Software Engineering

Module 7

React JS 2/2

© 2022 Institute of Data

Institute of Data

Agenda

Section 1 : Hooks

Section 2: Context

Section 3 : Routing

Section 4 : Integration with Libraries

Section 5: React UI Frameworks

Section 6 : Other Advanced Concepts

Institute of Data

Section 1: Hooks

Hooks are new to React 16.8. They enable you to use state and other React features without having to write a class.

Hooks

Motivation

- With `render props` and `HOC`, you need to **restructure your components** when you use them, which can be **cumbersome** and make code harder to follow
- With **Hooks**, you can **extract stateful logic** from a component so it can be **tested independently** and **reused**. Hooks allow you to **reuse** stateful logic **without changing** your component **hierarchy**.
- **Hooks** let you **split** one **component** into **smaller functions** based on what pieces are **related** (such as setting up a **subscription** or **fetching data**)

©2022 Institute of Data

Hooks

Definition

Hooks are **functions** that let you **"hook into"** React **state** and **lifecycle** features from **function** components. Hooks do **not work** inside **classes**.

Type of Hooks

- **Built-in** hooks
- **Custom** hooks

©2022 Institute of Data

Hooks

useState Hook

```
 1:  import React, { useState } from 'react';
 2
 3:  function Example() {
 4:    const [count, setCount] = useState(0);
 5:
 6:    return (
 7:      <div>
 8:        <p>You clicked {count} times</p>
 9:        <button onClick={() => setCount(count + 1)}>
10:         Click me
11:        </button>
12:      </div>
13:    );
14:  }
```


©2022 Institute of Data

Hooks

- Line **1**: We **import** the useState Hook from React. It lets us keep local state in a **function** component.
- Line **4**: Inside the Example component, a new **state** variable is declared by calling the `useState` Hook. It returns a **pair of values**. We are calling our variable `count` because it holds the number of button clicks. We initialize it to zero by passing 0 as the only `useState` argument. The second returned item is itself a function. It lets us update the count so we name it `setCount`.
- Line **9**: When the user clicks, we call `setCount` with a **new** value. React will then **re-render** the Example component, passing the **new count value** to it.

©2022 Institute of Data

Hooks

`useEffect` Hook

speaking of **lifecycle methods**, you can think of `useEffect` Hook as **componentDidMount**, **componentDidUpdate**, and **componentWillUnmount** combined

Use Cases:

- **Data** fetching.
- Setting up a **subscription**.
- manually **changing the DOM**.

©2022 Institute of Data

Hooks

useEffect Hook

- **lifecycle methods** in **Class** component

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }
  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }
  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }
  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
```

©2022 Institute of Data

Hooks

`useEffect` Hook

- combined **"lifecycle methods"** in Hook

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

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
```

©2022 Institute of Data

Hooks

useEffect Hook with Cleanup

- In **Class** component.

```
class FriendStatus extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }
  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
  componentWillUnmount() {  // clean up and unsubscribe
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }
  render() {
    if (this.state.isOnline === null) {
      return 'Loading...';
    }
    return this.state.isOnline ? 'Online' : 'Offline';
  }
```

©2022 Institute of Data

Hooks

`useEffect` Hook with Cleanup

- In **function** component.

```
function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // Specify how to clean up after this effect:
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
```

©2022 Institute of Data

Hooks

`useEffect` Hook

Optimizing **Performance** by **Skipping Effects**.

- How it is achieved in **Class** component.

```
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    document.title = `You clicked ${this.state.count} times`;
  }
}
```

- How it is achieved in **Function** component.

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // Only re-run the effect if count changes
```


©2022 Institute of Data

Rules of Hooks

Only Call Hooks at the Top Level

Don't call Hooks inside loops, conditions, or nested functions. Instead, always use Hooks at the **top level** of your React function, **before** any early **returns**. By following this rule, you ensure that Hooks are **called in the same order** each time a component renders. That's what allows React to correctly **preserve the state** of Hooks between multiple `useState` and `useEffect` calls.

©2022 Institute of Data

Rules of Hooks

Only Call Hooks from React Functions

Do not call Hooks from regular JavaScript functions. Instead, you can:

✅ Call Hooks from React **function components**.

✅ Call Hooks from **custom Hooks**.

By following this rule, you ensure that all **stateful logic** in a component is clearly **visible** from its **source code**.

©2022 Institute of Data

Custom Hooks

Customer Hook lets you **extract component logic** into **reusable functions**.

```
// extract custom hooks
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}
//use custom cooks
function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```


©2022 Institute of Data

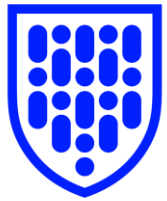
Custom Hooks

Pass Information Between Hooks

Since Hooks are **functions**, we can pass **information between** them.

```
const friendList = [
  { id: 1, name: 'Phoebe' },
  { id: 2, name: 'Rachel' },
  { id: 3, name: 'Ross' },
];

function ChatRecipientPicker() {
  const [recipientID, setRecipientID] = useState(1);
  const isRecipientOnline = useFriendStatus(recipientID); // recipientID state is passed to hook: useFriendStatus.
  return (
    <>
      <Circle color={isRecipientOnline ? 'green' : 'red'} />
      <select
        value={recipientID}
        onChange={e => setRecipientID(Number(e.target.value))}
      >
        {friendList.map(friend => (
          <option key={friend.id} value={friend.id}>
            {friend.name}
          </option>
        ))}
      </select>
    </>
  );
}
```


©2022 Institute of Data



Exercise

- Create the React Hooks version of the app in lab2.
- Create a Clock component and add to the current app.

