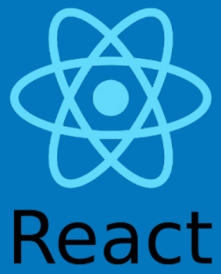


Fundamentals of React

About me – Jason Bell

Writing code since 1982 (TI-99/4A)

Native, Web, and Mobile Development

CTO for Successful Startup

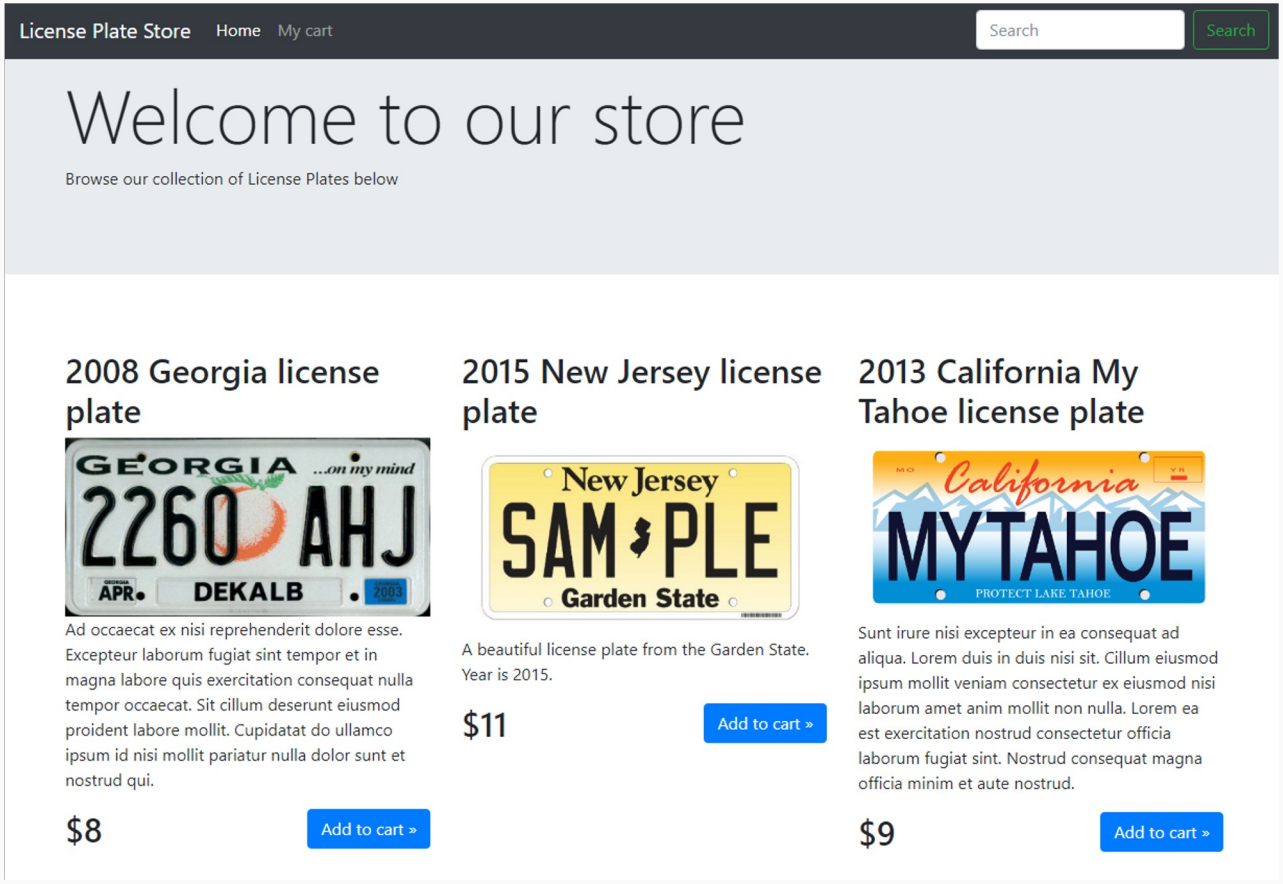
Published Author

Instructor since 2022

Quick poll

- How many of you are Java developers? C#, .Net?
- How many of you have any experience with React?
- With JavaScript? TypeScript?
- jQuery?
- Another JavaScript framework? Angular, Backbone, Vue?
- Your expectations for this class?

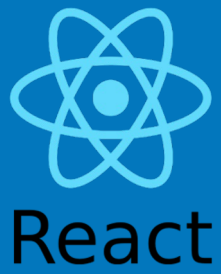
We are going to build a full application



How we're going to work

- Your questions are welcome, anytime!
- Being a web developer requires constant learning
- My goal is to give you the tools to work efficiently with React
  - We're going to practice a lot!

Chapter 1:
Introduction to React



What is React?

- React is a JavaScript library to build web applications
- Designed to be lightweight and simple to understand
- Not a framework: React is focused on rendering HTML. Additional features require third-party libraries
- Developed and used by Facebook, React is a favorite of Silicon Valley, also used by Netflix, Uber, Twitter, AirBnb, Pinterest, Instagram, and more

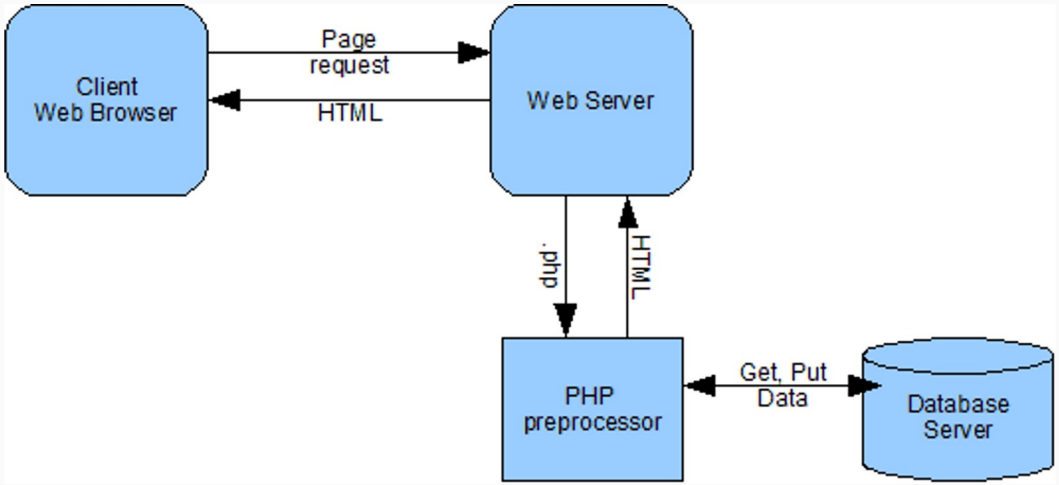
What is React?

- **React** lets you compose complex UIs from small and isolated pieces of code called “components”
- When our data changes, **React** will efficiently update and re-render our components. It **“reacts”** to data updates
- Creating components is equivalent to creating new, custom HTML elements that can be used in multiple different places within an app (think tabs, date pickers, forms, etc.)

How is it different from PHP, ASP, JSP?

In the past, all of the front-end code (HTML, JS, CSS) was generated from the back-end.

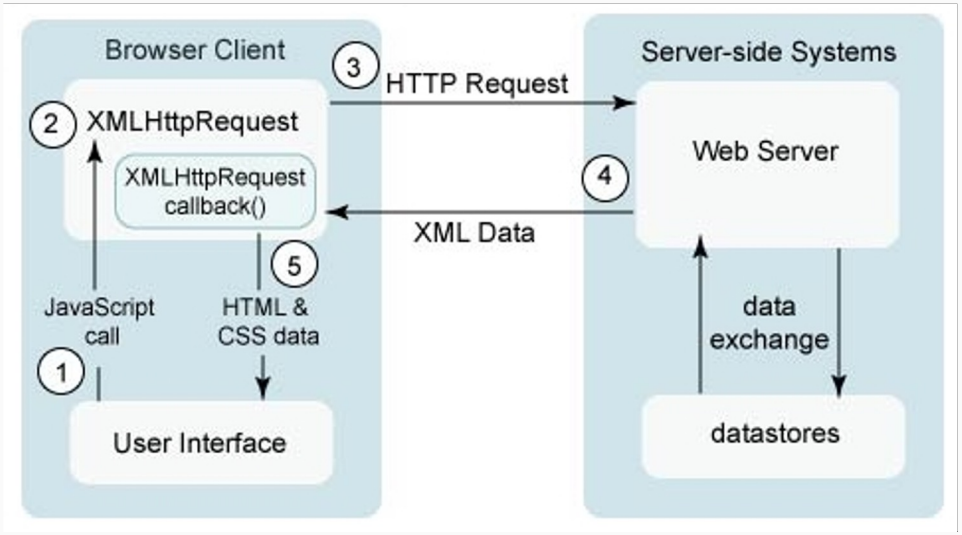
User interactions with the webpage often required a full-page refresh.



How is it different from PHP, ASP, JSP?

Then came AJAX and jQuery

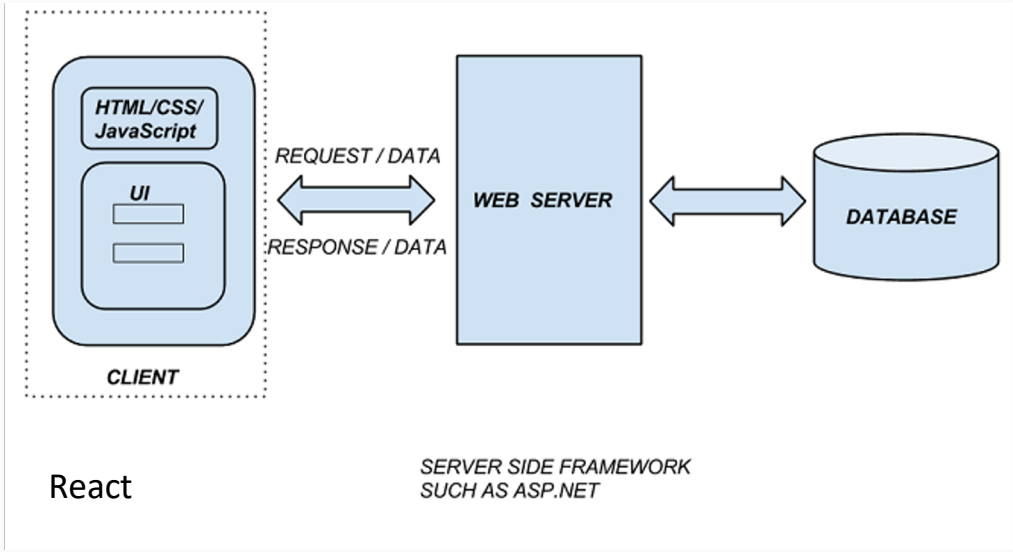
The main idea was to load content asynchronously in the background to refresh portions of a webpage



How is it different from PHP, ASP, JSP?

With React, the front-end code is independent from the back-end

The web server becomes a web-service that typically outputs JSON data, not dynamic HTML or CSS



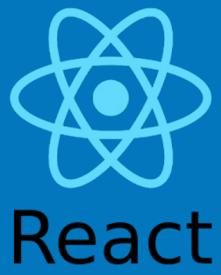
Creating a New React App

- For a new app, the React team recommends using **full-stack framework** like Next.js that takes full advantage of React's architecture to enable full-stack React apps
- You can also build a React app **without a framework** and add additional capabilities as you need them
- A **build tool** is required to run source code, provide a development server, and to package an app for deployment to a production server (**Vite**, Parcel, or Rsbuild)

Lab 1: Create a New React App

- Create a new React app by using **create-vite**
- Open the app's code in **Visual Studio Code**
- Use the **development server** to test the app

Chapter 2:
JSX



Introducing JSX

Consider this variable declaration:

```
const element = <h1>Hello, world!</h1>;
```

- It is called **JSX**, and it is a syntax extension to JavaScript. We recommend using it with React to describe what the UI should look like.
- **JSX** may remind you of a template language, but it comes with the full power of JavaScript.
- **JSX** produces React “elements”.

