

React Development

<https://olsensoft.com/react-course>



Contents

1. HTML Essentials
2. CSS Essentials
3. JavaScript Essentials
4. Getting Started with React
5. Components
6. JSX
7. State Management

Contents / Continued

8. Creating a Complete React Application
9. Introduction to Routing
10. User Input Techniques
11. Routing Techniques
12. Component Techniques
13. React Redux
14. Redux Saga
15. Testing a React Application

Appendices

- A. Modern ECMAScript
- B. TypeScript Essentials
- C. Custom Hooks

HTML Essentials

1. Introduction to HTML
2. Defining Simple Content
3. Additional Types of Content
4. Displaying a Form and Getting User Input

Section 1: Introduction to HTML

- Structure of an HTML document
- Understanding HTML tags
- Understanding HTML attributes

Structure of an HTML Document

- Here's the overall structure of an HTML document:

```
<html>
  <head>
    <title>This is my simple page</title>
    <meta charset="UTF-8" />
    <meta name="author" content="John Smith" />
  </head>

  <body>
    <div>Here is some content</div>
    <div>And some more content</div>
  </body>
</html>
```

Metadata

Content to display

- Example: 01-`HtmlStructure.html`

Understanding HTML Tags

- An HTML document comprises many tags (elements)
- Some tags enclose content
 - Comprise a **start tag**, **content**, and **end tag**:
- Some tags are empty
 - Just comprise a **start tag**:

```
<p>Here is some content</p>
```

```
<br>
```

Understanding HTML Attributes

- Tags can contain attributes
 - Additional info, defined in the start tag
- Attribute syntax:
 - attributeName="attributeValue"
- E.g. it's common to define an id attribute
 - Uniquely identifies a tag in a document

```
<p id="messageArea">Some message</p>
```

Section 2: Defining Simple Content

- Displaying section headings
- Displaying block-level content
- Displaying inline content
- Styling content

Displaying Section Headings

- HTML defines six section heading tags, to help you organize your page content into sections:

```
<h1>This is an h1</h1>
<h2>This is an h2</h2>
<h3>This is an h3</h3>
<h4>This is an h4</h4>
<h5>This is an h5</h5>
<h6>This is an h6</h6>
```

- Example:
 - 02-SectionHeadings.html

Displaying Block-Level Content

- You can display block content using `<div>` or `<p>`
 - `<div>` tags have no inherent styling
 - `<p>` tags have a blank line between them

```
<div>This is a div</div>
<div>This is another div</div>
<div>This is yet another div</div>

<p>This is a p</p>
<p>This is another p</p>
<p>This is yet another p</p>
```

- Example:
 - `03-BlockContent.html`

Displaying Inline Content

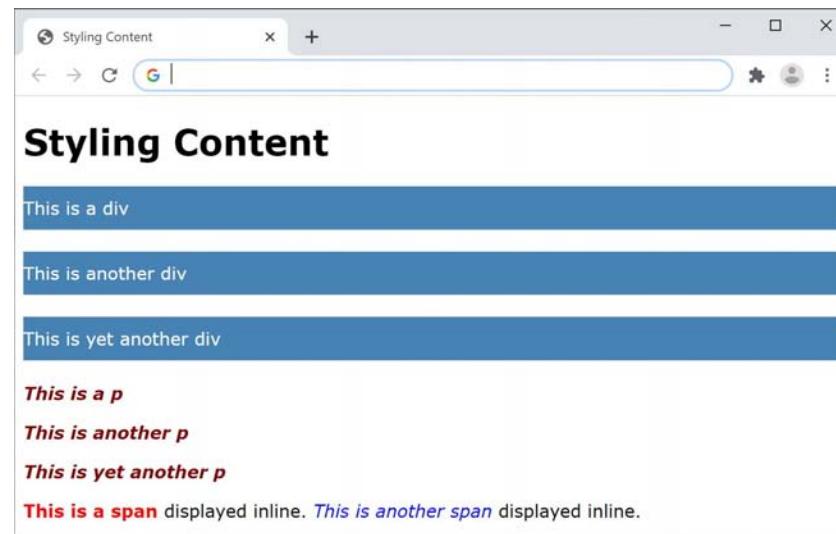
- You can display inline content using ``
 - `` tags have no inherent styling, they just demarcate the start/end of a piece of text

```
<span>This is a span</span> displayed inline.  
<span>This is another span</span> displayed inline.
```

- Example:
 - `04-InlineContent.html`

Styling Content

- You can use Cascading Style Sheets (CSS) to style tags
 - Example: 05-StylingContent.html



- We discuss CSS in more detail later...