© 2022 Institute of Data

Institute of Data

Section 2: Context

Context

Context provides a way to **pass data through** the **component tree without** having to **pass props** down **manually at every level**.

- **When** to Use Context: **Context** is designed to **share data** that can be considered **"global"** for **a tree** of React components, eg. the current **authenticated user**, **UI theme**, **locale preference**, **data cache** etc.
- **Before** You Use Context: Context is **primarily used** when some **data** needs to be **accessible** by many **components** at different **nesting levels**

©2022 Institute of Data

Context

API

- `React.createContext`: Creates a **Context object**. When React renders a **component** that **subscribes** to this **Context object** it will read the current **context value** from the **closest matching Provider** above it in the tree

```
const MyContext = React.createContext(defaultValue);
```

- The **defaultValue** argument is only used when a component **does not have a matching Provider** above it in the tree.

©2022 Institute of Data

Context

API

- `Context.Provider`: Every Context object comes with a Provider React component that allows **consuming components** to **subscribe** to **context changes**.

```
<MyContext.Provider value={/* some value */}>
```

- `value` prop changes cause **rendering for all its consuming components** that are **descendants** of this `Provider`. The consumer is **updated** even when an **ancestor** component **skips** an update.

©2022 Institute of Data

Context

API

- `Class.contextType`: this property on a class can **be assigned** a **Context object** created by `React.createContext()`.

```
class MyClass extends React.Component {
  componentDidMount() {
    let value = this.context;
    /* perform a side-effect at mount using the value of MyContext */
  }
  componentDidUpdate() {
    let value = this.context;
    /* ... */
  }
  componentWillUnmount() {
    let value = this.context;
    /* ... */
  }
  render() {
    let value = this.context;
    /* render something based on the value of MyContext */
  }
}
MyClass.contextType = MyContext;
```

©2022 Institute of Data

Context

API

- `Context.Consumer`: A React **component** that **subscribes** to context **changes**.

```
<MyContext.Consumer>
  {value => /* render something based on the context value */} // the function as child
</MyContext.Consumer>
```

- The function **receives** the current **context value** and **returns** a React **node**. The **value argument** passed to the function will be equal to the value prop of the **closest Provider** for this context **above** in the tree.

©2022 Institute of Data

Context

API

- `Context.displayName`: React **DevTools** uses this string to determine what to **display for the context**.

```
const MyContext = React.createContext(/* some value */);
MyContext.displayName = 'MyDisplayName';

<MyContext.Provider> // "MyDisplayName.Provider" in DevTools
<MyContext.Consumer> // "MyDisplayName.Consumer" in DevTools
```


©2022 Institute of Data

Context

Consuming Multiple Contexts

To keep **context re-rendering fast**, React needs to make each **context consumer** a **separate node** in the tree.

```
const ThemeContext = React.createContext('light');
const UserContext = React.createContext({ name: 'Guest'});
class App extends React.Component {
  render() {
    const {signedInUser, theme} = this.props;
    return (
      <ThemeContext.Provider value={theme}>
        <UserContext.Provider value={signedInUser}>
          <Content />
        </UserContext.Provider>
      </ThemeContext.Provider>
    );
  }
}
```

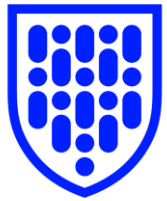

©2022 Institute of Data

Context

Consuming Multiple Contexts

```
function Content() {
  return (
    <ThemeContext.Consumer>
      {theme => (
        <UserContext.Consumer>
          {user => (
            <ProfilePage user={user} theme={theme} />
          )}
        </UserContext.Consumer>
      )}
    </ThemeContext.Consumer>
  );
}
```


©2022 Institute of Data



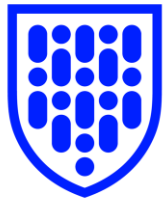
useContext Hook

To consume contexts in React, we simply use the `useContext()` hook.

It accepts a context object (the value returned from `React.createContext`) and returns the current context value for that context. The current context value is determined by the value prop of the nearest `<MyContext.Provider>` above the calling component in the tree.

```
import React from 'react';

const App = () => {
const value = React.useContext(MyContext);

  return (
    <div>
      <h1>Posts</h1>
    </div>
  );
};

export default App;
```


© 2022 Institute of Data

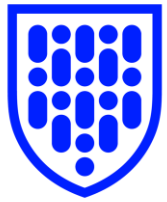


useContext Hook

Don’t forget that the argument to `useContext` must be the *context object itself*:

```
// Correct:
useContext(MyContext)

// Incorrect:
useContext(MyContext.Consumer)

// Incorrect:
useContext(MyContext.Provider)
```


