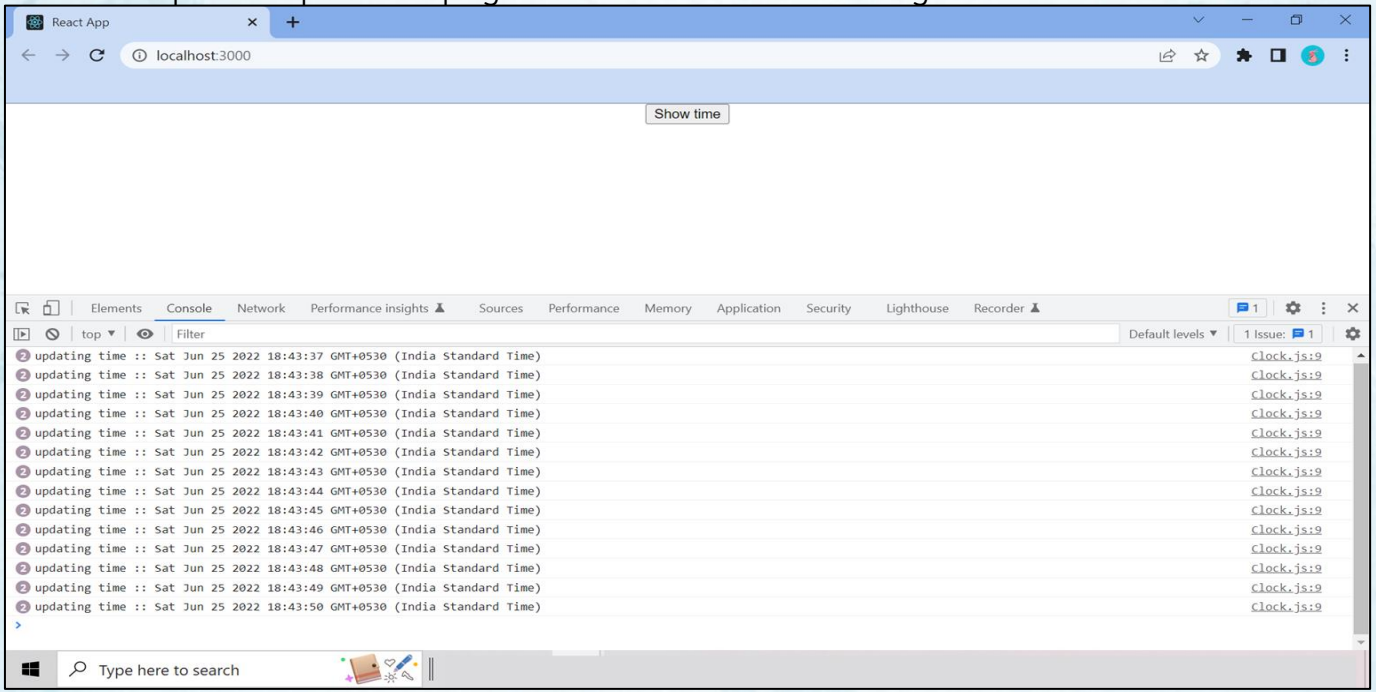
- View the output: Inspect the page and view the console log.



7.3.2 React Functional Component

Example:

- View the output: Inspect the page and view the console log.



7.3.2 React Functional Component

Example:

- You can see the time is getting updated while the time is shown and hidden.
- In general, when a component produces a side-effect, you should remember to clean it up. JavaScript gives us the *clearInterval()* function. *setInterval()* can return an ID, which you can then pass into *clearInterval()* to clear it.

```
useEffect(() => {
    timerInterval = setInterval(() => {
        console.log(`updating time :: ${new Date()}`);
        setCurrentTime(new Date());
    }, 1000);

    return () => {
        if (timerInterval) {
            clearInterval(timerInterval);
        }
    }
}, []);
```

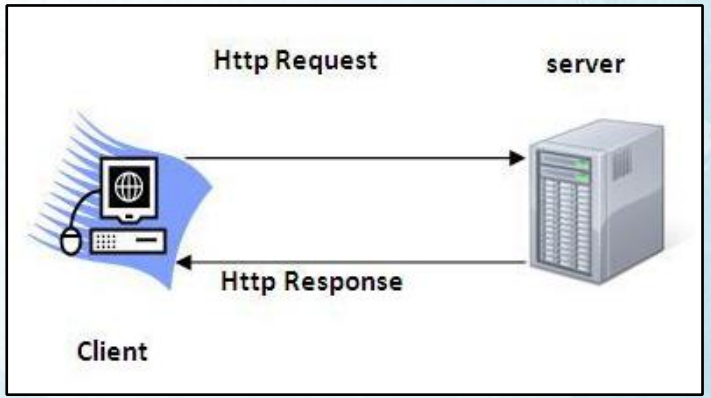
- View the output: Refresh the screen, now you can see the time getting updated only when it is shown.

