Institute of Data

Software Engineering

Module 7

React JS 2/2

Agenda

Section 1: Hooks

Hooks are a recent addition to React that enable developers to tap into (or ***'hook'*** into) different useful built-in functionalities of React from any component.

The most commonly used hook is **useState**, which we covered in the previous module. However, there are numerous **other hooks** available, and we can even create our **own**!

Hooks are **functions** identified by the prefix "**use**" to make their purpose clear. They always need to be called at the **top** of a component, usually the first thing a component does.

React Hooks - Rules of Hooks

Use Hooks at the Top Level : Avoid calling Hooks inside loops, conditions, or nested functions. Always use Hooks at the top level of your React function, before any early returns. Following this rule ensures that Hooks are called consistently in the same order during component rendering, allowing React to preserve their state correctly.

Call Hooks from React Functions : Don't call Hooks from regular JavaScript functions. Instead, you should:

- Call Hooks from React function components.
- Call Hooks from custom Hooks (more on this soon).

Institute of Data

React Hooks - useState recap

useState is a hook that lets us both **remember** a component-specific **state variable** in between renders, and **update** this variable, usually as a result of user interaction.

We pass it a **default value** for the state variable, and it returns an **array** (that we usually destructure) of two items: the **variable** itself, and a **function** for updating it.

useState allows us to:

- Manage A **Snapshot** Of All Data Needed To **Render** A Component At A Particular Point In Time
- **Share State** Data Between Components By Passing It As Props
- **Store**, **Organise** And **Update** State Data According To The React Guidelines And Best Practices.

React Hooks - useEffect

useEffect is another built-in React hook that lets us **synchronise our components** with external systems or data. It serves as an **"escape hatch"** that allows us to connect with **external** code or applications outside of React.

By using **effects**, we can run **additional code after rendering**, as they execute independently, at the **end** of a commit once the screen updates.

Common use-cases for using an effect include:

- **Fetching data** from an external system, ie. results for a search query
- **Synchronising** non-React JS widgets used by our components
- **Subscribing** to events, both native such as scroll listeners, and external APIs

Important: adding effects to components should be done **thoughtfully**. Effects are typically used to interact with external systems like browser APIs, third-party widgets, or networks. *If your effect only involves state, it might not be necessary.*

React Hooks - useEffect

React components follow a lifecycle: **mounting**, **updating**, and **unmounting**:

- **Mounting** occurs when a component is added to the screen.
- **Updating** happens when new props or state trigger a re-render of the component, often due to user interactions.
- **Unmounting** takes place when a component is removed from the screen.

Effects work **within** this lifecycle, but the component lifecycle **doesn't** directly apply to Effects.

Instead, consider Effects as **independent operations** that take place on one or more of the above lifecycle actions, which describe how to synchronize data from an external system with the current props and state (and vice versa).

useEffect Hook

useEffect accepts two arguments:

- A **function** to call when the effect runs (on **mounting** or **rendering**)
- An **array** of dependency variables

This effect sets up an interval to run every second, updating the time.

If the dependency array is **empty**, it means the effect will run only **once**, when the component is first **mounted**.

If the array is **missing**, the effect will run on **every re-render**.

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

// Renders a digital time that updates every second
function Clock() {

    const [date, setDate] = useState(new Date());

    useEffect(() => { // first arg is usually an arrow function
        setInterval(() => tick(), 1000);
        console.log('Clock component mounted');
    }, []); // second arg is an array of dependencies

    const tick = () => {
        setDate(new Date());
        console.log('tick'); // track the effect frequency
    };

    return (
        <div className="Clock">
            <h3>Digital Clock</h3>
            {date.toLocaleTimeString()}
        </div>
    );
}
// ++ Try removing the dependency array from useEffect
// and notice the difference in ‘tick’ log messages
// ++ Why do the console messages appear double?
```

useEffect Hook

So far we can see our **effect** running when the component is **mounted** into the DOM on initial render.

To also see what happens when the component is **unmounted**, i.e., **removed** from the DOM, we can add a parent component that **conditionally renders** the clock.

Clicking the button will **unmount** and **remount** the **Clock** component.

```
function ClockDisplay() {

    const [showClock, setShowClock] = useState(false);

    const toggleClock = () => {
        setShowClock(!showClock);
    }

    return (
      <div className="ClockDisplay componentBox">
        {showClock && <Clock />}
        <button onClick={toggleClock}>Toggle Clock</button>
      </div>
    )
}
export default ClockDisplay;
// ++ Add both components into Clock.jsx and render
// <ClockDisplay /> from App.jsx

// ++ Watch the console when the Clock is removed -
// does the ticking still continue?
```

useEffect Hook

The **return** value of **useEffect** (if used) should always be a **function**.

This function is automatically run when the component is **unmounted**, and is used to do any **clean up** or **finalisation** required when this component is no longer present.

In our case we want to **clear the ticking interval** when the clock is not shown, to prevent it running unnecessarily in the background.

```
useEffect(() => {
    let clockInterval = setInterval(() => tick(), 1000);
    console.log('Clock component mounted');

    // function returned from useEffect is run on unmount
    // and used to cleanup any side effects
    return () => {
        console.log('Clock component unmounted');
        clearInterval(clockInterval);
    }
}, []);

// ++ Watch the console when the Clock is removed -
// does the ticking still continue now?

// ++ Try to add another state variable tickCount, to keep
// track of (and display) the number of seconds the clock
// has ticked since mounting. Reset it to 0 on unmount.
// Hint: see here if the counter has issues incrementing
```

useEffect Hook

Another common use-case for **useEffect** is **fetching external data** to be displayed.

This effect **depends** on the ‘**participants’** variable to fetch the matching data, so we need to include it in the **array argument** for **useEffect**.

Our effect will now run **both** when the component is **mounted** (initial render) and **updated** (re-rendered) due to a change in the ‘**participants’** state.

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

function ActivityFinder() { // Fetches a random activity

    const [participants, setParticipants] = useState(1);
    const [activity, setActivity] = useState('');

    useEffect(() => {
        fetch('https://bored.api.lewagon.com/api/activity?' +
                           'participants=' + participants)
            .then(response => response.json())
            .then(json => {
                setActivity(json.activity);
            });
    }, [participants]);

    return (
        <div className="ActivityFinder componentBox">
            <h3>Activity Finder</h3>
            <label>Choose number of participants:
                <select value={participants}
                                        onChange={e =>
setParticipants(e.target.value)}>
       <option>1</option><option>2</option><option>3</option>
                </select>
            </label>
            <div><strong>Suggested Activity: </strong>{activity}</div>
        </div>
    )
}

// ++ Reference https://bored.api.lewagon.com/documentation and add a
// new dropdown to suggest an activity based on type
```

useEffect Hook

If an effect fetches data with a **dependency** (*ie. fetching a new activity when the number of participants changes*), the **cleanup function** can make sure the fetched results are still valid.

This is important **if the fetch operation has a delay**. Without cleanup, a user may select a new dependency value before the old effect has finished loading, making those results invalid.

See here for more details.

```
// updated useEffect hook from previous slide
useEffect(() => {
    console.log('running effect');
    let ignore = false; // flag to allow ignoring of the fetch result

    fetch('https://bored.api.lewagon.com/api/activity?' +
                     'participants=' + participants)
        .then(response => response.json())
        .then(json => {
            // only update state if the effect is still valid
            // ie. the dependency hasn't changed since the request
            if (!ignore) setActivity(json.activity);
        });

    // cleanup function - runs when unmounted or dependencies change
    return () => {
        ignore = true; // ignore now invalid fetch results
        console.log('cleanup effect');
    };
}, [participants]); // effect dependencies

// ++ Watch/edit the console log messages when using this component to
// understand when the effect and its cleanup run
```

React Hooks - useEffect

Keep these tips in mind for writing effective effects:

- Effects are triggered by **component rendering**, not specific interactions or events.
- Effects allow you to **synchronize a component** with external systems.
- By default (no dependencies), Effects run after **every render**, including the initial one.
- An **empty dependency array** ([ ]) means the Effect runs **only** on **initial render** (mounting).
- React **skips** the Effect if its **dependencies** have the **same values** as the **previous render**.
- You cannot manually select dependencies; they are determined by the code inside the Effect.
- In **Strict Mode** (development only), React mounts components **twice** to stress-test Effects.
- If your Effect encounters issues due to **remounting**, you should implement a **cleanup** function.
- React calls the **cleanup** function **before the next run** of the Effect and during **unmounting**.

useEffect and CORS

CORS (Cross-Origin Resource Sharing) is a security feature that controls which websites can access a back-end service. Many third-party APIs block requests from **localhost** due to strict CORS rules. To bypass this, a common solution is to use a **proxy**, which forwards requests and makes them appear to come from the same origin as the API.

You may see an error such as this when fetching external data via **useEffect**:

```
Access to fetch at 'https://goweather.herokuapp.com/weather/Auckland' from
origin 'http://localhost:5173' has been blocked by CORS policy: No
'Access-Control-Allow-Origin' header is present on the requested resource.
```

To solve this, set up a proxy in `vite.config.js`. Now, you can fetch using any relative request to `/api/weather` and it will be forwarded via a proxy to the external back-end service.

More information can be found at : https://vite.dev/config/server-options#server-proxy

```
// https://vite.dev/config/
export default defineConfig({
  plugins: [react()],
  server: {
    proxy: {
      '/api/weather': {
        target: 'https://goweather.herokuapp.com/',
        changeOrigin: true,
        rewrite: (path) => path.replace(/^\/api/, ''),
      },
    },
    logLevel: 'debug',
  },
})
```

Exercise 1

- Using the following starter code, complete the **BitcoinRates** component to fetch and display the current price of Bitcoin in the selected currency.
- Use a **useEffect** hook with cleanup and appropriate dependencies.

```
const currencies = ['USD', 'AUD', 'NZD', 'GBP', 'EUR', 'SGD'];

function BitcoinRates() {

    const [currency, setCurrency] = useState(currencies[0]);
    // fetch URL:
https://api.coingecko.com/api/v3/simple/price?ids=bitcoin&vs_currencies=${currency}
    const options = currencies.map(curr => <option value={curr}
key={curr}>{curr}</option>);

    return (
        <div className="BitcoinRates componentBox">
            <h3>Bitcoin Exchange Rate</h3>
            <label>Choose currency:
                <select value={currency} onChange={e =>
setCurrency(e.target.value)}>
                    {options}
                </select>
            </label>
        </div>
    )
}
```

React Hooks - useRef

useRef is another built-in React hook, that stores information in between renders.

Unlike **useState**, which stores data used for **rendering** and **triggers re-renders** when that data is updated, **useRef** stores a **reference** to an **object** that is **NOT** used for rendering and does **NOT** trigger re-renders when updated.

We can use **refs** for **non-state variables** that are **remembered between renders** (normal variables are **reset**). More commonly they are used as an '**escape hatch**' to interact directly with **DOM elements** or non-React JS.

The **useRef** hook function takes a **single argument**, which is the **initial** value of the ref. It returns an **object** with a single **property** called **current**, which we can directly modify to make updates.

useEffect Hook

This example uses both a **ref** and a standard **variable** to keep track of the number of times the button has been clicked.

Because the component is **not re-rendering**, **both** counts work as expected in the **alert** message, and **neither** will be updated in the JSX.

This is because **alerts** are **outside** React, but **React** doesn't know that the content in the JSX needs updating.

```
export default function RefCounter() {

    let countRef = useRef(0); // one counter stored in a ref
    let count = 0; // one counter stored in a normal variable

    function handleClick() {
        // update countRef object when clicking, via current property
        countRef.current = countRef.current + 1;
        count = count + 1; // update count variable when clicking
        // both counts should be the same value
        alert(`You clicked ${countRef.current} (${count}) times!`);
    }

    return (
        <div className="RefCounter componentBox">
            <button onClick={handleClick}>
                REFCOUNTER: Click me!
            {/* try rendering {count} and {countRef.current} here */}
            </button>
        </div>
    );
}

// Any variables rendered inside the returned JSX should be
// part of STATE, so that updates trigger re-rendering and keep
// everything in sync. Updates to refs and normal variables DO NOT
// trigger re-renders, so the updated count values don't show.
```

useEffect Hook

If we add a **state variable** to our example, we can see the difference between **refs** and **variables**.

State updates **trigger a re-render** of the component, which **re-initialises the variable** while **the ref value is remembered**. The **new** ref value still **isn't updated** in the JSX until the state variable triggers a re-render.

Rendering refs is therefore **unreliable** and should be avoided.

```
export default function RefCounter() {
    // new counter stored in state
    const [countState, setCountState] = useState(0);

    let countRef = useRef(0); // one counter stored in a ref
    let count = 0; // one counter stored in a normal variable

    function handleClick() {
        // update countRef object when clicking, via current property
        countRef.current = countRef.current + 1;
        count = count + 1; // update count variable when clicking
        // both counts should be the same value
        alert(`You clicked ${countRef.current} (${count}) times!`);
    }

    return (
        <div className="RefCounter componentBox">
            <button onClick={handleClick}>
                REF COUNTER: Click me!
            </button> Ref: {countRef.current} Var: {count} <br/>
            <button onClick={() => setCountState(countState + 1)}>
                STATE COUNTER: Click me to update!
            </button>
            State: {countState}
        </div>
    );
}
```

React Hooks - useRef

A common use of refs is to obtain a **reference** to a **native DOM element**, in order to run **built-in**, **non-mutating** browser functionality on those elements, such as scrolling, focusing, or playing and pausing audio or video.