© 2022 Institute of Data



useContext Hook

Example:

```
const themes = {
  light: {
    foreground: "#000000",
    background: "#eeeeee"
  },
  dark: {
    foreground: "#ffffff",
    background: "#222222"
  }
};
const ThemeContext = React.createContext(themes.light);

function App() {
  return (
    < ThemeContext.Provider value={themes.dark}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar(props) {
  return (
    <div> <ThemedButton /> </div>
  );
}

function ThemedButton() {
  const theme = React.useContext(ThemeContext);
  return (
    <button style={{ background: theme.background, color: theme.foreground
}}>
      I am styled by theme context!
    </button>
  );
}
```


© 2022 Institute of Data

Context

Caveats

`context` uses **reference identity** to determine **when to re-render**,

- The code below will **re-render** all consumers every time the **Provider re-renders** because a **new object** is always created for `value`.

```
class App extends React.Component {
  render() {
    return (
      <MyContext.Provider value={{something: 'something'}}>
        <Toolbar /> // will be rerendered
      </MyContext.Provider>
    );
  }
}
```


©2022 Institute of Data

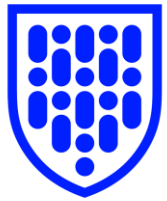
Context

Caveats

- Solution: **lift** the value into the **parent's state**.

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: {something: 'something'},
    };
  }

  render() {
    return (
      <MyContext.Provider value={this.state.value}>
        <Toolbar />
      </MyContext.Provider>
    );
  }
}
```


©2022 Institute of Data



Exercise

- Completely migrate the Emoji component into a separate one with its own state (leave App component to be stateless).

- Display another emoji, which is the same as that in the Emoji component and reflects when ‘Change Mood’, next to the clock. (Hint: useContext).

© 2022 Institute of Data

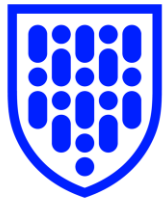
Institute of Data

Section 3 : Routing

You might be wondering if we have so many pages on an app and we want to navigate between them. How can we do it?

Then, if we want to pass some data from one page to another when we are navigated, how can we do it? And what can we use to do so?

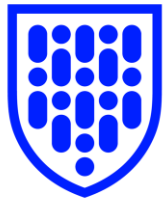
In this section, we will explore React Router which is a widely-used library for React application routing.



Introduction

React Router is a fully-featured client and server-side routing library for React, a JavaScript library for building user interfaces. React Router runs anywhere React runs; on the web, on the server with node.js, and on React Native.

© 2022 Institute of Data

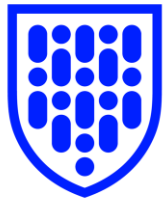


Routing Concepts

- **URL**:
  The URL in the address bar. Many use the term "URL" and "route" interchangeably, but this is **not a route in React Router**, it's **just a URL**.

- **Location**:
  This is a React Router specific object that is based on the built-in browser's `window.location` object. It represents "where the user is at". It's mostly an object representation of the URL but has a bit more to it than that.

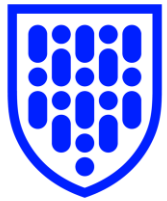
© 2022 Institute of Data



Routing Concepts

- **History**:
  An object that allows React Router to subscribe to changes in the URL as well as providing APIs to manipulate the browser history stack programmatically.
- **History Action**:
  One of **POP, PUSH**, or **REPLACE**. Users can arrive at a URL for one of these three reasons:
  - A push when a new entry is added to the history stack (typically a link click or the programmer forced a navigation).
  - A replace is similar except it replaces the current entry on the stack instead of pushing a new one.
  - Finally, a pop happens when the user clicks the back or forward buttons in the browser chrome.

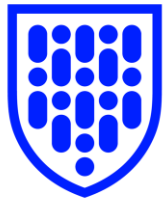
© 2022 Institute of Data



Routing Concepts

- **URL Params:**
  The parsed values from the URL that matched a dynamic segment.
- **Router:**
  Stateful, top-level component that makes all the other components and hooks work.
- **Route:**
  An object or Route Element typically with a shape of `{path,element}` or `<Routepath element>`.
  The `path` is a path pattern. When the path pattern matches the current URL, the element will be rendered.
- **Route Element:**
  Or `<Route>`. This element's props are read to create a route by `<Routes>`, but otherwise does nothing.

© 2022 Institute of Data



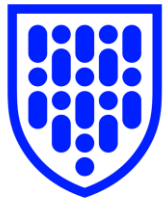
BrowserRouter

`<BrowserRouter>` is the recommended interface for running React Router in a web browser.

A `<BrowserRouter>` stores the current location in the browser's address bar using clean URLs. It navigates using the browser's built-in history stack.

```
import React from "react";
import ReactDOM from "react-dom";
import { BrowserRouter } from "react-router-dom";

ReactDOM.render(
  <BrowserRouter>
    {/* The rest of your app goes here */}
  </BrowserRouter>,
  root
);
```


© 2022 Institute of Data



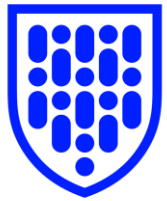
Link

A `<Link>` is an element that lets the user navigate to another page by clicking or tapping on it. In react-router-dom, a `<Link>` renders an accessible `<a>` element with a real `href` that points to the resource it's linking to.

This means that things like right-clicking a `<Link>` work as you'd expect. You can use `<Link reloadDocument>` to skip client side routing and let the browser handle the transition normally (as if it were an `<a href>`).

