Institute of Data

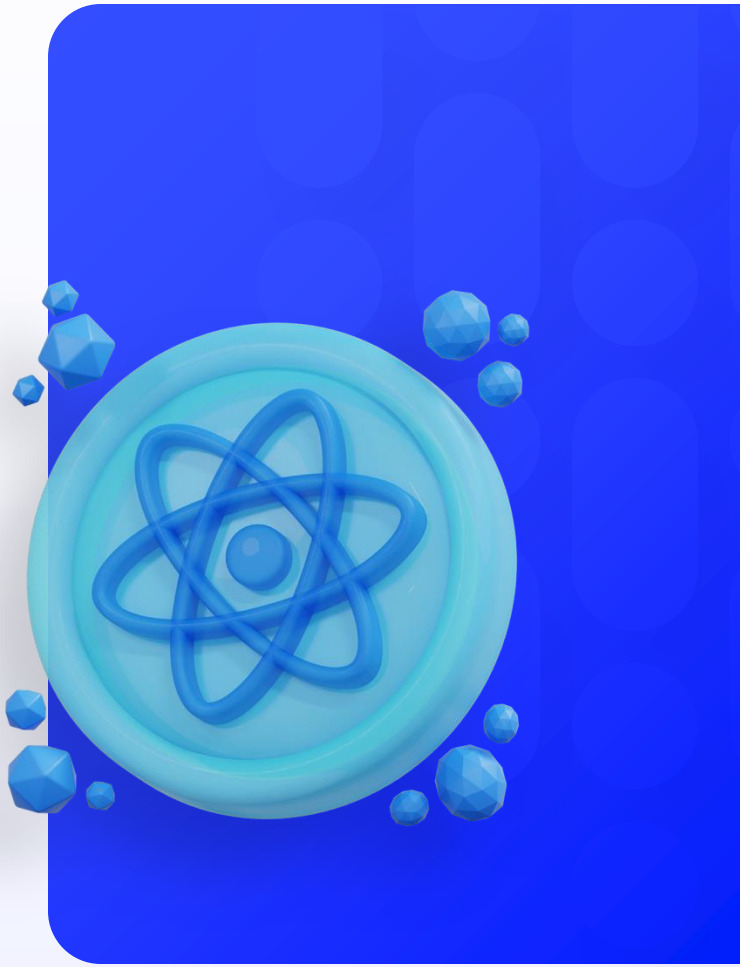
Software Engineering

Module 6 React JS 1/2

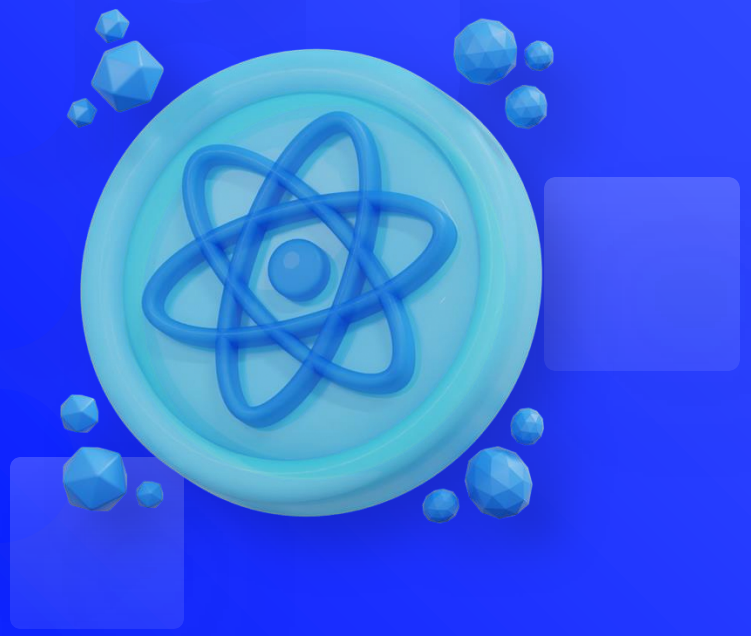
Agenda

Section 1 : React Basics

- JSX
- Components
- Props
- Conditional Rendering
- List and Keys
- State
- Handling Events
- Forms
- Error Boundaries
- Thinking in React



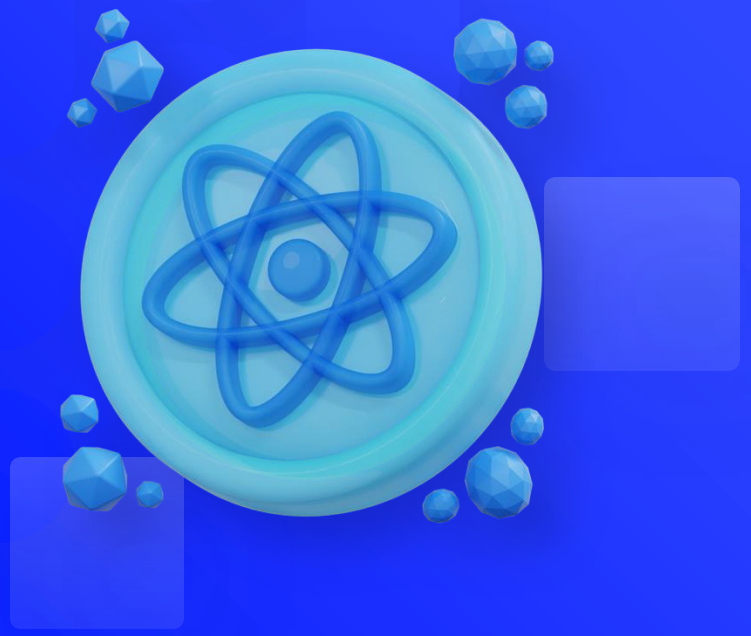
Introduction



React JS has been the world's most popular frontend framework for years. It was originally developed by and for Facebook, and is used by many other big companies such as Instagram, Netflix, Uber, Discord, Airbnb and more.

There are so many features and benefits to using React in frontend development. It makes web application development **easier** for developers and also easier for people to learn. Moreover, it is also known to be **declarative**, **powerful**, **fast**, **extensible** and **component-based** (almost everything you render on a web page is a component).

Introduction



Declarative programming focuses on **‘what’** should happen rather than **‘how’** it needs to be done, and is a good way to hide complexity and make code easier to write. React does this by enabling developers to specify what the UI should do (how it should **‘react’**) in response to user activity.

Instead of directly manipulating the DOM, it builds and manages a **‘virtual DOM’** which tracks and bundles component updates and renders changes seamlessly and quickly for the user.

React also **scales well** for both small and large applications, and has a well-established **developer community**, with lots of powerful extensions, great documentation and helpful support when needed.

Introduction

Why use React?

Speed : React is fast because it is optimised to bundle multiple screen updates together, and only update components that have changed since last render.

Modularity : React components provide a logical way to modularise large amounts of code into efficient, re-usable chunks.

Scalability : React was specifically designed to handle large amounts of constantly changing data in complex user interfaces, and its component-based architecture scales well.

Flexibility : React can be used to make simple websites, large and complex sites, for mobile apps and even for server-side applications (with additional frameworks).

Popularity : Since it’s proven to be so useful, learning React makes you more employable and gives current, relevant marketplace skills as well as access to a thriving developer community.

React Basics

To get started using React, there are two main strategies, depending whether you want to start a new React application from scratch, or add it to an existing project.

Both ideally require a Node.js-type environment that supports modular coding via import/export statements and which can make use of npm packages.

The simplest, modern way to get started for beginners is to use a build tool such as Vite, (pronounced *veet*) which aims to 'provide a faster and leaner development experience for modern web projects' by getting you started with all the recommended configuration defaults, to make your React applications easier to develop, build, test and run.

There are many other options including Webpack, Parcel and Rollup, and some now legacy options such as create-react-app, but Vite takes advantage of modern JS bundling features to ensure fast build and run times and improve the developer experience, both for beginners and professionals.

React Basics

First create a new GitHub repo (or folder) for Module 6. From this new (empty) folder, run:

```
npm create vite@latest
```

When it asks you, provide a name for your React app (eg. ‘first-react-app’), and use the arrow keys to select a **React** application that uses **JavaScript**:

```
Ok to proceed? (y) y
√ Project name: ...  first-react-app
√ Select a framework: » React
√ Select a variant: » JavaScript

Scaffolding project in D:\IOD\SE\Module6_React\first-react-app...

Done. Now run:

  cd first-react-app
  npm install
  npm run dev
```

Finally, run the last 3 commands as Vite suggests above.

```
VITE v4.3.3  ready in 4598 ms

  ➜  Local:   http://127.0.0.1:5173/
  ➜  Network: use --host to expose
  ➜  press h to show help
```

You should see a message like this after running **npm run dev**. Ctrl-click the link to open your React app in a browser.

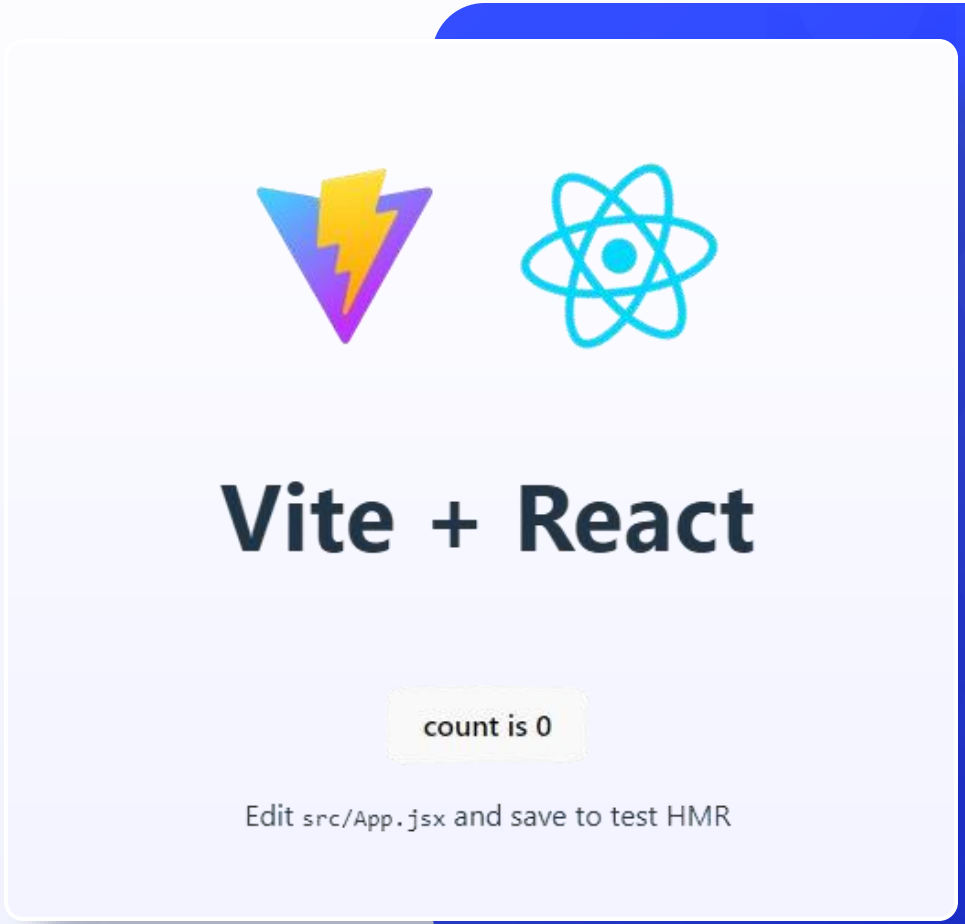
React Basics

JSX Features

Take a look in **App.jsx** inside your new React application (in the **src** folder). This contains some starter JSX code and provides an example of a **component** (more to come later).