JSX is turned into DOM commands

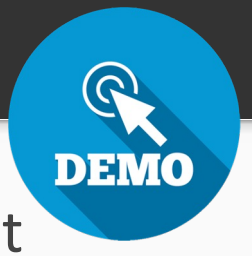
Babel compiles JSX down to `React.createElement()` calls.

These two examples are identical:

```jsx
const element = (
    <h1 className="greeting">
      Hello, world!
    </h1>
  );
```

```jsx
const element = React.createElement(
    'h1',
    {className: 'greeting'},
    'Hello, world!'
  );
```

Why JSX?



- React doesn't require using JSX, but most people find it helpful as a visual aid when working with UI markup inside of JavaScript code
- Here is how we can ask React to render a JSX element in the DOM:

```
const root = createRoot(document.getElementById('root')!);
root.render(<h1>Hello, world!</h1>);
```

Embedding Expressions in JSX

In this example, we declare a variable called name and use it inside JSX by wrapping it in curly braces

You can put any valid JavaScript expression inside the curly braces in JSX.

For example, `2 + 2`, `user.firstName`, or `formatName(user)` are all valid JSX expressions.


```
const name = "New React Dev";

const root = createRoot(
  document.getElementById("app")
);

root.render(<h1>Hello, {name}</h1>);
```

JSX is Javascript

After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects.

This means that you can use JSX inside of `if` statements and `for` loops, assign it to variables, accept it as arguments, and return it from functions.

```
function getGreeting(user) {
    if (user) {
      return <h1>Hello, {formatName(user)}!</h1>;
    }
    return <h1>Hello, Stranger.</h1>;
  }
```

Specifying Attributes with JSX

You may use quotes to specify string literals as attributes:

```
const element = <div tabIndex="0"></div>;
```

You may also use curly braces to embed a JavaScript expression in an attribute:

```
const element = <img src={user.avatarUrl}></img>;
```

Specifying Attributes with JSX

Don’t put quotes around curly braces when embedding a JavaScript expression in an attribute.

You should either use quotes (for string values) or curly braces (for expressions), but not both.

```
<img src={user.avatarUrl} />
```

```
<img src=”{user.avatarUrl}” />
<img src={“user.avatarUrl”} />
```

Warning:

Since JSX is closer to JavaScript than to HTML, React DOM uses `camelCase` property naming convention instead of HTML attribute names.

For example, `class` becomes `className` in JSX, and `tabindex` becomes `tabIndex`.

Specifying Children with JSX

If a tag is empty, you may close it immediately with `/>`, like XML:

```
const element = <img src={user.avatarUrl} />;
```

JSX tags may contain children:

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

Creating sibling elements with JSX

Sometimes, we want to return sibling elements with no defined parent:

```
<td>Hello</td>
<td>World</td>
```

In that case, we can wrap those elements in a **fragment element** since a JSX expression must represent a single element:

```
<React.Fragment>
  <td>Hello</td>
  <td>World</td>
</React.Fragment>
```

OR

```
<>
  <td>Hello</td>
  <td>World</td>
</>
```

Lab 2: Dynamic Content

- Modify the app to display a JavaScript constant
- Display the current time

JSX Prevents Injection Attacks

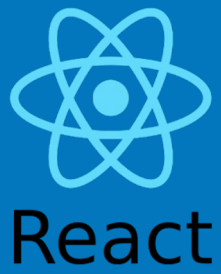
It is safe to embed user input in JSX:

```
const title = response.potentiallyMaliciousInput;
// This is safe:
const element = <h1>{title}</h1>;
```

By default, React DOM escapes any values embedded in JSX before rendering them.

This helps prevent XSS (cross-site-scripting) attacks.

Chapter 3:
Elements and React DOM / Virtual DOM



What are Elements?

- **`Elements`** are the smallest building blocks of React apps.
- Unlike browser DOM elements, React elements are plain objects, and are cheap to create.
- React DOM takes care of updating the DOM to match the React elements.

An element describes what you want to see on the screen:

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

Rendering an Element into the DOM

We need a ** ** somewhere in our main index HTML file:

```
<div id="root"></div>
```

We call this a “root” DOM node because everything inside it will be managed by React DOM.

Applications built with just React usually have a single root DOM node in their **`index.html`**

If you are integrating React into an existing app, you may have as many isolated root DOM nodes as you like.

Updating the Rendered Element

React elements are immutable. Once created, you can't change its children or attributes.

An element is like a single frame in a movie: it represents the UI at a certain point in time.

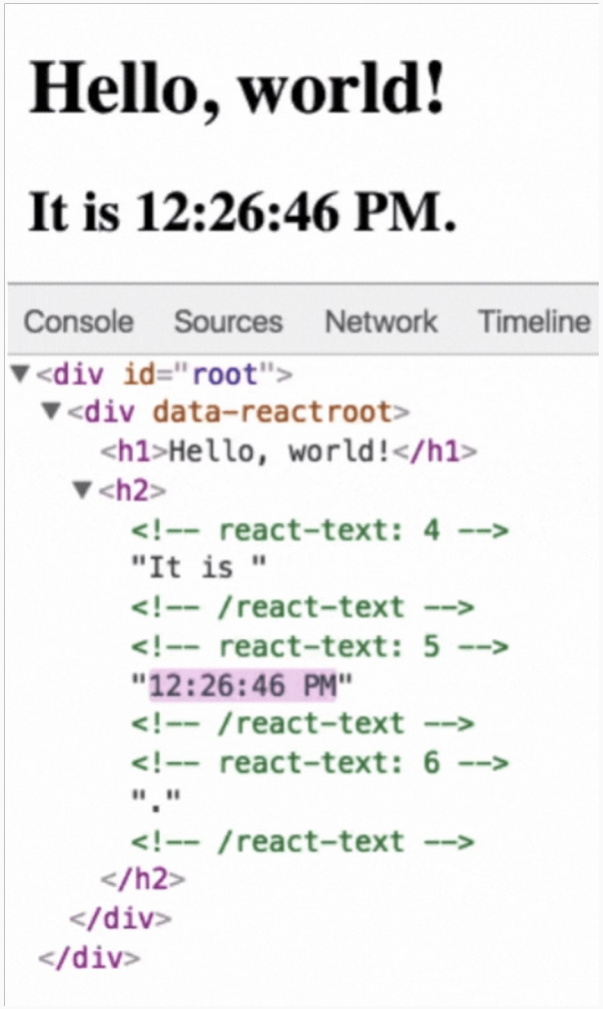
The only way to update the UI is to create a new element, and pass it to render( )


```
function tick() {
  root.render(
    <>
      <h1>Hello, {name}</h1>
      <h3>The time is {new Date().toLocaleTimeString()}</h3>
    </>
  );
}
setInterval(tick, 1000);
```

React Only Updates What's Necessary

React DOM compares the element and its children to the previous version of the DOM, and only applies the updates necessary to bring the DOM to the desired state.

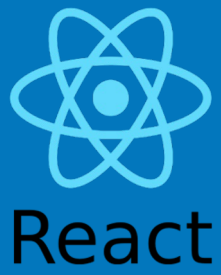
Even though we recreate the whole UI tree on every tick, only the text node whose contents have changed gets updated by React DOM, also known as Virtual DOM.



Virtual DOM defined

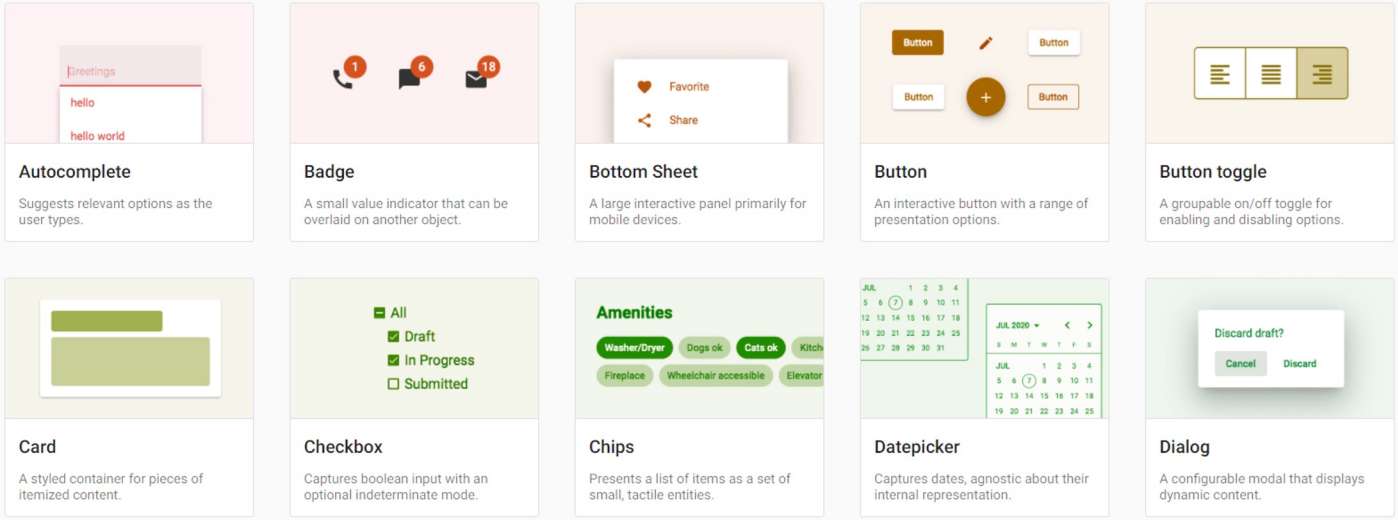
- React uses a virtual, in-memory representation of the DOM tree, known as the **“virtual DOM”**
- When a component re-renders, **React is creating a diff** of the virtual DOM **before and after** re-rendering.
- If the diff has any changes to apply to the DOM then React makes these **incremental changes** in the browser
- As a result, only the portions of the UI that need to be re-rendered actually get re-rendered

Chapter 4:
Components and Props



What are Components?

- Components let you split the user interface into independent, reusable pieces (think of them as widgets or custom HTML elements).
- A React app is made of several components.
- We can create our own components or build upon components from third-party libraries (e.g., https://material-ui.com):

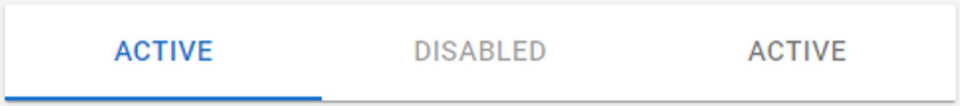


What are Components?

- Components are like JavaScript functions:
  They accept data inputs (called “Props”) and return React elements describing what should appear on the screen.
- Here is an example of using **`material-ui`** components with props:

```
<Tabs indicatorColor="primary" textColor="primary">
  <Tab label="Active" />
  <Tab label="Disabled" disabled />
  <Tab label="Active" />
</Tabs>
```

- The above code would render:



Function and Class Components

- The simplest way to define a component is to write a function:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

- This function is a valid React component because it accepts a single “props” (which stands for properties) object and returns a React element
- We call such components “**function components**” because they are literally JavaScript functions

Function and Class Components

You can also use an ES6 class to define a component:

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

- A component class has to extend **`React.Component`**
- A component class has a **`render()`** method
- Class and function components are equivalent from React's point of view

Rendering a Component

Previously, we only encountered React elements that represent DOM tags:

```
const element = <div />;
```

However, elements can also represent user-defined components:

```
const element = <Welcome name="Sara" />;
```

Rendering a Component

Let's recap what happens in this example:

1. We call `render()` with the `<Welcome name="Sara" />` element.
2. React calls the `Welcome` component with `{name: 'Sara'}` as the props.
3. Our `Welcome` component returns a `<h1>Hello, Sara</h1>` element as the result.
4. React DOM efficiently updates the DOM to match `<h1>Hello, Sara</h1>`

For example, this code renders "Hello, Sara" on the page:


```jsx
// This is a component with props
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}


const element = <Welcome name="Sara" />;
root.render(element);
```

Composing Components

Components can refer to other components in their output.