```jsx
import React from "react";
import { Link } from "react-router-dom";

function UsersIndexPage({ users }) {
  return (
    <div>
      <h1>Users</h1>
      <ul>
        {users.map(user => (
          <li key={user.id}>
            <Link to={user.id}>{user.name}</Link>
          </li>
        ))}
      </ul>
    </div>
  );
}
```


© 2022 Institute of Data

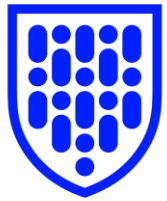


Navigate

A `<Navigate>` element changes the current location when it is rendered. It's a component wrapper around `useNavigate` and accepts all the same arguments as props.

```jsx
import React from "react";
import { Navigate } from "react-router-dom";

class LoginForm extends React.Component {
  state = { user: null, error: null };

  async handleSubmit(event) {
    event.preventDefault();
    try {
      let user = await login(event.target);
      this.setState({ user });
    } catch (error) {
      this.setState({ error });
    }
  }

  render() {
    let { user, error } = this.state;
    return (
      <div>
        {error && <p>{error.message}</p>}
        {user && (
          <Navigate to="/dashboard" replace={true} />
        )}
        <form onSubmit={event => this.handleSubmit(event)}>
          <input type="text" name="username" />
          <input type="password" name="password" />
        </form>
      </div>
    );
  }
}
```


© 2022 Institute of Data

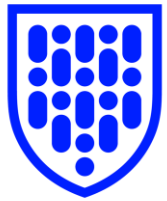


Outlet

An `<Outlet>` should be used in parent route elements to render their child route elements. This allows nested UI to show up when child routes are rendered. If the parent route matched exactly, it will render a child index route or nothing if there is no index route.

```
function Dashboard() {
  return (
    <div>
      <h1>Dashboard</h1>

      {/* This element will render either
<DashboardMessages> when the URL is
          "/messages", <DashboardTasks> at "/tasks", or
null if it is "/"
      */}
      <Outlet />
    </div>
  );
}

function App() {
  return (
    <Routes>
      <Route path="/" element={<Dashboard />}>
        <Route
          path="messages"
          element={<DashboardMessages />}
        />
        <Route path="tasks" element={<DashboardTasks />}
/>
      </Route>
    </Routes>
  );
}
```


© 2022 Institute of Data



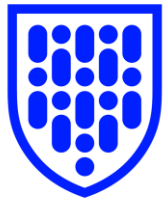
Route

`<Routes>` and `<Route>` are the primary ways to render something in React Router based on the current location. You can think about a `<Route>` kind of like an `if` statement; if its path matches the current URL, it renders its `element`! The `<Route caseSensitive>` prop determines if the matching should be done in a case-sensitive manner (defaults to false).

Whenever the location changes, `<Routes>` looks through all its `children` `<Route>` elements to find the best match and renders that branch of the UI. `<Route>` elements may be nested to indicate nested UI, which also correspond to nested URL paths.

```
<Routes>
  <Route path="/" element={<Dashboard />}>
    <Route
      path="messages"
      element={<DashboardMessages />}
    />
    <Route path="tasks" element={<DashboardTasks />} />
  </Route>
  <Route path="about" element={<AboutPage />} />
</Routes>
```


© 2022 Institute of Data



Nested Routes

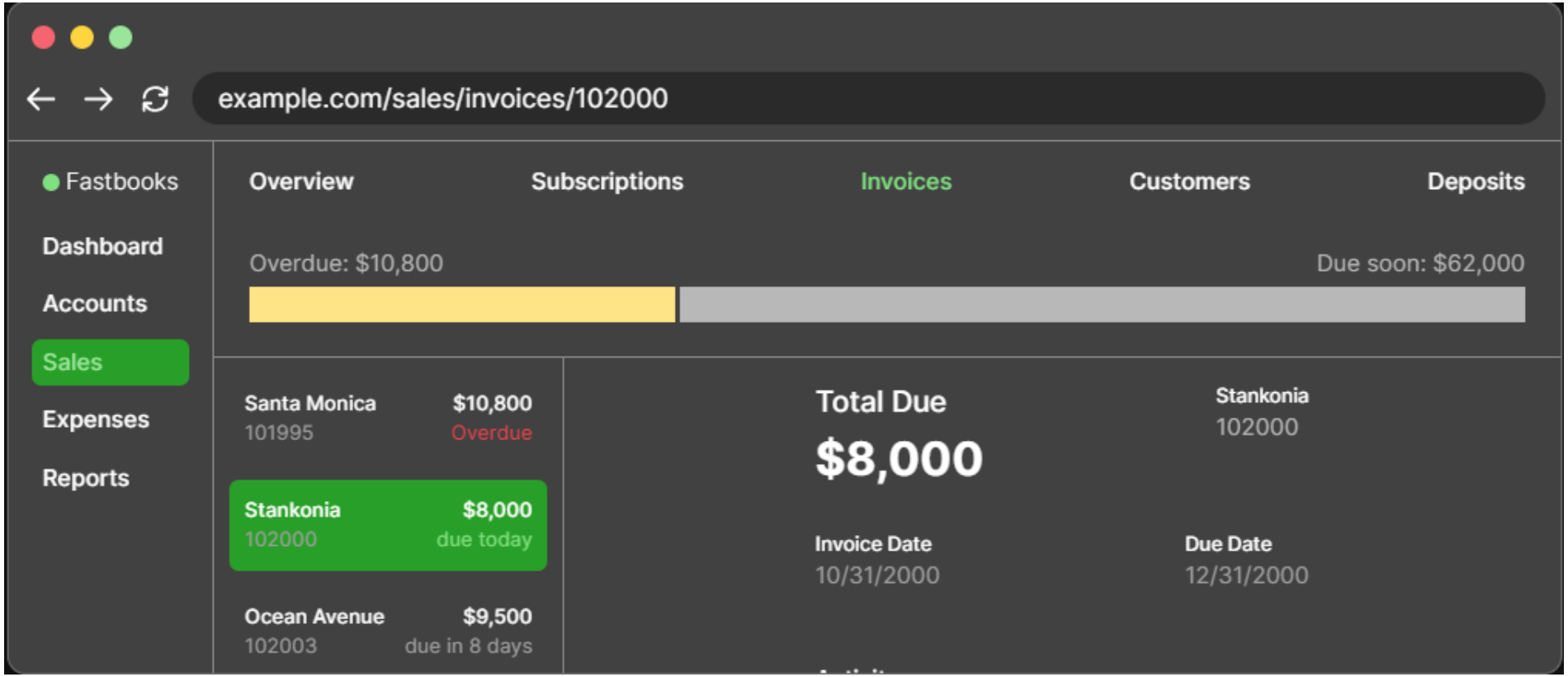
With React Router, this is all built-in. Nested routes add both segments to the URL and layouts to the UI hierarchy. As the URL changes, your layouts automatically change with it.