This involves 3 simple steps:

- **Create** the ref with a null initial value:

```
const videoRef = useRef(null);
```

- **Pass** the ref as a prop to the DOM element:

```
<video ref={videoRef}
```

- Use the **current** property to refer to the DOM element and **run** its native functions:

```
videoRef.current.play();
```

When React **creates** the `video` DOM element, it stores a **reference** to it in `videoRef.current`. We can then **use** that reference from our event handlers and **access** the built-in browser APIs defined on it.

React Hooks - useRef

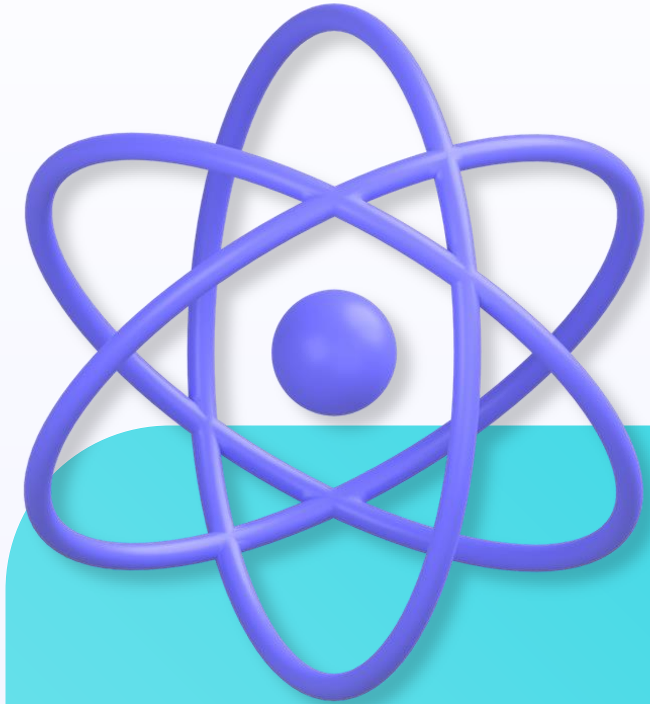
```
import { useState, useRef } from 'react';

export default function VideoPlayer() {
    const [isPlaying, setIsPlaying] = useState(false);
    const videoRef = useRef(null); // 1. Create the ref

    function handleClick() {
        // use the current property of the ref object to access play and pause functions
        isPlaying ? videoRef.current.pause() : videoRef.current.play(); // 3. Use the ref to call DOM functions
        setIsPlaying(!isPlaying); // switch between playing and paused
    }

    return (
        <div className="VideoPlayer componentBox">
            {/* button to play or pause the video */}
            <button onClick={handleClick}> {isPlaying ? 'Pause' : 'Play'} </button>
            {/* 2. Initialise the ref */}
            <video ref={videoRef} width="250">
                <source type="video/mp4"
                    src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4" />
            </video>
        </div>
    );
}
// see https://react.dev/learn/manipulating-the-dom-with-refs for more examples
```

React Hooks - useReducer

useReducer is another built-in React hook which provides an **alternative** to **useState**, best suited for **simplifying complex state updates**.

If there are many event handlers all modifying the same or related states in slightly different ways, it can make sense to consolidate this logic into a reducer instead.

useState and **useReducer** **both** manage state by **remembering** values and triggering re-renders on **updates**. **useState** is a **simpler** choice for **basic** state management of **simple**, **independent** and **isolated** variables, while **useReducer** requires **more coding**, but offers more **flexibility**, **scalability** and **control** for **advanced** state management scenarios.



useEffect Hook

useReducer takes **two** arguments:

1. The **reducer function** called via **dispatch** to handle state updates (*we need to write this*)
2. The **initial value** of the state variable.

It returns an **array** with **two** items (similar to **useState** return array):

1. The **state** variable
2. The **dispatch function**, called in event handlers when making (dispatching) all state updates.

```
import { useReducer } from "react";

function ReducerCounter() {

    // useReducer takes a reducer function and the initial state
value
    // returns array with the state variable and a dispatch function
    const [counter, dispatch] = useReducer(reducer, 0);

    const handleIncrement = () => {
        // we call the dispatch function to make all state updates
        dispatch({ type: "increment" });
    };

    const handleDecrement = () => {
        // dispatch takes a single argument - object passed to
reducer
        dispatch({ type: "decrement" });
    };

    return (
        <div className="ReducerCounter componentBox">
            <h2>Count: {counter}</h2>
            <button onClick={handleIncrement}>Reducer
Increment</button>
            <button onClick={handleDecrement}>Reducer
Decrement</button>
        </div>
    );
};

// see next slide for reducer function

export default ReducerCounter;
```

useEffect Hook

The **reducer** function (which can be named as needed) **returns the new state** and takes **two** arguments:

1. The **state** variable (in our case the **counter** integer)
2. The **action** passed from **dispatch**, which tells the **reducer** what action to take on the **state** variable.

Action is typically an **object** containing properties needed for the update to **state**, such as the **type** of update.

```
// reducer function - can be called anything
// takes two arguments: the current state, and the action to be taken
// action is passed via dispatch, state is stored in component
const reducer = (state, action) => {
    switch (action.type) { // switch statements are common in reducers
        case "increment":
            return state + 1;
        case "decrement":
            return state - 1;
        default:
            return state;
    }
};
// reducer function returns the new value of state after taking action

// ++ Try to add more buttons for incrementing and decrementing by 5:
// 1. Add the buttons
// 2. Add handler functions which dispatch new types
// 3. Update the reducer function to handle the new types
```

Institute of Data

React Hooks - useReducer

A common use-case for **useReducer** is to handle the **various possible outcomes** of fetching remote data. We often want to know **when** the data has finished loading, **what** the successful result was, and if there was any **error**. Instead of managing separate **useState** calls, we can combine it:

```
import { useEffect, useReducer } from "react"                                            // continued on next slide
import axios from 'axios' // first do 'npm install axios' - alternative to fetch

export default function PostListReducer() {

    const [postsResult, dispatch] = useReducer(reducer, { // initial state for postsResult state variable
        loading: true, // true when loading and no data in posts
        posts: [], // empty until data is fetched
        error: '' // empty unless there was an error
    })

    useEffect(() => {
        axios.get('https://jsonplaceholder.typicode.com/posts?_limit=5') // modify this URL to test the error case
            .then(response => {
                // object passed to dispatch holds all data needed for updating state: both type of update and associated data
                dispatch({ type: "FETCH_SUCCESS", payload: response.data }) // dispatch calls reducer function and triggers re-render
            })
            .catch(error => {
                dispatch({ type: "FETCH_ERROR", payload: error.message }) // lets us handle different types of state changes differently
            })
    }, []);
```

React Hooks - useReducer

```
                                                                                  // continued from previous slide
    // returned JSX uses the 3 things stored in postsResult state object to conditionally render data or an error message
    return (
        <div className="PostList componentBox">
            {postsResult.loading ? <div>Loading posts...</div> :
                postsResult.posts.map(post => ( // list of posts is just one of the things stored in the postsResult state object
                    <div className="post" key={post.id}><h3>Post #{post.id}: {post.title}</h3><p>{post.body}</p></div>
            ))}
            <div className="error">{postsResult.error}</div>
        </div>
    )
}

// reducer function for axios fetch response
// called from dispatch when state is updated, lets us handle different actions
// return object should match same structure as initial state passed to useReducer
function reducer (postsResult, action) {
    switch (action.type) {
        case 'FETCH_SUCCESS':
            return { loading: false, posts: action.payload, error: '' }
        case 'FETCH_ERROR':
            return { loading: false, posts: [], error: action.payload }
        default:
            return { ...postsResult, loading: false }
    }
} // What would this component look like using useState instead of useReducer?
```

React Hooks - Custom Hooks

React has several other **built-in hooks** (see here for details). **useContext** is widely used and we will cover it shortly, but most others are less common.

useMemo and **useCallback** are used for **caching** variables and functions respectively, to **optimise performance** in large, complex and slow components.

There are also some variations on **useEffect** with differences in timing, and other **niche** hooks such as **useId**, mainly used by package or library authors.

However, we are **not limited to React's hooks**! We can define our own **custom hooks** to **extract out** any **often repeated** or **complex** logic and **simplify** our components, making the code easier to write and read. Ideally **components** should describe **what** their purpose is, and extract out the **how** it is done.

React Hooks - useEffect

Tips for custom hooks:

1. **Follow the naming convention**: All hooks should be prefixed with '**use**' followed by **capitalised** word/s concisely describing the feature of this hook.
2. **Extract reusable logic**: Identify a **specific** piece of logic or functionality that can be **reused** across multiple components and **extract** it into a custom hook.
3. **Follow the rules of hooks**: Ensure that your custom hook function **calls other hooks** (otherwise it's a function, not a hook), and call hooks at the **top level** of your components. **Avoid** using hooks **conditionally** or in **loops**.
4. **Keep hooks pure**: Your custom hooks will run on **every component re-render**, so they need to **avoid mutating** external variables and return **consistent** output given the same input repeatedly.
5. **Separate hooks and components**: Components should describe **what** happens, by returning **JSX**. Hooks should describe **how** it happens, and return **data**, state or functions.

Custom Hooks

This simple subscribe form has two very **similar** input fields with some **repetitive logic**:

- Setting up **state** variables
- Creating **handler** functions
- Defining **input** elements with **value** and **onChange** props

We can **extract** this common logic out into a **custom useFormInput hook** to simplify our component and avoid repetition.

```
export default function SubscribeForm() {
    const [status, setStatus] = useState('');

    // similar state variables mapped to form inputs
    const [firstName, setFirstName] = useState('Mary');
    const [email, setEmail] = useState('mary@poppins.com');

    // similar handler functions
    const handleNameChange = (e) => setFirstName(e.target.value);
    const handleEmailChange = (e) => setEmail(e.target.value);

    function handleSubscribe() {
        setFirstName(''); setEmail('');
        setStatus('Thanks for subscribing!')
    }

    return (
       <div className="SubscribeForm componentBox">
          <label>First name: {/* form inputs with similar props
*/}
             <input value={firstName} onChange={handleNameChange}
/>
          </label>
          <label>Email: {/* form inputs with similar props */}
             <input value={email} onChange={handleEmailChange} />
          </label>
          <button onClick={handleSubscribe}>Subscribe</button>
          <div>{status}</div>
       </div>
    );
}
```

Institute of Data

Custom Hooks

Our **custom hook** function can be **re-used** to set up **many** form inputs.

It sets up a **single** generic **state** variable with an initial value, and a simple **handler function** that **updates** the **state** from the **event target** value.

It then returns an array containing an object with two properties, called **value** and **onChange** to match the required **input props**, and a **function** to **reset** the **state** value back to empty.

```
export function useFormInput(initialValue) {
    // our custom hook still uses useState internally
    const [value, setValue] = useState(initialValue);

    // generic handler function to update state
    function handleChange(e) {
        setValue(e.target.value);
    }

    // generic function to reset input value
    const reset = () => setValue('');

    // object containing the state value and handler function
    // can be unpacked to set as props for input element
    const inputProps = {
        value: value,
        onChange: handleChange
    };

    // returns data to be used by a component
    return [inputProps, reset];
}

// Name this file useFormInput.jsx and store in a separate
folder
// to your components, typically called 'hooks'
```

Custom Hooks

The updated component is now **simplified** with less repetition.

Form input state is **extracted** into the custom hook, and the **returned input props** object is easily **destructured** into **value** and **onChange** props.

A **custom hook** function can return any **variables** that are **useful** to the **components** using the hook. Here we need both the **input props** object and a function to **reset** the value.

```
import { useState } from 'react';
import { useFormInput } from '../hooks/useFormInput';

export default function SubscribeForm() {
    const [status, setStatus] = useState('');

    // use our custom hook instead of useState directly
    const [nameInputProps, resetName] = useFormInput('Mary');
    const [emailInputProps, resetEmail] =
useFormInput('mary@pop.com');

    function handleSubscribe() {
        resetName(); resetEmail();
        setStatus('Thanks for subscribing!')
    }

    return (
      <div className="SubscribeForm componentBox">
        <label>First name: <input {...nameInputProps} /></label>
        <label>Email: <input {...emailInputProps} /></label>

        <button onClick={handleSubscribe}>Subscribe</button>
        <div>{status}</div>
      </div>
    );
}
// ++ Try adding another form input using the custom hook
// ++ Try creating another form component using the custom hook
```

Custom Hooks

One common use of custom hooks is to **extract** any **effects** out of components.

Effects are an '**escape hatch**' to break out of React and **synchronize** data with **external systems**, so it often makes sense to **move this logic out** of a component.

This custom hook uses both **useState** and **useEffect internally** to extract the process of **fetching data** from a **third-party API**.

```
// hooks are usually named exports rather than default
export function useData(url) {
    // state variable for holding fetched json data
    const [data, setData] = useState(null);

    useEffect(() => {
        if (url) {
            let ignore = false;
            fetch(url)
                .then(response => response.json())
                .then(json => {
                    if (!ignore) {
                        setData(json);
                    }
                });

            // cleanup function, in case url changes before complete
            return () => {
                ignore = true;
            };
        }
    }, [url]); // re-run effect if url changes

    // return the data fetched from the given url
    return data;
}
// save as useData.jsx in the 'hooks' folder
```

Custom Hooks

Putting this **useData** custom hook into practice could look like this ->

Our **ActivityFinder** component is now much **simpler**, as all the complexity of running an effect to retrieve and sync external data is **extracted** into our hook.