7.4 Make HTTP request in React

7.4 Make HTTP request in React

HTTP

- HTTP stands for **H**yper **T**ext **T**ransfer **P**rotocol.
- Communication between clients and servers is done by requests and responses.
  - A **client** (a browser) sends an HTTP request to the web
  - A **web server** receives the request.
  - The server runs an application to process the request.
  - The server returns an HTTP response (output) to the browser.
  - The client (the browser) receives the response.



7.4 Make HTTP request in React

- In a typical web application, the client makes an HTTP request through the browser and the server sends an HTML page in the response with data.
- But in a single-page application (SPA), we have only one page, and whenever a client makes an HTTP request to the server it generally responds with JSON/XML formatted data.
- For making HTTP requests we have some of the below options –
  - XmlHttpRequest
  - Windows fetch
  - Axios
- Where XmlHttpRequest and Windows fetch methods are default JavaScript methods to use them we need not install any libraries.
- In the initial stages, XMLHttpRequest is used to fetch XML data over HTTP hence the name. But today it can be used with protocols other than HTTP and it can fetch data not only in the form of XML but also JSON, HTML, or plain text.

7.4 Make HTTP request in React

- The Fetch API allows you to make network requests similar to XMLHttpRequest (XHR). The main difference is that the Fetch API uses Promises, which enables a simpler and cleaner API, avoiding callback hell and having to remember the complex API of XMLHttpRequest.
- Axios is a JavaScript library that allows you to connect with the backend API and manage requests made via the HTTP protocol.
- The advantage of Axios lies in its being promise-based, thus allowing the implementation of asynchronous code. The asynchronous code will allow, on a page, to load multiple elements simultaneously instead of sequentially, significantly reducing loading times.
- The Promise, which Axios is based on, is instead a JavaScript object that allows you to complete requests asynchronously, passing through three states (pending, satisfied, rejected).

7.4 Make HTTP request in React

Adding Axios to Your Project

- Create a new react app.

```
C:\workspace\react tutorial\create react app>create-react-app react-axios-example

Creating a new React app in C:\workspace\react tutorial\create react app\react-axios-example.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...
```

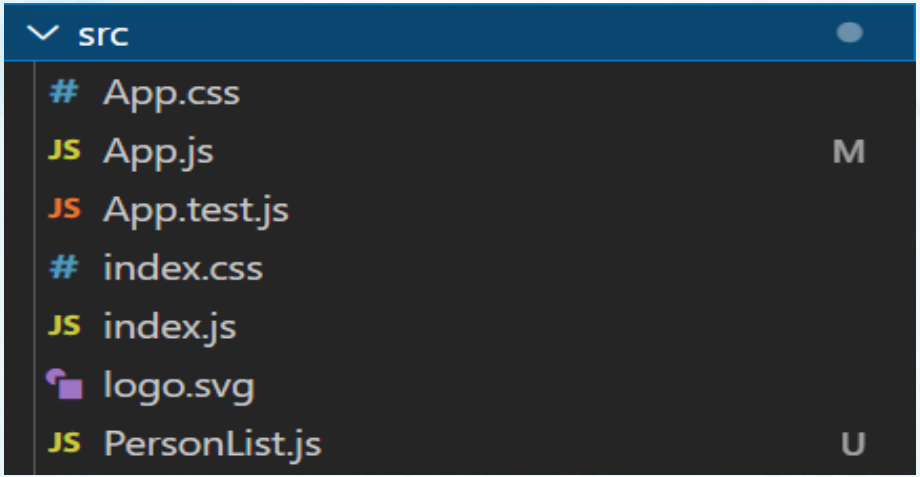
- The first part is to install Axios into your project with React.
- This step is very simple, as you just need to move to the project directory and then launch the Axios installation command.

```
C:\workspace\react tutorial\create react app>cd react-axios-example

C:\workspace\react tutorial\create react app\react-axios-example>npm install axios
```

7.5 Make a GET Request

7.5 Make a GET Request

- Create a new component and import Axios into it to send a GET request.
- Open the React project in the Visual Studio code, and create a new component name *PersonList*.