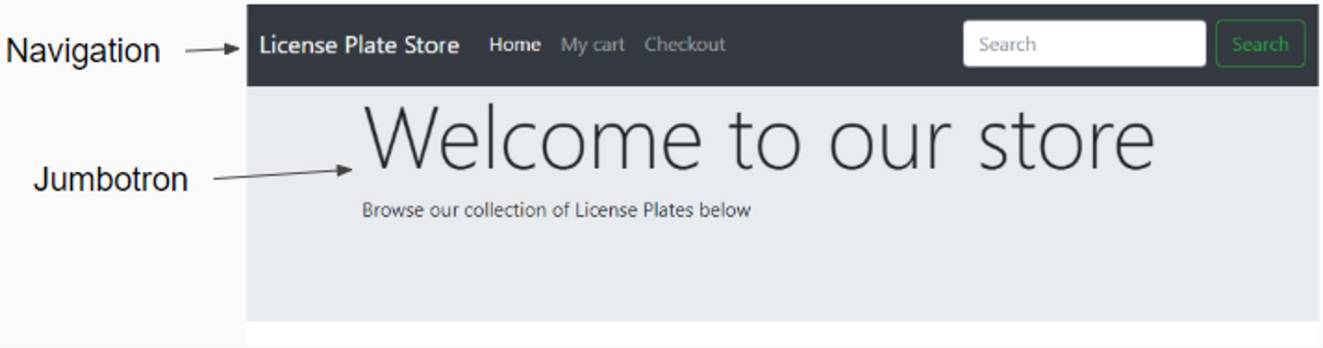
This lets us use the same component abstraction for any level of detail.

A button, a form, a dialog, a screen: in React apps, all those are commonly expressed as components.

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}
```

Lab 3: Defining Components

- Configure the app to use **Bootstrap** for styling
- Define a **navigation** component and a **jumbotron** component



Props are Read-Only

Whether you declare a component as a function or a class, it must never modify its own props. Consider this sum function:

```
function sum(a, b) {
  return a + b;
}
```

Such functions are called “pure” because they do not attempt to change their inputs, and always return the same result for the same inputs.

Props are Read-Only

In contrast, this function is impure because it changes its own input:

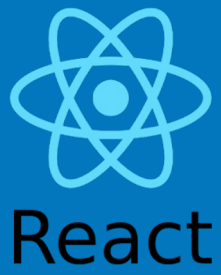
```
function withdraw(account, amount) {
  account.total -= amount;
}
```

Props are Read-Only

React is pretty flexible but it has a strict rule:
All React components must act like pure functions with respect to their props.

- Of course, application UIs are dynamic and change over time. Soon, we will introduce the concept of “state”.
- State allows React components to change their output over time in response to user actions, network responses, and anything else, without violating this rule.

Chapter 5:
Introduction to Typescript



Let’s get started - What is TypeScript?

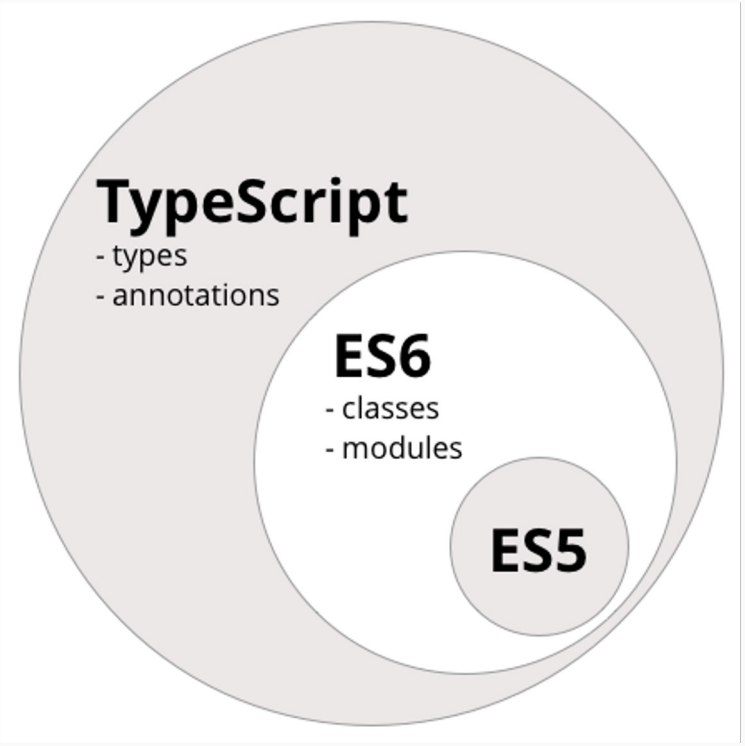
- TypeScript is a typed superset of JavaScript
- Converted (transpilied) to JavaScript
- All JavaScript code is valid TypeScript code
- https://www.typescriptlang.org/

What is TypeScript?

ES5 is yesterday's Javascript

ES6 is modern Javascript

Typescript = ES6 + types and annotations



TypeScript and types

- Typing is optional and can be used anywhere:

```
function sayHello(person : string) : string {
    return "Hello, " + person;
}

var user : string = "World";

document.body.innerHTML = sayHello(user);
```

Basic Types

- boolean, number, string
- Array:

```
var list : number[] = [1,2,3];

var list : Array<number> = [1,2,3];
```

Interfaces

- Here is an interface that describes something that has a firstName, a lastName, and a **`toString`** method:

```
interface Person {
    firstName : string,
    lastName : string,
    toString() : string
}
```

implements is not required. Having the right “shape” is enough.

Basic Types

- **any**, **unknown** and **void**:

```
// Any type works!
let notSure: any = 4;

// This function does not return anything
function warnUser(): void {
    alert("This is my warning message");
}
```

Generics

- Allow writing code that can work over a variety of types rather than a single one

```
class Greeter<T> {
    greeting: T;
    constructor(message: T) {
        this.greeting = message;
    }
    sayHello() : T {
        return this.greeting;
    }
}

let greet = new Greeter<string>("Hello "+ user.getFullName());
```

Type information for class component props

- Typescript enables type checking of our props, which also enables IDE features such as autocomplete.
- With class components, props are a generic type of the base component class from React.

```
export interface JumbotronProps {
    title: string;
    description: string;
}

export class Jumbotron extends React.Component<JumbotronProps> {
```

Type information for function component props

- Function components are even more straightforward:

```
export interface JumbotronProps {
    title: string;
    description: string;
}

function Jumbotron(props: JumbotronProps) {
```

Lab 4: Using Typescript with React

- Define an interface for JumbotronProps
- Refactor the Jumbotron component

Union Types

- Allows for multiple possible types - Here a string OR a number:

```
/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
 * If 'padding' is a number, then that number of spaces is added to the left side.
 */
function padLeft(value: string, padding: string | number) {
    // ...
}

let indentedString = padLeft("Hello world", true); // errors during compilation
```

Advanced Types

- Intersection Types - like a logical AND:

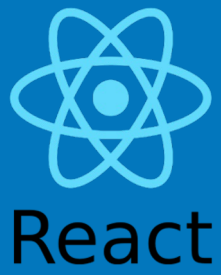
```
type Animal = {name: string}

type HoneyLover = {honey: boolean}

type Bear = Animal & HoneyLover;

const bear: Bear = {name: 'Smokey', honey: false};
```

Advanced Types - Alternative syntax

- Types do not have to be named - though that syntax can be confused with the one used for object creation:

```
type Animal = {name: string}

type Bear = Animal & {honey: boolean}

const bear: Bear = {name: 'Smokey', honey: false};
```

Chapter 6:
Handling Events



Handling Events

Handling events with React elements is very similar to handling events on DOM elements. There are some syntax differences:

- React events are named using **camelCase**, rather than **lowercase**.
- With JSX you pass a function as the event handler, rather than a string.

Handling Events

For example, the HTML:

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

is slightly different in React:

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

Handling Events: Function Component

With function components, event handlers are usually other functions defined outside the scope of the current component.

```
function handleClick() {
  console.log('clicked');
}

function Button() {
  return <button onClick={handleClick}>Click Me!</button>;
}

const element = <Button />;
ReactDOM.render(element, document.getElementById('root'));
```

Handling Events: Class Component

With class components, the best practice is to use arrow functions in order to preserve the scope of **`this`**.

This example is a good recipe to follow to listen to events in a class component.

```
class LoggingButton extends React.Component {
  // This syntax ensures `this` is bound within handleClick.

  handleClick = () => {
    console.log('this is:', this);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

Handling Events: Class Component

The problem with this syntax is that a different callback is created each time the **`LoggingButton`** renders. In most cases, this is fine.

As a result, the previous syntax is usually recommended.

If you aren't using class fields syntax, you can use an arrow function in the callback:

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }

  render() {
    // This syntax ensures `this` is bound within handleClick
    return (
      <button onClick={() => this.handleClick()}>
        Click me
      </button>
    );
  }
}
```

Passing Arguments to Event Handlers

Inside a loop, it is common to want to pass an extra parameter to an event handler. For example, if `id` is the row ID, either of the following would work:

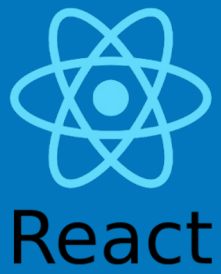
```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

The above two lines are equivalent, and use arrow functions and `Function.prototype.bind` respectively.

Lab 5: Events and Props

- Define a **LicensePlate** component
- Use **props** to pass information to the component (the plate to be displayed)
- Listen for button **click** event to display an alert when added to the cart

Chapter 7:
Repeating content for tables and lists



Array.map( )

- The **`map()`** function can be used to apply a transformation to all elements of an array and return a new array.
- In this example, we assign the new array returned by **`map()`** to the variable **`doubled`** and log it:

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);
console.log(doubled);
```

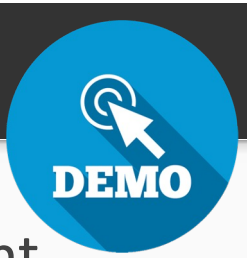
- This code logs **`[2, 4, 6, 8, 10]`** to the console.

Rendering Multiple Elements

- You can build collections of elements and include them in JSX using expressions with curly braces `{}`.
- Below, we loop through an array using the JavaScript `map()` function.
- We return a `<li>` element for each item and assign the resulting array of elements to `listItems`:

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li>{number}</li>
);
```

Rendering Multiple Elements



We include the entire **`listItems`** array inside a **`<ul>`** element, and render it to the DOM:

```
root.render(<ul>{listItems}</ul>);
```

This code displays a bullet list of numbers between 1 and 5.

Basic List Component

When you run this code, you'll be given a warning that a key should be provided for list items.

A "key" is a unique string attribute you need to include when creating lists of elements.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li>{number}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Keys

- Keys help React identify which items have changed, are added, or are removed.
- Keys should be given to the elements inside the array to give the elements a stable identity:

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li key={number.toString()}>
    {number}
  </li>
);
```

Keys

- The best way to pick a key is to use a string that uniquely identifies a list item among its siblings.
- Most often you would use IDs from your data as keys:

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}>
    {todo.text}
  </li>
);
```

Keys

- When you don’t have stable IDs for rendered items, you may use the item index as a key as a last resort.

- This is not ideal and can have a negative impact on performance:

```
const todoItems = todos.map((todo, index) =>
  // Only do this if items have no stable IDs
  <li key={index}>
    {todo.text}
  </li>
);
```

Keys must only be unique among siblings

- Keys serve as a hint to React but they don't get passed to the component
- If you need the same value in your component, pass it explicitly as a prop with a different name:

```
const content = posts.map((post) =>
  <Post
    key={post.id}
    id={post.id}
    title={post.title} />
);
```

- With the example above, the **Post** component can read **`props.id`**, but not **`props.key`**.