This example will run in your browser and any changes you make in App.jsx will be reflected without needing restarts or refreshes. This is achieved through **Hot Module Replacement** (HMR), a feature of Vite and similar tools to speed development.

Everything inside the **return** block is JSX code. Take a close look and note the differences and similarities between HTML and JS.



React Basics

What is JSX?

Take a look at the code on the right.

What kind of code do you think it is? Is it HTML? JavaScript?

It is JavaScript.

=> The part that looks like HTML, <h1>Hello world</h1>;, is something called ***JSX* (Javascript XML)**.

```
const h1 = <h1>Hello world</h1>;
```

React Basics

Attributes In JSX

JSX elements can have attributes, just like HTML elements can.

A JSX attribute is written using HTML-like syntax: a **name**, followed by an **equals** sign, followed by a **value**. The attribute value should be wrapped in **quotes**, like this:

```
// This <p> element has an id attribute with the value large
const p1 = <p id='large'>foo</p>;
```

React Basics

JSX is a **syntax extension** to JavaScript. React uses it to **describe** what the **UI** should look like. JSX puts **markup in the JavaScript**, which makes it look like a **template** language, and it also comes with the full **power of JavaScript**.

```
const whatIsJSX = <div>
        <p>Is it JavaScript?</p><p>Is it HTML?</p>
        <p>No, it's JSX!</p>
    </div>
```

Syntax extension: Since JSX is not valid JavaScript, and not quite HTML either, web browsers can't read or interpret it directly.

If a JavaScript file contains JSX code, then that file will first have to be **compiled** before being sent to the browser. React's JSX compiler (Babel) will translate any JSX into regular JavaScript. *JSX and React are two separate things, but are often used together*.

React Basics - JSX

Writing markup in JSX

JSX looks very similar to HTML and uses many of the same syntax rules, although it is **more strict**. There are some key differences to be aware of:

- All tags need to be closed.
 and <img> is **valid HTML** for example, but **invalid JSX** - it should be
 and <img/>
- All elements and attributes are **case-sensitive** in JSX and need to use camel case (including event listeners like onClick), while HTML is less strict
- Instead of using the **class** attribute for CSS, JSX elements need to use **className**
- *(continued…)*

React Basics - JSX

Writing markup in JSX

- To write a **comment** in JSX use {/* comment here */} instead of the HTML syntax <!-- comment here --> or the JS syntax //
- All JSX expressions need to use a **single parent element**. So the following causes a syntax error, because both **p** tags are at the top level:

```
const invalidJSX = <p>paragraph 1</p><p>paragraph 2</p>;
// JSX expressions must have one parent element
```

Try out the above in App.jsx, and then fix the error by ensuring that the expression has a single parent element.

React Basics - JSX

Embedding expressions in JSX

- Curly braces are used in JSX to ‘escape back’ into JavaScript and evaluate standard JS expressions, including **variables**, **functions**, **conditional statements** and more.
- To use a JS expression as the value of an **attribute** in JSX, replace the quotes with curly braces.

In **App.jsx**, take a close look at each use of curly braces:

- Rendering locally hosted images
- Setting the event handler function for clicking on a button
- Displaying the value of a variable.

React Basics - JSX

Embedding JSX in expressions

After compilation, JSX expressions become **regular JavaScript function calls** and **evaluate to JavaScript objects**. This means you can use JSX inside of **if statements** and **for loops**, **assign it** to variables, accept it as **arguments** and **return** it from **functions**. We start by assigning it to variables as below:

```
const spiderman = { name: 'Spiderman', alterEgo: 'Peter Parker', catchPhrase: 'With great power comes great responsibility' };
const spideyJSX = (<div>
    <h3>{spiderman.name}</h3>
    <blockquote>{spiderman.catchPhrase}</blockquote><cite>{spiderman.alterEgo}</cite>
</div>);
```

Incorporate the above code into App.jsx so it displays on screen.

See JavaScript in JSX with Curly Braces for more examples.

React Basics - JSX

Fragments - <> & </>

You may remember that one of the rules of JSX is that it must have a **single parent element**.

We can follow this rule by wrapping everything in a **parent div**.

To avoid using **unnecessary markup**, another solution is to use a **fragment**.

Fragments **don't render** in the HTML, but still satisfy the single parent element rule.

App.jsx has an example of this, but we can also use our own fragments:

```
// single parent <div> element
const spideyJSX = (<div>
  <h3>{spiderman.name}</h3>

<blockquote>{spiderman.catchPhrase}</blockquote>
  <cite>{spiderman.alterEgo}</cite>
</div>);

// single parent fragment element
const spideyJSXFragment = (<>
  <h3>{spiderman.name}</h3>

<blockquote>{spiderman.catchPhrase}</blockquote>
  <cite>{spiderman.alterEgo}</cite>
</>);

// render each of these examples and check the
difference in the HTML in your browser.
```

React Basics - JSX

React can be used without JSX, although in practice this is rarely done. React's **Babel** compiler transforms JSX into React.createElement() **calls** under the hood, which are then used as standard JS objects.

```
// standard JSX syntax for creating an element - render with {jsxElement}
 const jsxElement = <h1 className="greeting">Hello World</h1>

 // bypassing JSX and creating the element in React directly - render with {nojsxElement}
 const nojsxElement = React.createElement('h1', { className: 'greeting' }, 'Hello World')

 // the actual JS representation of the element as an object - can't render directly
 const jsElement = { type: 'h1', props: { className: 'greeting', children: 'Hello World' } }
```

React Basics

What is a component?

A component is a small, reusable chunk of code that is responsible for one job - usually to render some HTML.

Take a look at the 3 code samples on the right, then copy the first 2. This code will **create**, then **render** a new React component, then **convert** it all to HTML and JS in the browser:

```
// This creates a new component as a function that returns
some JSX.
// Add this in App.jsx ABOVE the App component
function ExampleComponent() {
     return (
        <div className="ExampleComponent componentBox">
            <h1>My Example Component</h1>
            <p>My first React component!</p>
        </div>
    )
}
```

```
{/* This renders the component, calling the function and
including its JSX output at this point in the code. Add this
INSIDE the return block of the App component. */}
<ExampleComponent/>
```

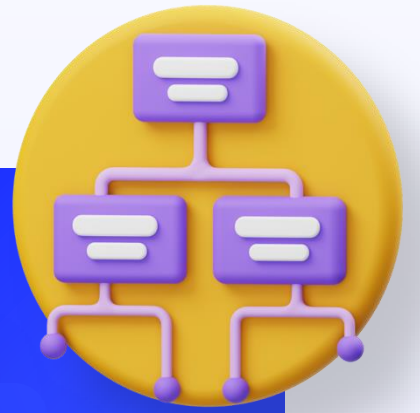
```
// This code in main.jsx handles rendering the App component
// into the DOM element with an id of 'root' (in index.html)
ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

React Basics

Embedding JSX in expressions

A React application can have dozens, even hundreds, of components.

Each of them may only do one very small thing. But when combined, they can make huge and complex applications.



React Basics - Components

Rendering a component

React can render **native DOM components**, differentiated by lowercase:

```
<a href="https://react.dev" target="_blank">
    <img src={reactLogo} className="logo react" alt="React logo" />
</a>
```

React also renders custom, **user-defined** components. We can pass **JSX attributes** and **children** to these components as a **single object** (named props).

```
function Welcome(props) { // custom Welcome component
  return (
    <div className="Welcome">
      {/* if the 'name' prop exists, render it on the screen */}
      <h3>Welcome {props.name}!</h3>
      {/* if this component has children, render them here */}
      {props.children}
    </div>
  )
}
```

```
{/* Renders the Welcome component with a name
prop and a child (nested) element */}

<Welcome name="students">
    <p>Children of Welcome</p>
</Welcome>
```

React Basics - Components

Styling a component

Since components render **HTML**, we can still use **CSS** to style them, in several ways. A neat, concise method is to import **external** CSS rules from a separate file, for styling **classes** specified in the JSX using the **className** attribute.

```
import './App.css' // at the top of App.jsx

// componentBox class styles a component into a box
// Welcome class identifies this component
function Welcome(props) {
  return (
    <div className="componentBox">
      <h3>Welcome {props.name}!</h3>
    </div>
  )
}
```

```
/* add into App.css to apply to entire component tree
*/
.componentBox { border: 1px solid #aaa; margin: 1em;
padding: 1em; text-align: center; background:
#efefef; }
```

Complex components often have a dedicated CSS file with a matching name, and we can put global styles into an application-wide CSS file.

React Basics - Components

Styling a component

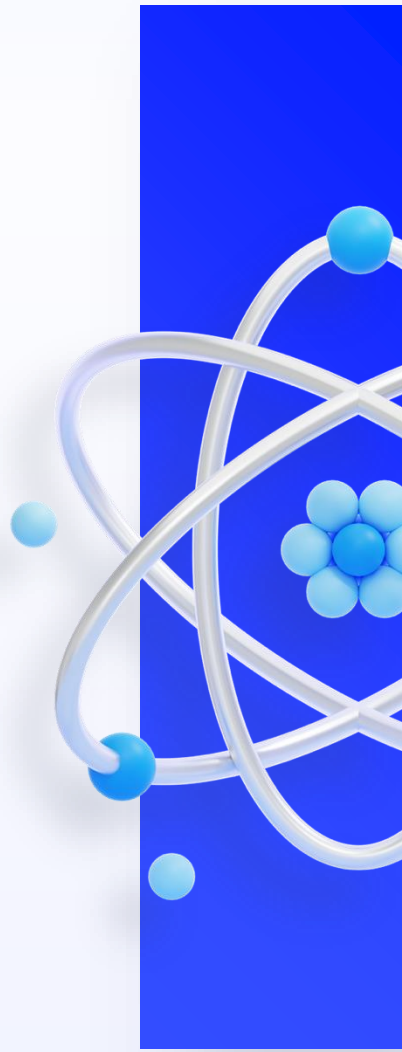
We can also specify **in-line** CSS rules using the **style** attribute. The style attribute is handled as an **object**, with individual CSS rules as **properties**.

```
<h3 style={{color: 'blue', textTransform: 'uppercase'}}>Welcome!</h3> {/* style object directly in JSX */}
```

Inline style objects use **double curly braces**. The **outer** braces signify **JS goes here** (a JSX expression) and the **inner** braces contain the style object itself.

```
const divStyle = { background: 'lightblue', padding: '1em', fontWeight: 'bold' };
...
<div style={divStyle}>This is a styled div.</div> {/* style object as a variable - neater and re-usable */}
```

Remember to camel-case CSS property names and quote the values as strings.



React Basics - Components

Components and Props

Components let us **split** the UI into **independent**, **reusable pieces**, and think about each piece in **isolation**, with distinct logic and appearance.

Components are represented as **JavaScript functions**. They accept **arguments** (called props) and return **React elements (JSX)** describing what **should appear** on the screen. Props are passed from the **parent** component to the **child** component, following a **"top-down"** or "unidirectional" flow.

By convention, each component begins with a **capital letter** (to distinguish it from a regular function). Historically, components were represented as **classes**. This syntax is now **legacy** but you may still see it in older codebases.

React Basics - Components

Importing & Exporting

To keep our components **independent** and **reusable**, we usually want to store each one in its **own file** with a **matching uppercase name**. (A single-use component only used by a parent may sometimes break this rule.)

To use these independent components across our app, we need to do two steps:

1. **Export** the component from the file where it lives
2. **Import** the component where it needs to be rendered

Exports can be either **default** (the main or only thing exported from that file) or **named** (potentially one of many exports from the same file).

Imports usually go at the **top** of a file. Named imports need to use curly braces.

Props

Every component has something called **props**.

A component's props is an **object**. It holds **key-value pairs passed** to that component when it is **rendered**.

If we render this component just as **<PropsDisplayer />**, it has no props and the object is empty.

We can include **any number** of attributes (props) with different names and values.