We could also **refactor** our **useData** hook (or create a new one) to use a **reducer** to handle the **state** inside the hook and return an **object** containing loading status and error as well as data.

```
import { useState, useEffect } from "react";
import { useData } from "../hooks/useData";

function ActivityFinder() { // Fetches a random activity

    const [participants, setParticipants] = useState(1);

    // uses custom hook to handle the effect for loading external
data
    const data = useData('https://www.boredapi.com/api/activity?'+
        'participants='+participants);

    // get the activity from the json or show error message if failed
    const activity = data ? data.activity : 'not found';

    return (
        <div className="ActivityFinder componentBox">
            <h3>Activity Finder</h3>
            <label>Choose number of participants:
                <select value={participants}
                    onChange={e => setParticipants(e.target.value)}>
           <option>1</option><option>2</option><option>3</option>
                </select>
            </label>
            <div><strong>Suggested Activity:
</strong>{activity}</div>
        </div>
    )
} // ++ Add a second use of useData to fetch activities based on type
```

React Hooks - Custom Hooks

You may not need a custom hook if:

- the logic is simple, isolated or specific to only a few components
- it's only used for extracting state
- it doesn't result in simplified component code.

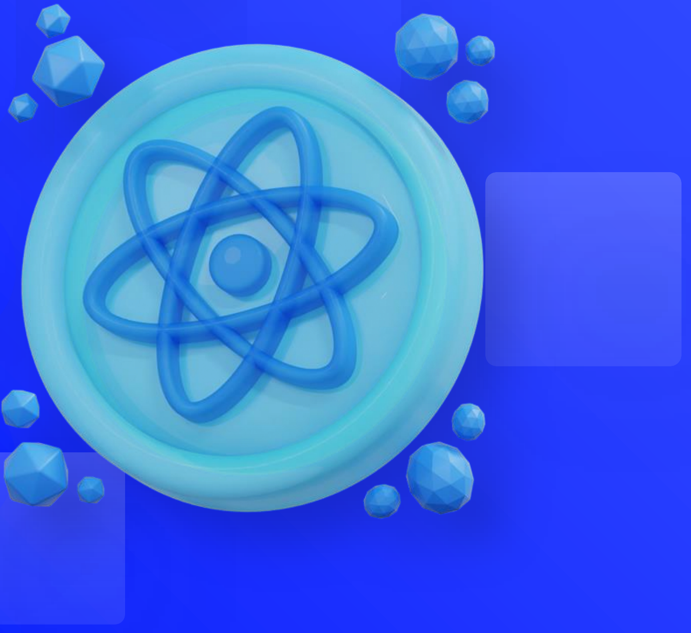
You could benefit from a custom hook if:

- you have logic that can be shared among multiple components
- you need to abstract complex logic for better code readability
- you want to organize and separate concerns in your codebase
- it makes the calling code more declarative by constraining what it does
- there are multiple similar effects
- you can extract code that synchronizes with an external system.

Exercise 2

- Update your **BitcoinRates** component to use a custom hook for extracting the external data synchronization process
- **Extension**: Implement **useReducer** to handle the internal state management of your custom hook

Section 2: Context



So far, the only way one component can pass data to another that we have seen is via **props**. **Context** provides a way to **pass data** through the entire **component tree** without having to **pass props down manually** at every level.

A **context provider component** can provide data in **context** to all its **children**, no matter how **deeply nested**.

This is especially useful when **many components** in an app **need the same pieces of information** to do their job. This kind of **global data** can include **theming** information, data about the currently **authenticated user**, a **locale** or **language preference**, etc.

Context

Working with context requires 3 steps, all of which have built-in support in React:

1. **Create a context** - `createContext`. We can potentially have many contexts (**UserContext**, **ThemeContex**t etc) so we need to **name each one uniquely** according to the information it provides.
2. **Provide the context** - `Context.Provider`. Usually this is a **top-level component**, that specifies the **data to be passed** to all the **children** that need to use it.
3. **Use that context** - `useContext`. We can **use** this hook **as many times as needed**, from **any component** that needs the provided context data.

Context lets even a distant parent at the very top of the component hierarchy provide some data to the entire tree of child components inside of it.

Context

This code template is a handy way to **set up any context** with all required steps.

First we **create the context**. React includes a **createContext** function for this which returns a new **Context** object.

Next we **wrap** the **Context.Provider** in a custom **UserProvider** component that also manages the relevant user **state**, **passing** it to all **children** via the **value** prop.

Finally we create a simple **custom hook** that **bundles** our new **UserContext** into the **useContext** hook, ready for use.

```
// 1. Create the context
const UserContext = React.createContext();

// Custom provider component for this context.
// Use it in App.jsx like <UserProvider>...</UserProvider>
export const UserProvider = (props) => {
    // store the current user in state at the top level
    const [currentUser, setCurrentUser] = useState({});

    // sets user object in state, shared via context
    const handleUpdateUser = (user) => {
        setCurrentUser(user);
    }

    // 2. Provide the context.
    // The Provider component of any context (UserContext.Provider)
    // sends data via its value prop to all children at every level.
    // We are sending both the current user and an update function
    return (
        <UserContext.Provider value={{currentUser,
handleUpdateUser}}>
            {props.children}
        </UserContext.Provider>
    );
}

// 3. Use the context. This custom hook allows easy access
// of this particular context from any child component
export const useUserContext = () => {
    return useContext(UserContext);
}

// Save as UserContext.jsx in a separate 'context' folder
```

Context

The **useContext** hook is simple to use but can be tricky to understand.

Two main points to remember:

- It takes a **single argument**, which must be a **Context object** created via `React.createContext` (in our example, **UserContext**)
- It returns whatever **custom data** has been defined in the **value prop** of `Context.Provider`. This will be different for every context, and is usually an **object**, though it can be **any type** of variable.

If you forget to include or render a **Provider**, **useContext** will return the **default value** passed to `React.createContext`.

Any **changes** to provided **data** will **trigger re-renders** in any component using it.

Context

We can use our new **context** to make sure the logged-in **user** is **remembered** across **all the components** in our application.

First we need to include our custom **UserProvider** component, usually as a **parent** to **all components** at the **top level** of the hierarchy in **App.jsx**.

Storing the current user in **state** at this **top level** and **passing via context** means that any component can easily **access data** about the currently logged-in user.

```
import { UserProvider } from './context/UserContext'

// Modify App.jsx to include the UserProvider component
function App() {

  return (
    <>
      <UserProvider> {/* provider component is at top level */}
        <ClockDisplay /> {/* so all children can use context data */}
        <MoviesList /> {/* even if they have children of their own */}
        <ActivityFinder />
        <PostListReducer />
        {/* logging in here can now set up current user everywhere */}
        <LoginForm />
      </UserProvider>
    </>
  )
}
```

Institute of Data

Context

We can now update the **LoginForm** component from the previous module to make use of **context**.

All the context setup is done, so we only need to **use** the **values** returned from the **useContext** hook. These are the **same values** provided by the **UserContext.Provider** component: the **currentUser object** and the **update handler function**.

When the user logs in successfully, we **call the function** with their details. If the current user exists, we **welcome them** and show a **logout button** instead of the **form.**

```
import { useState } from "react";
import { useUserContext } from "../context/UserContext";

function LoginForm() {
    const [userEmail, setUserEmail] = useState('');
    const [userPassword, setUserPassword] = useState('');
    const [submitResult, setSubmitResult] = useState('');
    // destructure the context values passed via UserProvider
    const {currentUser, handleUpdateUser} = useUserContext();

    // alternative using the useContext hook directly, either works
    //const {currentUser, handleUpdateUser} = useContext(UserContext);

    const handleSubmit = (e) => {
        e.preventDefault();
        if (userPassword.length < 5) {
           setSubmitResult('Password must be at least 5 chars long');
        } else if (userPassword === userEmail) {
           setSubmitResult('Password must not match email address');
        } else {
           setSubmitResult('Successful login.');
           handleUpdateUser({ email: userEmail }); // context function
        }
    }

    {/* if the user is already logged in, show msg instead of form */}
    if (currentUser.email) return (
        <><p>Welcome {currentUser.email}!</p>
        <button onClick={() => handleUpdateUser({})}>Log Out</button>
        </>
    );
    // otherwise render same form as previously, no changes
```

Context

After **setting up** and **providing** the **context**, actually **using** it from a component is quite simple.

If we want to display the **current user** from any component in the tree, we only need to:

1. **Import** the context or its hook
2. **Call** the built-in **useContext** or our custom version to get the user object
3. **Render** the details of the user as needed.

```
// ---- EITHER use our custom hook for UserContext:
    import { useUserContext } from "../context/UserContext";

    // and use the values provided by the context
    const {currentUser} = useUserContext();


// ---- OR call the useContext hook and pass in UserContext
    import { useContext } from "react";
    import { UserContext } from "../context/UserContext";

    // and use the values provided by the context
    const {currentUser} = useContext(UserContext);


// ---- THEN use/display the context values as needed

    {/* display the current user from any component */}
    <p>Welcome {currentUser.email}!</p>


// ++ Display the current user's email from another component
// ++ Add a 'name' field to the login form and include it in
// the user object stored in context, then display it too
```

Institute of Data

Context

Context recap:

Most of the time we can easily **pass data** from one component to another via **props**. However, if the same data needs to be accessed by **many components** at **many different levels** (global data), it can be simpler to handle this with **context**.

Data passed via context needs to be sent from a **provider component** at a **top level** of the component hierarchy.

Context can provide any data of **any type**. Multiple values are often bundled into an **object** which can be **destructured**.

We can have **multiple contexts** in one app that each provide different data. Typical examples include authenticated users, theming or locale data.

Context

This example uses the same basic **template**, to combine everything related to this context in the one file.

Instead of **storing** and **providing** **user** data, this context stores the **current theme** in **state**.

Via the **value** prop, the context **provider** makes available to any component:

- this **theme object**,
- a **function** for updating it, and
- a **boolean flag** to tell if **dark mode** is currently enabled.

```
// theme options with specific colour values
export const themes = {
    light: {
        foreground: "#333333",
        background: "#BAE2FF"
    },
    dark: {
        foreground: "#ffffff",
        background: "#222222"
    }
};

// named export for this context (to be used via useContext elsewhere)
export const MyThemeContext = React.createContext(
    {theme: themes.light});

// provider wrapper. uses its own state to track which theme is in use
// use it in App.jsx like <MyThemeProvider>...</MyThemeProvider>
export default function MyThemeProvider(props) {
    const [theme, setTheme] = React.useState(themes.light);

    // helper boolean to tell if we're currently in dark mode
    const darkMode = theme.background === themes.dark.background;

    return (
        <MyThemeContext.Provider value={{theme, setTheme, darkMode}}>
            {props.children}
        </MyThemeContext.Provider>
    );
}
// ++ Try to use this context to style some existing components
// ++ Try to add a component with a button/checkbox to switch themes
```

Context

If an app requires **multiple contexts**, they can be **nested** at the very top level.

Context providers can also be applied at **any high level** where the **provided values** are **available** to all **relevant child components**.

Context lets us write components that display themselves **differently** depending on the **context** in which they are rendered, and which can easily use **global** data to maintain **consistency** across an app.

```
function App() {
  // Multiple context providers can be nested at the top level
  return (
    <>
      <UserProvider> {/* provider components at top level */}
        <MyThemeProvider>
          <ClockDisplay />
          <ActivityFinder />
          <PostListReducer />
          <LoginForm />
        </MyThemeProvider>
      </UserProvider>
    </>
  )
}


// to use this context in a component, first employ useContext
hook
const {theme, darkMode} = useContext(MyThemeContext);

// then use the theme object for inline styling
<div className="LoginForm componentBox"
    style={{background: theme.background, color:
theme.foreground}}>

// or the boolean to create a CSS class
<div className={darkMode ? 'dark' : 'light'}>
```

Exercise 3

- Modify the **Emoji** component from Module 6 Exercise 3 and create a context for storing the current emoji/mood
- Display the current emoji from the **BitcoinRates** component, and make sure it updates when clicking the ‘Change Mood’ button

Section 3: Routing

React is designed to render **single page applications** (SPAs) where a single HTML file is served to the browser and updated via JS.

However, we can still simulate multiple page applications via **client-side routing**, which uses JS to update the browser URL and render different components on a new ‘page’.

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

React Router: Introduction

React Router is a **routing library** for React, including many features to support client and server-side routing. React Router operates wherever React does: on the **web**, on the **server** with node.js, and on React Native (for **mobile** apps).

Client-side routing allows your app to change the website address (URL) in the browser **without** asking the server for a new page. Instead, your app can quickly show dynamic UI and use **fetch** to get new information to update the page content.

This makes things **faster** for users because the browser doesn't have to fetch a whole new page from the server or reload the CSS and JavaScript for each new page. It also allows for more **interactive** experiences, like smooth animations.

React Router: Concepts

The concept of **routing** in **back** and **front end** frameworks works slightly differently, and the terms involved can have different meanings. In React Router we use the following:

URL - The URL in the address bar. A lot of people use the term "**URL**" and "**route**" interchangeably, but a URL 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.

Location State - A **value** that **persists** with a location that isn't encoded in the URL. Much like hash or search params (data encoded in the URL), but stored invisibly in the browser's memory.

History Stack - As the user navigates, the browser keeps track of each location in a **stack**. If you click and hold the back button in a browser you can see the browser's history stack right there.

React Router: Concepts

Client-Side Routing - enables developers to **manipulate** the browser history stack without making a document request to the server.

History - An **object** that allows React Router to **subscribe** to changes in the URL and **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** is when a new entry is added to the history stack (typically a link click or a forced 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.

Segment - The **parts** of a URL or path pattern (see below) between the `/` characters. For example, `/users/123` has two segments.