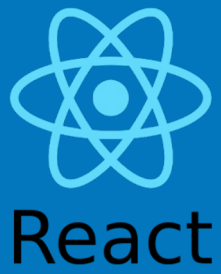
Embedding map() in JSX

JSX allows embedding any expression in curly braces so we could inline the `map()` result:

```
function NumberList(props) {
  const numbers = props.numbers;
  return (
    <ul>
      {numbers.map((number) =>
        <ListItem key={number.toString()}
                  value={number} />
      )}
    </ul>
  );
}
```

Lab 6: Lists and Keys

- Use **`Array.map`** function to render an array of license plates
- Render every other plate with a light grey background (HTML color code **`#F5F5F5`**)



Chapter 8:
Conditional Rendering

Conditional Rendering

- Conditional rendering in React works the same way conditions work in JavaScript.
- We can use `if` or the inline conditional operators to decide whether to display an element or not:

```
<div>
  The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.
</div>
```

- In this example, the expression will render “currently”, when **`isLoggedIn`** is **`true`**, “not” otherwise

Conditional Rendering

Let’s consider these two function components:

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}
```

Conditional Rendering

We'll create a `Greeting` component that displays either of these components depending on whether a user is logged in.

This example renders a different greeting depending on the value of `isLoggedIn` prop.

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}
```

Element Variables

You can use variables to store elements.

This can help you conditionally render a part of the component while the rest of the output doesn't change.

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  let button;
  if (isLoggedIn) {
    button = <LogoutButton onClick={this.handleLogoutClick} />;
  } else {
    button = <LoginButton onClick={this.handleLoginClick} />;
  }

  return (
    <div>
      <Greeting isLoggedIn={isLoggedIn} />
      {button}
    </div>
  );
}
```

Inline If with Logical && Operator

You may embed expressions in JSX by wrapping them in curly braces.

This includes the JavaScript logical && operator.

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
        </h2>
      }
    </div>
  );
}
```

Inline If with Logical && Operator

The previous example works because in JavaScript, **`true && expression`** always evaluates to **`expression`**, and **`false && expression`** always evaluates to **`false`**.

Therefore, if the condition is **`true`**, the element right after **`&&`** will appear in the output.

If it is **`false`**, React will ignore it and won't render it.

Inline If-Else with Conditional Operator

Another method for conditionally rendering elements inline is to use the JavaScript conditional operator **`condition ? true : false`**

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.
    </div>
  );
}
```

In the example, we use it to conditionally render a small block of text.

Inline If-Else with Conditional Operator

It can also be used for larger expressions although it can be less obvious to read and understand:

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn
        ? <LogoutButton onClick={this.handleLogoutClick} />
        : <LoginButton onClick={this.handleLoginClick} />
      }
    </div>
  );
}
```

Inline If-Else with Conditional Operator

- Just like in JavaScript, it is up to you to choose an appropriate style based on what you and your team consider more readable

- Remember that whenever conditions become too complex, it might be a good time to extract a component

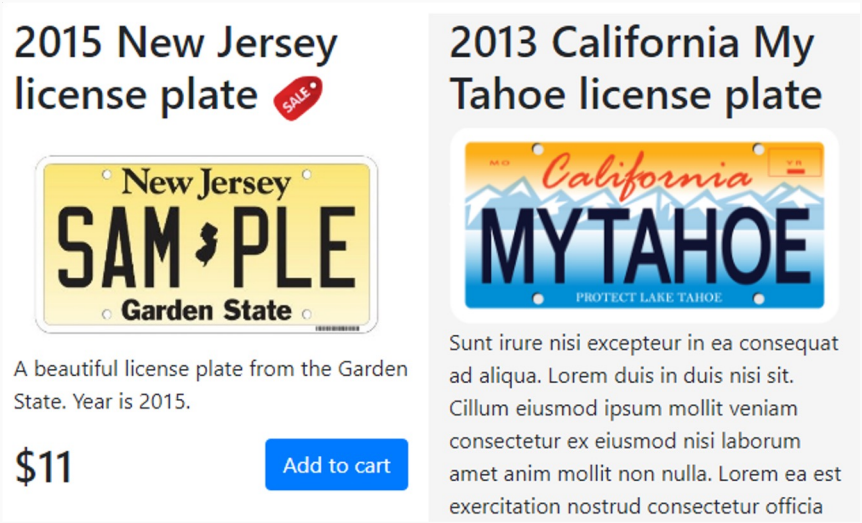
Preventing a Component from Rendering

- In rare cases you might want a component to hide itself even though it was rendered by another component. To do this, return `null`

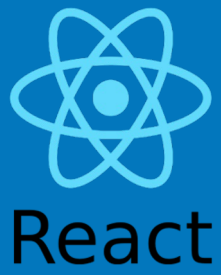
```
function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }

  return (
    <div className="warning">
      Warning!
    </div>
  );
}
```

Lab 7: Conditional Rendering

- Conditionally render an image
- Some license plates are on sale (property **`onSale = true`**).
- We want to showcase those products by adding an image next to their title (**`sale.png`**)



Chapter 9:
React State



useState() for function components

useState returns the current state value and a function to update that value - in an array.

It takes a default value as a parameter.


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

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

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

useState() Hook

You can use the useState Hook more than once in a single component

Note the **array destructuring syntax** that lets us give different names to the state variables we received when calling **useState**.

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

function MyComponent() {
  const [age, setAge] = useState(28);
  const [name, setName] = useState('Taylor');
  const [todos, setTodos] = useState(() => createTodos());
  // ...
```

useState() Hook

If the new state is computed based on the previous state, we can pass a function to the setter to compute that new value as follows:

```
const [count, setCount] = React.useState(0);

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

Adding Local State to a Class Component

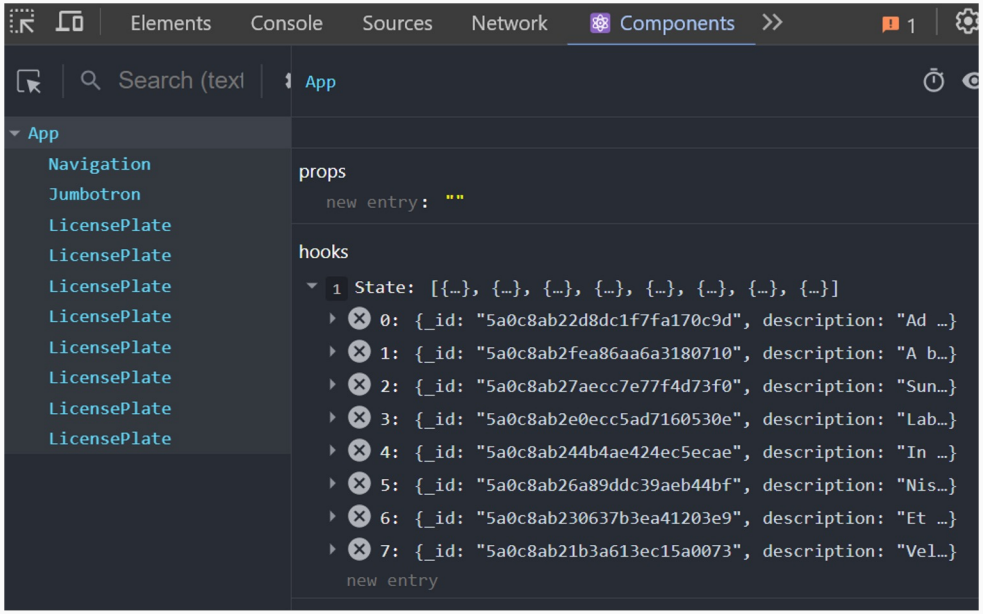
`this.state` is a property of **`React.Component`**

We can add any property we want on the state object and assign a default value when we initialize it.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

Lab 8: Using State

- Add some state to the **App** component
- Explore the **React Dev Tools** in the browser



Using State Correctly

Do not Modify State Directly

For example, this will not re-render a component:

```
// Wrong
this.state.comment = 'Hello';
```

Instead, use **`setState()`**

```
// Correct
this.setState({comment: 'Hello'});
```

Using State Correctly

State Updates May Be Asynchronous

- React may batch multiple **`setState()`** calls into a single update for performance.
- Because **`this.props`** and **`this.state`** may be updated **asynchronously**, you should not rely on their values for calculating the next state.

For example, this code may fail to update a counter:

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

Using State Correctly

State Updates May Be Asynchronous

To fix it, use a second form of **`setState()`** that accepts a function rather than an object as a parameter.

That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument:

```
// Correct
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

Using State Correctly

State Updates are Merged

When you call **`setState(),`** React merges the object you provide into the current state.

Your state may contain several independent properties:

```
constructor(props) {
  super(props);
  this.state = {
    posts: [],
    comments: []
  };
}
```

Using State Correctly

State Updates are Merged

You can update state independently with separate **`setState()`** calls.

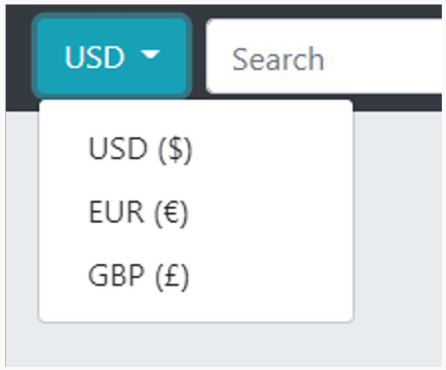
Merging is shallow, so **`this.setState({comments})`** leaves **`this.state.posts`** intact, but completely replaces **`this.state.comments`**.

```
this.setState({
  posts: response.posts
});
```

```
this.setState({
  comments: response.comments
});
```

Lab 9: Class component migration and useState

- In this lab, we will turn a **CurrencyDropdown** class component into a function component that uses Hooks to store its state.



With React, Data Flows Down

This is commonly called a “top-down” or “unidirectional” data flow. Any state is always owned by **some specific component**, and any data or UI derived from that state can only affect components **“below”** them in the tree.

If you imagine a component tree as a **waterfall** of props, each component’s state is like an additional water source that joins it at an arbitrary point but also **flows down**.

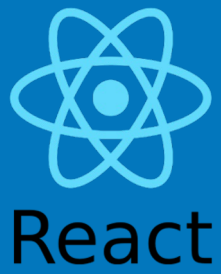
Data Flows Down

A component may choose to pass its state down as props to its child components:

```
<FormattedDate date={this.state.date} />
```

The **`FormattedDate`** component would receive the date in its props and wouldn't know whether it came from the Clock's state, from the Clock's props, or was typed by hand:

```
function FormattedDate(props) {
  return <h2>It is {props.date.toLocaleTimeString()}.</h2>;
}
```



Chapter 10:
Component Lifecycle

Mounting and Unmounting

- In applications with many components, it's very important to **free up resources** taken by the components when they are destroyed.

- When a component is rendered to the DOM for the first time, this is called **"mounting"** in React.

- Whenever the DOM produced by a component is removed, this is called **"unmounting"** in React.

Adding Lifecycle Methods to a Class

We can add special methods on the component class to run some code when a component mounts and unmounts.

These methods are called **“lifecycle methods”**

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
  }

  componentWillUnmount() {
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

componentDidMount()

The **componentDidMount()** method runs after the component output has been rendered to the DOM.

This is a good place to set up a timer or run async tasks:

```
componentDidMount() {
  this.timerID = setInterval(
    () => this.tick(),
    1000
  );
}
```

Note how we save the timer ID right on this (**`this.timerID`**).

componentWillUnmount()

Using our previous example, we will tear down the timer in the **componentWillUnmount()** lifecycle method:

```
componentWillUnmount() {
  clearInterval(this.timerID);
}
```

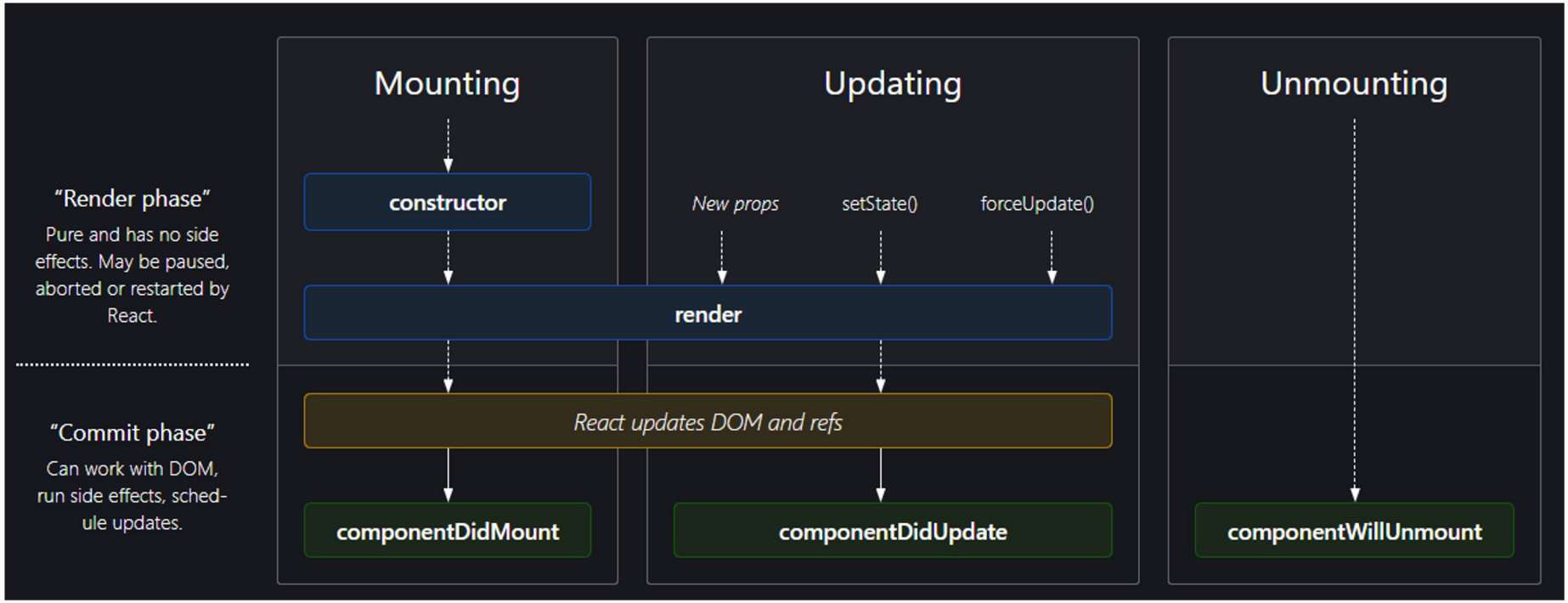
Adding Lifecycle Methods to a Class

We add a method called **tick()** that the **Clock** component will run every second.

It uses **this.setState()** to schedule updates to the component local state.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Lifecycle Methods Diagram



useEffect() Hook

useEffect is the equivalent of lifecycle methods for function components.

It takes a function as a parameter.

By using this Hook, you tell React that your component needs to do something after **each render**.

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

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

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });
```

useEffect() Hook with empty array

useEffect takes a second parameter that determines when the effect should run.

Using an **empty array** means that we want the effect to run **just once** after the initial render (equivalent to the **componentDidMount** lifecycle method for class components)

```
let [todos, setTodos] = React.useState([]);

React.useEffect(() => {
  fetch('https://jsonplaceholder.typicode.com/todos?userId=1')
    .then(response => response.json())
    .then(json => setTodos(json))
}, []);
```

Cleaning up before unmounting

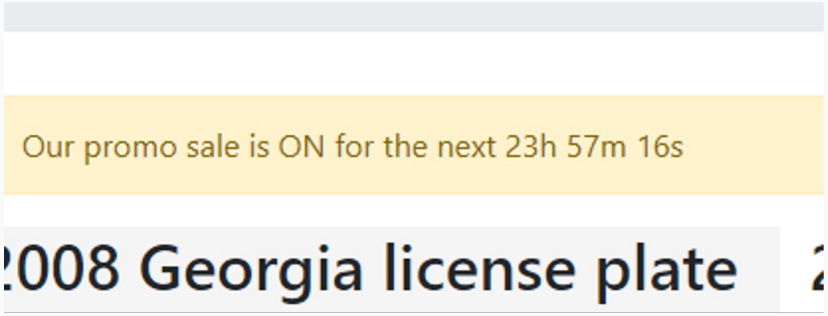
useEffect also accepts a returned function that will run before unmounting a component.

This is a perfect solution to unsubscribe/clear some timers as we would do with **componentWillUnmount** in a class component.

```
function FriendStatus(props) {
  // ...
  useEffect(() => {
    // ...
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
```

Lab 10: useEffect and Updating State

- Add a promo banner with a countdown that gets updated every second

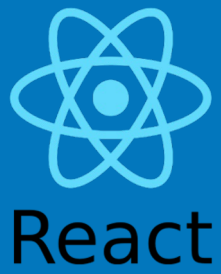


useEffect() Hook with non-empty array

The second parameter of **useEffect** is an **array of dependencies** that need to change values for the effect to actually run.

This is a perfect solution when you have multiple side-effects based on different props/state values.

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



Chapter 11:
HTTP requests and Promises

Using fetch()

- The **Fetch API** is standard in modern browsers and does not require an additional install
- **fetch()** returns a promise that resolves to a **Response** object

```
fetch('http://localhost:8000/data').then(response => response.json())
```

- The interface for the **Response** object can be found here: https://developer.mozilla.org/en-US/docs/Web/API/Response

- In our example, we call the **json()** method, which deserializes the response body from a JSON string into a Javascript object.

What are promises?

A **Promise** is an object representing the eventual completion or failure of an asynchronous operation.

We use the **then** method to register callback functions that will be triggered upon success or failure

```
const promise = doSomething();

promise.then(successCallback,
failureCallback);
```

Promises can be chained

The code on the right side of the screen would run in the order in which it is declared, no matter how much time each individual step takes

```
doSomething()
.then(result => doSomethingElse(result))
.then(newRes => doThirdThing(newRes))
.then(finalRes => console.log(`Final
result: ${finalRes}`))
.catch(failureCallback);
```

Promise.all()

Creates a new promise that resolves once all given promises resolve, or fails if any of those promises fail.

```
var promise1 = Promise.resolve(3);
var promise2 = 42;
var promise3 =
new Promise(function(resolve, reject) {
    setTimeout(resolve, 100, 'foo');
});


Promise.all(
  [promise1, promise2, promise3]
).then(function(values) {
    console.log(values);
});
// output: Array [3, 42, "foo"]
```

When to call fetch()

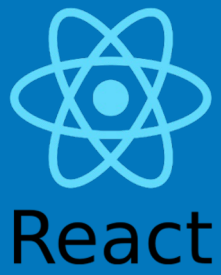
- As we saw in the previous chapter, **componentDidMount** (or **useEffect**) is the right place to call any asynchronous tasks such as HTTP requests.
- In this example, we fetch some data, get the body of the response, and assign that deserialized body to the state of our component:

```
componentDidMount() {
    fetch('http://localhost:8000/data')
        .then(response => response.json())
        .then(data => this.setState({licensePlates:  data }));
}
```

Lab 11: Making HTTP Requests with fetch

- Connect the **App** component to an HTTP server
- Use the **fetch** API to download license plate data

Chapter 12: Forms



Forms

HTML form elements work a little bit differently from other DOM elements in React, because form elements naturally keep some internal state.

For example, this form accepts a single name that will be stored in the **`value`** property of the **`input`** HTML element:

```
<form>
  <label>
    Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

Forms

- This form has the default HTML form behavior of browsing to a new page when the user submits the form
- If you want this behavior in React, it just works
- In most cases, it's convenient to have a JavaScript function that handles the submission of the form and has access to the data that the user entered into the form
- The standard way to achieve this is with a technique called **controlled components**

Controlled Components

In React, mutable state is kept in the state property of components, and only updated with `setState()`

- We can make the **React state** be the **"single source of truth"** for our form
- Then the React component that renders a form also controls what happens in that form on subsequent user input
- This is called a **controlled component**

Controlled Components

In order to have a controlled component, we need two things:

1) The current value of the HTML form element is read from our React state
2) Changing the value of the form updates the React state

```
class FlavorForm extends React.Component {

  state = {value: 'coconut'};

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

  handleSubmit(event) {
    alert('Your favorite flavor is: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={(e) => this.handleSubmit(e)}>
        <label>
          Pick your favorite flavor:
          <select value={this.state.value} onChange={(e) => this.handleChange(e)}>
```

Controlled Components for different HTML form elements

`<input type="text">`, **`<textarea>`**, and **`<select>`** all work very similarly - they all have a value attribute that you can use to implement a controlled component.

You can pass an array into the value attribute, allowing you to select multiple options in a select tag:

```
<select multiple={true} value={['B', 'C']}>
```

Handling Multiple Form Elements

When you need to handle multiple controlled input elements, you can add a **`name`** attribute to each element and let the handler function choose what to do based on the value of **`event.target.name`**

```
class Reservation extends React.Component {

  state = {
      isGoing: true,
      numberOfGuests: 2
   };

  handleInputChange = (event) => {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
    const name = target.name;
    this.setState({[name]: value});
  }

  render() {
    return (
      <form>
        <label>
          Is going:
          <input
            name="isGoing" type="checkbox" checked={this.state.isGoing}
            onChange={this.handleInputChange} />
        </label>
```

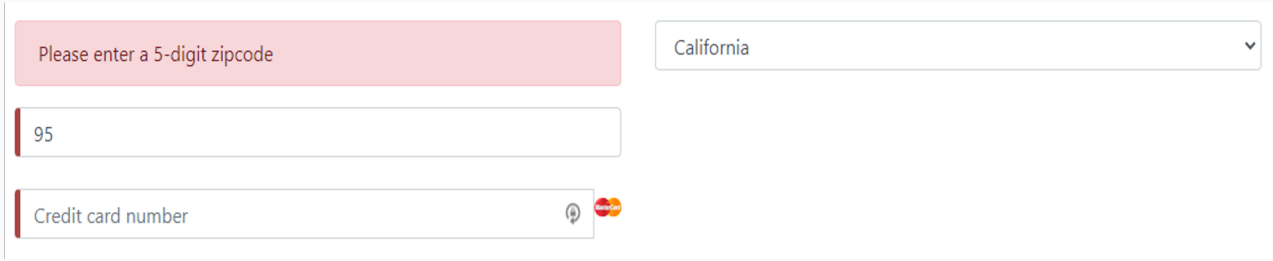
Handling Multiple Form Elements

Note how we used the ES6 computed property name syntax to update the state key corresponding to the given input name:

```
this.setState({
  [name]: value
});
```

Lab 12: Using a Controlled Component Form

- Implement a **CheckoutForm** component as a controlled component
- Implement some form validation with feedback to the user
- Submit the form to an HTTP server



Uncontrolled Components

- It can be tedious to use controlled components, because we need to write an event handler for every way our data can change and pipe all of the form state through component state

- There is an alternative technique for implementing input forms called **uncontrolled components**, which relies on **Refs**

- With that approach, we rely on the internal DOM state as the single source of truth and we read that state when needed

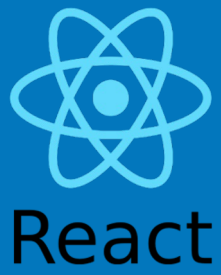
Uncontrolled Components

A **ref** gives us a reference to an HTML DOM element, which we can access using the `current` property of the ref.

Refs can be created with the `createRef()` function or the **useRef()** hook for function components

```
class NameForm extends React.Component {

  input = React.createRef();


  handleSubmit = (event) => {
    alert('A name was submitted: ' + this.input.current.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Chapter 13:
React Router



React Component Router

React was designed for Single Page Applications, but very often we need more than one page!