```
// Create a new file called PropsDisplayer.jsx in a components folder
function PropsDisplayer(props) {
    // convert object to string
    const stringProps = JSON.stringify(props);

    return (
        <div className="PropsDisplayer componentBox">
            <h2>Check out my props!</h2>
            <h3>{stringProps}</h3>
            {/* what happens if we try to render the object {props} ? */}
        </div>
    )
}
// export this component so we can import it elsewhere
export default PropsDisplayer
```

```
{/* Include the below in App.jsx */}
import PropsDisplayer from './components/PropsDisplayer' // up the top

{/* Renders the component with no props */}
<PropsDisplayer />

{/* Renders the component with a single prop 'myProp' */}
<PropsDisplayer myProp="first prop"/>

{/* Renders the component with multiple props - add your own! */}
<PropsDisplayer prop1="first" prop2="second" prop3={3}/>
```

Props

Almost any kind of data can be passed to a component as a prop. *Experiment with your PropsDisplayer component*.

You can pass `string`, `number`, `array`, `object`, `function`, or even another `component`, as props when rendering a component.

Individual props can be referenced from the props object using **props.<propName>**, e.g. `props.name`, `props.age`, `props.pets`

```
{/* String prop value uses quotes, numeric prop value uses curly braces
*/}
<PropsDisplayer name="Harry Styles" age={29}/>

{/* Array prop value - uses curly braces */}
<PropsDisplayer pets={["cat", "dog", "goldfish"]}/>

{/* Variable prop values - uses curly braces */}
<PropsDisplayer reactLogo={reactLogo} buttonCount={count}/>

{/* Our PropsDisplayer component won't handle stringifying other
components */}
{/* <PropsDisplayer component={<ExampleComponent />}/> - fails for this
example but the concept is still valid */}
```

```
{/* Place this inside the return block of PropsDisplayer to access and
display individual props */}
<p>Name: {props.name} is {props.age} years old</p>
```

See Passing Props to a Component for more info

Props

Because props is an **object**, we can take advantage of object **destructuring** to simplify our use of props in two handy ways:

1. **Extracting named properties** into individual variables
2. To provide **default values** for certain props.

```
// add this to a new file City.jsx in the components folder
// destructures props object into 3 variables, two with defaults
function City({name, state = 'NSW', country = 'Australia'}) {

    return (
        <div className="City componentBox">
            <strong>{name}</strong> is in {state}, {country}
        </div>
    )
}

export default City;
```

```
{/* state and country are not specified, will use defaults */}
<City name="Sydney" />

{/* country is not specified, will use default */}
<City name="Melbourne" state="VIC" />

{/* all values are specified, won't use defaults */}
<City name="Chicago" state="Illinois" country="USA" />
```

Create a new component to display an Address, using destructured props with default values where relevant.

Props - Children

Props has a special property called **children**.

This is used when **nesting** child components **inside** our component between the opening and closing tags, making it a parent.

We can include both **native browser** components and **user-defined** components as children.

Try it using this **City** example.

```
function City({name, state = 'NSW', country = 'Australia', children}) {
    // destructuring the props.children property as well

    return (
        <div className="City">
            <strong>{name}</strong> is in {state}, {country}
            {children}
        </div>
    )
}
```

```
{/* Everything in between <City> and </City> is passed as props.children
*/}
<City name="Newcastle">
    <div>Newcastle is a harbour city in the Australian state of New South
Wales.</div>
    <div><strong>Population:</strong> 322,278 (2016)</div>
</City>
```

React Basics - Props

Props are Read-only

We can **access** the props passed to a component via the props object, but we should **NEVER** change or modify (**'mutate'**) them.

Technically this is valid code:

```
function PropsDisplayer(props) {

    props.newProp = "never do this!"; // adds or modifies the newProp object property - BAD PRACTICE
    const stringProps = JSON.stringify(props); // convert object to string
```

BUT it violates a key rule of components - that they must be **'pure'** and therefore reliable. Components can modify their **own** local/internal variables, but **not** those created elsewhere. See here for more.

React Basics - Conditional Rendering

Although we **can't modify props**, we often need to check their value or **existence** and **change what we are rendering** as a result.

We can use the standard **conditional syntax** features of JavaScript to achieve this: `if` statements, ternary `?` operators or `&&`. See here for more detail.

```
function Pet(props) { // add this to a new file Pet.jsx in the components folder
    let type = 'unknown'; // set default value for type
    if (props.type) type = props.type; // override with prop type if set (could also
destructure with default)

    return (
        <div className="Pet componentBox">
            <h2>My Pet</h2> <p>Type: {type}</p>

            {/* if props.name is truthy (not undefined), render it in a <p> tag,
otherwise render nothing */}
            {props.name ? <p>Name: {props.name}</p> : null}

            {/* Only render the <p> tag if the expression before the && is truthy */}
            {props.colour && <p>Colour: {props.colour}</p>}
        </div>
    )
}
export default Pet; // render this component in App.jsx. Try out different prop values,
add support for more
```

Exercise 1

- Create a **Greeting** component, in its own file, which renders the text “Hello World”.
- Import it into App.jsx, and pass in a prop called ‘name’ when rendering the Greeting component
- If the name prop exists, make sure the Greeting component replaces ‘World’ with that name value, i.e. “Hello John”.
- Include support for a greeting message via children.

React Basics - Virtual DOM

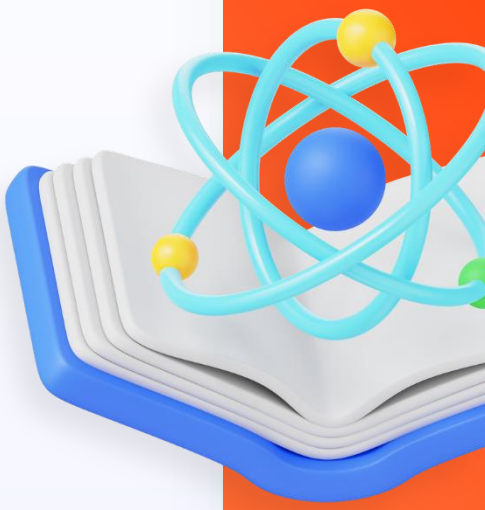
Document Object Model

Working with vanilla JS in the browser, any modifications to the HTML need to be achieved by **manipulating the DOM directly**, eg.

```
document.getElementById('main').innerHTML = '<p>new content</p>';
```

When making many updates, this quickly becomes **inefficient** and **slow**, both when **writing** code and **executing** it. To handle this issue, React creates and manages a **virtual DOM** internally.

When a change occurs requiring a DOM update, all component updates are first **collected** and made to the **virtual DOM**, almost instantaneously. Then React **compares** the new virtual DOM to the browser DOM, and efficiently updates **only** the parts that have **changed**.



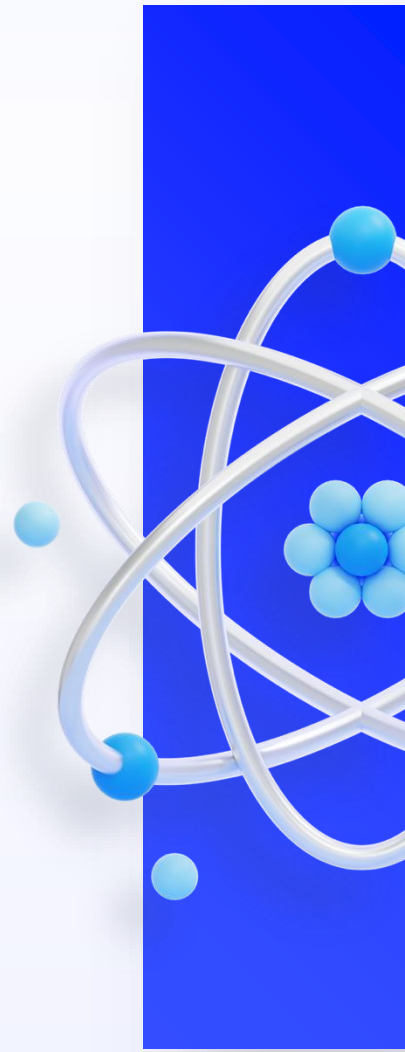
React Basics - Virtual DOM

Render method

- Syntax (main.jsx)

```
ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode> // Strict Mode does extra checks during development
    <App /> // App is App.jsx - where all components are rendered
  </React.StrictMode>,
) // this syntax can vary between builder tools and apps but achieves the same purpose
```

- First we **create a root DOM element** to display **React components** inside a browser DOM node - in this case `<div id="root"></div>` from index.html
- Next we **render** our React component tree into the DOM
- Any React elements that were **previously rendered** into the container will be **updated** and **only mutate** the DOM as **necessary** to reflect the **latest** version of each React element.

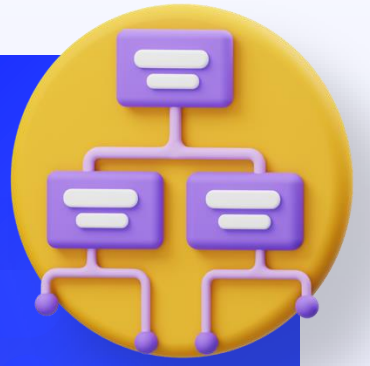


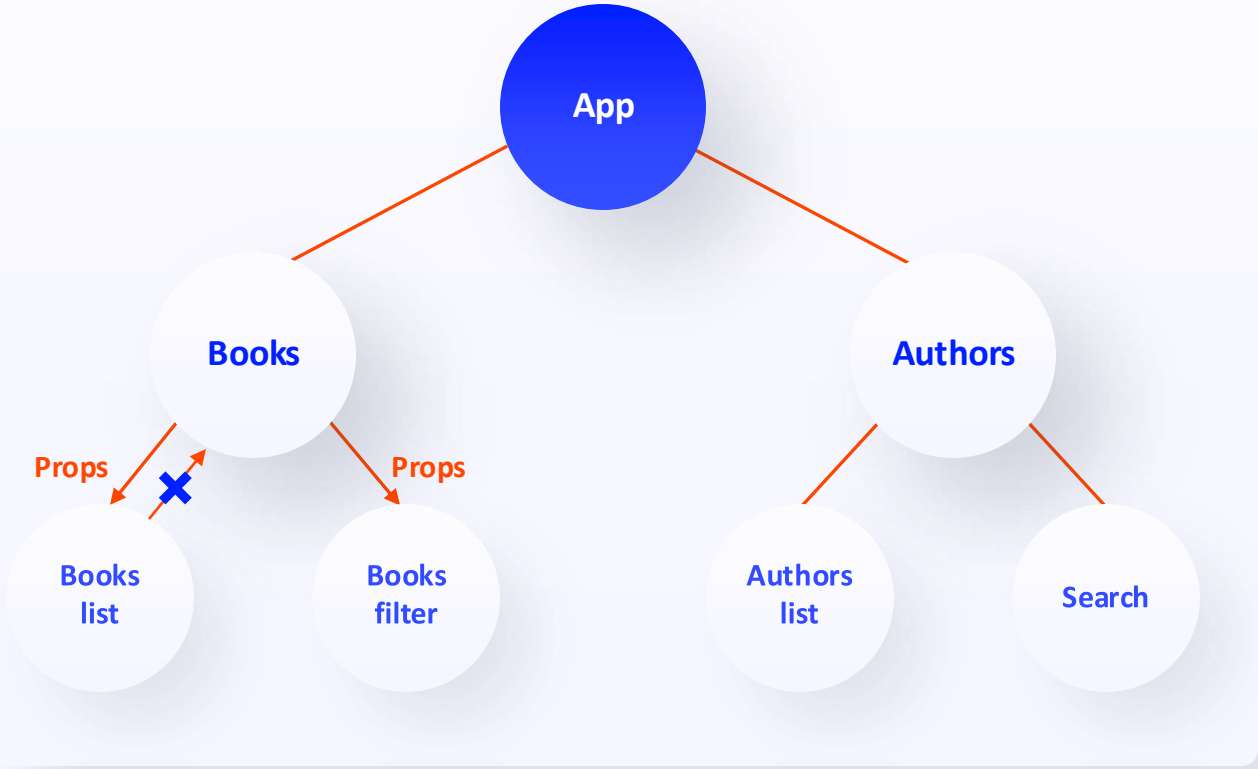
React Basics - Hierarchy

React organizes (**composes**) components into a **hierarchical tree**.

This allows it to only update **parts** of the tree which have changes. Data is passed **down** the tree, from **parent** to **child** components using **props**.

If multiple components need to use the same data, it should be **stored in a parent** and passed down as needed, via props.





All books data is managed on the **Books** component. It can be passed down as **props** to **BooksList** or **BooksFilter**, but no data can be passed in the opposite direction.

React Basics - Composition

Composing Components

When components render other components in their output, this is called **composition**. Composition allows us to build more **complex components** by combining **simpler** ones. This approach enables code **reusability** and makes managing and maintaining large applications easier.