Section 3: Additional Types of Content

- Displaying images
- Displaying hyperlinks
- Displaying an unordered list
- Displaying an ordered list
- Displaying a table
- Styling a table
- Defining a table header, body, and footer

Displaying Images

- HTML documents can contain images:

```

```

- You can specify a relative or absolute URL:

```

```

```

```

- Example:
 - 06-DisplayingImages.html

Displaying Hyperlinks

- HTML documents can contain hyperlinks:

```
<a href="aLocation">text for the hyperlink</a>
```

- You can specify a relative or absolute URL:

```
<a href="resources/page1.html">page 1</a>
```

```
<a href="https://www.olsensoft.com/">Olsen Software</a>
```

- Example:

- 07-DisplayingHyperlinks.html

Displaying an Unordered List

- Use the `` tag to display an unordered list
 - Represent each item with an `` tag:

```
<ul>
  <li>St. Anton</li>
  <li>Kitzb&uuml;hel</li>
  <li>Val d'Is&egrave;re</li>
  <li>Meribel</li>
  <li>Villars</li>
</ul>
```

- St. Anton
- Kitzbühel
- Val d'Isère
- Meribel
- Villars

- Example:
 - `08-UnorderedLists.html`

Displaying an Ordered List

- Use the `` tag to display an ordered list of items
 - Represent each item with an `` tag:

```
<ol>
  <li>St. Anton</li>
  <li>Kitzb&uuml;hel</li>
  <li>Val d'Is&egrave;re</li>
  <li>Meribel</li>
  <li>Villars</li>
</ol>
```

```
1. St. Anton
2. Kitzb&uuml;hel
3. Val d'Is&egrave;re
4. Meribel
5. Villars
```

- Example:
 - `09-OrderedLists.html`

Displaying a Table

- Use the `<table>` tag to display a table:

```
<table>
  <tr>
    <td>Ola</td> <td>M</td> <td>25</td>
  </tr>
  <tr>
    <td>Kari</td> <td>F</td> <td>24</td>
  </tr>
</table>
```

- In the table, use `<tr>` to represent a row
- In a row, use `<td>` to represent a column
- Example: `10-Tables.html`

Styling a Table

- To style a table, you should use CSS styles
 - Don't use deprecated attributes (e.g. width, valign)
- Example: 11-TableStyles.html

```
<style>
  table {
    width: 25%;
    background-color: lightgrey;
  }

  td {
    border: 1px solid white;
  }
</style>
```

Defining a Table Header, Body, and Footer

- You can group rows into a header, body, and footer
 - Via `<thead>`, `<tbody>`, `<tfoot>`
 - Enables you to apply different CSS styles for each part
- Example: `12-TableHeaderBodyFooter.html`

```
<table>
  <thead> ...
  <tbody> ...
  <tfoot> ...
</table>
```

Section 4: Displaying a Form and Getting User Input

- Overview of HTML forms
- Adding input controls to a form
- Getting text input
- Getting user selection
- Selecting option(s) from a list
- Submitting or resetting form data
- Form example

Overview of HTML Forms

- An HTML <form> contains input controls, to gather input from the user:

```
<form ... >  
    ... input controls ...  
    ... button to submit data to server application ...  
</form>
```

Adding Input Controls to a Form

- A form can contain many types of input control
 - Text boxes, check boxes, radio buttons, lists, buttons, etc.
- To add an input control in a form:

```
<form ... >
  <label for="fullname">Enter full name:</label>
  <input type="text" id="fullname" name="fullname">
  ...
</form>
```

- type - type of input control (e.g. text)
- id - unique id for the element
- name - name of param to pass to server

Getting Text Input

- To get text input from the user:

```
<input type="text" name="fullname" size="25"  
      value="some initial value">
```

- To get a password from the user:

```
<input type="password" name="password" size="25">
```

- To get multi-line text from the user:

```
<textarea name="bio" rows="4" cols="25">  
  Some initial multi-line value  
</textarea>
```