```
<Route path="/" element={<App />}>
  <Route path="sales" element={<Sales />}>
    <Route path="invoices" element={<Invoices />}>
      <Route path=":invoice" element={<Invoice />}
/>
    </Route>
  </Route>
</Route>
```

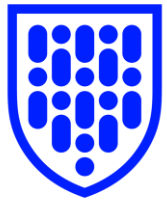

© 2022 Institute of Data

Route Declaration

Example output:



© 2022 Institute of Data



Exercise

- Create a basic routing app with 3 pages: Home, Posts, Dashboard.

© 2022 Institute of Data

Institute of Data

Section 4: Integrate with Libraries

Integrating with Other Libraries

- React is **unaware of changes** made to the DOM **outside** of React. It determines **updates** based on its own **internal representation**, and if the same DOM nodes are **manipulated by another library**, React gets confused and has no way to recover.
- The easiest way to **avoid conflicts** is to **prevent** the React component from **updating**. You can do this by **rendering elements** that React has **no reason** to **update**, like an empty `<div />`.

©2022 Institute of Data

Integrating with Other Libraries

- Integrate with `jQuery`

```
class Chosen extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
    this.$el.chosen();
  }
  componentWillUnmount() {
    this.$el.chosen('destroy');
  }
  render() {
    return (
      <div>
        <select className="Chosen-select" ref={el => this.el = el}>
          {this.props.children}
        </select>
      </div>
    );
  }
}
function Example() {
  return (
    <Chosen>
      <option>vanilla</option>
      <option>chocolate</option>
      <option>strawberry</option>
    </Chosen>
  );
}
```

©2022 Institute of Data

Integrating with Other Libraries

- Integrate with `Backbone.js`