```
function NamePart(props) {
  return ( // reusable component to display part of a name from the value prop
    <span className="NamePart">{props.value}</span>
  )
}

function FullName(props) {
  return ( // composes the NamePart component to display a full name
    <div className="FullName componentBox">
      Full name: <NamePart value={props.first} /> <NamePart value={props.last} />
    </div>
  )
}
// add the above to FullName.jsx, then export it and import into App.jsx
```

```
<FullName first="Kendrick" last="Lamar" />
```

Our App.jsx is a **top-level parent component** composing many of our examples so far.

Add the above as well, and modify the **first** and **last** props, adding support for **middle** names.

React Basics - Extracting Components

React components are designed to be **small**, **single-purpose** and **reusable**, to make the most of React's modular, scalable and performance benefits.

Often this means we need to **simplify** a **large**, **complex component** by **extracting** smaller components from it and **composing** them together.

```
function ComplexComment(props) { // complex component which displays different elements of a comment

    return (
        <div className="Comment componentBox">
            <div className="UserInfo"> {/* the user info is one aspect of the comment */}
                <img className="Avatar" src={props.author.avatarUrl} alt={props.author.name} />
                <div className="UserInfo-name">{props.author.name}</div>
            </div>
            <div className="Comment-text"> {/* the actual comment text is another aspect */}
                {props.text}
            </div>
            <div className="Comment-date"> {/* the comment date is another aspect */}
                {props.date.toLocaleString()}
            </div>
        </div>
    );
} // save in a new file ComplexComment.jsx, then export it and import into App.jsx
```

React Basics - Extracting Components

When the complex component is rendered, at that **top level** it will make no difference to the generated HTML whether or not it has been broken down into smaller extracted components **under the hood**.

The below code (add to **App.jsx**) should work both prior to extraction and afterwards (just replace **ComplexComment** with **Comment**):

```
// object storing comment data - passed as props
const comment = {
    date: new Date(),
    text: 'I hope you enjoy learning React!',
    author: { // author is also an object
        name: 'Hello Kitty',
        avatarUrl: 'https://placekitten.com/g/64/64',
    },
};
```

```
{/* render the component, passing object data as props */}
<ComplexComment author={comment.author}
    date={comment.date}
    text={comment.text}/>
```

React Basics - Extracting Components

To simplify a complex component, we first need to **identify** related chunks or **sections** of JSX that would logically make sense to exist as individual, extracted components. We then need to **extract** those sections into their own named components, and finally **pass required data** down to them as **props**.

For our comment example, a simplified version may look like this:

```
// simpler Comment component with the user info section
extracted out into its own component
function Comment(props) {

    return (
        <div className="Comment componentBox">
            <UserInfo user={props.author} /> {/* here we pass
the author prop down to the UserInfo component */}
            <div className="Comment-text">
                {props.text}
            </div>
            <div className="Comment-date">
                {props.date.toLocaleString()}
            </div>
        </div>
    );
}
```

React Basics - Extracting Components

Class Exercise

Try to create the extracted **UserInfo** component to make the code on the previous slide run correctly. Make sure it displays data from the **user** prop.

Also try extracting an **Avatar** component from **UserInfo**, and a **FormattedDate** component from **Comment**. Make sure that the end result in the browser is the same as the original, but now you have simple, easy-to-read, compositional component code.

(sample answers/hints are available here if you need them, but try it first)

React Basics

Browser elements are often **nested** inside one another, eg.

```
<div><h1>Child Heading</h1></div>
<!-- h1 is a child of div, div composes h1 -->
```

React can do this with **custom components** as well, with a special prop called **children**, used inside the **parent**. This is a flexible pattern that can be used to **nest components**, often for achieving different forms of **layout** or common styling.

Try nesting other components in FancyBox, and making a FancyBorder!

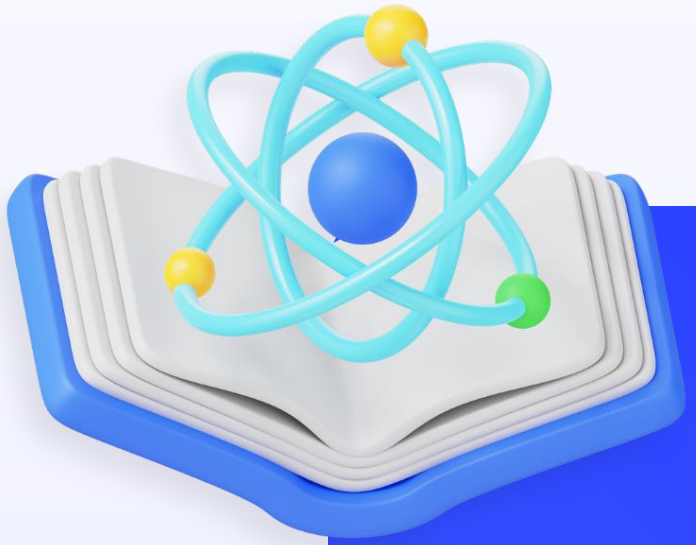
```
function FancyBox(props) {
    return (
        <div className={'FancyBox FancyBox-' + props.color}>
            {props.children} {/* everything in between the opening
           <FancyBox> and closing </FancyBox> tags */}
        </div>
    );
}

function Callout(props) {
  return (
    <FancyBox color="blue">
          <h1 className="Callout-title">{props.title}</h1>
          <p className="Callout-message">{props.message}</p>
          {props.children} {/* everything in between the opening
           <Callout> and closing </Callout> tags */}
    </FancyBox>
  );
}

// render the Callout component with FullName as children
<Callout title="Nested React Component"
    message="Simple message with a fancy box applied via composition">
    <FullName first="Elon" last="Musk" />
</Callout>

// sample CSS to make things fancy
.FancyBox { --mask: linear-gradient(#000 0 0) 50%/calc(100% - 37.00px)
calc(100% - 37.00px) no-repeat,
  radial-gradient(farthest-side,#000 98%,#0000) 0 0/2em 2em round; mask:
var(--mask); -webkit-mask: var(--mask); padding: 2em;
  background: linear-gradient(135deg, #1f005c, #5b0060, #870160,
#ac255e, #ca485c, #ffb56b) fixed; color: white; }
.FancyBox-blue { background: linear-gradient(135deg, #140061, #50006E,
#64018C, #6955FF, #4684FF, #1FA7FF) fixed; }
```

React Basics - Lists

Very often in React we need to display a **list of items**, such as products, users, posts, comments, etc, from a **collection** of data.

The usual process for this involves creating a **component** (or template) for displaying each item, and then **iteratively rendering** that component with **different data for each item**.

We can use the built-in JS **Array** method **map** to achieve this, since it **iterates** over list items and **transforms** the raw data from each one into a populated component.



React Basics - Lists

In this example we have a **collection** (array) of **movies**, represented as simple objects.

The JSX for our **MoviesList** component renders an unordered list element, and then uses **map** to create a list item for each movie title.

Since this is a JS expression, we evaluate it inside **curly braces**.

```
function MoviesList() {
  // collection of objects representing movies
  const movies = [
    {
      title: "The Shawshank Redemption",
      year: 1994,
      synopsis: "Two imprisoned men find redemption.",
    },
    {
      title: "The Dark Knight",
      year: 2008,
      synopsis: "Batman fights the menace known as the Joker.",
    },
    {
      title: "Interstellar",
      year: 2014,
      synopsis: "Explorers travel through a wormhole in space.",
    },
  ];

  return (
    <div className="MoviesList componentBox">
      <ul> {/* iterate over each movie, print the title in a list
*/}
          { movies.map(movie => (
            <li>{movie.title}</li>
          )) }
      </ul>
    </div>
  )
}

export default MoviesList;
```

React Basics - Lists

It is also common practice to create a **variable** storing the JSX nodes returned from **map**, and then to render that variable.

This helps to keep the return block tidy and manageable.

*Try rendering the **year** and **synopsis** of each movie in the list item as well as the **title**.*

```
const movieItems = movies.map(movie => (
    <li>{movie.title}</li>
))

return (
  <div className="MoviesList">
    <ul>{ movieItems }</ul>
  </div>
)
```

NOTE: The above code works because the arrow function inside .map implicitly returns everything after the arrow.

The round brackets allow us to format the JSX more neatly over multiple lines, but it's still a single statement.

If we instead use curly braces, we need to explicitly include a return statement as it's no longer implied. Try it and see!

React Basics - Lists

The **MoviesList** example currently displays a warning in the browser console:

```
Warning: Each child in a list should have a unique "key" prop.

Check the render method of `MoviesList`. See https://reactjs.org/link/warning-keys for more information.
```

An important **rule of lists** in React is that each one needs a **unique field** included in its data, to help React know which **array item** each **JSX node** corresponds to.

```
// collection of objects representing movies
// now modified to include unique IDs
const movies = [
    {
        id: 1, // items in data collections need unique IDs
        title: "The Shawshank Redemption",
        year: 1994,
        synopsis: "Two imprisoned men find redemption.",
    },
    {
        id: 2, // unique ID
        title: "The Dark Knight",
        year: 2008,
        synopsis: "Batman fights the menace known as the Joker.",
    },
    {
        id: 3, // unique ID
        title: "Interstellar",
        year: 2014,
        synopsis: "Explorers travel through a wormhole in space.",
    },
];
```

React Basics - Lists & Keys

```
const movieItems = movies.map(movie => (
        <li key={movie.id}>{movie.title}</li> // key prop is required for lists
    ))
```

Key is a special **prop** only used for rendering **individual items in a collection**. It helps React to **track each list item uniquely**, especially when they are later sorted, filtered, added or removed. This is important for **performance** and **reliability** to ensure only **modified** items are updated in the DOM.

Rules of keys:

- Ensure keys are **unique** among siblings (different lists can re-use keys)
- Avoid generating or changing keys during rendering - they should be **stable**.

React Basics - Lists

More commonly, we would create a **separate component** for displaying each item in a collection.

We still need to use a **key** prop at the **top level** inside the **.map** function. Previously this was the native **<li>** component, now it is the custom **<Movie>** component.

Since **key** is a **special prop**, it is not accessible inside **Movie** (try it!). If we need to use or render the **id**, we have to pass it as a separate **id** prop.