7.5 Make a GET Request

- Import React and Axios so that both can be used in the component.

```
import React from 'react';
import axios from 'axios';
```

- Then you hook into the *componentDidMount* lifecycle hook and perform a GET request.
- Use *axios.get(url)* with a URL from an API endpoint to get a promise which returns a response object. Inside the response object, there is data that is then assigned the value of person.

```
export default class PersonList extends React.Component {
  state = {
    persons: []
  }

  componentDidMount() {
    axios.get(`https://jsonplaceholder.typicode.com/users`)
      .then(res => {
        const persons = res.data;
        this.setState({ persons });
      })
  }
```

7.5 Make a GET Request

- Other information about the request, such as the status code under *res.status* or more information inside of *res.request*.
- Add this component to App.js

```
render() {
  return (
    <ul>
      {
        this.state.persons
          .map(person =>
            <li key={person.id}>{person.name}</li>
          )
      }
    </ul>
  )
}
```

App.js > ...

```
import './App.css';
import PersonList from './PersonList.js';


function App() {
  return (
    <div ClassName="App">
      <PersonList/>
    </div>
  )
}

export default App;
```

7.5 Make a GET Request

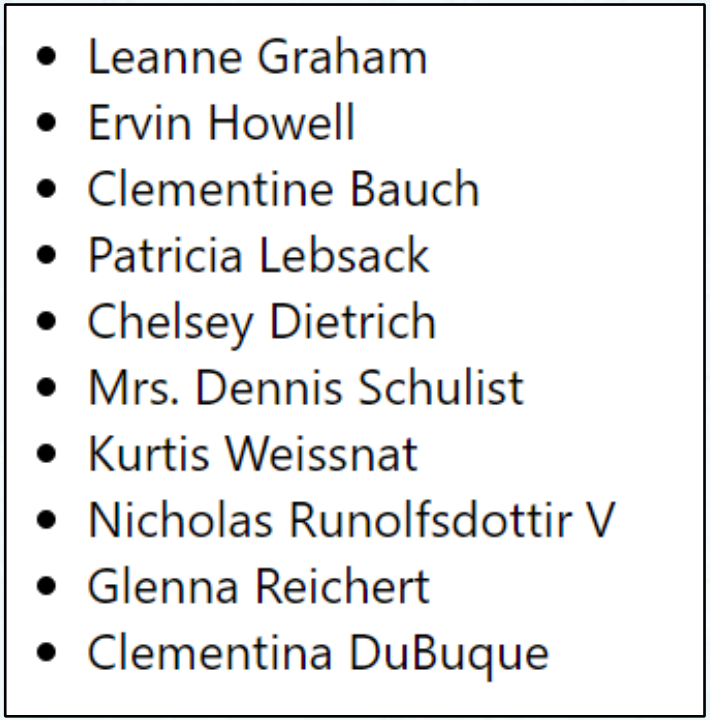
- Run the app and view the output in the browser.

```
C:\workspace\react tutorial\create react app\react-axios-example>npm start

> react-axios-example@0.1.0 start
> react-scripts start

(node:2308) [DEP_WEBPACK_DEV_SERVER_ON_AFTER_SETUP_MIDDLEWARE] DeprecationWarning: 'onA
(Use `node --trace-deprecation ...` to show where the warning was created)
(node:2308) [DEP_WEBPACK_DEV_SERVER_ON_BEFORE_SETUP_MIDDLEWARE] DeprecationWarning: 'on
.
Starting the development server...
```

```
import React from 'react';
import axios from 'axios';

export default class PersonList extends React.Component {
  state = {
    persons: []
  }

  componentDidMount() {
    axios.get(`https://jsonplaceholder.typicode.com/users`)
      .then(res => {
        const persons = res.data;
        this.setState({ persons });
      })
  }

  render() {
    return (
      <ul>
        {
          this.state.persons
            .map(person =>
              <li key={person.id}>{person.name}</li>
            )
        }
      </ul>
    )
  }
}
```

7.5 Make a GET Request

- Output:



7.6 Make a POST Request

7.6 Make a POST Request

- Create a new component named *PersonAdd*.
- Create a form that allows for user input and subsequently POSTs the content to an API.

```
<form onSubmit={this.handleSubmit}>
  <label>
    Person Name:
    <input type="text" name="name" onChange={this.handleChange} />
  </label>
  <button type="submit">Add</button>
</form>
```

- Import React and Axios so that both can be used in the component.

```
import React from 'react';
import axios from 'axios';
```

7.6 Make a POST Request

- Inside the *handleSubmit* function, you prevent the default action of the form. Then update the state to the user input.

```
handleSubmit = event => {
  event.preventDefault();

  const user = {
    name: this.state.name
  };
```

- Using POST gives you the same response object with information that you can use inside of a then call.