React Router: Concepts

Path Pattern - These look like URLs but have **special characters** for matching URLs to routes, like **dynamic** segments (`/users/:userId`) or **star** segments (`/docs/*`). They aren't URLs, they're **patterns** that React Router will match.

Dynamic Segment - A segment of a path pattern that is **dynamic**, meaning it can match any values in the segment. For example the pattern `/users/:userId` will match URLs like `/users/123`

URL Params - The parsed values from the URL that matched a dynamic segment.

Router - Stateful, top-level **component** that makes all the other routing components and hooks work.

Route Config - A tree of **route** components under `<Routes>` that will be matched against the **current location** to create a branch of route matches.

React Router: Concepts

Route - A component like `<Route path element>`. The **path** is a path pattern. When the path pattern matches the current URL, the **element** will be rendered.

Nested Routes - Because routes can have children and each route defines a portion of the URL through segments, a **single** URL can match **multiple** routes in a **nested** "branch" of the tree. This enables automatic layout nesting through `<Outlet>`, relative links, and more.

We need to install the **React Router** library before we can take advantage of its features. First stop your app from running, then run:

```
npm install react-router-dom
```

BrowserRouter

`<BrowserRouter>` is the recommended interface for running React Router in a **web browser**. It needs to be added to the very **top** level. There are other Router options, but this is the best to begin with.

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**.

No other React Router components will work if you miss this step!

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
import './index.css'
import { BrowserRouter } from 'react-router-dom'

// This code in main.jsx handles rendering the App component
// into the DOM element with an id of 'root' (in index.html)
ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    {/* All other routing components need this to work */}
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>,
)
```

Routes

`<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**.

Whenever the location **changes**, `<Routes>` looks through all its child `<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**.

```
// special component containing all the possible routes for this app
// any props passed into AppRoutes will also be passed onto
// child components using {...props}
function AppRoutes(props) {

    return (
        <Routes>
            {/* index matches on default/home URL: / */}
            <Route index element={<Homepage {...props} />} />

            {/* nested routes, matches on /dash/messages etc */}
            <Route path="dash" element={<DashboardPage {...props} />}>
                <Route path="messages" element={<DashboardMessages />}
/>
                <Route path="tasks" element={<DashboardTasks />} />
            </Route>

            <Route path='/about' element={<AboutPage {...props} />} />

            {/* special route to handle if none of the above match */}
            <Route path="*" element={<PageNotFound />} />
        </Routes>
    )
}

export default AppRoutes;
// Name this file AppRoutes.jsx and store in new folder 'routes'
```

useNavigate

`useNavigate` is a **hook** that comes with React Router DOM.

It returns a **function**, usually called **navigate**. The most common way to use this function is to pass the **URL to navigate to** as a single parameter.

We can also use it to navigate the **history stack**, by passing in a number, such as -1 to go back one location.

```
import { useNavigate, Outlet } from 'react-router-dom'

export default function DashboardPage() {
    // built-in hook in React Router DOM, returns a function
    const navigate = useNavigate();

    return (
        <div className="DashboardPage componentBox">
            <h1>Dashboard</h1>

            <Outlet /> {/* see next slide */}
            {/* Will render either <DashboardMessages> when the URL
is
            "/dash/messages", <DashboardTasks> at "/dash/tasks",
            or null if it is "/dash" */}

            <button onClick={() => navigate('/dash/tasks')}>
                Show Tasks</button>
            <button onClick={() => navigate('/dash/messages')}>
                Show Messages</button>
            <button onClick={() => navigate(-1)}>
                Back</button>
        </div>
    )
}
// ++ Move the <Outlet/> below the buttons and observe the change
// Save as DashboardPage.jsx in a new folder called 'pages'
// See next slide for additional content for this file
```

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** (eg. `/dash/messages`). If the parent route matched **exactly**, it will render a child index route, or nothing if there is no index route.

Think of it as a **placeholder** where **matching** child route elements will be displayed, depending on the URL.

```
// Add this code to DashboardPage.jsx : nested route path components
// rendered by matching nested routes in AppRoutes.jsx

// These components will appear in the <Outlet /> placeholder spot
// in the DashboardPage component if the route matches

export function DashboardMessages(props) {
    const { currentUser } = useUserContext();

    return (
        <div className="DashboardMessages">
            <p>Welcome to your dashboard, {currentUser.email}</p>
        </div>
    )
}

export function DashboardTasks(props) {

   const tasks = [
      { id: 1, name: 'Learn React' },
      { id: 2, name: 'Decide on capstone project' },
      { id: 3, name: 'Learn databases' }
   ]

   return (
      <div className="DashboardTasks">
         <ul className="tasks">
            { tasks.map(task => <li key={task.id}>{task.name}</li>)
}
         </ul>
      </div>
   )
}
```

Link

A `<Link>` is an element of React Router DOM that renders a **link** that the user can click on to **navigate** to another page. A `<Link>` renders an accessible `<a>` element with a real `href` that points to the path in the `to` prop.

This means that things like right-clicking a `<Link>` work as expected.

You can use `<Link reloadDocument>` to **skip** client-side routing and let the browser handle the transition normally, **BUT** this will **lose all current state**.

```
import { Link } from 'react-router-dom'

function PageNotFound() {
    return (
        <div className="PageNotFound">
            <h1>Page Not Found</h1>
            <p>What were you looking for?
                Maybe going back <Link to="/">home</Link>
                will help you find it.</p>
        </div>
    )
}

export default PageNotFound

// Save as PageNotFound.jsx in the 'pages' folder
// ++ Add a Back button to navigate one page back in the
history
// ++ Add a standard <a href> link as well and observe the
difference
```

Rendering Routes

Now that we have set up our **routing** (via **BrowserRouter**), **routes** (via **AppRoutes**) and associated **pages** and **components**, we are almost ready to see it all in action.

First we need to add the remaining pages and render our **AppRoutes** →

Next we need to add some way of **navigating** to each of the **routes** in our app. Usually we do this with a **NavBar** component.

```
// Simple page components for completing the routes
// ++ Render some existing components on these pages to add content

export default function Homepage() { // Save in pages/Homepage.jsx
    return (
        <div className="Homepage">
            <h1>Home</h1>
        </div>
    )
}
export default function AboutPage() { // Save in pages/AboutPage.jsx
    return (
        <div className="About">
            <h1>About</h1>
        </div>
    )
}
// ++ Update imports in AppRoutes.jsx to include these new files

// Update App.jsx as shown - all components rendered via routes
function App() {
  return (
      <UserProvider>
        <MyThemeProvider>
          <AppRoutes />
        </MyThemeProvider>
      </UserProvider>
  )
} // ++ Create a Footer component as well and add under AppRoutes
```

Institute of Data

NavLink

React Router DOM has a special component `<NavLink/>` for rendering links within **navigation menus**. It works mostly the same as **Link** BUT will also include a `class=”active”` attribute on the `<a>` element of the **currently matched route**.

The **NavBar** component should be rendered **before** the **AppRoutes** component (in App.jsx). This will ensure it renders on **every route**, **above** the matched page content.

```
import { useContext } from 'react'
import { NavLink } from 'react-router-dom'
import { MyThemeContext } from '../context/MyThemeContext'

export default function NavBar() {
    const {theme} = useContext(MyThemeContext);

    return (
        <nav className="NavBar"
style={{backgroundColor: theme.background, color:
theme.foreground}}>
            <ul className="menu">
                <li><NavLink to="/">Home</NavLink></li>
                <li><NavLink to="/dash">Dashboard</NavLink></li>
                <li><NavLink to="/about">About</NavLink></li>
            </ul> {/* ++ Add another page with route and component */}
        </nav>
    )
}
// Save as components/NavBar.jsx and render in App.jsx
// ABOVE <AppRoutes />

/* basic NavBar CSS, add to App.css and customise as needed */
.NavBar ul { display: flex; list-style: none; justify-content:
center; padding: 0; }
.NavBar ul li a { padding: 1em; color: inherit; text-transform:
uppercase; display: inline-block; text-decoration: none; }
.NavBar ul li a.active { font-weight: bold; }
```

Exercise 4

- Create an app with 3 different pages: Home, Login and Bitcoin Rates
- Use existing components to add content to each page
- Include a navbar to navigate between pages

React Router: Recap

Our application is now starting to support more **complexity**. As it grows, it is important to **structure** and **separate** your code logically, to make it easier to **debug**, **update**, **maintain** and **understand**, both for yourself and your team.

There is no single right way to structure your application and many organisations have their own preferred patterns. A common option is to create folders for each type of element: **components**, **hooks**, **context**, **routes**, **pages.**

Client-side routing allows us to separate component rendering into **different pages**, which still manipulate the browser history as expected when navigating between them. React Router also supports more advanced concepts such as **dynamic routes** and **protected routes**.

Dynamic Routes

This component displays a list of posts, each with a unique **Link** to load details about a **specific post** based on its `id`.

We don’t want to **manually** set up **routes** for **every possible post id**, because there could potentially be **many**, we want to **avoid repetition**, and we **don’t know** the **id**s in advance.

Therefore we need a **dynamic route** that can handle **whatever** the **post id** value is.

```
import { Outlet, useParams, Link } from "react-router-dom";
import { useData } from "../hooks/useData"

// save as pages/PostsPage.jsx
export default function PostsPage() {
    return (
        <div className="Posts">
            <h1>Posts</h1>
            <Outlet />
        </div>
    )
}

export function PostList() {
    const postsData =
useData('https://jsonplaceholder.typicode.com/posts?_limit=5');

    // the ? means only call map if postsData is not null
    const postList = postsData?.map(post => (
        <li key={post.id}><Link to={"/posts/" + post.id}>
            Post #{post.id}: {post.title}</Link></li>
    ));

    return (
        <ul>{postList}</ul>
    )
}
```

Dynamic Routes

Dynamic routes use a special syntax in the **path** prop, which is a **colon** : followed by the **param** (variable) **name**.

Any **matching route** (in this case `/posts/1` or `/posts/55` or `/posts/anything`) will render the **Post** component and store the value after the slash (**1** or **55** or **anything**) in the **id param**.

The **useParams** hook returns an **object** of all the matched **params** and their values, which we can use in the **Post** component to render the **right post** for the route.

```
// add to AppRoutes.jsx
import PostsPage, { Post, PostList } from "../pages/PostsPage"

// add new Route branch to AppRoutes.jsx
<Route path='/posts' element={<PostsPage {...props} />} >
  <Route index element={<PostList />} />
  {/* dynamic param taken from route, stored in variable called id
*/}
  <Route path=":id" element={<Post />} />
</Route>

// add to NavBar.jsx so we can access our new page
<li><NavLink to="/posts">Posts</NavLink></li>

// add to PostsPage.jsx
export function Post() {
   const { id } = useParams(); // custom hook to access dynamic
params
   const post =
useData('https://jsonplaceholder.typicode.com/posts/'+id);

   return (
       <div className="Post">
           {post ?
               <><h3>Post #{post.id}: {post.title}</h3>
                 <p>{post.body}</p></>
           : "Loading ..." }
       </div>
   )
}
// ++ Add a Next Post button to the Post component
```

Query Parameters

As well as the **useParams** hook for obtaining **dynamic** request parameters, we can also use the **useSearchParams** hook to get and manage any **query** parameters from the request.

It returns two values in an array :

- The searchParams object for the URL
- A function for updating the searchParams.

The **setSearchParams** function works like **navigate**, in that it modifies the current route, but only for the search portion of the URL after the '?'.

```
export function PostList() { // updated from slide 60, replace old version
  const [searchParams, setSearchParams] = useSearchParams(); // import this
hook
  const limit = searchParams.get('limit') ? searchParams.get('limit') : 5;
  const postsData =
useData("https://jsonplaceholder.typicode.com/posts?_limit="+limit);

  const handleChangeLimit = (e) => {
    setSearchParams({limit: e.target.value})
  }

  // the ? means only call map if postsData is not null
  const postList = postsData.map((post) => (
    <li key={post.id}>
      <Link to={"/posts/" + post.id}>
        Post #{post.id}: {post.title}
      </Link>
    </li>
  ));
  return (
    <>
      <ul>{postList}</ul>
      <Link to='/posts?limit=10'>Load 10 Posts</Link>
    </>
  );
}
// ++ Change the 'Load 10 Posts' link to a drop-down allowing users to
choose
// 5, 10 or 20 posts. Use the handleChangeLimit function in the onChange
event.
```

Protected Routes

Some routes need to include **security**, to ensure that they are only accessed by **authenticated users**.

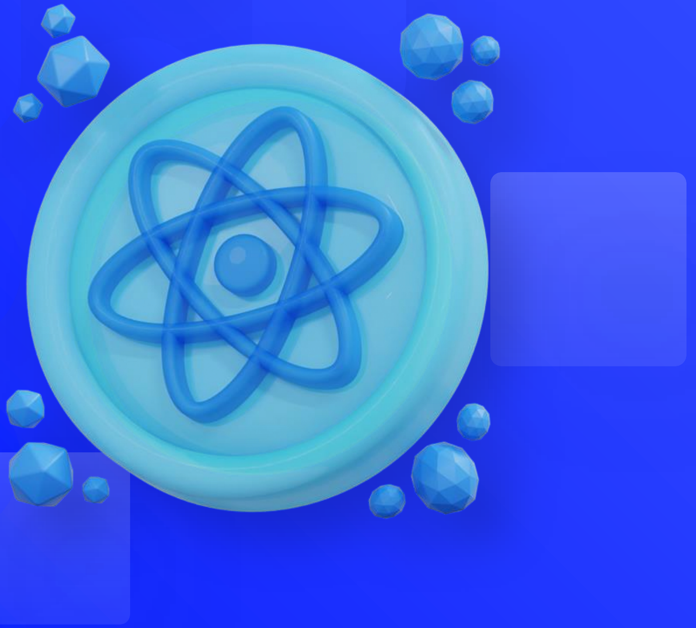
This **ProtectedRoute** component uses the **Navigate** component from React Router DOM to **automatically redirect** the user to a specific page (default `/login`) if there is no valid user object in context.

We can wrap any of our routes in this **ProtectedRoute** component to easily add some basic security.