Getting User Selection

- To enable the user to choose a single option:

```
<input type="checkbox" name="salaried" value="yes">
```

- To enable the user to choose an option from a group of radio buttons:

```
<input type="radio" name="emptype" id="emptype1">
<label for="emptype1">Full-time</label>

<input type="radio" name="emptype" id="emptype2">
<label for="emptype2">Part-time</label>

<input type="radio" name="emptype" id="emptype3">
<label for="emptype3">Contractor</label>
```

Selecting Option(s) from a List

- To enable the user to select option(s) from a list:

```
<select name="languages" multiple size="5">
  <option value="">--Choose an option--</option>
  <option value="1">English</option>
  <option value="2">Mandarin Chinese</option>
  <option value="3">Hindi</option>
  <option value="4">Spanish</option>
  <option value="5">French</option>
  <option value="6">Arabic</option>
  <option value="7">Bengali</option>
  <option value="8">Russian</option>
</select>
```

Submitting or Resetting Form Data

- A form typically has a "submit" button:

```
<form ... >
  <input type="submit" value="Submit!" >
  ...
</form>
```

- A form can also have a "reset" button:

```
<form ... >
  <input type="reset" value="Reset!" >
  ...
</form>
```

Form Example

- 13-FormClient.html is a web page that lets a user enter data and submit to the server
- server.py is a Python REST service that stores employee data; you can run it as follows:

```
pip install flask
pip install flask_restful
python server.py
```

Summary

- Introduction to HTML
- Defining Simple Content
- Additional Types of Content
- Displaying a Form and Getting User Input

CSS Essentials

1. Introduction to CSS
2. Understanding CSS selectors
3. Setting common style properties
4. Understanding the CSS box model

Section 1: Introduction to CSS

- Overview of Cascading Style Sheets
- Defining `<style>` elements
- A closer look at style rules
- Linking to an external style sheet

Overview of Cascading Style Sheets

- Cascading Style Sheets (CSS) defines styles for HTML content
- Here's a simple example with inline styles
 - Set the `style` attribute on an element

```
<h1 style="color: red">  
    Here's an h1 element  
</h1>  
  
<p style="color: blue">  
    Here's a p element  
</p>
```

01-Inlinestyles.html

Here's an h1 element

Here's a p element

Defining <style> Elements

- Rather than defining inline styles, a more reusable approach is to define style rules in a <style> element
 - Put the <style> element in the HTML <head> section
- Example:

```
<style>
  h1 {
    color: red;
  }

  p {
    color: blue;
  }
</style>
```

02-styleElements.html

Here's an h1 element
Here's another h1 element
Here's a p element
Here's another p element

A Closer Look at Style Rules

- Each style rule has a { } declaration block
 - The declaration block contains a series of declarations
 - Each declaration is a *property:value* pair terminated by ;
- Example:

```
h1 {  
  color: red;  
  font-style: italic;  
}  
  
p {  
  color: blue;  
  margin-bottom: 10px;  
  border: lightblue solid 5px;  
}
```

03-StyleRules.html

Here's an h1 element
Here's another h1 element
Here's a p element
Here's another p element

Linking to an External Style Sheet

- If you want to define common styles for many web pages:
 - Define the style rules in a .css style sheet file

```
body {  
    font-family: consolas, sans-serif;  
}  
  
h1 {  
    color: red;  
    font-style: italic;  
}
```

MyStylesheet1.css

- To link a web page to the CSS style sheet file:
 - Use a <link> tag

```
<link rel="stylesheet" type="text/css" href="MyStylesheet1.css">
```

Section 2: Understanding CSS Selectors

- Introduction to CSS selectors
- Selecting all elements
- Element selectors
- Class selectors
- ID selectors

Introduction to CSS Selectors

- Each style rule specifies a CSS selector
 - The selector defines which parts of the HTML document will be affected by the declarations

```
css-selector {  
    declarations...  
}
```