```
const movieItems = movies.map(movie => (
    <Movie
        key={movie.id} // key prop is required for lists
        title={movie.title}
        year={movie.year}
        synopsis={movie.synopsis}/>
      // can also destructure movie into individual props
      // <Movie key={movie.id} {...movie}/>
    )
);

// separate component for displaying each movie
function Movie({title, year, synopsis}) {
    return (
        {/* no key prop here - only at top level inside .map
*/}
        <li>
            <h3>{title}</h3> <span>({year})</span>
            <div>{synopsis}</div>
        </li>
    )
}
```

See Rendering Lists for more information and challenges.

Exercise 2

- Create a **BigCats** component, in its own file, which uses the cats array (below) to display a styled list of cats.
- Include a unique **id** and fix the warning about keys
- Create a **SingleCat** component for rendering each individual cat, and add an image property for each one.

```
const cats = [
  { name: 'Cheetah', latinName: 'Acinonyx jubatus' },
  { name: 'Cougar', latinName: 'Puma concolor' },
  { name: 'Jaguar', latinName: 'Panthera onca' },
  { name: 'Leopard', latinName: 'Panthera pardus' },
  { name: 'Lion', latinName: 'Panthera leo' },
  { name: 'Snow leopard', latinName: 'Panthera uncia' },
  { name: 'Tiger', latinName: 'Panthera tigris' },
]
```

React Basics - State

State

React components will often need ***dynamic information*** in order to render. For example, a component that displays the total of a shopping cart will need to **change** whenever a product is added or removed from the cart.

Our component needs to **remember** the prices and quantities of each product in the cart to render properly, and it needs to **re-render** whenever that data changes. *We can't store this data in standard variables, because they **aren't** remembered between renders, and updating them **won't** re-render the component and show the new value on screen*.

In React, component-specific data that changes over time is called **state**.

React Basics - State

State

A React component can access dynamic information in two ways: **props** and **state**.

Unlike props, which are passed into a component, a component's state is defined **inside itself**.

To make a component have state, we use the **useState** hook.

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

// save in MoodChanger.jsx
function MoodChanger() {
    const [mood, setMood] = useState('happy');

    return (
        <div className="MoodChanger componentBox">
            Current Mood: {mood}
        </div>
    )
}

export default MoodChanger;
```

React Basics - State

Hooks like **useState** allow us to 'hook' into built-in features of React, such as **state management**.

useState is a **function** that returns **two values**:

- The first is the **value to be stored in state** - a **dynamic** value that will change over time
- The second is a **function** we can call to **update** the first value.
- We use the **value**, **setvalue** naming convention for these two variables.

```
// useState is a React hook
import { useState } from "react";

// save in MoodChanger.jsx
function MoodChanger() {
          // two variables :
     // mood stores current mood, default happy
     // setMood is a function for updating mood
    const [mood, setMood] = useState('happy');

    return (
        <div className="MoodChanger componentBox">
            Current Mood: {mood}
        </div>
    )
}

export default MoodChanger;
```

React Basics - State

Updating State

So far our state is **not dynamic** because the mood value never changes.

If we add buttons, we can use **setMood** to **update** the mood in response to a **button click**, making it dynamic.

The value passed to **setMood** will become the **new value** of **mood**.

Calling **setMood** will trigger a **re-render** of the component so that the **new value** of **mood** will be displayed.

```
Current Mood: {mood}

{/* Change the mood state by calling setMood when a
button is clicked */}
<button
    onClick={() => setMood('tired')}>
    Stay Up Late
</button>

<button
    onClick={() => setMood('hungry')}>
    Skip Lunch
</button>

{/* ++ Add these buttons and some of your own to
MoodChanger and try it out! */}
```

React Basics - State

State management in React allows us to do **two important things**, represented by the two variables returned from the useState hook:

1. **Remember** the values needed to display each component at any point in time - a **snapshot** of that component.
2. **Update** these values - primarily through user interaction.

These two are closely linked. Whenever the **snapshot** for a component (all of its dynamic state values) **changes**, the component needs to **re-render** to reflect this, and **remember** the new snapshot. This is a **three-step process**:

1. A change in state is **triggered** by some event, eg. a mouse click
2. All state changes are bundled and **rendered** in React's virtual DOM
3. React compares its DOM to the actual DOM and **updates** as needed.

React Basics - Component Lifecycle

There are 3 main stages in the lifecycle of a component.

1. ***Mounting***: Happens **once**, when loading the component for the first time. Also called the **'initial render'**, when the component is created and first inserted into the DOM.

2. ***Updating***: Results from **changed state** or **changed props**. This **'re-rendering'** usually happens multiple times, whenever a change occurs.

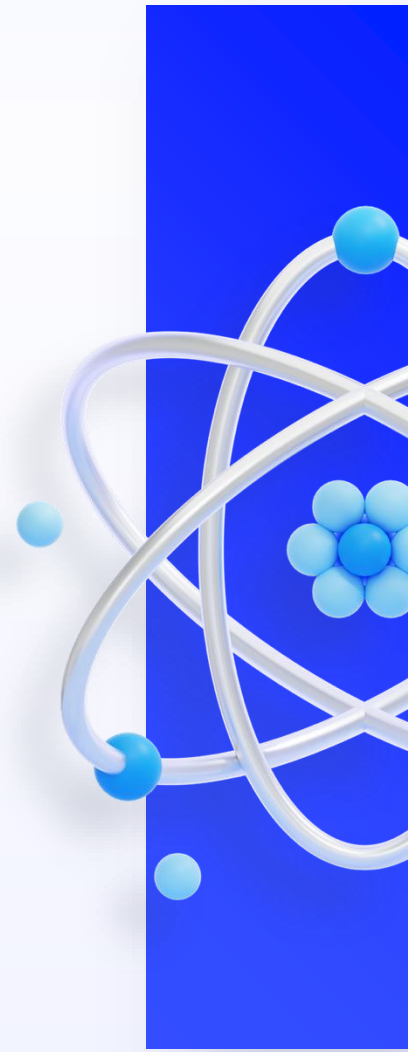
3. ***Unmounting***: **Removing** the component from the DOM, either when the entire DOM is removed/refreshed, or when a condition to prevent the particular component from rendering is satisfied.

React Basics - State

Event Handlers

State can change in response to many different **actions**, often triggered by the **user** executing an **event**, such as clicking on a **button**, entering data in a **form field**, clicking a **link**, choosing a value from a **drop-down**, etc.

Event handlers in React are **camel-cased props** (eg. **onClick**) that specify a **function** to be called when the **event** is triggered. In the previous example we used arrow functions for this, but we can also define our own.



***IMPORTANT**: Functions passed to event handlers must be **passed**, not **called**.* Calling the function like:

```
<button onClick={setMood('tired')}>Stay Up Late</button>
```

executes it immediately on **rendering**, instead of on the **click** action! Try it, but be warned!

React Basics - State

Event Handlers

If the event handler function is very short and **simple**, it is common to use an **arrow function**.

More typically, we define custom event handler functions separately, **inside** the component, with a name beginning with **'handle'**. These functions may perform validation or test for conditions, etc. as well as **updating state**.

```
function MoodChanger() {
  const [mood, setMood] = useState('happy');

  // Calls setMood with a fixed value of 'ecstatic'
  // begin with 'handle' prefix by convention
  const handleWinLotto = () => {
    setMood('ecstatic')
  }

  // Calls setMood with a conditional state value
  const handleRunningLate = () => {
    let newMood = 'stressed';
    if (mood === 'stressed') newMood = 'really stressed'
    else if (mood === 'really stressed') newMood = 'giving up';

    setMood(newMood)
  }

  return (
    <div className="MoodChanger">
       Current Mood: {mood}
       {/* Using arrow functions */}
       <button onClick={() => setMood('tired')}>Stay Up
Late</button>

       {/* Using custom event handler functions */}
       <button onClick={handleRunningLate}>Running Late</button>
       <button onClick={handleWinLotto}>Win Lotto</button>
    </div>
  )
}
// ++ add your own buttons to handle different moods!
```

React Basics - State

Updating State

There are a few tips to remember when using a state updater function from the **useState** hook to change the value of a state variable:

- Updating state **triggers a re-render of the component**, so it can display the new state value. **Avoid** doing this inside the **return block** of your component, or it will **re-render infinitely** and crash your browser.

- State updates **don't happen immediately**. They are **asynchronous** so they can be efficiently bundled (rendered) together. So this code **won't** log the new value of **mood**:

```
const handleWinLotto = () => {
    setMood('ecstatic')
    console.log(mood) // will be the previous/old value, not 'ecstatic'
    // since the re-render has been triggered but not yet completed
}
```

React Basics - Strict Mode

By default in most build tools and during development phase for React projects, **Strict Mode** is enabled. This is a built-in feature of React, designed to help developers **find common bugs** in their components **early in development**.

Take a look in main.jsx and note how the App component is rendered using <StrictMode>.

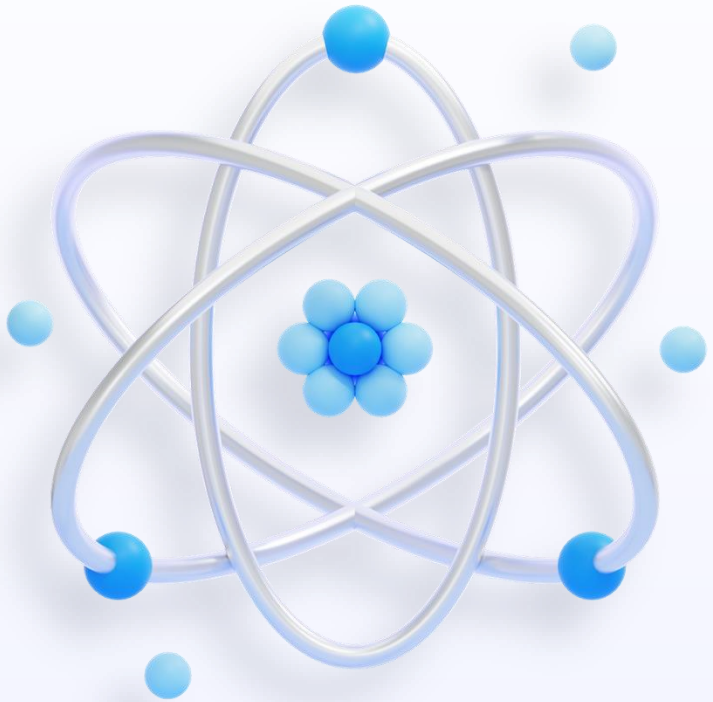
Strict Mode enables the following **development-only component behaviors**:

- **One extra re-render** to find bugs caused by impure rendering.
- **One extra effect execution** to find bugs caused by missing effect cleanup.
- **Checking** for usage of deprecated APIs.

Strict mode may cause debug logging to output more often than you expect, and will give extra and more complete warning messages.

Exercise 3

- Create an **Emoji** component, in its own file, which initially renders a happy emoji.
- Add a ‘Change Mood’ button in the component to switch between 2 emojis when clicked.



React Basics - State

Multiple State Values

So far we have only used a **single** state value **(mood)**. But components can (and often do!) have **multiple** dynamic values to remember and manage in state. There is **no limit** to the **amount** or the **type** of state values, **but** there are some things to keep in mind:

- **Unrelated** state values should be kept **separate** so that they can be updated and remembered independently
- **Related** state values may be better stored **together**, in an object or an array, to streamline updates and simplify variables.

React Basics - State

This example uses **two state variables**: one for the **current language**, and one for the **corresponding phrase**.

Technically this works fine, and is not a bad solution.