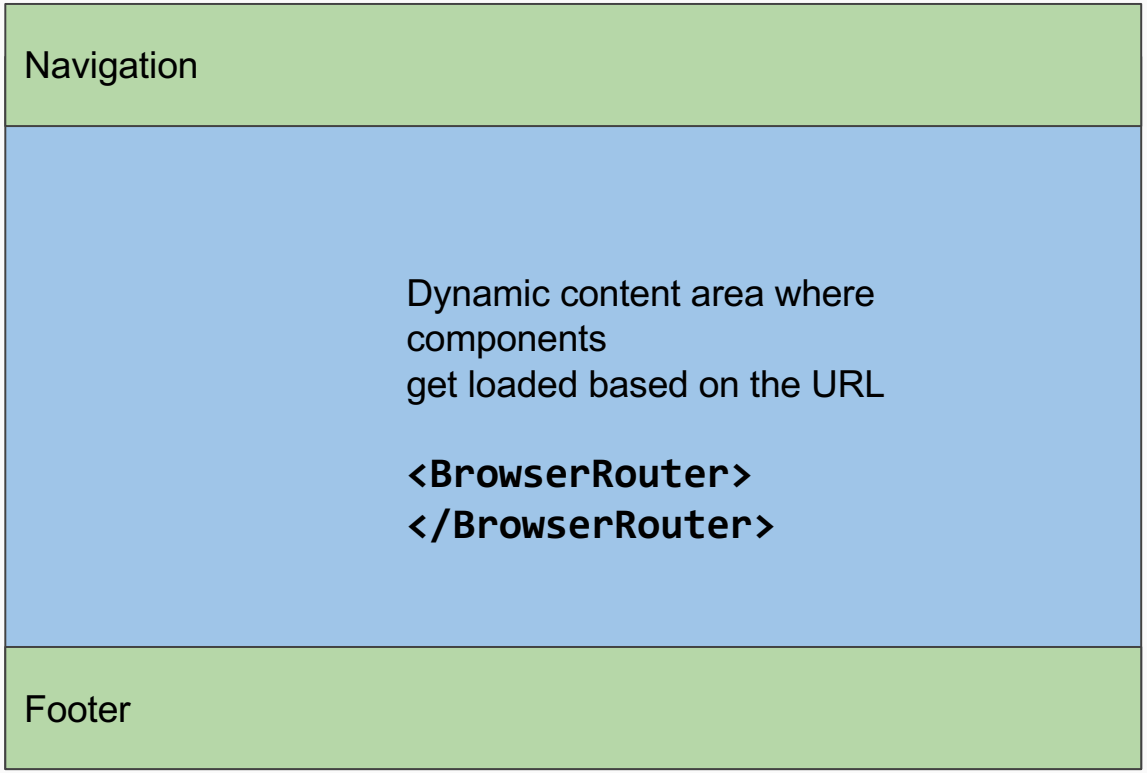
The React Router, which is a separate library (https://reactrouter.com), allows us to emulate this by loading components based on URLs

For instance, **`/cart`** would load a Cart component, **`/login`** for login, etc.

Example of router

Here the navigation component and the footer are always the same on all pages.

The only dynamic part is the blue section, determined by the **`BrowserRouter`** component



React Router Config

- Requires **`BrowserRouter`** and **`Route`** components that map URLs to components:

```
<BrowserRouter>
  <Routes>
    <Route path="/" element={<App />} />
    <Route path="expenses" element={<Expenses />} />
    <Route path="invoices" element={<Invoices />} />
  </Routes>
</BrowserRouter>,
```

React Router Links

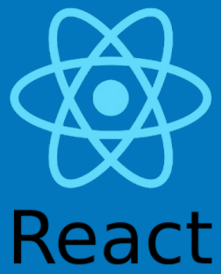
Router links can be used to navigate between routes using the **`Link`** component and the **`to`** prop to set the destination:

```
<Link to="/invoices">Invoices</Link>
<Link to="/expenses">Expenses</Link>
```

Lab 13 - Component Router

- We would like three different views: A store view, a checkout view, and a cart view
- **`StoreViewComponent`** should be the default route
- **`CartViewComponent`** should be at **`/cart`**
- **`CheckoutViewComponent`** should be at **`/checkout`**

Chapter 14: Component Architecture at scale



Reusable components with props.children

A special prop called **children** can be used to pass JSX content to a component. Put it in the body of the component's element:

```
<PopupWindow title={"Test"} show={true} >
    My custom content with <b>HTML</b> and components:
    <Jumbotron/>
</PopupWindow>
```

Content can be displayed in the component using **props.children**:

```
<div className="overlay-content alignLeft">
    {props.children}
</div>
```

Extracting Components

Don’t be afraid to split components into smaller components.

For example, consider this **`Comment`** component:

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}
        />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Extracting Components

It accepts author (an object), text (a string), and date (a date) as **props**, and describes a comment on a social media website.

This component can be tricky to change because of all the nesting, and it is also hard to reuse individual parts of it.

Let's extract a few components from it.

Extracting Components

First, we will extract **`Avatar`**:

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}
    />
  );
}
```

The Avatar doesn't need to know that it is being rendered inside a Comment. This is why we have given its prop a more generic name: user rather than author.

Extracting Components

We can now simplify Comment a tiny bit:

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <Avatar user={props.author} />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Extracting Components

Next, we will extract a **`UserInfo`** component that renders an **`Avatar`** next to the user's name:

```
function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user} />
      <div className="UserInfo-name">
        {props.user.name}
      </div>
    </div>
  );
}
```

Extracting Components

This lets us simplify `Comment` even further:

```
function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Extracting Components

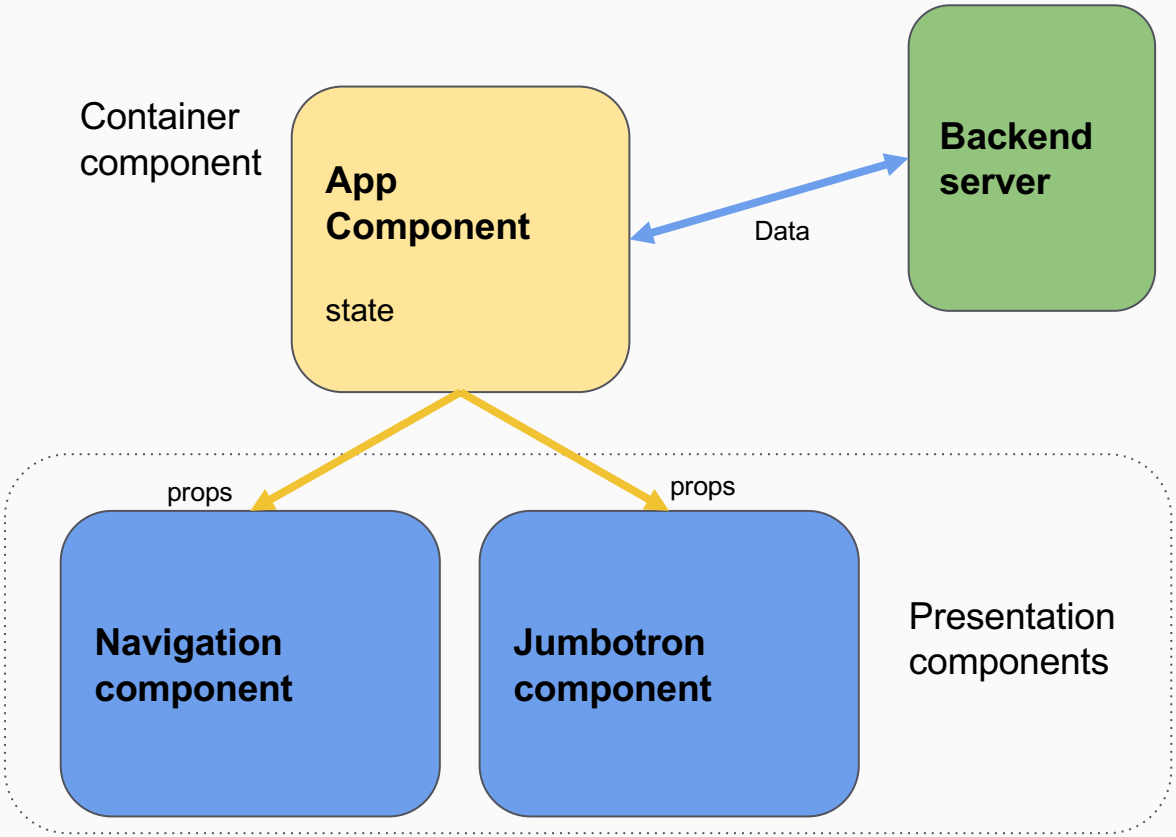
Extracting components might seem like grunt work at first, but having a palette of reusable components pays off in larger apps.

A good rule of thumb is that if a part of your UI is used several times (**`Button`**, **`Panel`**, **`Avatar`**), or is complex enough on its own (**`App`**, **`FeedStory`**, **`Comment`**), it is a good candidate to be extracted to a separate component.

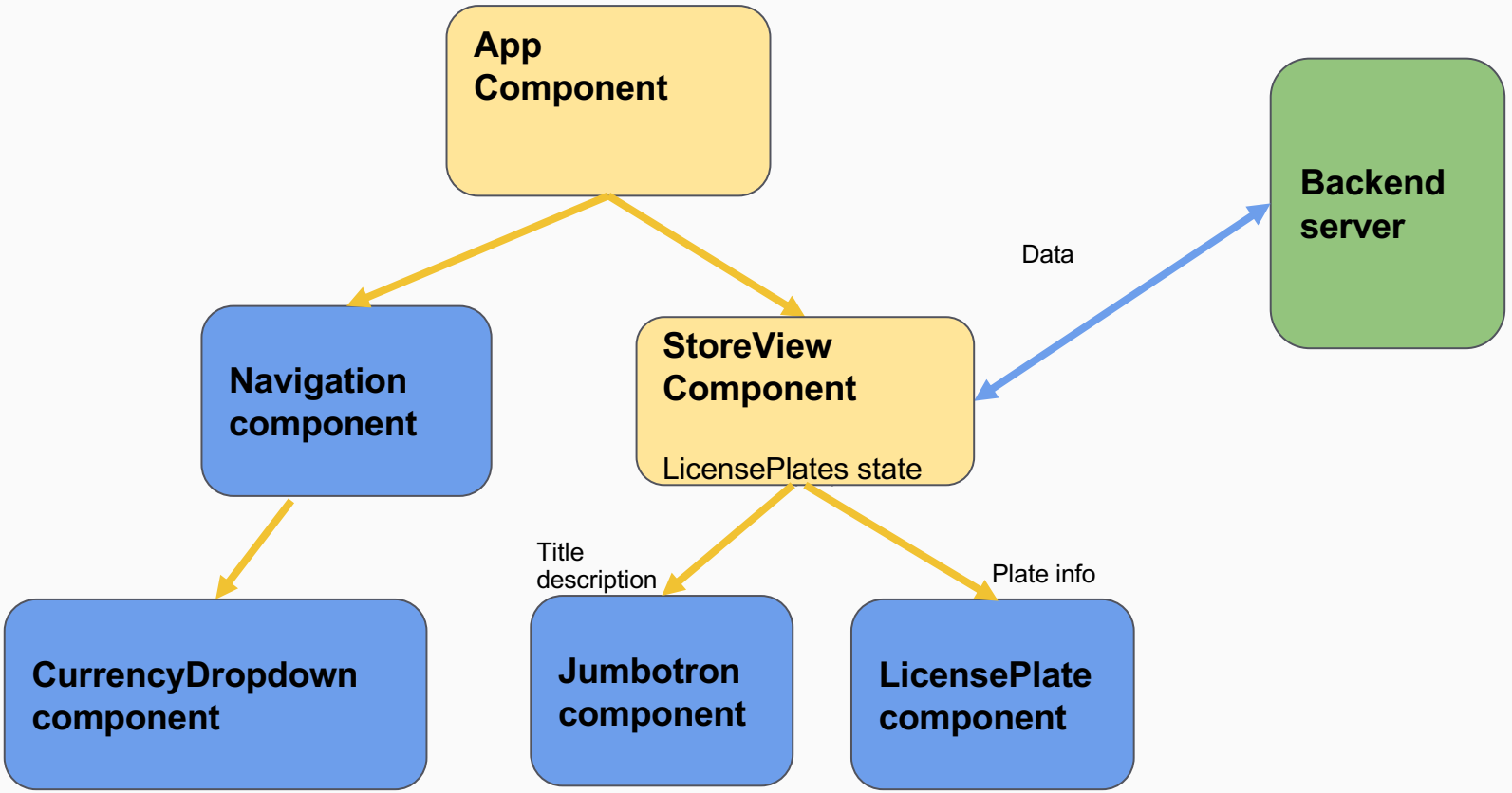
Component architecture and design

- React embraces **reactive programming**, where components know how to react to external changes
- This pattern relies on two types of components:
  - **Container components** (or “smart” components) know how to get data and work with servers and store that info in their internal state
  - **Presentation components** (a.k.a. “dumb” or “lazy” components) have to be fed with data through props and are usually stateless