```
import { Outlet, Navigate } from "react-router-dom";
import { useUserContext } from "../context/UserContext";

// wrap around logged-in user only routes to protect them
function ProtectedRoute({ redirectPath = '/login', children }) {
    const { currentUser } = useUserContext();

    if (!currentUser.email) {
        return <Navigate to={redirectPath} replace />;
    }
    // works for both nested and standalone routes
    return children ? children : <Outlet/>;
}
export default ProtectedRoute
// save as routes/ProtectedRoute.jsx

// update AppRoutes.jsx to protect the Dashboard
<Route path="dash" element={<ProtectedRoute>
       <DashboardPage {...props} /></ProtectedRoute>}>
   <Route path="messages" element={<DashboardMessages />} />
   <Route path="tasks" element={<DashboardTasks />} />
</Route>
<Route path="login" element={<LoginForm/>} />

// update NavBar.jsx to include our login page
<li><NavLink to="/login">Login</NavLink></li>
```

Institute of Data

Section 4: React Frameworks



React is technically a **library**, not a **framework**. It can be included in an existing non-React app and only used for displaying specific pages, or sections of pages, in a UI.

There are a number of frameworks based on React that provide full stack application potential. **Next.js** is used in tens of thousands of production-facing websites and web applications, including many of the world's largest brands.

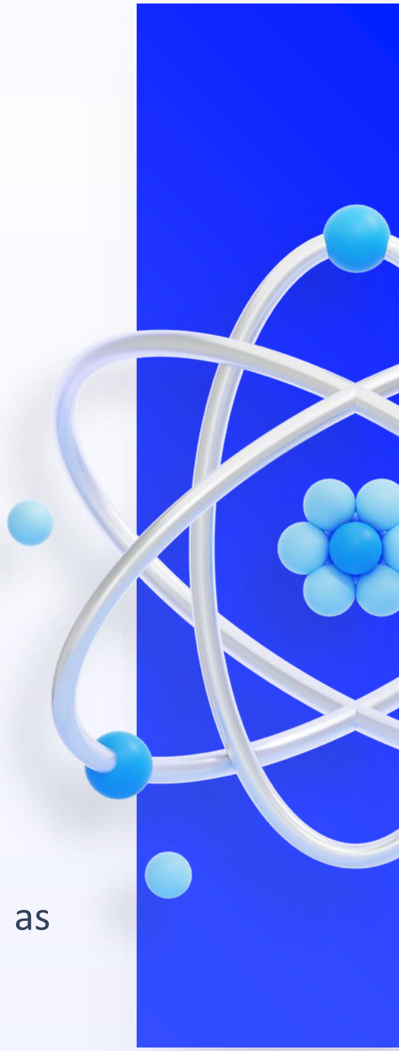
Next.js is a flexible **React framework** that gives you building blocks to create fast web applications. It handles the tooling and **configuration** needed for React, and provides additional **structure**, **features**, and **optimisations** for your application.

Next.js Introduction

To build a **complete web application** with **React** from scratch, there are many important tasks:

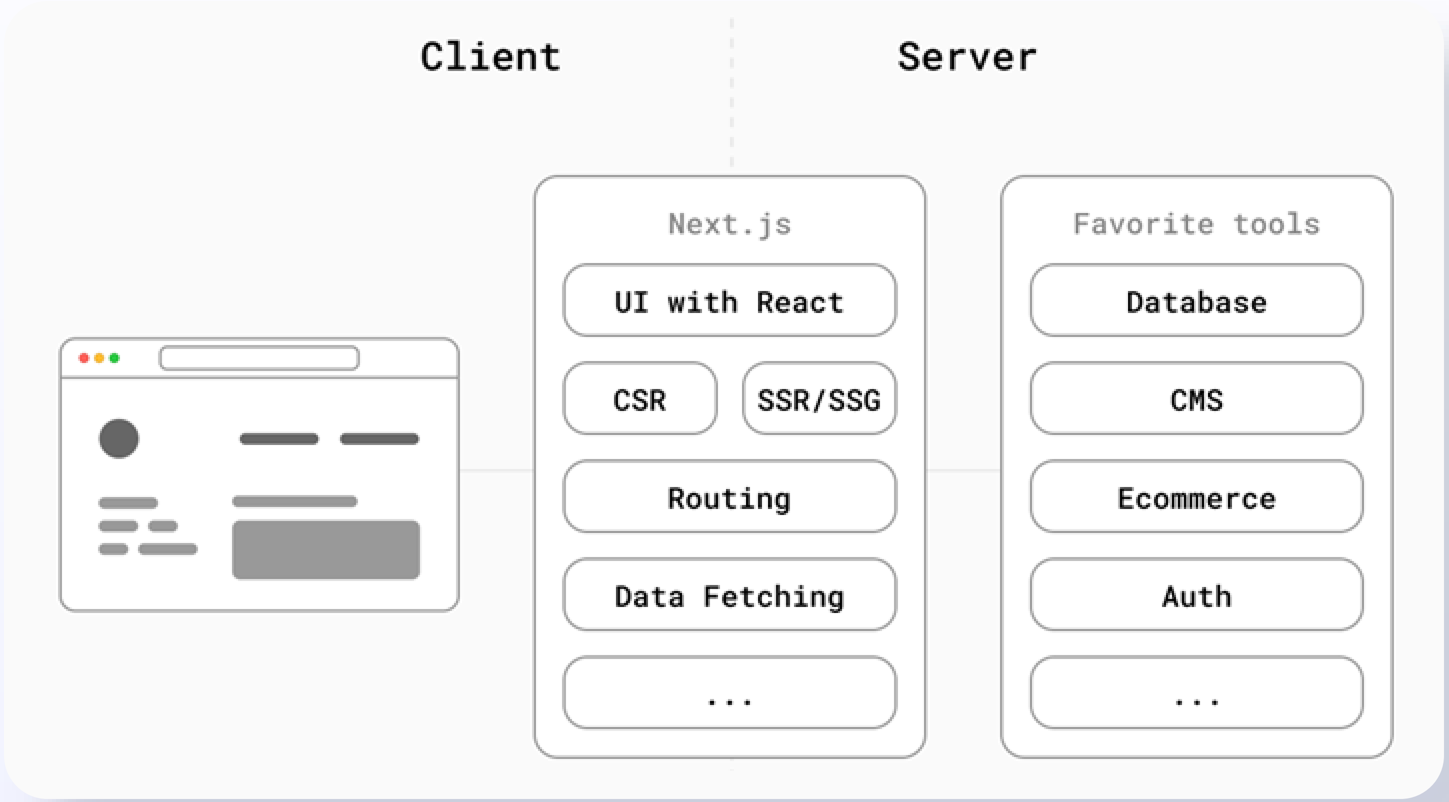
- **Bundle** your code: Use tools like Vite or webpack to bundle code and make it **compatible** for the web.
- **Transform** your code: Use a **compiler** like Babel to transform your code into a format that can be understood by different browsers.
- **Optimise** for production: Implement **code splitting** to improve performance and load times.
- Choose **rendering** methods: Decide whether to use **server-side** rendering or **client-side** rendering based on your specific needs (eg. SEO) and desired user experience.
- Connect to **external data** sources: If your app requires data from a data store or API, you may need to write **server-side code** to support this.

These tasks can mostly be achieved using React with the addition of extra tools and libraries. A framework such as **Next.js** can also resolve them in a more streamlined and supported way.



Next.js Introduction

As **Next.js** is a **full framework**, it builds on the core UI-building features of **React** and provides many extra features, not all of which we will cover in this course.



Probably its main point of difference is the ability to create **server-rendered React applications**, where the UI code is generated **before** it hits the browser.

CSR: Client-side rendering

SSR: Server-side rendering

SSG: Static site generation

Next.js Introduction

Some extra features included in **Next.js**:

Routing (both **client-side** and **server-side**). Client-side routing requires an extra package like React Router DOM to achieve in React itself. **Next.js** includes support for routing on both the client and the server.

Server-Side Rendering (**SSR**). React uses **client-side rendering**, in that JS is run directly in the browser to build the UI and display content. **Next.js** can render content on the **server** and send pre-built pages to the browser. This is especially beneficial for SEO (Search Engine Optimisation) as search engines can process the generated static content better than dynamic JS.

Automatic Code-Splitting. Since **Next.js** can process code on the server, it has better support for splitting and bundling the JS sent to the browser for each page.

Next.js Introduction

Getting started with **Next.js** is similar to the process we have already followed using **Vite** to bundle a **React** app.

```
npx create-next-app first-next-app
```

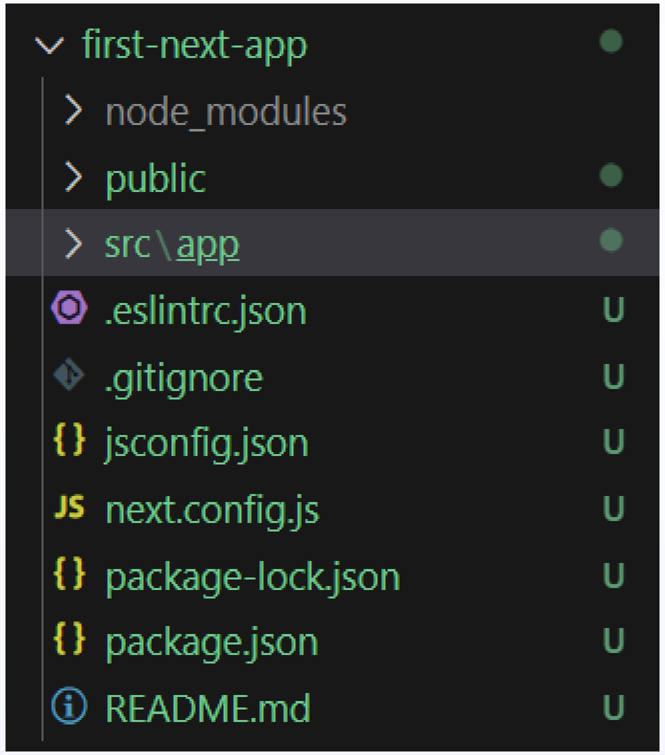
The above command will create a **new Next.js app** in a folder called '**first-next-app**'.

The tool will then ask a series of questions about how your app should be configured:

```
Ok to proceed? (y) y
√ Would you like to use TypeScript with this project? ... No / Yes
√ Would you like to use ESLint with this project? ... No / Yes
√ Would you like to use Tailwind CSS with this project? ... No / Yes
√ Would you like to use `src/` directory with this project? ... No / Yes
√ Use App Router (recommended)? ... No / Yes
√ Would you like to customize the default import alias? ... No / Yes
```

Select the underlined responses using the arrow keys: **No** Typescript, **Yes** ESLint, **No** Tailwind, Yes src directory, **Yes** App Router, **No** import customisation.

Next.js Introduction



```
first-next-app
  node_modules
  public
  src\app
  .eslintrc.json      U
  .gitignore          U
  jsconfig.json       U
  next.config.js      U
  package-lock.json   U
  package.json        U
  README.md           U
```

Your folder structure should look similar to this, with **public** and src folders at the top level along with various **config** files.

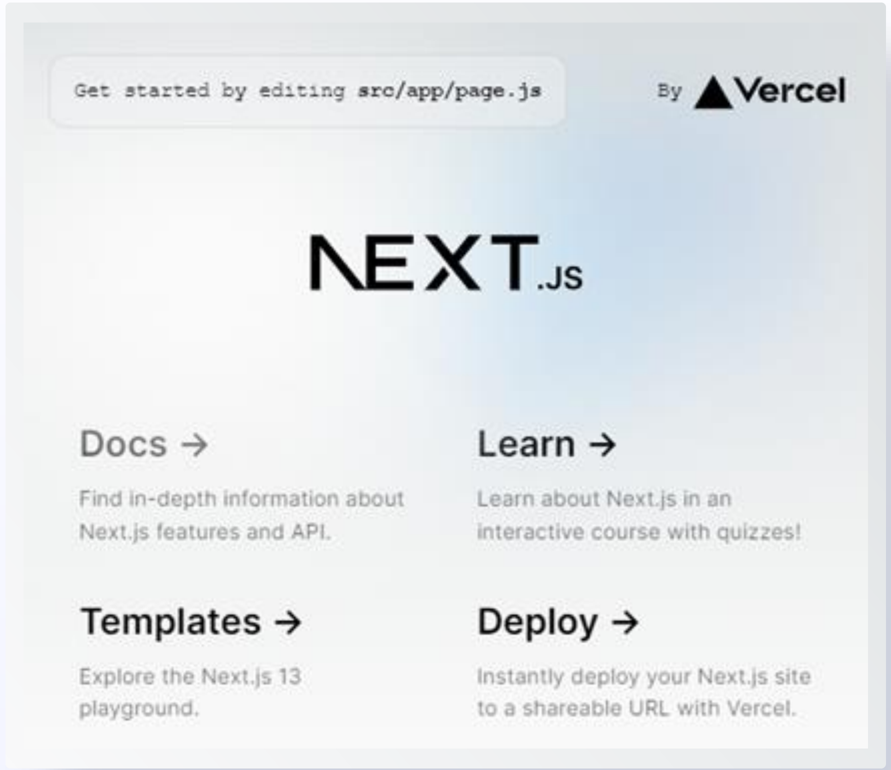
app/page.js is responsible for loading the content shown in the browser.

app/layout.js includes the HTML structure.

Now we're ready to run our app. **First change** into the new folder, then **run** it using `npm run dev:`