However, since both values **always change together** and need to stay in sync, it may be more reliable to combine them into a single state value.

```
const phrases = new Map([
    ['english', 'Happy Birthday'],
    ['german', 'Alles Gute zum Geburtstag'],
    ['spanish', 'Feliz Cumpleaños']
]);

function BirthdayTranslator() {

    const [currentLanguage, setCurrentLanguage] =
useState('english');
    const [phrase, setPhrase] = useState(phrases.get('english'))

    const handleChangeLanguage = (lang) => {
        setCurrentLanguage(lang);
        setPhrase(phrases.get(lang));
    }

    return (
        <div className="BirthdayTranslator componentBox">
            <h3>{phrase}! ({currentLanguage})</h3>
            <button onClick={() =>
                  handleChangeLanguage('english')}>English</button
            >
            <button onClick={() =>
                  handleChangeLanguage('german')}>German</button>
            <button onClick={() =>
                  handleChangeLanguage('spanish')}>Spanish</button
            >
        </div>
    )
}

// Add this to BirthdayTranslator.jsx and render it in App.jsx
// ++ Add support for another language!
```

React Basics - State

We could combine both states into a single object as shown.

Technically both approaches work, but unifying state can help to make sure that **multiple related things are always updated together**.

This is especially useful if the amount of state values to be stored is **unknown** (dependent on user input) or when submitting a complex form.

```
const phrases = new Map([
    ['english', 'Happy Birthday'],
    ['german', 'Alles Gute zum Geburtstag'],
    ['spanish', 'Feliz Cumpleaños']
]);

function BirthdayTranslator() {

    const [currentPhrase, setCurrentPhrase] = useState(
        {lang: 'english', phrase: 'Happy Birthday'}
    )

    const handleChangeLanguage = (newlang) => {
       setCurrentPhrase({lang: newlang, phrase:
phrases.get(newlang)})
    }

    return (
        <div className="BirthdayTranslator componentBox">
            <h3>{currentPhrase.phrase}! ({currentPhrase.lang})</h3>
            <button onClick={() =>
                  handleChangeLanguage('english')}>English</button>
            <button onClick={() =>
                  handleChangeLanguage('german')}>German</button>
            <button onClick={() =>
                  handleChangeLanguage('spanish')}>Spanish</button>
        </div>
    )
}

// ++ Try generating the buttons from the phrases Map dynamically
// instead of hardcoding them, then adding support for more
languages
```

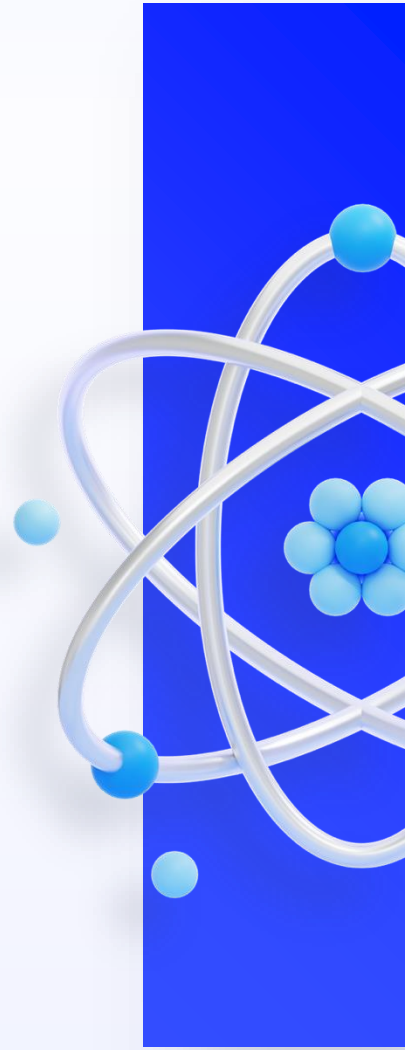
React Basics - State

Simplifying State (see here for details)

When identifying state for your components, try to keep things **simple**:

1. Consider **merging** separate state values that need to stay in sync.
2. Try to make sure different state values can't **contradict** each other.
3. Avoid **redundant** state that can be calculated from other values.
4. Don't **duplicate** information stored in state - this can lead to redundancy and contradiction.
5. Don't store **deeply nested** hierarchical data in state - flatten it instead to simplify updates.

These rules will make more sense the more you work with stateful components. It takes some experience to recognise and avoid the pitfalls, so revisit this list.



React Basics - State

Applying rule 3 (**avoid redundancy**) from the previous slide can **simplify** our component even further.

The birthday phrase being displayed can be **calculated** from the current language, so we **don't** need to store it **separately** in state.

This avoids redundancy, simplifies the code, improves performance, and reduces chances for errors.

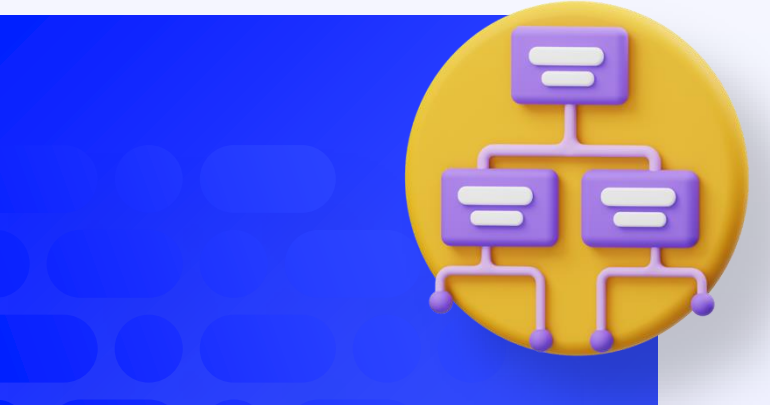
```
function BirthdayTranslator() {

    const [currentLanguage, setCurrentLanguage] =
useState('english');

    const handleChangeLanguage = (newlang) => {
        setCurrentLanguage(newlang)
    }

    return (
        <div className="BirthdayTranslator componentBox">
            <h3>{phrases.get(currentLanguage)}!
                ({currentLanguage})</h3>
            <button onClick={() =>
                handleChangeLanguage('english')}>English</button>
            <button onClick={() =>
                handleChangeLanguage('german')}>German</button>
            <button onClick={() =>
                handleChangeLanguage('spanish')}>Spanish</button>
        </div>
    )
}

// It is common to adjust your state values during development
// as you gain experience and grow the complexity of your code.
```

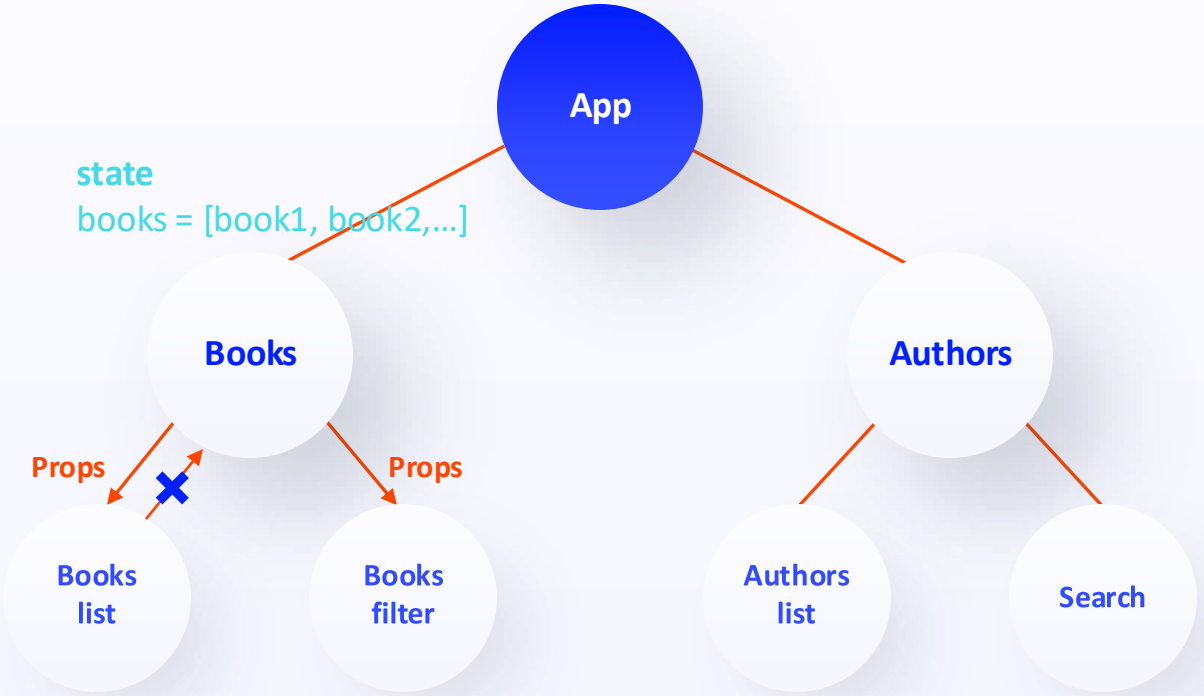
React Basics - Data Flow

Since data in React always **flows down** the component hierarchy, state data follows these same rules.

We can **pass** state values from a **parent** component to one or multiple **child** components as **props**, but we **can't** pass data back up from child to parent components.

Since components should be small, single-purpose and reusable, we often need to share data between them.





The state *books* are kept track on the **Books** component. It can be passed as **props** to **Books list**, but nothing can be passed in the opposite direction.

React Data Flow

Lifting State Up

To **share state** among **multiple** components, we need to define it in a **common parent** component and pass it **down as props** to all children.

This is a common practice in React called **lifting state up**.

In this example the parent **Weather** component **defines** the state and **passes** it to two child components to be displayed (in **Temperature**) and **updated** (in **CheckWeather**).

```
// Parent component storing common state data
function Weather() { // copy to Weather.jsx

    // two separate state values to store weather data
    const [weather, setWeather] = useState('sunny')
    const [tempCelcius, setTempCelcius] = useState(27)

    // handler function to update both state values at once
    const handleWeatherChange = (newWeather, newTemp) => {
        setWeather(newWeather)
        setTempCelcius(newTemp)
    }

    return (
       <div className='Weather componentBox'>
           <h2>Today's Weather</h2>
           <div>
               <strong>{weather}</strong> with a temp of
               {/* Child component to display temp -
                   needs temp value as prop */}
               <Temperature temp={tempCelcius} units="C" />
           </div>
           {/* Child component to update the weather -
               needs handler function as prop */}
           <CheckWeather
onWeatherChange={handleWeatherChange} />
       </div>
    )
}
export default Weather;
```

React Data Flow

Lifting State Up

Our two child components can access the **parent state values** via **props**. They don't store any local state of their own, which makes them **controlled components**.

Since props can be **anything**, we can pass down **both** the **data** from the state and any **functions** for updating that data.

In this way, **child components can access and update parent state**.

```
// Child component - receives parent state handler via props
function CheckWeather(props) {
   const weatherTypes = ['sunny', 'windy', 'raining', 'cloudy'];

   // generates new random weather data and updates state via prop
   const randomWeather = () => {
      let newTemp = Math.floor(Math.random() * 40);
      let newWeatherIndex = Math.floor(
                              Math.random() *
weatherTypes.length);
      // ++ try to destructure this function from the props object
      props.onWeatherChange(weatherTypes[newWeatherIndex],
newTemp)
   }

   return (
      <button onClick={randomWeather}>Check Weather</button>
   )
}
// ++ Add some more weather types of your own

// Child component to display and convert temp if needed
function Temperature({temp, units = 'C'}) { // default to celcius

   // convert to Fahrenheit if units is F (or not C)
   let displayTemp = units === 'C' ? temp : (temp * 9/5) + 32

   return (
      <span class="Temperature">
         <strong> {parseInt(displayTemp)}&deg;{units} </strong>
      </span>
   )
}
// ++ Try adding a button to convert between C and F temps
```

React Basics - State

Lifting State Up (see here for details)

To **share** state data among **multiple** components, we need to:

- **Identify** a common **parent** component
- **Remove local state** from the child component/s
- **Lift** this **state** data **up** into the parent component instead
- **Pass** the relevant **state** data **down** to each child component as **props**.