7.6 Make a POST Request

- To complete the POST request, you first capture the user input. Then you add the input along with the POST request, which will give you a response. You can then *console.log* the response, which should show the user input in the form.

```
axios.post(`https://jsonplaceholder.typicode.com/users`, { user })
  .then(res => {
    console.log(res);
    console.log(res.data);
  })
```

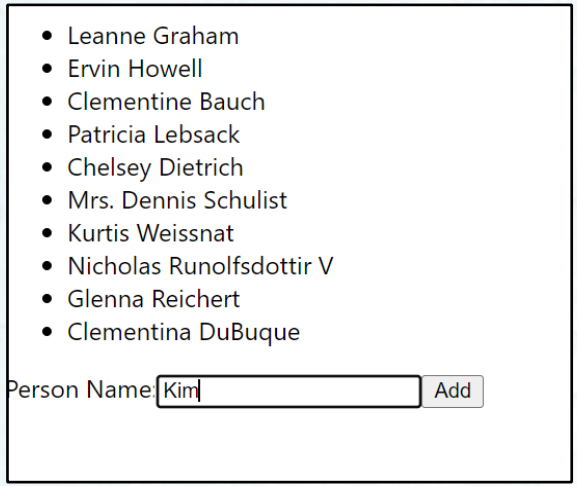
- Add this component to App.js

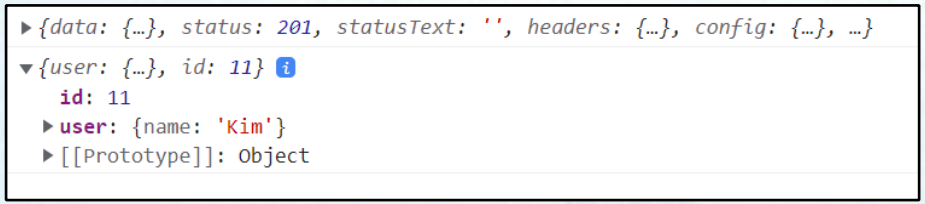
```
import PersonAdd from './PersonAdd';
```

```
<div ClassName="App">
  <PersonList/>
  <PersonAdd/>
</div>
```

7.6 Make a POST Request

- View the output in the browser and Check the console after submitting the form.





7.6 Make a POST Request

- Complete code

```
import React from 'react';
import axios from 'axios';

export default class PersonAdd extends React.Component {
  state = {
    name: ''
  }

  handleChange = event => {
    this.setState({ name: event.target.value });
  }

  handleSubmit = event => {
    event.preventDefault();

    const user = {
      name: this.state.name
    };

    axios.post(`https://jsonplaceholder.typicode.com/users`, { user })
      .then(res => {
        console.log(res);
        console.log(res.data);
      })
  }

  render() {
    return (
      <div>
        <form onSubmit={this.handleSubmit}>
          <label>
            Person Name:
            <input type="text" name="name" onChange={this.handleChange} />
          </label>
          <button type="submit">Add</button>
        </form>
      </div>
    )
  }
}
```

7.7 Make a DELETE Request

7.7 Make a DELETE Request

- Let's see how to delete items from an API using *axios.delete* and passing a URL as a parameter.
- Create a new component PersonRemove.
- The *res* object provides you with information about the request. You can then *console.log* that information again after the form is submitted.

```
import React from 'react';
import axios from 'axios';

export default class PersonRemove extends React.Component {
  state = {
    id: ''
  }

  handleChange = event => {
    this.setState({ id: event.target.value });
  }

  handleSubmit = event => {
    event.preventDefault();

    axios.delete(`https://jsonplaceholder.typicode.com/users/${this.state.id}`)
      .then(res => {
        console.log(res);
        console.log(res.data);
      })
  }
```

7.7 Make a DELETE Request

```
render() {
  return (
    <div>
      <form onSubmit={this.handleSubmit}>
        <label>
          Person ID:
          <input type="number" name="id" onChange={this.handleChange} />
        </label>
        <button type="submit">Delete</button>
      </form>
    </div>
  )
 }
}
```