```
cd first-next-app
npm run dev
```

Then open http://localhost:3000 in your browser:



Get started by editing src/app/page.js

By Vercel

NEXT.JS

Docs →

Find in-depth information about Next.js features and API.

Learn →

Learn about Next.js in an interactive course with quizzes!

Templates →

Explore the Next.js 13 playground.

Deploy →

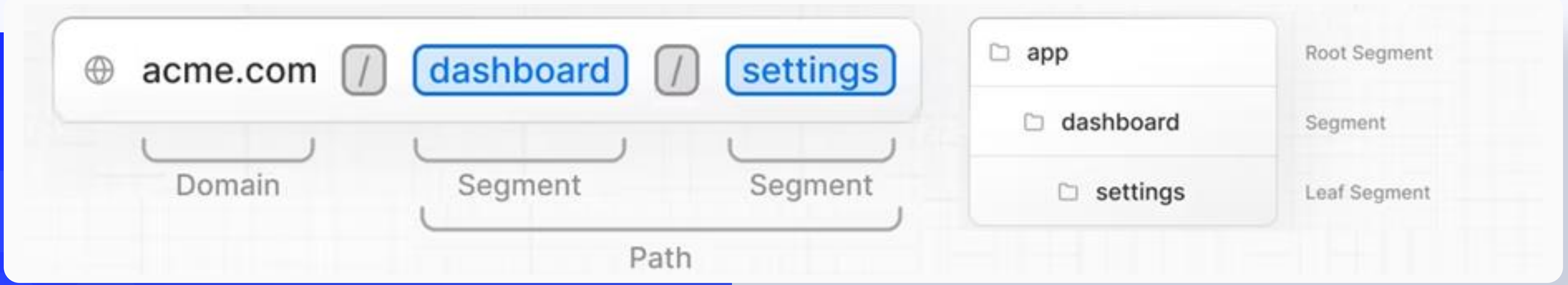
Instantly deploy your Next.js site to a shareable URL with Vercel.

Next.js Routing

So far, our app only has a single page - the **homepage** at `app/page.js`. Websites generally have many different pages, and the integrated **App Router** with **Next.js** supports multiple pages along with shared layouts, nested routing, loading states, error handling, and more.

In **Next.js**, a **page** is a **React Component** exported from a (potentially nested) `page.js` file in the **app** directory. Next.js uses a **file-system-based router** where:

- **Folders** are used to define **routes**. A **route** is a single path of **nested folders**, following the file-system hierarchy from the parent (**root**) folder down to a final child (**leaf**) folder that includes a `page.js` file.
- **Files** matching specific **naming conventions** are used to create UI that is shown for a **route segment**. This includes `page.js` but also others for error handling, layout, and more.



Next.js Routing

To add an **About** page to our **app** is as simple as creating an `about` **folder** with a `page.js` **file** (`page.jsx` works as well) inside it.

The **page component** in this file must be the **default export**, and will be rendered whenever the **route** matches the **folder structure** (relative to **app**).

The **app** folder handles all **routing**. We can still have regular **components**, **hooks**, **context** etc folders - but store these under the **src** folder, NOT in **app**.

```
import Link from 'next/link'
import styles from '../page.module.css'

// Save as page.jsx in app/about
export default function About() {
    return (
        <main className={styles.main}>
            <div className="About">
                <h1>About</h1>
                <p>This is the about page.
                  Nothing to see, go <Link
href="/">home</Link>.</p>
            </div>
        </main>
    )
}
// Check http://localhost:3000/about to see it in action
```

Institute of Data

Next.js Layouts

A **layout** refers to UI that is **shared** across multiple pages. When navigating between pages, layouts **maintain state**, remain **interactive**, and **do not re-render**. Layouts can also be **nested** within each other.

To create a layout, you can default export a **React component** from a file named `layout.js` (or `jsx`). This component should have a **children prop** that will be filled with either a **child layout** (if available) or a **child page** when it is rendered.

By **separating the layout** from the actual **page content**, we are able to construct **shared** elements such as headers, footers, sidebars, and menus more easily.

```
import styles from '../page.module.css'

// Save as app/about/layout.jsx
export default function PageLayout({ children }) {
    // Supports nested structures via the children prop
    return (
        <main className={styles.main}>
            {children}
        </main>
    )
}


// Updated page.jsx: <main> layout structure moved to
layout.jsx
export default function About() {
    return (
        <div className="About">
            <h1>About</h1>
            <p>This is the about page.
                Nothing to see, go <Link
href="/">home</Link>.</p>
        </div>
    )
}
```

Institute of Data

Next.js Layouts

The **root layout** is defined in the **top level app** directory and applies to **all routes**. This layout is **mandatory** and enables us to modify the initial HTML returned from the server.

The root layout must define `<html>` and `<body>` tags since **Next.js** does not automatically create them.

We can use the **built-in SEO support** to manage `<head>` elements: for example, the `<title>` and `<meta>` elements.

The **root** layout (`app/layout.js`) will wrap our **about** layout (`app/about/layout.jsx`), which would wrap any nested **about** route segments.

```
// Modified from app/layout.js - the root app layout
import NavBar from '@/components/NavBar' // see next slide
import './globals.css'
import { Inter } from 'next/font/google' // supports google fonts

const inter = Inter({ subsets: ['latin'] })

// Exported metadata will appear in the <head> section
export const metadata = {
  title: 'Create Next App',
  description: 'Generated by create next app',
}

// All layouts support nesting via children prop
export default function RootLayout({ children }) {
  // Root layout must render the <html> and <body> elements
  return (
    <html lang="en">
      <body className={inter.className}><NavBar/>{children}</body>
    </html>
  )
}

// View the source of http://localhost:3000 and /about and match
// the HTML structure back to the layout files
// ++ Modify the metadata and try to change fonts
// https://nextjs.org/docs/app/building-your-application/optimizing/fonts#google-fonts
```

Next.js Layouts

Components **common to all pages** such as a **NavBar** can be included in a **layout**.

This **NavBar** is similar to the original React version but with a few differences:

- The Next.js **Link** component uses a **href** prop instead of **to**
- There is no **NavLink** equivalent, but we can include an '**active**' class by using the **usePathname** hook
- Context and extra pages have been left out for simplicity.

*Try adding a basic **Dashboard** page to avoid the 404 when clicking on it in the **NavBar**.*

```
'use client' //          client component, not server rendered
import Link from "next/link"
import { usePathname } from 'next/navigation'

// copied from previous NavBar.jsx component, modified for Next.js
// save as src/components/NavBar.jsx
function NavBar() {
  const path = usePathname(); // hook to check current path

  return (
    <nav className="NavBar"
         style={{backgroundColor: '#09193b', color: '#14bbe5'}}>
      <ul className="menu">
        {/* Next.js Link components use href instead of to prop
*/}
        <li><Link href="/">Home</Link></li>
        <li><Link href="/dash" className={path.startsWith('/dash')
? 'active' : null}>Dashboard</Link></li>
        <li><Link href="/about"
className={path.startsWith('/about') ? 'active' :
null}>About</Link></li>
      </ul>
    </nav>
  )
}
export default NavBar

/* basic NavBar CSS, add to globals.css */
.NavBar ul { display: flex; list-style: none; justify-content:
center; padding: 0; }
.NavBar ul li a { padding: 1em; color: inherit; text-transform:
uppercase; display: inline-block; text-decoration: none; }
.NavBar ul li a.active { font-weight: bold; }
```

Next.js Linking & Navigating

The **Next.js router** uses **server-centric routing** with **client-side navigation**. It supports instant loading states and concurrent rendering. This means navigation maintains client-side **state**, avoids expensive re-renders, is interruptible, and synchronises well.

There are two main ways to navigate between routes:

- `<Link>` **Component** - demonstrated already, used similarly to **Link** in React Router DOM
- `useRouter` **Hook** - returns an **object** with **push**, **replace**, **back** and **forward** functions

Most commonly the **Link** component is used. For **events** such as **onClick**, the **push** function available through **useRouter** can **navigate** to a given **path**.

When a route **transition** is initiated using `<Link>` or calling `router.push()`, the router **updates** the **URL** in the browser. It maintains an in-memory **client-side cache**, split by **route segments**, that stores the **rendered result** of **Server Components**.

This **improves performance** by re-using segments that haven't changed, avoiding re-fetching data and re-rendering components unnecessarily.

Next.js Server vs. Client Components

Server and **Client Components** allow developers to build applications that span the server and client, combining the rich interactivity of client-side apps with the improved performance of traditional server rendering.

React Server Components introduce a **new way of thinking**, in order to build complex hybrid applications that leverage both the server and the client.

Instead of **React** rendering your whole single-page application **client-side**, **Next.js** gives us the flexibility to **choose where to render** your components based on their purpose.

By default **all** components in **Next.js** are **server components**. These are rendered into HTML on the **server**, and the generated HTML is sent to the **browser**. Any **client components** then enhance this HTML with browser-based JS for **client-side interactivity**.

To differentiate a **client** component, simply add the '**use client**' directive as the first line of the file.

NavBar *already does this, in order to use the* ***usePathname*** *hook.*

Next.js Server vs. Client Components

Server and Client components are intended to work together in Next.js. Anything in the **app** directory, such as **pages** and **layout**, will generally be a **server** component.

These server components can import and render **client** components.

What do you need to do?	**Server Component**	**Client Component**
Fetch data	✓	×
Access backend resources (directly)	✓	×
Keep sensitive information on the server (access tokens, API keys, etc)	✓	×
Keep large dependencies on the server / Reduce client-side JavaScript	✓	×
Add interactivity and event listeners (onClick(), onChange(), etc)	×	✓
Use State and Lifecycle Effects (useState(), useReducer(), useEffect(), etc)	×	✓
Use browser-only APIs	×	✓
Use custom hooks that depend on state, effects, or browser-only APIs	×	✓

Data Fetching

Data fetching in **Next.js** is usually the job of **server components**.

This example is a Next.js equivalent to the previous **PostsPage** React Router example.

Next.js extends the **fetch** function to simplify its **return** and support **caching** and **revalidating**, as well as including helper functions for **headers** and **cookies**.

The recommended pattern is to create an **async getData** function which is called from the **server component**.

```
import Link from "next/link"

// Save as app/posts/page.jsx and copy layout.jsx from /about
async function getPostsData(limit, page = 1) {
    const res = await fetch('https://jsonplaceholder.typicode.com/' +
                        'posts?_limit='+limit+'&_page='+page);

    if (!res.ok) { // Recommendation: handle errors
        // This will activate the closest `error.js` Error Boundary
        throw new Error('Failed to fetch posts')
    }
    return res.json()
}

export default async function Posts() {
    const posts = await getPostsData(5);
    const postList = posts.map(post => (
        <li key={post.id}><Link href={"/posts/" + post.id}>
            Post #{post.id}: {post.title}</Link></li>
    ))

    return (
        <div className="Posts">
            <h1>Posts</h1>
            <ul>{postList}</ul>
        </div>
    )
} // ++ Update the NavBar to include this new /posts page route
```

Institute of Data

Dynamic Routes

Since **Next.js** uses **file-system-based routing**, its support for **dynamic** routes (where the **path segments** are not fixed or known in advance) leverages **file system naming conventions**.

A **Dynamic Segment** can be created by wrapping a folder name in **square brackets**. For example, `[id]`.

The bracketed folder name (`id`) is passed with the matched path segment (`1` or `2` etc) as the **params** prop to **layout** and **page** components.

```jsx
import Link from "next/link"
// Save as app/posts/[id]/page.jsx
async function getPostData(id) {
    const res = await fetch('https://jsonplaceholder.typicode.com/'
                + 'posts/' + id);

    // Recommendation: handle errors
    if (!res.ok) {
        // This will activate the closest `error.js` Error Boundary
        throw new Error('Failed to fetch post #'+id)
    }
    return res.json()
}
// Uses params prop to get value of [id] from path segment
export default async function Post({params}) {
    // so for /posts/3/, params will be { id:3 }
    const post = await getPostData(params.id);
    return (
        <><div className="post">
            {post ?
                <><h3>Post #{post.id}: {post.title}</h3>
                  <p>{post.body}</p></>
            : "Loading ..." }
        </div>
        <Link href="/posts">All Posts</Link></>
    )
} // ++ Try adding Next Post and Previous Post links
```

Institute of Data

Server & Client

Posts is currently a **server** component which fetches data. To implement the **page limit** drop-down from earlier, we will also need to introduce some **client** functionality to allow the user to choose the limit value and re-render the screen.

We can access the **search** (or **query**) parameters from both client and server components.

useSearchParams is a client-side hook allowing **read-only** access. We can **update** search params by changing the route via **useRouter**.