Component Architecture: Recap

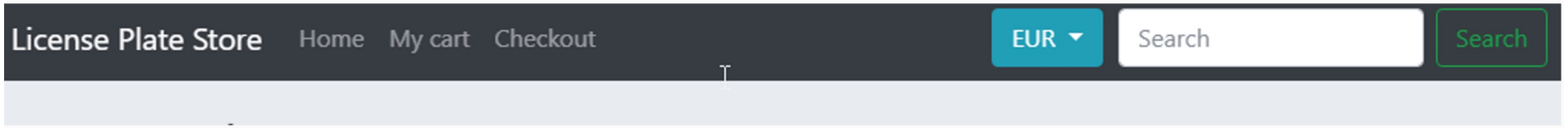


Component Architecture of our License Plate store



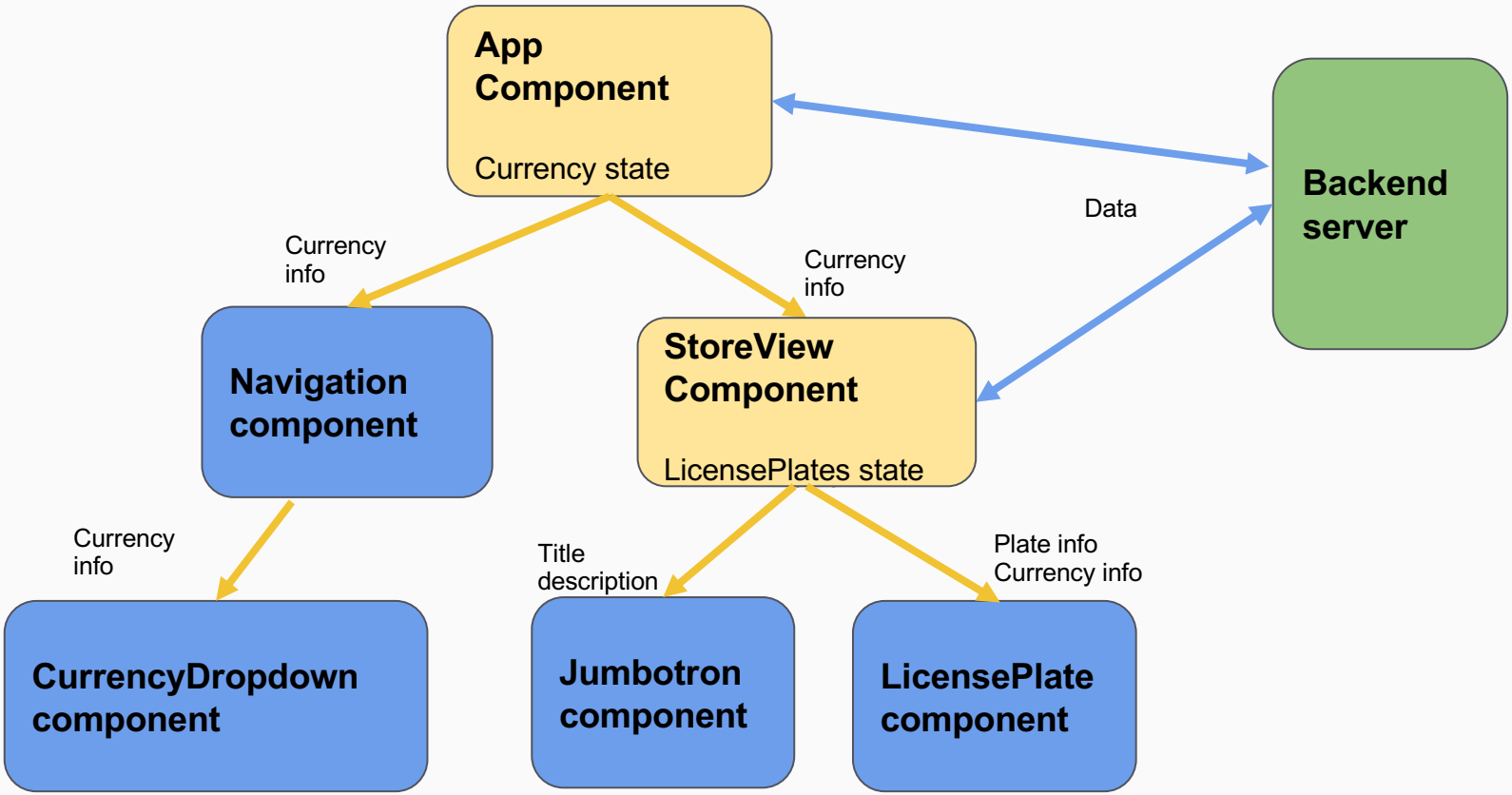
Lab 14 - Component Communication - Lifting state up

Make use of the **`CurrencyDropdown`** in our webapp header:



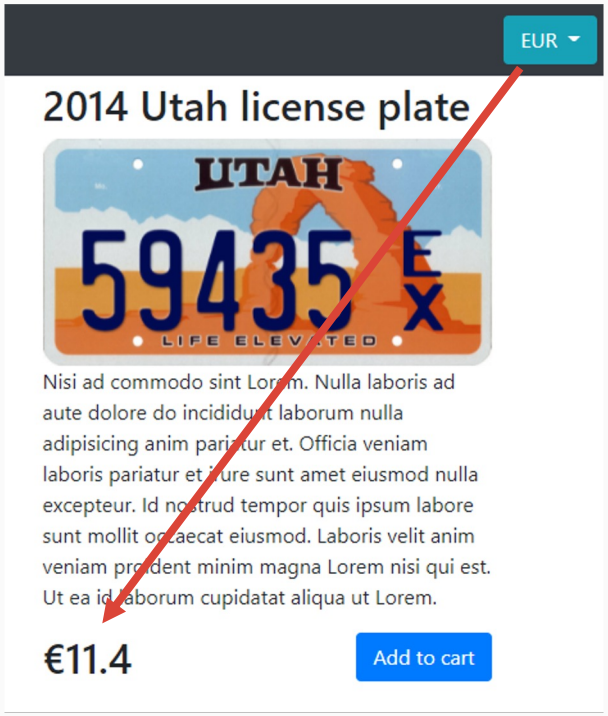
Update our application so that when a user selects a new currency with that component, the change is propagated to the state of the **`App`** component.

Component Architecture of our License Plate store



Lab 15 - Component communication

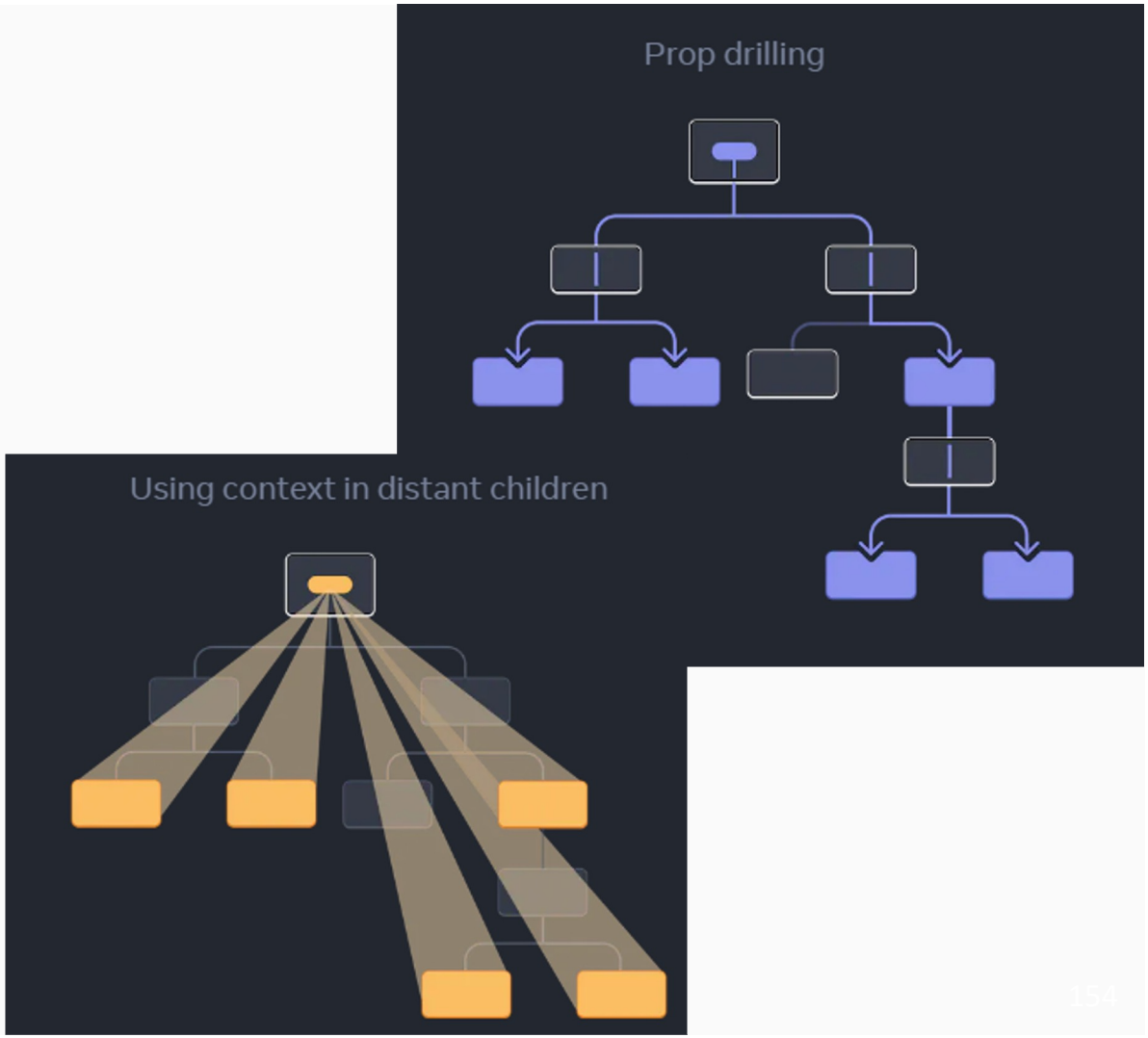
Complete the propagation of our state update so that the currency is displayed by each **`LicensePlateComponent`**



Using context as an alternative to passing props

Context lets a parent component provide data to the entire tree below it.

Components that need data from that context can just request is using the **useContext** hook



Context example

Here the **CurrencyContext** allows us to access the current currency value in any descendent of **CurrencyContext.Provider** in the component hierarchy.