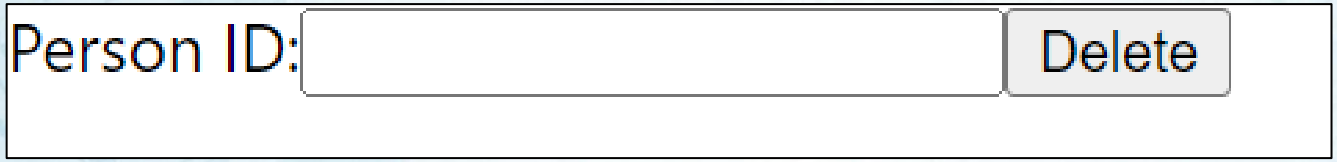
- Add this component to App.js

```
import PersonRemove from './PersonRemove';
```

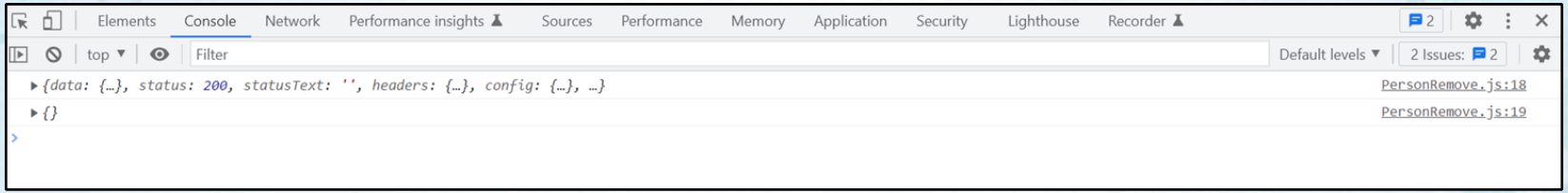
```
<div ClassName="App">
  <PersonList/>
  <PersonAdd/>
  <PersonRemove/>
</div>
```

7.7 Make a DELETE Request

- View the application in the browser. You will be presented with a form for removing users.



- View the console log after submitting the form.



7.8 Use Async and Await

7.8 Use Async and Await

- The *await* keyword resolves the promise and returns the value. The value can then be assigned to a variable.
- Let's use *async* and *await* to work with promises in DELETE Request.

```
handleSubmit = async event => {
  event.preventDefault();

  const response = await axios.delete(`https://jsonplaceholder.typicode.com/users/${this.state.id}`);
  console.log(response);
}
```

- View the output, there won't be any change in the output.

7.9 Deploy React Apps in GitHub Pages

7.9 Deploy React Apps in GitHub Pages

GitHub Pages

What is GitHub pages?

- GitHub Pages is a static website hosting service and it is free of charge.
- It takes resources like HTML, CSS, and JavaScript files directly from the GitHub repository.
- We can host our website on GitHub's with github.io domain or our own custom domain name.

Types of GitHub pages:

- There are three types of GitHub Pages sites which are,
  - User site
  - Project site
  - Organization site.

7.9 Deploy React Apps in GitHub Pages

GitHub Pages

- Project site:

- Project sites are connected to a specific project hosted on GitHub, such as a JavaScript library.
- User and organization sites are connected to a specific GitHub account.
- User site:
  - The user site is connected to a specific GitHub account.
  - You should have a user account on GitHub.
  - To publish a user site, you must create a GitHub repo owned by your user account and the repo name should be **username.github.io.** For Example, If your user name is **TestName** your repo name should be **testname.github.io.**
  - If you are not using any custom domain, then your website will be available at https://username.github.io.

7.9 Deploy React Apps in GitHub Pages

GitHub Pages

- Organization site:
  - The organization site is connected to a specific GitHub account.
  - To publish an organization site, you must create a repo owned by an organization and the repo name should be **organization.github.io.**
  - If you are not using any custom domain, then your website will be available at https://organization.github.io
  - You can create only one user or organization site for each user account on GitHub. But we can create unlimited Project sites which can be owned by an organization or a user account.

7.9 Deploy React Apps in GitHub Pages

GitHub Pages

How to Publish the React application on GitHub pages?

- Step 1: Create an account on GitHub.
- Step 2: Create a Repository on GitHub.
- Step 3: Deploy the React Application using the GitHub pages.
- Step 4: Commit and Push the codebase(React Application) into GitHub repo.