```
'use client'
// save as components/PostsLimit.jsx

import { useRouter, usePathname, useSearchParams } from 'next/navigation'

// Client component allowing user to choose the number of posts displayed
// and set new value in search params
export default function PostsLimit(defaultLimit) {
    const searchParams = useSearchParams(); // next.js hook for search
params
    const router = useRouter(); // next.js hook for client side navigation
    const pathname = usePathname(); // next.js hook for current URL path

    const limit = searchParams.has('limit') ?
                  searchParams.get('limit') : defaultLimit;

    const handleChangeLimit = (e) => {
        // change the route to the existing path plus the new search param
        router.replace(pathname + '?limit=' + e.target.value)
    }

    return (
        <label className="PostsLimit">Number of posts:
            <select onChange={handleChangeLimit} value={limit}>
                <option>5</option><option>10</option><option>20</option>
            </select>
        </label>
    )
}

// See next slide for changes to server-side Posts component
```

Server & Client

Once the user has chosen a new posts limit via the client-side component, the **route** is updated to include the new **limit** value as a **search parameter**, eg. *localhost:3000/posts?limit=10.*

Server components can then access **read-only search parameters** via the **searchParams** prop object.

We can then use that **limit** value to fetch the required amount of posts.

Since this is a new route and a new server request, the updated markup is sent to the browser for rendering and the user sees the updated post list.

```
// Updated app/posts/page.jsx

// We can use searchParams to send the user's chosen value from
// client component to server component

// Page server components can access searchParams as a parameter
export default async function Posts({searchParams}) {
    const limit = searchParams.limit ? searchParams.limit : 5;
    const posts = await getPostsData(limit);
    const postList = posts.map(post => (
        <li key={post.id}><Link href={"/posts/" + post.id}>
            Post #{post.id}: {post.title}</Link></li>
    ))

    return (
        <div className="Posts">
            <h1>Posts</h1>
            <ul>{postList}</ul>
            <PostsLimit defaultLimit={limit}/>
        </div>
    )
}

// ++ Make sure to import the PostsLimit component

// ++ View the HTML page source (not just via the Dev Inspector) to verify
// that the server-rendered HTML includes the updated list of posts

// ++ Add another drop-down to let the user choose the current page
```

Error Handling

Next.js includes built-in support for **Error Boundaries**.

Creating an error handler is as simple as creating a file called `error.js` (or `jsx`).

Any **runtime errors** that occur will render the **closest error component** for that **route**, instead of crashing the application.

When this **fallback error component** is active, layouts **above** the error boundary maintain their **state** and remain **interactive**, and the error component can attempt to **recover** from the error.

```
'use client' // Error components must be Client Components

import { useEffect } from 'react'

// Save as app/posts/error.jsx
export default function Error({ error, reset }) {
    useEffect(() => {
        // Log the error to an error reporting service
        console.error(error)
    }, [error])

   return (
      <div>
         <h2>Something went wrong!</h2>
         <p>{error.message}</p>
         {/* Attempt to recover by trying to re-render the segment
*/}
         <button onClick={() => reset()}>
            Try again
         </button>
      </div>
   )
}
// Test it out by requesting a post id that doesn't exist, like
// http://localhost:3000/posts/500
```

Institute of Data

Next.js Summary

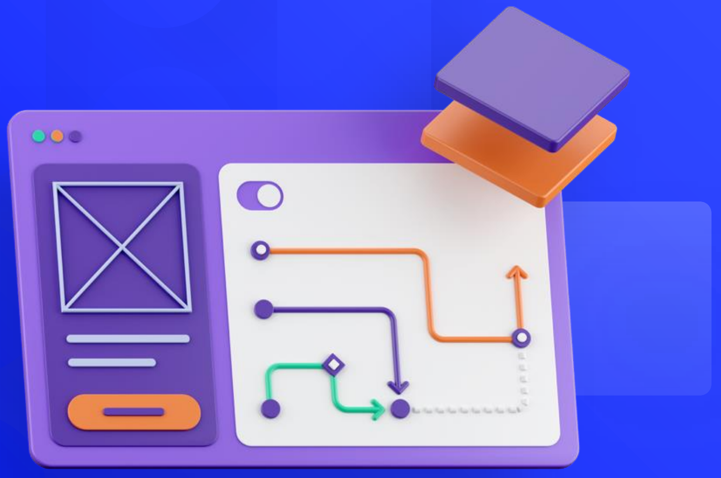
Next.js is a rapidly-evolving, widely-used, feature-rich **React framework** that provides **full-stack application** potential. It includes built-in support for features such as **routing**, **performance optimisations** and **error handling**, to simplify some common React tasks.

It also introduces several new concepts, and a new way of thinking involving **server-side vs client-side** task separation. These ultimately increase flexibility and support for complex applications, but can be overwhelming for beginners.

Although this course includes an **introduction** to the key concepts of **Next.js**, there is much more that cannot be covered in this limited timeframe. To explore more, please see the documentation at Next.js.

***Optional extension exercise**: Try re-doing Exercise 4 using Next.js!*

Section 5:
React UI Libraries



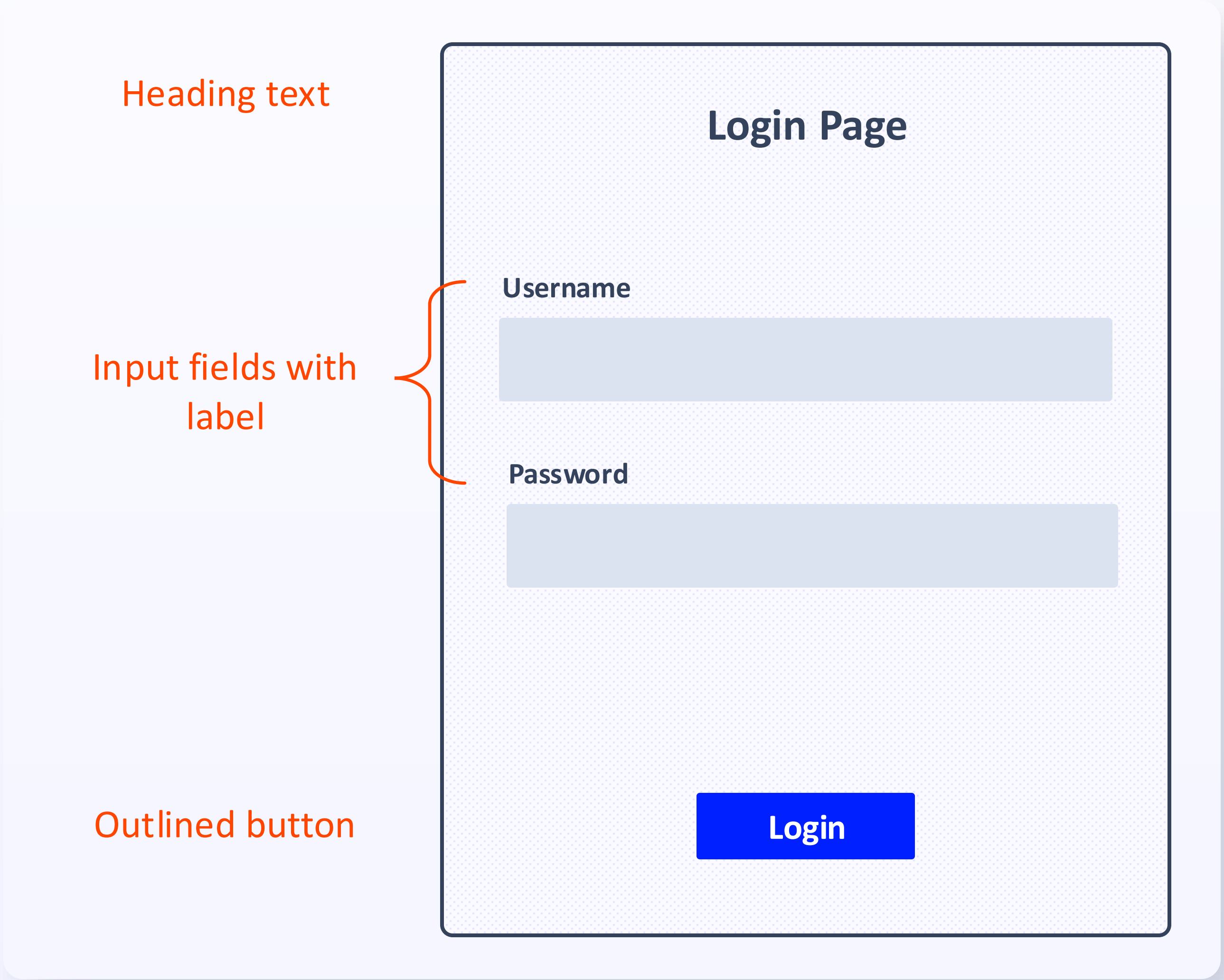
Web applications are always required to be attractive, informative and user-friendly. A web application with a nice colour scheme and good looking layout is definitely preferred to its counterparts with boring and old-fashioned appearance.

However, not all programmers are good at creating good-looking layouts and components. Therefore, the creation of UI libraries saves us and makes this process easier.

What is a UI Library?

In React, everything is a **component**. So there are countless components that we will need throughout a project. Each needs to look both **consistent** and **attractive** as well as being **user-friendly**. Take the **Login Page** on the right for example. It includes several elements to be all styled individually.

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



Popular React UI Libraries

- **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. Each component has been built from scratch as a true React component, without unneeded dependencies.
- **Ant Design React**: Dedicated to providing high quality components and demos for building rich, interactive user interfaces.
- **Semantic UI React, Chakra UI**, etc.

MUI and Bootstrap are probably the most popular. Each framework has its own style of design and syntax. However, they all have the same objective:
to provide easier and quicker frontend developing experiences for developers.

MUI vs. Bootstrap

Material UI (MUI) is based on Google's Material Design.

- Hundreds of pre-built, highly configurable components
- Modern, attractive styling out-of-the-box
- Many built-in styling options
- Breadth of configuration options can make some components overly complex (eg. AppBar)
- Can be difficult/confusing to implement custom styling/theming

React Bootstrap is based on existing Bootstrap CSS/JS

- Hundreds of pre-built, easily configurable components
- Basic, user-friendly styling out-of-the-box
- Add custom styling with CSS
- Intuitive configuration focused on commonly used options
- Usually requires more custom work to style as needed and include all desired features

MUI vs. Bootstrap

Although **React Bootstrap** can be easier to work with initially, and is still a popular choice for many applications, this course will focus on **MUI** for its modern look and wider range of components.

The process for using any UI library is quite similar:

1. Use the **documentation** (Material UI - Overview) or (React Bootstrap)
2. Find a close **example** of the component you want to implement
3. **Copy and paste** the code into your application (usually as a new component)
4. **Test** and **modify** and **integrate** as needed.

Getting Started with MUI

First we need to install the **MUI library** into our **React** project:

```
npm install @mui/material @emotion/react @emotion/styled
```

To use components that include **icons**, install the **icon library** too:

```
npm install @mui/icons-material
```

Setup is as easy as that! Now we can copy and adapt any of the examples from the MUI documentation.

MUI Card

A **Card** is a basic building block component containing **content** and **actions** about a **single subject**.

MUI has several variations at React card component (this one is towards the end).

Usually we wrap a **custom component** around the **base** MUI version to **extract** the setup details (sizing, colours, variants, etc) and allow **customisation** as needed via **props**. Then we can render it like this:

```
<CustomCard title="Iguana">Green Lizard Card</CustomCard>
```

```
import Card from '@mui/material/Card';
import CardContent from '@mui/material/CardContent';
import CardMedia from '@mui/material/CardMedia';
import Typography from '@mui/material/Typography';
import { Button, CardActionArea, CardActions } from
'@mui/material';

// wraps the default MUI Card component to customise it with props
export default function CustomCard({title, children}) {
  return (
     <Card sx={{ maxWidth: 345 }}>
        <CardActionArea>
           <CardMedia component="img" height="140" alt="iguana"
  image="https://mui.com/static/images/cards/contemplative-
reptile.jpg" />
             <CardContent>
               <Typography gutterBottom variant="h5"
component="div">
                   {title}
               </Typography>
               <Typography variant="body2" color="text.secondary">
                   {children}
               </Typography>
           </CardContent>
         </CardActionArea>
         <CardActions>
             <Button size="small" color="primary">Share</Button>
         </CardActions>
     </Card>
  );
}
// ++ Add support for a button text prop as well, test rendering
it
```

MUI Grid

Grid is a responsive layout component that uses CSS Flexbox to adapt to screen size, ensuring consistency across layouts.

It uses a similar 12-column setup to the Bootstrap grid used in earlier modules.

There are two types of **Grid** layout: **containers** and **items**. Make sure to use **both**.

There are five grid **breakpoints**: **xs**, **sm**, **md**, **lg**, and **xl**.

Breakpoints can be assigned integer values, representing the number of **columns occupied** (out of the 12 available), when the **viewport** width matches the **breakpoint**.