```
function Paragraph(props) {
  return <p>{props.text}</p>;
}

const ParagraphView = Backbone.View.extend({
  render() {
    const text = this.model.get('text');
    ReactDOM.render(<Paragraph text={text} />, this.el);
    return this;
  },
  remove() {
    ReactDOM.unmountComponentAtNode(this.el);
    Backbone.View.prototype.remove.call(this);
  }
});
```

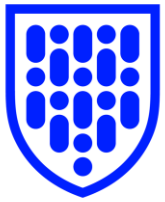

©2022 Institute of Data

Institute of Data

Section 5 : React UI Framework

Web applications are always required to be attractive, informative and user-friendly. A web application with a nice colour scheme and good looking layout definitely is preferred to its counterparts with boring and old-fashioned appearance.

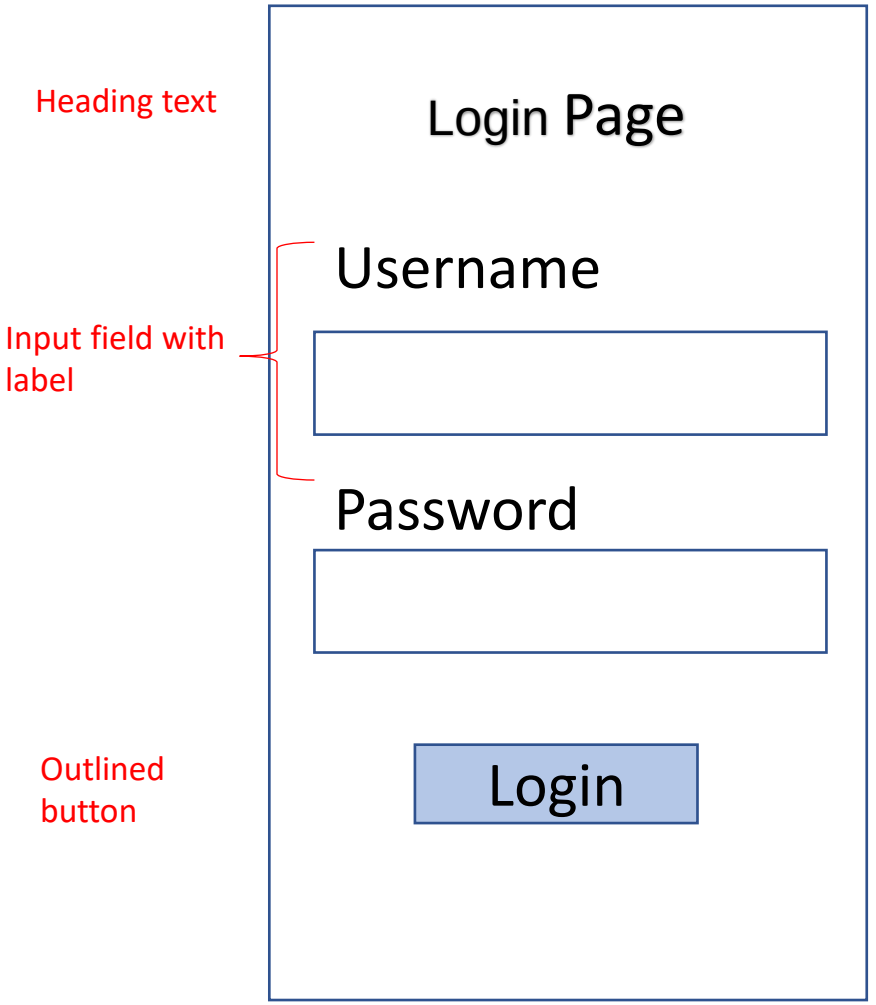
However, not all programmers are good at creating good-looking components. Therefore, the creation of UI frameworks saves their live and make it easier.



What is a UI Framework?

As you know, in React, everything is a component. So, there are countless of components that we will need it everywhere in a project. And they need to be consistent regardless where they are placed. Take the **Login Page** on the right for example.

UI Frameworks are the results of the effort to provide **reusable, easy-to-use, good looking**, and, very importantly**, consistently designed** components for everyone.



© 2022 Institute of Data

Popular React UI Frameworks

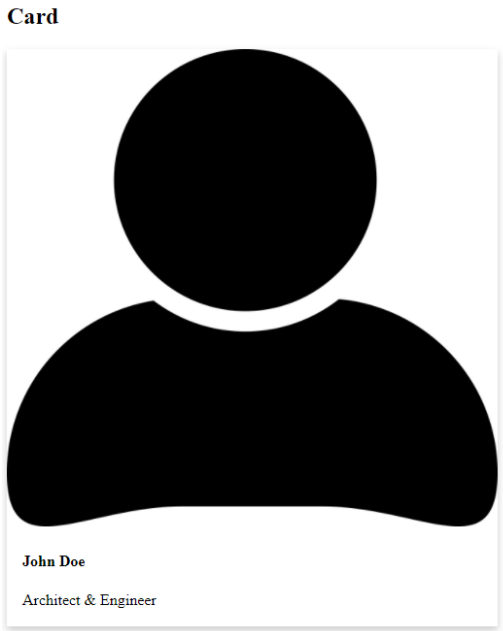
- **Material UI (MUI):** provides a robust, customisable, and accessible library of foundational and advanced components, helping you to build your own design system and develop React applications faster.
- **React Bootstrap**: The most popular front-end framework rebuilt for React.
- **Ant Design React**: dedicated to providing a good development experience for programmers.
- **Semantic UI React, Chakra UI,** etc.

Each framework has its own style of design and syntax. However, they all have the same objective to bring easier and quicker frontend developing experiences for developers.

© 2022 Institute of Data

HTML Card

Let’s have a look at how we create card in normal HTML.

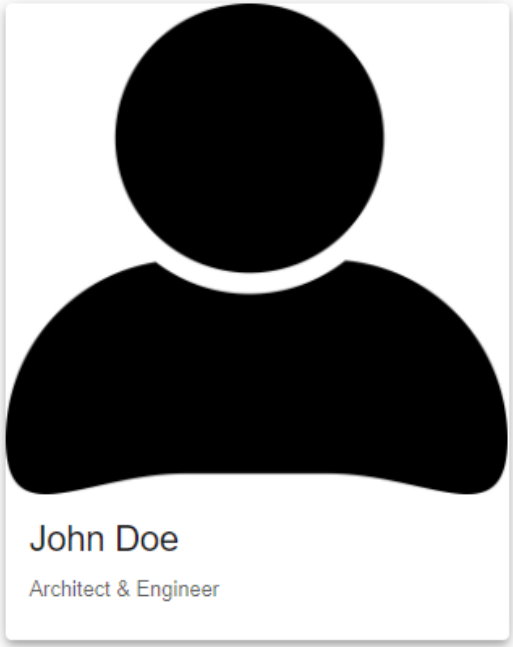



```
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width, initial-
scale=1">
<style>
.card {
  box-shadow: 0 4px 8px 0 rgba(0,0,0,0.2);
  transition: 0.3s;
  width: 40%;
}

.card:hover {
  box-shadow: 0 8px 16px 0 rgba(0,0,0,0.2);
}

.container {
  padding: 2px 16px;
}
</style>
</head>
<body>

<h2>Card</h2>

<div class="card">
  <img src="img_avatar.png" alt="Avatar" style="width:100%">
  <div class="container">
    <h4><b>John Doe</b></h4>
    <p>Architect & Engineer</p>
  </div>
</div>

</body>
</html>
```


© 2022 Institute of Data

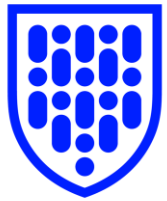
MUI Card

Now, let’s create the same card using MUI.