1) We create the context

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

export const CurrencyContext = createContext("USD");
```

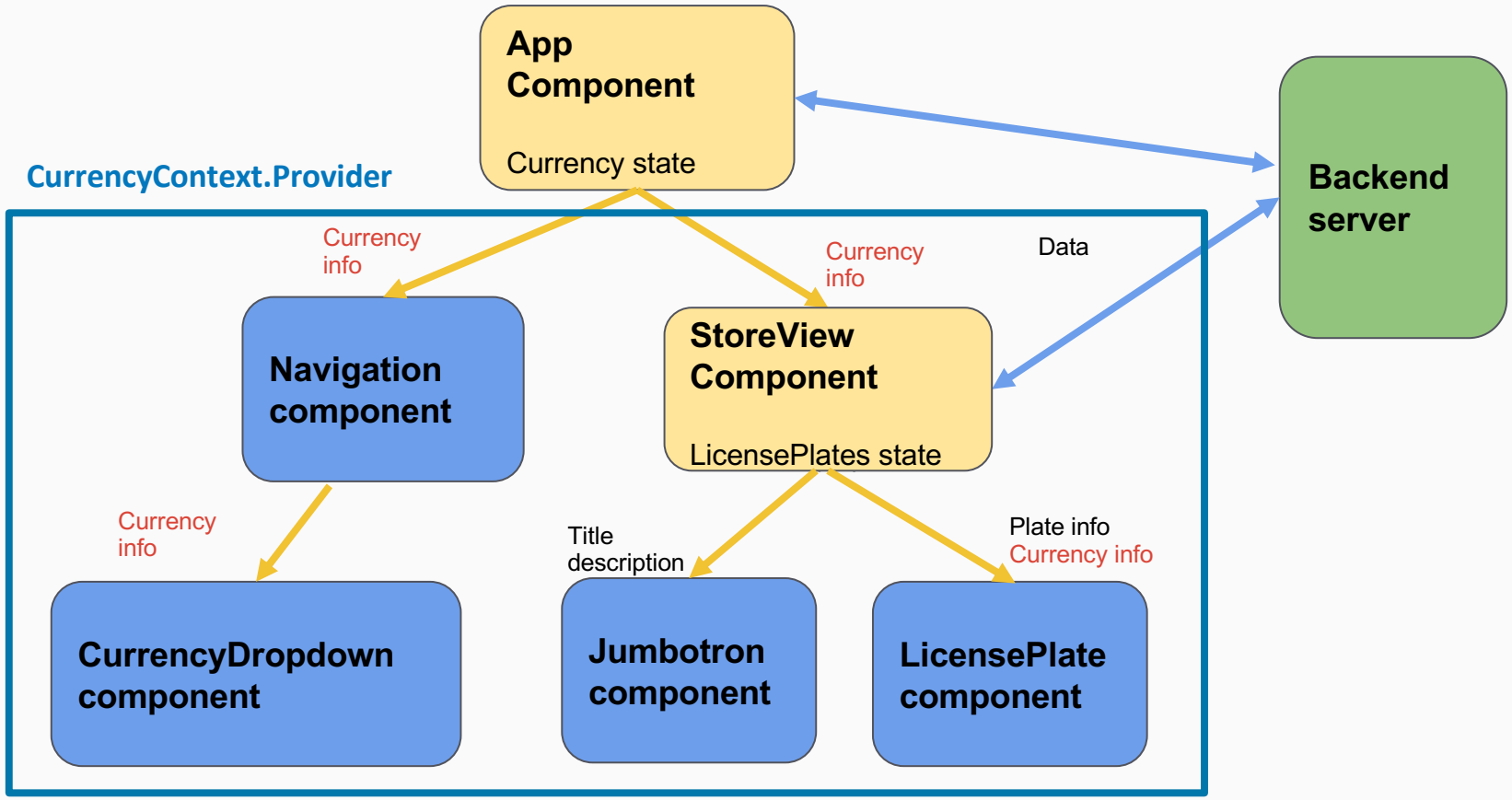
2) We wrap the component tree that needs that context in a Provider component

```
<CurrencyContext.Provider value="USD">
  <Section />
</CurrencyContext.Provider>
```

3) We use the **useContext** hook to read information from that context

```
export default function Content() {
  const currency = useContext(CurrencyContext);
  return <span>{currency} 21</span>;
}
```

License Plate store with Context



Custom Hooks

Another way to improve our component architecture is to refactor common features into **custom hooks**.

A custom Hook is a JavaScript function whose name starts with **”use”** and that may call other Hooks.

Unlike a React component, a custom Hook doesn’t need to have a specific signature. We can decide what it takes as arguments, and what, if anything, it should return.

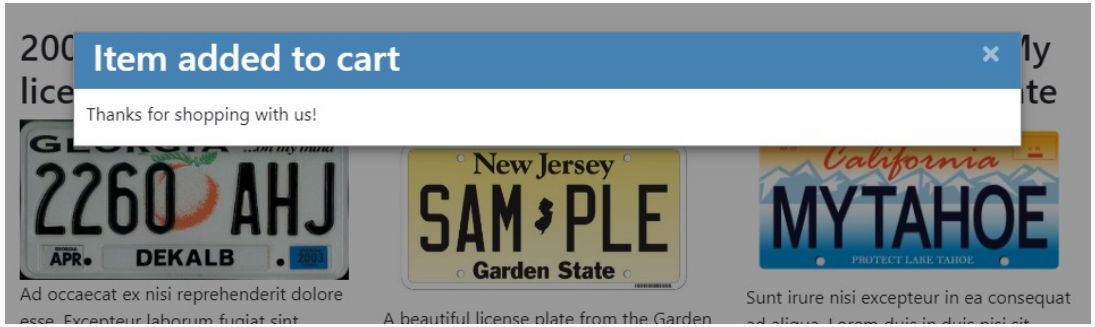
Custom Hook example

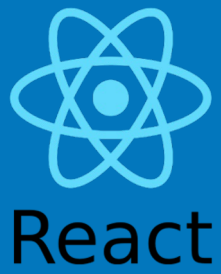
- Here we wrap all cart features into a single hook that exposes the current state as well as utility update functions:

```
export function useCart() {
    const [cartContents, setCartContents] = useState([]);

    useEffect(() => {
        refreshCart();
    }, []);

    const refreshCart = () => getCartContents().then(data => setCartContents(data));

    const addPlateToCart = async (plate) => {...};

    const removePlateFromCart = async (plate) => {...};

    return [cartContents, addPlateToCart, removePlateFromCart];
}
```

Lab 16 - “Add to cart” Feature with a Hook

- Use the cart-hook.ts hook features to add a license plate to the user’s cart.
- When the API returns successfully, display a PopupWindow to confirm success





Chapter 15:
Useful React Hooks

Performance issues with function components

- The following function component filters its data using a function, which works but can be problematic. Can you tell why?

```
function TodoList(props) {
    const visibleTodos = filterTodos(props.todos);
    // ...
}
```

- Function components run with every render, which means that **filterTodos** might run multiple times for no good reason.

useMemo Hook

- The solution is to cache the result of that function and only call it again if the **todos** change. useMemo is a hook that does exactly that. It uses an array of dependencies like **useEffect**:

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

function TodoList(props) {
    const visibleTodos = useMemo(
        () => filterTodos(props.todos),
        [props.todos]
    );
    // ...
}
```

Performance issues with function components

- The following function component filters isn't ideal either. Can you tell why?

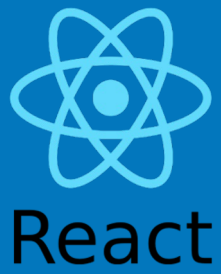
```
function ProductPage({ productId, referrer }) {
   const handleSubmit = (orderDetails) => {
       post('/product/' + productId + '/buy', {referrer, orderDetails});
   };
       // ...
}
```

- Function components run with every render, which means that **handleSubmit** will be created and re-created many times for no good reason.

useCallback Hook

- The solution is to create the function once and re-create it only if its dependencies change.
- **useCallback** is a hook that does exactly that:

```
function ProductPage({ productId, referrer}) {
    const handleSubmit = useCallback((orderDetails) => {
        post('/product/' + productId + '/buy', {referrer, orderDetails});
    }, [productId, referrer]);

}
```



Chapter 16:
Unit testing React code

Jest

Jest is an open source testing framework for JavaScript.

Jest aims to be easy to read: Code can almost read just like plain English sentences

https://jestjs.io

Example of code

```
function helloWorld() {
 return 'Hello world!';
}
```

Jest test for that code

```
describe('Hello world test', function () {
 test('says hello', function () {
   expect(helloWorld()).toEqual('Hello world!');
 });
});
```

Jest: Describe

The describe function is for grouping related specs (tests), typically each test file has one at the top level.

The string parameter is for naming the collection of specs, and will be concatenated with specs to make a spec's full name.

```
describe('Hello world test', function () {

 test('says hello', function () {
   expect(helloWorld()).toEqual('Hello world!');
 });

});
```

Jest: test()

Specs are defined by calling the **`test`** (or **`it`**) function, which takes two parameters:

The string is the title of the spec and the function is the test itself.

A spec contains one or more expectations that test the state of the code.

```
describe('Hello world test', function () {

  test('says hello', function () {
    expect(helloWorld()).toEqual('Hello world!');
  });

});
```

Expectations and matchers

Expectations are built with the function **`expect`** which takes a value, called the actual.

It is chained with a Matcher function, which takes the expected value.

```
expect(helloWorld()).toEqual('Hello world');

expect(1+1).toBe(2);

expect(false).not.toBe(true);

expect([1, 2, 3]).toContain(2);

expect(2).toBeLessThan(3);
```

React Testing Library

A library that adds APIS for working with React components.

We can instantiate a component with the *render* method.

Then we can query the resulting DOM to check its contents.

```
import React from "react";
import {Jumbotron} from "./Jumbotron";
import {render} from "@testing-library/react";

test('renders Jumbotron with the proper title',() => {
   const { getByText } = render(
         <Jumbotron title="Welcome to our store"/>
   );
   const titleElement = getByText("Welcome… ");
   expect(titleElement).toBeInTheDocument();
});
```

https://testing-library.com

React Testing Library

Testing Library allows us to fire events such as clicks, wait for an element to be present on the screen, and more.

As a result, we can simulate user actions and test their expected result.

```
import {render, rerender, fireEvent, screen, waitFor}
   from "@testing-library/react";

let curr = "USD";
let { getByText, getByRole, rerender } = render(
  <CurrencyDropdown currency={curr}
                    onCurrencyChange={(c) => curr = c} />
);
const button = getByRole("button");
fireEvent.click(button);
await waitFor(() => expect(
     getByText('EUR (€)')).toBeInTheDocument()
);
fireEvent.click(getByText("EUR (€)"));
rerender(
  <CurrencyDropdown currency={curr}
                    onCurrencyChange={(c) => curr = c} />
);
expect(getByRole("button").textContent).toBe("EUR");
```

Mocking and stubbing

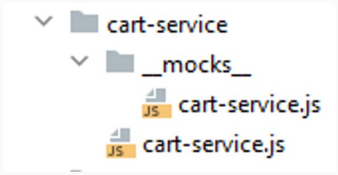
Manual mocks are used to stub out functionality with mock data.

Instead of accessing a remote resource (API, database), we create a manual mock that allows us to use fake data.

This ensures our tests will be fast and not flaky.

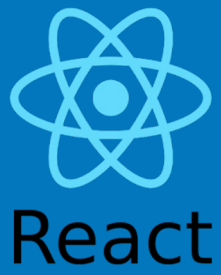
```
// Mocks a dependency in node_modules

jest.mock('jquery');


// Mocks a specific module in our code

jest.mock('../cart-service/cart-service');
```

The above example requires a `__mocks__` folder with the mocked data in the same folder as the mocked object:



Test Driven Development (TDD)

- Most unit-testing libraries promote the idea of "first testing, then coding", which emphasizes on setting up the test data for a piece of code that can be tested first and then implemented.
- This approach is like "test a little, code a little, test a little, code a little."
- Increases developer productivity and code stability, which in turn reduces development stress and time spent on debugging.
- A very powerful approach to software development as developers have to think about edge cases and complex scenarios first, which results in a better architecture/application right from the start.



Chapter 17:
Redux and State Management

What is state management?

- We saw that React relies on state and props to pass information.
- While this works, it can become really verbose to pass props data up and down our component tree.
- State management with Redux is an alternative solution that relies on using a **global application state** (not to be confused with our React components state) that components can access and update using well-defined patterns and APIs.

Redux: Core Concepts

- **Redux** (http://redux.js.org) relies on simple core concepts and principles to manage the state of any application:
  - The entire state of a web application should be described as a single, plain object
  - When the state needs to change, an action has to be dispatched
  - A function called reducer is in charge of handling the action to transition the app from state A to state B

Redux: Three Principles

- **Single source of truth:** The state of the app is stored in an object tree within a single store
- **State is read-only:** The only way to change the state is to emit an action, an object describing what happened
- **Changes are made with pure functions:** Reducers are pure functions that take the previous state and an action, and return the next state. They return new state objects instead of mutating the previous state.

THE END