Standalone components that manage their own state are referred to as **uncontrolled components**, whereas components reliant on a parent to pass down state as props are called **controlled components**. There is often some overlap between the two.

React Basics - State

Our previous **MoviesList** component doesn't manage any state as the data doesn't change.

Adding a **button** to **reverse the order** means the data being rendered will **change in response to user input**, so we need to manage it using **state**.

We need to treat any arrays (or objects) stored in state as **read-only**.

```
function MoviesList() {

    const [currentMovies, setCurrentMovies] =
useState(movies);

    const movieItems = currentMovies.map(movie => (
        <Movie key={movie.id} title={movie.title}
               year={movie.year} synopsis={movie.synopsis}/>
    ))

    const handleReverseMovies = () => {
        // first clone the original, so we don't mutate it
        let newMovies = [...currentMovies];
        newMovies.reverse(); // 2. modify the clone
        setCurrentMovies(newMovies); // 3. set updated clone
in state
    }

    return (
       <div className="MoviesList">
           <ul>
               { movieItems }
           </ul>
           <button onClick={handleReverseMovies}>Reverse
List</button>
       </div>
    )
}

// ++ Try adding buttons to sort by year and by title
// ++ Try extracting a SortButton component to replace the
above
```

React Basics - State

State is Immutable (see here for details)

Although objects and arrays in JS are **mutable** (can be modified), **when storing them in state** we need to treat them as **immutable**. To make updates, we need to:

1. First, **clone** the original object or array (usually using the spread operator …)
2. Then **make changes** to the **clone** (push, sort, modify properties, etc)
3. Finally, **set** the updated clone as the new state value.

Updating a single object property directly modifies the original object. Instead, we need to clone it and replace the affected property:

```
// BAD WAY - mutates original object/array
movie.title = 'New Title';
movies[0].title = 'New Title';
```

```
// GOOD WAY - read-only originals
let newMovie = {...movie, title = 'New Title'};
let newMovies = movies.map((movie, i) => i === 0 ? {...movie,
title: 'New Title'} : movie);
```

React Basics - State

State is Immutable (see here for details)

Array functions like **filter** and **map**, which return a new array, are fine as they **don't mutate the original**. Functions like **push**, **pop**, **sort**, **reverse**, etc **mutate** the original array and mean we need to instead follow the **cloning first** process to **avoid mutation** of our state array values.

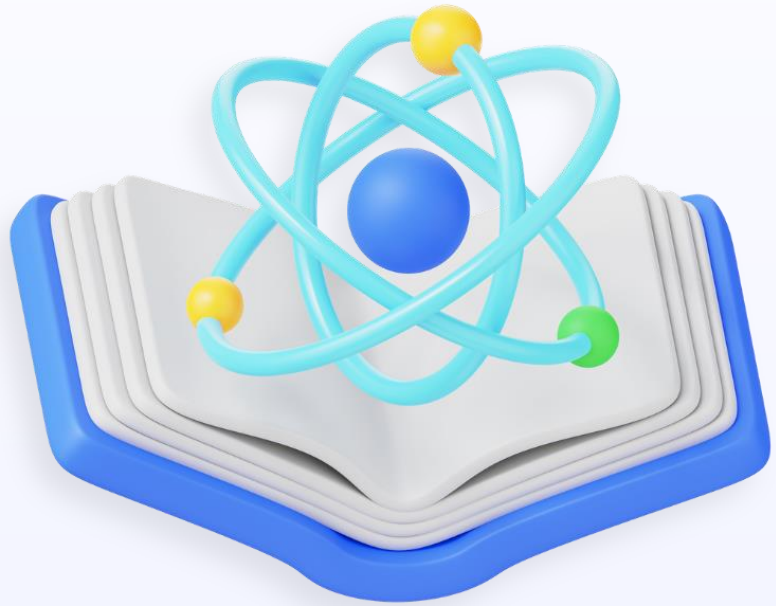
Array operation	Avoid - mutates the original	Instead - modify a copy
adding	push, unshift	concat, [...arr, newItem]
removing	pop, shift, splice	filter, slice
replacing	splice, arr[i] = (assignment)	map
sorting	reverse, sort	clone first

React Basics - State

```
// add a new item to an array or object - RIGHT WAY using spread to clone original then adding in new item
 let newMovies = [...currentMovies, {
     id: 4, title: "The Whale", year: 2022,
     synopsis: "A morbidly obese teacher attempts to reconnect with his daughter.",
   }]

 // remove an item from an array - RIGHT WAY using filter to create a new array without the removed item
 let idToDelete = 2;
 let newMovies = currentMovies.filter(movie => movie.id != idToDelete);

 // replace/update an item in an array - RIGHT WAY using map to create a new array including the updated item
 let updatedSynopsis1 = 'Iconic heart-warming prison break movie';
 let newMovies = currentMovies.map(movie => movie.id === 1 ? {...movie, synopsis: updatedSynopsis1} : movie);

 // sort/reverse an array - RIGHT WAY cloning the original first
 let newMovies = [...currentMovies];
 newMovies.reverse();
 newMovies.sort();
```

Exercise 4

- Extend your **BigCats** component from Exercise 2 to include buttons for alphabetically sorting and reversing the list
- Add another button to filter the list and display only the cats in the ‘Panthera’ family, and one to reset and show the full list.



React Basics - Forms

Built-in form components such as **<input>** and **<select>** let us render and control form elements in React.

Controlled form elements use an **onChange** event handler with a value prop to keep their current **value** in sync with state at all times.

As the user types, the **value** of the input field (accessed via the **event** commonly named **e**) is **synchronously** (real-time) reflected in the state.

```
function LoginForm() {
  // input state values always need to be strings - empty initially
  const [userEmail, setUserEmail] = useState('');
  const [userPassword, setUserPassword] = useState('');

  return (
    <div className="LoginForm componentBox">
      <div className="formRow">
        <label>Email Address:
          {/* Controlled form element needs both value and onChange.
           onChange handler uses event param e to access target
value.
           Whenever user types, new value is stored in state. */}
          <input type="email" value={userEmail} name="userEmail"
            onChange={(e) => setUserEmail(e.target.value)} />
        </label>
      </div>
      <div className="formRow">
        <label>Password:
          <input type="password" value={userPassword} name="password"
            onChange={(e) => setUserPassword(e.target.value)} />
        </label>
      </div>
    </div>
  )
}
// try removing the onChange prop and typing in a field
export default LoginForm
```

React Basics - Forms

Controlled form elements

Passing a **value** prop to a form element (or **checked** for a radio/checkbox) makes it a **controlled** form element. It means that **React** controls the value, not the user.

To enable the user to still **interact** with a controlled element, we need to include an **onChange** event handler, which fires whenever the input's value is **changed** by the user, and updates this new value into the **state** stored by React.

Controlled form elements are **commonly used** as they have several advantages:

1. **Data flow control**: they provide a more predictable and controlled way of managing form data with React state, making it easier to track, reset and manipulate the data.
2. **Validation and error handling**: validation logic and error handling is easier because the form data is directly controlled via the code within the component.

React Basics - Forms

Uncontrolled form elements

Controlled components trigger **re-renders** on every input change, which can be costly in terms of performance, especially if the form has many inputs.

Uncontrolled components natively have better performance as they rely on the DOM for data management. Instead of **tracking each change** as it is made by the user, we wait until the form is **submitted** and capture all data together, usually via the built-in **FormData** object.

Instead of using **value** and **onChange**, uncontrolled form components can use **defaultValue** (or **defaultChecked**) to specify the initial value. Any changes made by the user after the default value is displayed are **not tracked**.

`<input type=”file”/>` for uploading files is always an uncontrolled component.

React Basics - Forms

Form tips

- Provide a **label** for each input - important for useability and accessibility
- Provide an **initial value** for each input - either with **value** or **defaultValue**
- Provide a **name** for each input - helps when processing form data together
- Use a `<form>` element with an **onSubmit** handler function to process the form values when the form is submitted
- Any **button** inside `<form>` will submit the form, unless you give it `type="button"`

See <input> component for more information.

React Basics - Forms

Since our **LoginForm** form elements are **controlled components**, values entered by the user are **immediately set** into the matching state variables.

When we add a **submit handler** function, we can use these variables to perform some basic **validation** and display a **message** to the user.

Try to extend this example by:

- adding another validation condition
- adding a limited amount of incorrect password attempts
- hiding the form if this limit is exceeded OR login is successful.

```
// new state value for showing submission messages to user
 const [submitResult, setSubmitResult] = useState('');

 const handleSubmit = (e) => {
   e.preventDefault(); // prevent page reloading on form submit

   // add some password validation
   if (userPassword.length < 5) {
     setSubmitResult('Password must be at least 5 characters long');
   } else if (userPassword === userEmail) {
     setSubmitResult('Password must not match email address');
   } else {
     setSubmitResult('Successful login.');
   }
 }

   return (
       <div className="LoginForm componentBox">
           <form onSubmit={handleSubmit}>
               {/* same form code as previously, BUT now includes
                   <form> and <button> */}
               <button>Log In</button>
               <p>{submitResult}</p>
           </form>
       </div>
   )
}
```

React Basics - Forms

We can add a form to provide extra features and **interactivity** to our existing components such as **MoviesList**.

The **individual movie fields title**, year and synopsis are stored **locally** in the **AddMovieForm** component state, which has a **function** passed down as a **prop** for adding the new movie details to the list in the **parent MoviesList** component when the form is submitted.

Try using uncontrolled components instead!

```
function AddMovieForm({onAddMovie}) {
    const [title, setTitle] = useState('')
    const [year, setYear] = useState('')
    // ++ add support for the synopsis field as well, here and below

    const handleSubmit = (e) => {
        e.preventDefault();
        onAddMovie({title, year})
    }
    return (
        <div className="AddMovieForm componentBox">
            <form onSubmit={handleSubmit}>
                <label>Movie Title:
                    <input name="title" value={title}
                       onChange={(e) => setTitle(e.target.value)} />
                </label>
                <label>Year Released:
                    <input name="year" type="number" value={year}
                       onChange={(e) => setYear(e.target.value)} />
                </label>
                <button>Add Movie</button>
            </form>
        </div>
    )
}

// add this in MoviesList component
const handleAddMovie = (newMovie) => {
    newMovie.id = currentMovies.length + 1; // unreliable but
succinct
    setCurrentMovies([...currentMovies, newMovie])
}
<AddMovieForm onAddMovie={handleAddMovie}/>
```

React Basics - Forms

Form submission

Submitting a form using a handler function, like `<form onSubmit={handleSubmit}>` passes the **submit event** as a parameter to the **handleSubmit** function. We usually capture this as **e** and use it to **prevent** the **default** browser behaviour of **refreshing** the page, with `e.preventDefault();` so that React state is maintained and we can process the form.

Using **uncontrolled** components, we need to capture all form field values from the **form event**, usually using the built-in **FormData** object:

```
const handleSubmit = (e) => {
    e.preventDefault();
    // Creates key-value pair object with form input names/values
    const data = new FormData(e.target);
    onAddMovie(Object.fromEntries(data));
}
```

Exercise 5

- Create a new **AddCatForm** component that renders a form with controlled components to capture name, **latinName** and image details for a new Big Cat (extending Exercises 2 and 4)
- Submitting the form should update the parent **BigCats** component and re-render the list
- Add a Delete link next to each cat allowing it to be removed from the list.

Error Handling

Error handling in vanilla JS usually involves **try … catch** blocks or **asynchronous** equivalents.

We can still use these within **individual** React components, but they **won't** handle errors across the **entire** nested component **hierarchy**.

Instead we can use an npm package called React Error Boundary.