```jsx
import React from 'react';

import Card from '@mui/material/Card';
import CardContent from '@mui/material/CardContent';
import CardMedia from '@mui/material/CardMedia';
import Typography from '@mui/material/Typography';

const App = () => {

  return (
    <Card sx={{ maxWidth: 345, margin: 10 }} elevation={5}>
      <CardMedia
        component="img"
        image="img_avatar.png"
        alt="avatar"
      />
      <CardContent>
        <Typography gutterBottom variant="h5" component="div">
          John Doe
        </Typography>
        <Typography variant="body2" color="text.secondary">
          Architect & Engineer
        </Typography>
      </CardContent>
    </Card>
  )
}

export default App;
```


© 2022 Institute of Data



React UI Frameworks

So, as we can see, creating components using UI frameworks is easier, quicker and less complicated than using plain HTML and CSS. Furthermore, using frameworks provide us a smoother look on the UI.

UI frameworks provide us almost all components that we need to build any web applications from basic to advanced components. There are basic components like *Card, Container, Table, etc.* More advanced components can be listed as: *Grid, Collapsible, Breadcrumbs, Modal, etc*. All of them can be implemented with ease.

© 2022 Institute of Data

Institute of Data

Section 6 : Other Advanced Concepts

Error Boundaries

React components that **catch JavaScript errors** anywhere in their **child component tree**, **log** those **errors**, and **display** a **fallback UI** instead of the component tree that crashed.

- use `static getDerivedStateFromError()` to render a **fallback UI** after **error** has been **thrown**.
- use `componentDidCatch()` to log error information.

©2022 Institute of Data

Error Boundaries

- Example

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }
  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
    return { hasError: true };
  }
  componentDidCatch(error, errorInfo) {
    // You can also log the error to an error reporting service
    logErrorToMyService(error, errorInfo);
  }
  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }
    return this.props.children;
  }
}
```


©2022 Institute of Data

Forwarding Refs

- Ref forwarding is a **technique** for automatically **passing a ref** through a component to one of **its children**

```
const FancyButton = React.forwardRef((props, ref) => (
  <button ref={ref} className="FancyButton">
    {props.children}
  </button>
));

// You can now get a ref directly to the DOM button:
const ref = React.createRef();
<FancyButton ref={ref}>Click me!</FancyButton>;
```


©2022 Institute of Data

Forwarding Refs

- Forwarding refs in **higher-order components** (HOC)

```
function logProps(Component) {
  class LogProps extends React.Component {
    componentDidUpdate(prevProps) {
      console.log('old props:', prevProps);
      console.log('new props:', this.props);
    }
    render() {
      const {forwardedRef, ...rest} = this.props;
      return <Component ref={forwardedRef} {...rest} />;
    }
  }
  return React.forwardRef((props, ref) => {
    return <LogProps {...props} forwardedRef={ref} />;
  });
}
```


©2022 Institute of Data

Fragments

Fragments let you **group** a list of **children without** adding extra **nodes** to the DOM.

- Usage:

```
class Columns extends React.Component {
  render() {
    return (
      <React.Fragment>
        <td>Hello</td>
        <td>World</td>
      </React.Fragment>
    );
  }
}
```


©2022 Institute of Data

Fragments

- Short Syntax

```
class Columns extends React.Component {
  render() {
    return (
      <>
        <td>Hello</td>
        <td>World</td>
      </>
    );
  }
}
```


©2022 Institute of Data

Fragments

- **Keyed** Fragments
  At the time of writing, `key` is the **only attribute** that can be passed to `Fragment`.

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        <React.Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </React.Fragment>
      ))}
    </dl>
  );
}
```


©2022 Institute of Data

Higher-Order Components

- An **advanced technique** in React for **reusing component** logic.
- Concretely, a higher-order component is a **function** that **takes** a component and **returns** a new component.
- Whereas a **component** transforms **props into UI**, a **higher-order** component transforms a **component into another component**.

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```


©2022 Institute of Data

Higher-Order Components

- **Conventionally**, HOC's name is prefixed with `with`.

```
function withSubscription(WrappedComponent, selectData) {
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.state = {
        data: selectData(DataSource, props)
      };
    }
    componentDidMount() {
      DataSource.addChangeListener(this.handleChange);
    }
    componentWillUnmount() {
      DataSource.removeChangeListener(this.handleChange);
    }
    handleChange() {
      this.setState({
        data: selectData(DataSource, this.props)
      });
    }
    render() {
      return <WrappedComponent data={this.state.data} {...this.props} />;
    }
  };
}
```


©2022 Institute of Data

Portals

Portals provide a **first-class way** to render **children** into a DOM node that exists **outside** the DOM **hierarchy** of the **parent component**.

- Syntax

```
ReactDOM.createPortal(child, container)
```

- Use case: when a **parent** component has an `overflow: hidden` or `z-index` style, but you need the **child** to visually **"break out"** of its **container**. eg. **dialogs**, **hovercards**, and **tooltips**.