Step 1: Create an account on GitHub

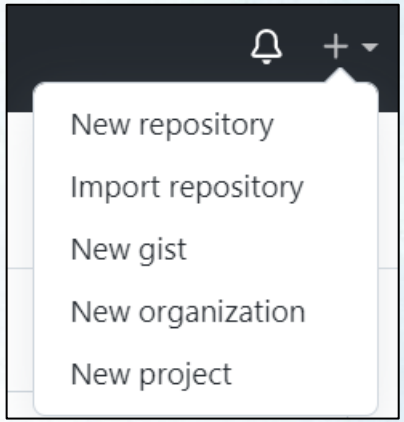
- Open the GitHub link
- Create an account using your personal email
- To secure your GitHub account, you can enable MFA (multi-factor authentication) login for your account.

7.9 Deploy React Apps in GitHub Pages

GitHub Pages

Step 2: Create Repository on GitHub

- Log in to your GitHub account.
- Create the new repo by clicking the "+" icon at the right top of the page
- Enter the repo name
- Click on the "Create repository" button.
- Now your repo has been created.



7.9 Deploy React Apps in GitHub Pages

GitHub Pages

Step 3: Deploy the React Application using GitHub pages

- **Add GitHub Pages dependency package:** Install "gh-pages" package.

```
webpack compiled successfully
Terminate batch job (Y/N)? y

C:\workspace\react tutorial\create react app\react-axios-example>npm install gh-pages - save-dev
```

- Add homepage property to package.json file: "homepage": "https://{username}.github.io/{reponame}"

```
"name": "react-axios-example",
"version": "0.1.0",
"private": true,
"homepage": "http://[redacted].github.io/git-pages-demo",
"dependencies": {
```

7.9 Deploy React Apps in GitHub Pages

GitHub Pages

- **Add deploy scripts to package.json file:** The "predeploy" command is used to bundle the react application and the "deploy" command helps to deploy the bundled file.

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject",
  "predeploy": "npm run build",
  "deploy": "gh-pages -d build"
},
```

- Create a remote GitHub repository: Initialize the Git using "git init" command. Add it as remote using "git remote add origin {your-github-repository-url.git}" command.

```
eact app\react-axios-example>git init
y in C:/workspace/react tutorial/create react app/react-axios-example/.git/

eact app\react-axios-example>git remote add origin https://github.com/[redacted].git
```

7.9 Deploy React Apps in GitHub Pages

GitHub Pages

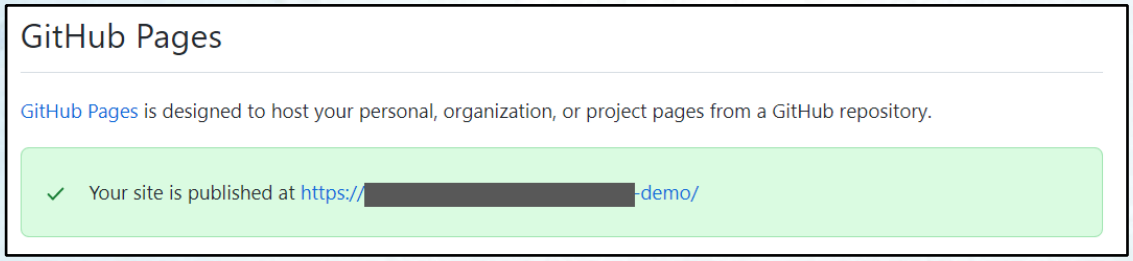
- **Deploy the Application to GitHub pages:** Deploy your react application to GitHub Pages using the command “npm run deploy”.

```
C:\workspace\react tutorial\create react app\react-axios-example>npm run deploy

> react-axios-example@0.1.0 predeploy
> npm run build


> react-axios-example@0.1.0 build
> react-scripts build

Creating an optimized production build...
Compiled successfully.
```

7.9 Deploy React Apps in GitHub Pages

GitHub Pages

- **Access deployed site:** To get the published URL, go to your GitHub Repo and follow the steps:
  - Click settings menu.
  - Go to the “Pages” Section.
  - You can see the “Your site is published” message.
  - Select branch to **"gh-pages"** and click on the **"Save"** button.
  - Access the deployed site using the published URL.



- **Step 4: Commit and Push the codebase:** Commit and push your code changes to the GitHub repo. It is an **optional** one. But, for maintaining the code base we can do this step.

7.10 Assignment

7.10 Assignment

Outline:

To create a counter app that increments and decrements the current value on the button clicks using the React **useState** hooks and deploy the app on GitHub pages.