There are several ways to use it. This example uses a custom **ErrorMessage** component which will be rendered **instead** of a white error screen of death.

```
// add to main.jsx or App.jsx wrapped around top components
// to catch errors thrown by ANY children and render a fallback
// component instead of an error

// first do npm install react-error-boundary
import { ErrorBoundary } from 'react-error-boundary'

<ErrorBoundary FallbackComponent={ErrorMessage}>
  <App /> {/* can wrap App or other high-level parent components */}
</ErrorBoundary>


// add to ErrorMessage.jsx
function ErrorMessage({ error, resetErrorBoundary }) {
  console.error(error);
  // Call resetErrorBoundary() to reset the error boundary and retry the render.
  // Will work for errors caused by changing state, but not syntax errors
  return (
    <div className="ErrorMessage">
      <p>An error occurred:</p>
      <pre>{error.message}</pre>
      <button onClick={() => resetErrorBoundary()}>Try
Again?</button>
    </div>
  );
}

export default ErrorMessage;
```

Error Handling

This example component can be used to **test** the Error Boundary.

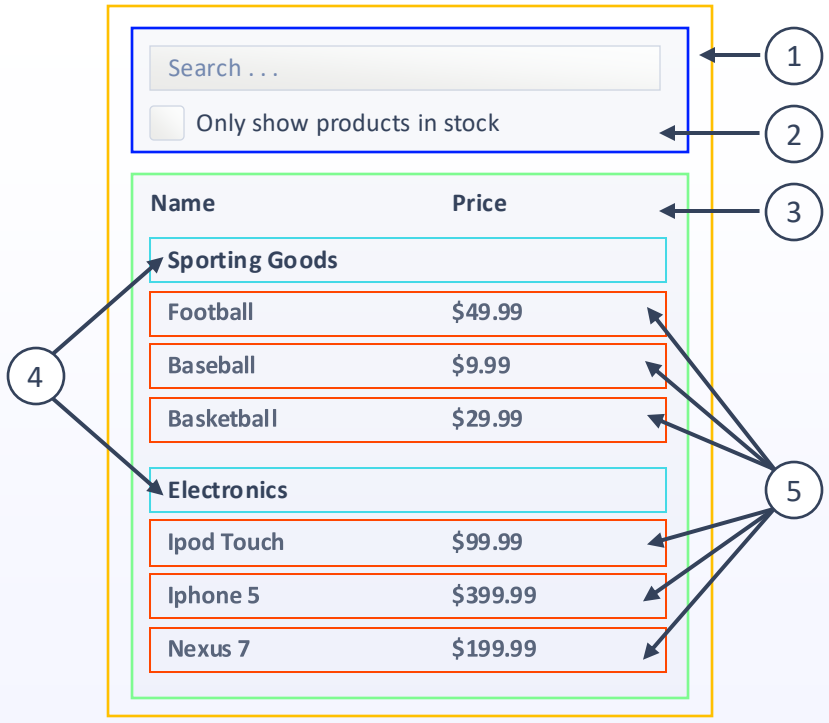
Try rendering it **without** the **ErrorBoundary** in place and click the button. This renders the **Bomb** component which **throws an error** and **crashes** React.

With the **ErrorBoundary** in place, the error is **still thrown**, but is **caught** by **ErrorHandler**. Clicking the **Try Again?** button will **reset** the error state and **reload** the React component tree.

```
function Bomb() {
    throw new Error('💥 KABOOM 💥')
}

function ExplodingBomb() {

    const [exploded, setExploded] = React.useState(false)

    return (
        <div className="ExplodingBomb componentBox">
            <button onClick={() => setExploded(!exploded) }>
                DANGER: Click to explode bomb!</button>

            {/* Renders the Bomb conditionally, depending on
state */}
            {exploded ? <Bomb /> : null}
        </div>
    )
}

export default ExplodingBomb;
```

React Basics - Thinking in React

Step 1: Break the UI into a Hierarchical Component Tree



Building a component-based React application from a design first involves **identifying the components**.

Draw a box around each distinct element, thinking about how you can re-use repeated elements by populating them with unique data.

Find logical names for each element, which can become your component names.

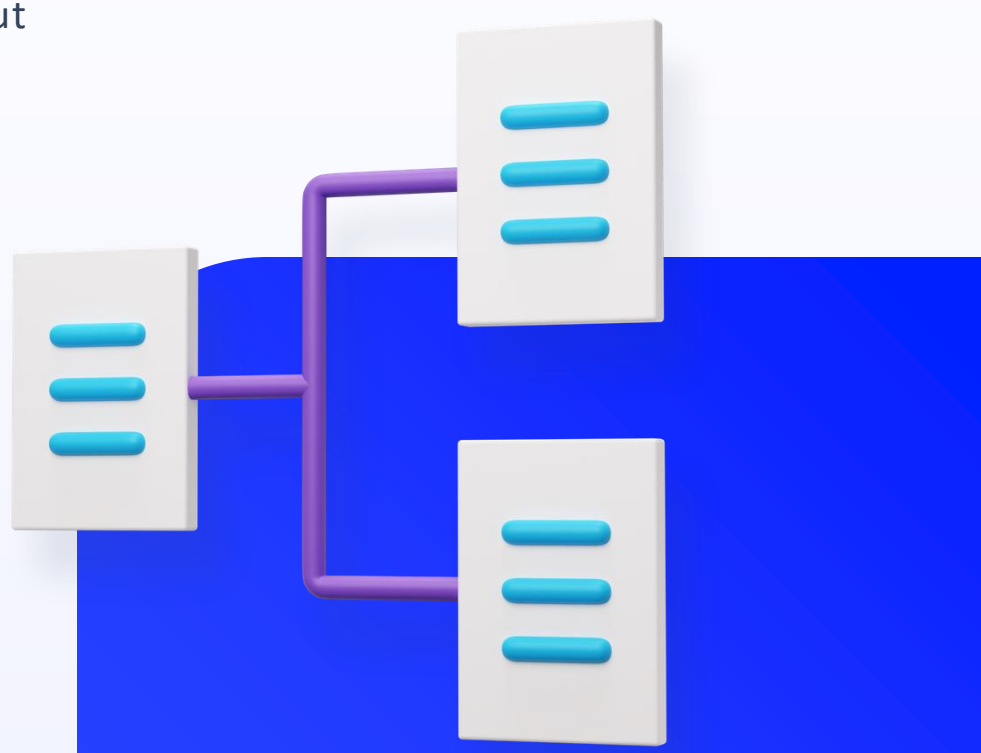
See : Thinking in React

React Basics - Thinking in React

1. **FilterableProductTable (orange)**: the parent component comprising the entire example
2. **SearchBar (blue)**: search component that handles user interaction by receiving user input
3. **ProductTable (green)**: displays product data matching the search/filter user input
4. **ProductCategoryRow (turquoise)**: displays a heading for each product category
5. **ProductRow (red)**: displays a row containing data for each product

The Hierarchy

- FilterableProductTable
  - SearchBar
    - ProductTable
      - ProductCategoryRow
      - ProductRow



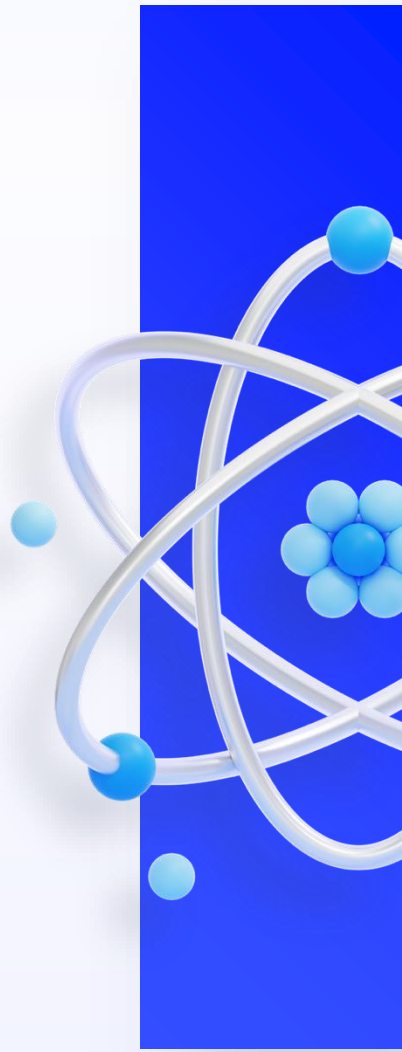
React Basics - Thinking in React

Step 2: Build a Static Version in React.

Once we have our **components** and their **hierarchy**, we can create a **static version** of our app that simply renders the UI based on the data. Implementing interactivity requires careful consideration, which we will address at a later stage.

To create this static version, we construct **components** that re-use other components and **pass data** through **props** (save state for later!).

There are two approaches to building the app: "**top down**" and "**bottom up**". In simpler cases, starting from the higher-level components in the hierarchy (e.g., **FilterableProductTable**) is usually easier, while for larger projects, it is often more convenient to begin from the lower-level components (e.g., **ProductRow**).



React Basics - Thinking in React

Step 3: Identity the Minimal (but complete) Representation of State.

State is a snapshot of the minimal set of changing data that your app needs to remember, to display on screen. To identify which data is state, ask yourself:

- Is it passed in from a parent **via props**? If so, it probably is **not** the state.
- Does it remain **unchanged** over time? If so, it probably is **not** the state.
- Can you **compute** it based on any other state or props in your component? If so, it definitely is **not** the state.
- Does it change **after the user interacts** with UI elements? If so, **yes** it is the state.

React Basics - Thinking in React

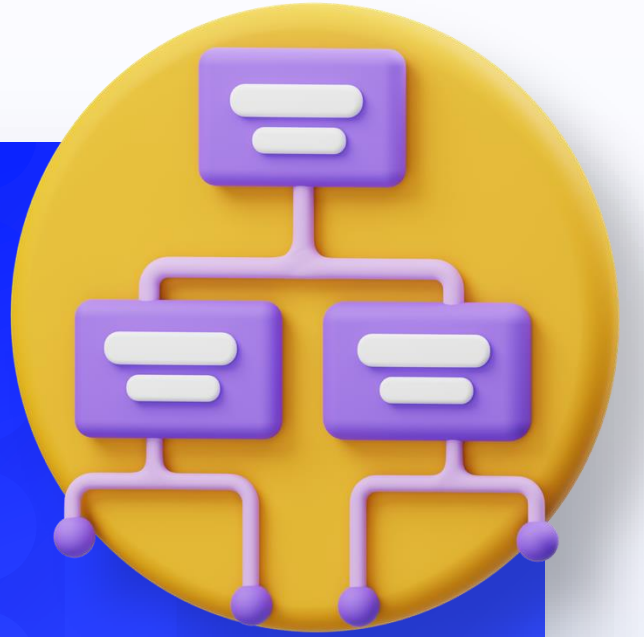
Step 4: Identify Where Your State Should Live.

We need to **identify** which component **mutates**, **or owns**, this **state**.

1. Identify every component that **renders** something **based on** that state.
2. Find a **common owner component** (a single component above all the components that need the state in the hierarchy).
3. Either the **common owner** or another **component higher up** in the hierarchy should **own the state**.
4. If you **can't find a component** where it makes sense to own the state, **create a new component** solely for **holding the state** and **add it** somewhere in the **hierarchy** above the common owner component.

React Basics - Thinking in React

Step 5: Add Inverse Data Flow



So far, we have built an app that renders correctly as a **function of props** and **state** flowing **down** the hierarchy. Now it is time to support data **flowing the other way**.

Any child components that need to update parent state need to be passed a **handler function** as a **prop** which can be called as needed, to update the state across the entire hierarchy.

Exercise 6

- Design and create a basic **Calculator** component which takes 2 numbers and the operator provided by the user and displays the result.