```
render() {
  // React does *not* create a new div. It renders the children into `domNode`.
  // `domNode` is any valid DOM node, regardless of its location in the DOM.
  return ReactDOM.createPortal(
    this.props.children,
    domNode
  );
}
```

©2022 Institute of Data

Portals

- **Event Bubbling** Through Portals
  - Even though a portal can be **anywhere** in the **DOM tree**, it **behaves** like a **normal React child** in every other way. Features like `context` work exactly the same **regardless** of whether the child is a portal, as the portal still **exists** in the **React tree** regardless of **position** in the **DOM tree**.
  - This includes **event bubbling**. An **event** fired from inside a portal will **propagate to ancestors** in the containing **React tree**, even if those elements are **not ancestors** in the **DOM tree**.
  - **Catching an event bubbling up** from a portal in a parent component **allows** the development of more **flexible abstractions** that are **not inherently reliant** on portals.

©2022 Institute of Data

Refs and the DOM

Refs provide a way to **access DOM nodes** or **React elements** created in the **render method**.

When to use Refs

- Managing **focus**, text **selection**, or media **playback**.
- Triggering imperative **animations**.
- **Integrating** with **third-party** DOM libraries.

When not to use Refs

- Avoid using refs for **anything** that can be done **declaratively**.

©2022 Institute of Data

Refs and the DOM

Accessing Refs

- **Native Dom element**

```
<input type="text" ref={this.textInput} />
```

- **Class Component**

```
<CustomTextInput ref={this.textInput} /> //CustomTextInput is a Class component
```

- **Functional component**

  Functional component does **not** support `ref` as **prop**, however you can use `forwardRef` (possibly in conjunction with `useImperativeHandle`), or you can **convert** the component to a **class**, or **explicitly** pass a **ref as** a differently **named prop**

©2022 Institute of Data

Refs and the DOM

- **Exposing** DOM Refs **to Parent** Components
  In **rare** cases, you might want to have **access** to a **child's DOM** node from a **parent component**. This is generally **not recommended** because it **breaks** component **encapsulation**, but it can occasionally be **useful** for triggering **focus** or **measuring the size** or **position** of a child DOM node.
- **Callback** Refs
  Instead of passing a `ref` attribute **created** by **createRef()**, you pass a **function**. The function receives the React **component instance** or HTML **DOM element** as its argument, which can be **stored and accessed** elsewhere.

```
<CustomTextInput inputRef={el => this.inputElement = el} />
```


©2022 Institute of Data

Render Props

- **Render Props** refers to a **technique** for **sharing code** between React components using a **prop whose value is a function**.

```
<DataProvider render={data => (
  <h1>Hello {data.target}</h1>
)}/>
```

- Use **Render Props** for **Cross-Cutting** Concerns.
  Components are the **primary unit** of code reuse in React, but it is not always obvious **how to share the state or behavior** that one component encapsulates to other components that need that same state.

©2022 Institute of Data

Strict Mode

`StrictMode` is a tool for **highlighting** potential **problems** in an application.

- **Identifying** components with **unsafe lifecycles**.
- **Warning** about legacy **string ref** API usage.
- **Warning** about **deprecated** `findDOMNode` usage.
- **Detecting** unexpected **side effects** by **"double-invoking"** the methods.
- **Detecting** legacy **context API**.

©2022 Institute of Data

Uncontrolled Components

The **uncontrolled component** is the **alternative** for **controlled component**, where **form data** is handled by the **DOM itself**.

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.input = React.createRef();
  }
  handleSubmit(event) {
    alert('A name was submitted: ' + this.input.current.value);
    event.preventDefault();
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

©2022 Institute of Data

Uncontrolled Components

Default Values

In an **uncontrolled** component, a `defaultValue` attribute is used **instead of** `value`. **Changing** the value of **defaultValue** attribute after a component has **mounted** will **not** cause any **update** of the **value** in the DOM.

```
<form onSubmit={this.handleSubmit}>
  <label>
    Name:
    <input
      defaultValue="Bob"
      type="text"
      ref={this.input} />
  </label>
  <input type="submit" value="Submit" />
</form>
```


©2022 Institute of Data

Uncontrolled Components

The file input Tag

In React, an `<input type="file" />` is **always** an **uncontrolled** component because its value can **only be set** by a **user**, and **not programmatically**.

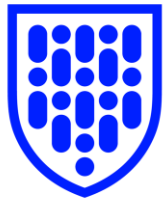
©2022 Institute of Data

Uncontrolled Components

- Example of **file input**

```
class FileInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.fileInput = React.createRef();
  }
  handleSubmit(event) {
    event.preventDefault();
    alert(`Selected file - ${this.fileInput.current.files[0].name}`);
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>Upload file: <input type="file" ref={this.fileInput} /></label>
        <button type="submit">Submit</button>
      </form>
    );
  }
}
```


©2022 Institute of Data



Exercise

- Create a basic media post app which can:
  - Display posts from the backend.
  - Create new posts.
- Use an UI framework for styling.

© 2022 Institute of Data