```
import * as React from 'react';
import { styled } from '@mui/material/styles';
import Grid from '@mui/material/Grid';
import CustomCard from './MUICard';

// layout cards in a grid
export default function BasicGrid() {
   // Outermost Grid is a container
   return (
      <Grid container spacing={2} my={2}>
         {/* Inner Grids (columns) are items */}
         <Grid item xs={4}> {/* use 4/12 columns on xs screens up
*/}
            <CustomCard title="First Column">
               First column is wider
            </CustomCard>
         </Grid>
         <Grid item xs={2}>middle item</Grid>
         <Grid item xs={2}>middle item</Grid>
         <Grid item xs={4}>
            <CustomCard title="Last Column">
               Last column is wider
            </CustomCard>
         </Grid>
      </Grid>
   ); // 4 + 2 + 2 + 4 = 12, so grid will be 4 columns on one row
}

// ++ Create some different Cards and lay them out in a Grid
// see https://mui.com/material-ui/react-grid/ for more options
```

MUI Forms

MUI includes several useful components for building great-looking **forms**, including **TextField**, **Button**, **Select**, and **Checkbox**.

These can all be **customised** in many different ways, using the supporting FormControl elements to build and lay out the form you need.

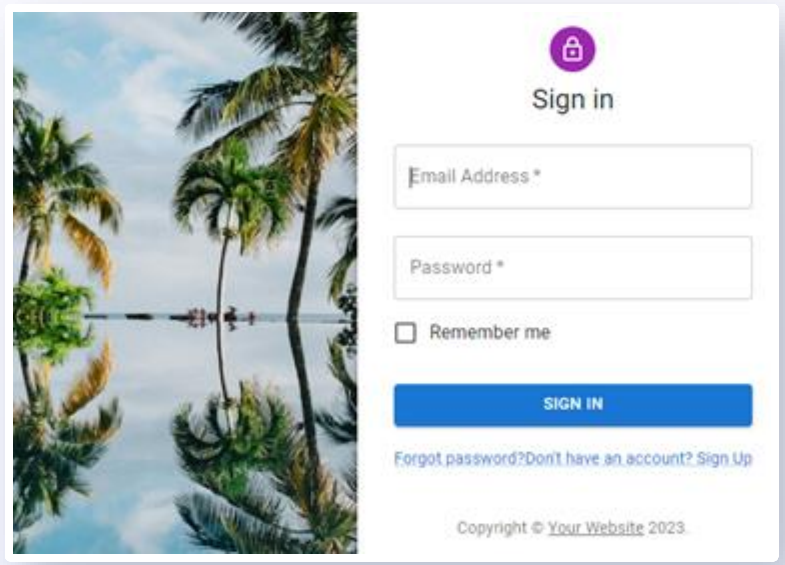
MUI form elements can be **controlled** or **uncontrolled** and integrated with built-in **React** features such as **state**, **refs** and **custom hooks**.

```
import { useState } from "react"
import { TextField, Checkbox, Select, MenuItem, Button } from
"@mui/material";
import { FormControl, FormControlLabel, InputLabel } from
"@mui/material";

export default function MUIForm() {
   const [age, setAge] = useState(10);
   return (
     <form>
        <TextField required id="outlined-required"
           label="Greeting" defaultValue="Hello World" />

         <FormControlLabel control={<Checkbox defaultChecked />}
           label="Uncheck Me" />

         <FormControl>
           <InputLabel>Age</InputLabel>
           <Select value={age} onChange={e =>
setAge(e.target.value)}>
              <MenuItem value={10}>Ten</MenuItem>
              <MenuItem value={20}>Twenty</MenuItem>
              <MenuItem value={30}>Thirty</MenuItem>
           </Select>
        </FormControl>

        <Button variant="contained">Submit</Button>
     </form>
   )
}
// ++ Build a MUI version of your LoginForm component
```

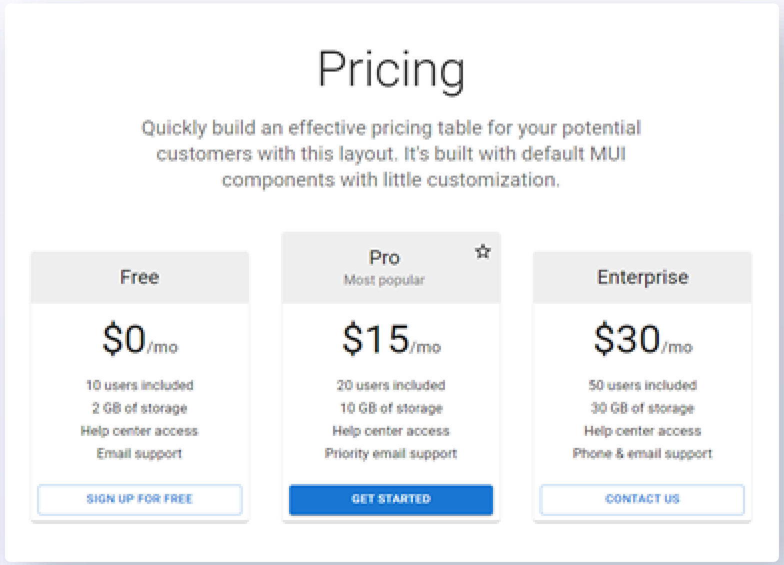
MUI Templates

For even faster page building, there are several useful templates available in the MUI documentation at React Templates.

These combine various MUI components in ready-to-use **page layouts** such as **sign-up**, **login**, **checkout**, and **pricing**. All include code ready for copying and adapting.

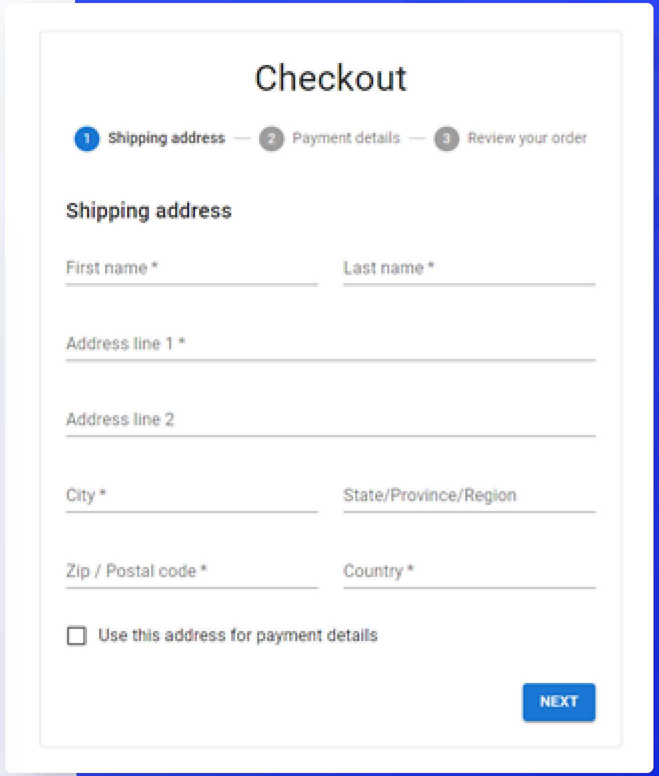


Copyright © Your Website 2023.



Quickly build an effective pricing table for your potential customers with this layout. It's built with default MUI components with little customization.

Free	Pro (Most popular)	Enterprise
$0/mo	$15/mo	$30/mo
10 users included	20 users included	50 users included
2 GB of storage	10 GB of storage	30 GB of storage
Help center access	Help center access	Help center access
Email support	Priority email support	Phone & email support
SIGN UP FOR FREE	GET STARTED	CONTACT US



MUI Dialogs

A **modal** window displaying a **dialog** is a common tool to display extra information or feedback to the user without them leaving the page.

Dialogs **disable** all app functionality when they appear, and **remain** on screen until confirmed, dismissed, or a required **action** has been taken. They are purposefully **interruptive**, so should be used **sparingly**.

Dialogs need **state** to determine when they are open, and a **trigger button** as well as the **content** inside the dialog.

```
import * as React from 'react';
import Button from '@mui/material/Button';
import Dialog from '@mui/material/Dialog';
import DialogActions from '@mui/material/DialogActions';
import DialogContent from '@mui/material/DialogContent';
import DialogContentText from '@mui/material/DialogContentText';

export default function MUIDialog({ text }) {
   const [open, setOpen] = React.useState(false);
   const handleClickOpen = () => setOpen(true);
   const handleClose = () => setOpen(false);

   return (
      <div>
         <Button variant="outlined" onClick={handleClickOpen}>
            Open Dialog
         </Button>
         <Dialog open={open} onClose={handleClose}>
            <DialogContent>
               <DialogContentText id="alert-dialog-description">
                  {text}
               </DialogContentText>
            </DialogContent>
            <DialogActions>
               <Button onClick={handleClose} autoFocus>OK</Button>
            </DialogActions>
         </Dialog>
      </div>
   );
} // Render as <MUIDialog text="My first MUI Dialog" />
```

MUI Styling

MUI provides several different options for customising a component's styles, so you can choose the most suitable:

1. **One-off** customisation (**sx** or **className** prop)
2. **Reusable** component (**styled** utility)
3. **Global theme** overrides (**ThemeProvider**)
4. **Global CSS** override (**GlobalStyles** component)

The **sx prop** is available on all MUI components and can be used to set specific CSS rules.

```
import { Box } from '@mui/system';

// Customising style using sx prop
export default function SxExample() {
    return (
        <Box sx={{ bgcolor: '#efefef', minWidth: 300,
                boxShadow: 1, borderRadius: 4,
                p: 2, // padding
            }}>
            <Box sx={{ color: '#55f', fontSize: '2em',
                    textTransform: 'uppercase' }}>Sessions</Box>
            <Box sx={{ color: '#333', fontSize: 34,
                    fontWeight: 'bold' }}>98.3 K</Box>
            <Box sx={{ fontSize: 14, mx: 0.5, // horizontal margins
                    display: 'inline', fontWeight: 'bold'}}>
                +18.77%
            </Box>
            <Box sx={{ color: '#999', display: 'inline',
                    fontSize: 14 }}>vs. last week</Box>
        </Box>
    );
}
// https://mui.com/system/getting-started/the-sx-prop/ has good tips
// Camel-case the CSS properties
// Some properties like margin and padding have special syntax
// ++ Experiment with the sx prop in some existing MUI components
```

MUI Styling

styled is a utility function that **wraps** a given element/component by **applying the given styles**, turning it into a '*styled*' version of itself while still retaining all other original functionality and props.

It is a handy way to define a common set of styles and apply them to any number of components.

It doesn't include all of the same CSS shortcuts as the **sx** prop.

styled can also take the current **theme** into account.

```
import { Box } from '@mui/material';
import { styled } from '@mui/system';

const styles = {
    color: 'darkslategray',
    backgroundColor: 'aliceblue',
    padding: '2em',
    borderRadius: '1em',
    border: '1px solid darkslategray',
    margin: '1em 0'
}

const StyledDiv = styled('div')(styles);

const StyledBox = styled(Box)(styles);

export default function BasicUsage() {
    return (
        <>
            <StyledDiv>This div includes styles</StyledDiv>
            <StyledBox>This box has the same styles</StyledBox>
        </>
    );
}

// see https://mui.com/system/styled/ for more tips
```

MUI Theming

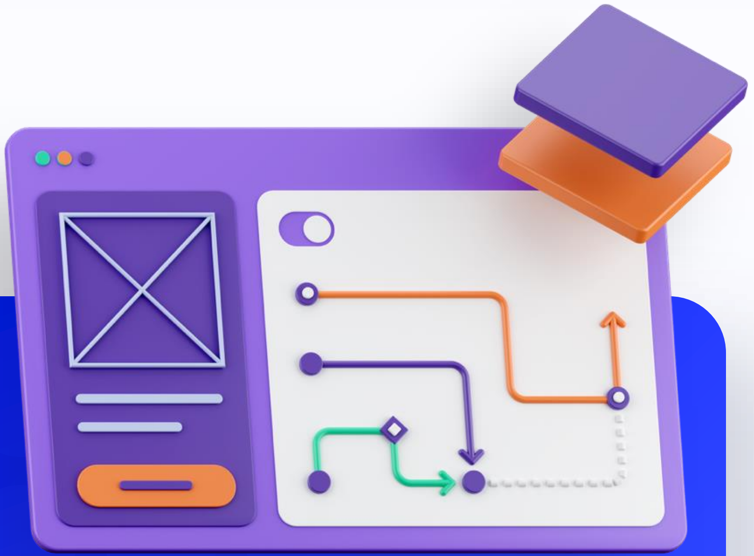
MUI **Themes** let you apply a consistent tone to your app. It allows you to customise all design aspects of your project in a central place, in order to brand and style all components in your app consistently.

To customise the theme, we first create a custom theme using **createTheme**. We then use the **ThemeProvider** component to set this theme at the top level of our app.

ThemeProvider relies on the **context** feature of React to **pass the theme** down to all child/nested components.

```
import { createTheme } from "@mui/material/styles";
// save as themes/tealTheme.jsx
// creates a new theme containing overrides for any defaults
// see https://mui.com/material-ui/customization/theming/
export const tealTheme = createTheme({
    palette: {
        primary: { main: '#214D4C', contrastText: '#efefef' },
        secondary: { main: '#3CA899', contrastText: '#ffffff' }
    },
    typography: {
        fontFamily: 'Montserrat',
        fontSize: 14,
        h1: { fontSize: 30 }
    },
    shape: { borderRadius: 0 },
    components: {
        MuiCssBaseline: {
            styleOverrides: `a { color: #3CA899; }`,
        },
        MuiButton: { defaultProps: { variant: 'contained' } },
        MuiTextField: { defaultProps: { variant: 'filled' } }
    }
});
// in App.jsx: import theme and provider, wrap around component
tree
import { ThemeProvider } from "@mui/material/styles";
import { tealTheme } from './themes/tealTheme';
<ThemeProvider theme={tealTheme}>{/* App.jsx components
*/}</ThemeProvider>
```

React UI Libraries



So, as we can see, creating components using UI libraries is easier, quicker and less complicated than writing our own. Furthermore, modern libraries provide us a smoother look in the UI and include common accessibility considerations.

UI libraries provide us almost all components that we need to build web applications. Basic components like *Card, Typography, List, and Box* can quickly display your content. More advanced components like *Grid, Collapsible, Breadcrumbs, Modal, etc* can ensure consistent, responsive layouts across your whole app. All can be implemented with ease by following the documentation.

Take time to experiment with MUI components and templates (or another library); your applications will look a lot better as a result!

Exercise 5

- Update your solution for **Exercise 4** to use MUI components for styling
- Use the **AppBar** for navigation and MUI form components for any form inputs
- **Extension**: Include the **PostList** component and style using MUI cards and grids
- **Extension**: Try to create a custom theme using **createTheme**