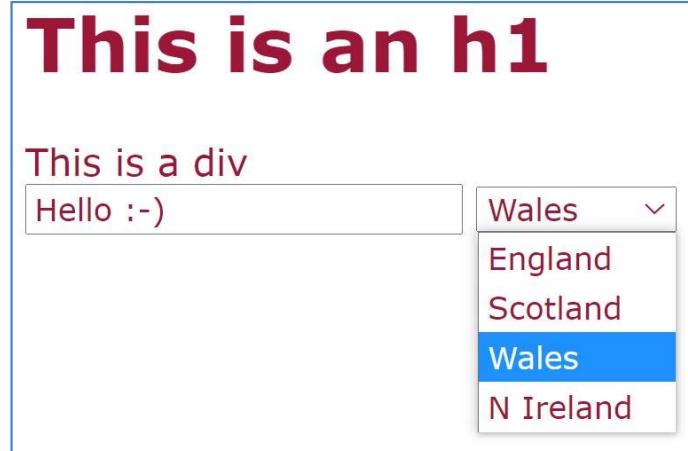
- There are several types of CSS selector:
 - Element selectors
 - Class selectors
 - ID selectors

Selecting All Elements

- You can define a rule that selects all elements

```
* {  
    font-family: verdana;  
    color: #9c1738;  
}
```

05-AllElementsSelector.html



Element Selectors

- Element selectors are the simplest kind of selector
 - Also known as tag selectors
 - Select all elements that have a particular tag name
- We've seen an example like this earlier

```
/* Applies to all <div> elements */
div {
  color: red;
}

/* Applies to all <p> elements */
p {
  color: blue;
}
```

Class Selectors

- A class selector applies to all elements in the document that have a particular CSS class
 - Lets you apply the same style to different kinds of element
- Example:

```
/* Applies to elements that have "optional" class */
.optional {
    border: 1px solid lightblue;
}

/* Applies to elements that have "required" class */
.required {
    border: 1px solid pink;
}
```

06-ClassSelectors.html

ID Selectors

- An ID selector applies to an element with a specified id
 - Targets a specific single element in the web page
- Example:

```
#mainContent {  
    color: blue;  
    background-color: #eeeeff;  
}  
  
#additionalContent {  
    color: red;  
    background-color: #ffeeee;  
}
```

07-IdSelectors.html

Section 3: Setting Common Style Properties

- Setting the background color
- Setting text color
- Aligning text horizontally
- Setting the font family
- Setting the font size
- Setting the font weight
- Setting the font italicization

Setting the Background Color

- To set the background color for an element:

```
aSelector {  
    background-color: aColor;  
}
```

- Colors are most often specified by:
 - Hexadecimal value, e.g. #fc7276
 - RGB value, e.g. rgb(252,186,3)
 - Keyword color name, e.g. green
- See 08-BackgroundColor.html

Setting Text Color

- To set the text color for an element:

```
aSelector {  
    color: aColor;  
}
```

- Colors are set as per background-color:
 - Hexadecimal value, e.g. #fc7276
 - RGB value, e.g. rgb(252,186,3)
 - Keyword color name, e.g. green
- See 09-TextColor.html

Aligning Text Horizontally

- You can specify horizontal text alignment in an element:

```
aSelector {  
    text-align: aTextAlignOption;  
}
```

- Valid values for `text-align`:

```
left, center, right, justify, start, end
```

- See `10-TextAlignment.html`

Setting the Font Family

- To set the font family for text in an element:

```
aSelector {  
    font-family: preferredFontFamily, nextBestFontFamily, ... fallbackFontFamily;  
}
```

- Generic families, such as:
 - serif, sans-serif, monospace
- Specific family names, such as:
 - Arial, Courier New
- See 11-FontFamily.html

Setting the Font Size

- To set the font size for text in an element:

```
aSelector {  
    font-size: aFontSize; /* Length, percent, relative size, or absolute size */  
}
```

- Relative sizes:

```
smaller, larger
```

- Absolute sizes:

```
xx-small, x-small, small, medium,  
large, x-large, xx-large, xxx-large
```

- See 12-FontSize.html

Setting the Font Weight

- To set the font weight (boldness) for text in an element:

```
aSelector {  
    font-weight: aFontWeight; /* Relative weight, absolute weight, or number */  
}
```

- Relative weights: `lighter, bolder`
- Absolute weights: `normal, bold`
- Numbers:
`100, 200, 300,
400 (same as normal),
500, 600, 700 (same as bold),
800, 900`
- See `13-FontWeight.html`

Setting the Font Italicization

- To set the font italicization for text in an element:

```
aSelector {  
    font-style: aFontStyle;  
}
```

- Valid values for `font-style`:

```
normal, italic, oblique, oblique <angle>
```

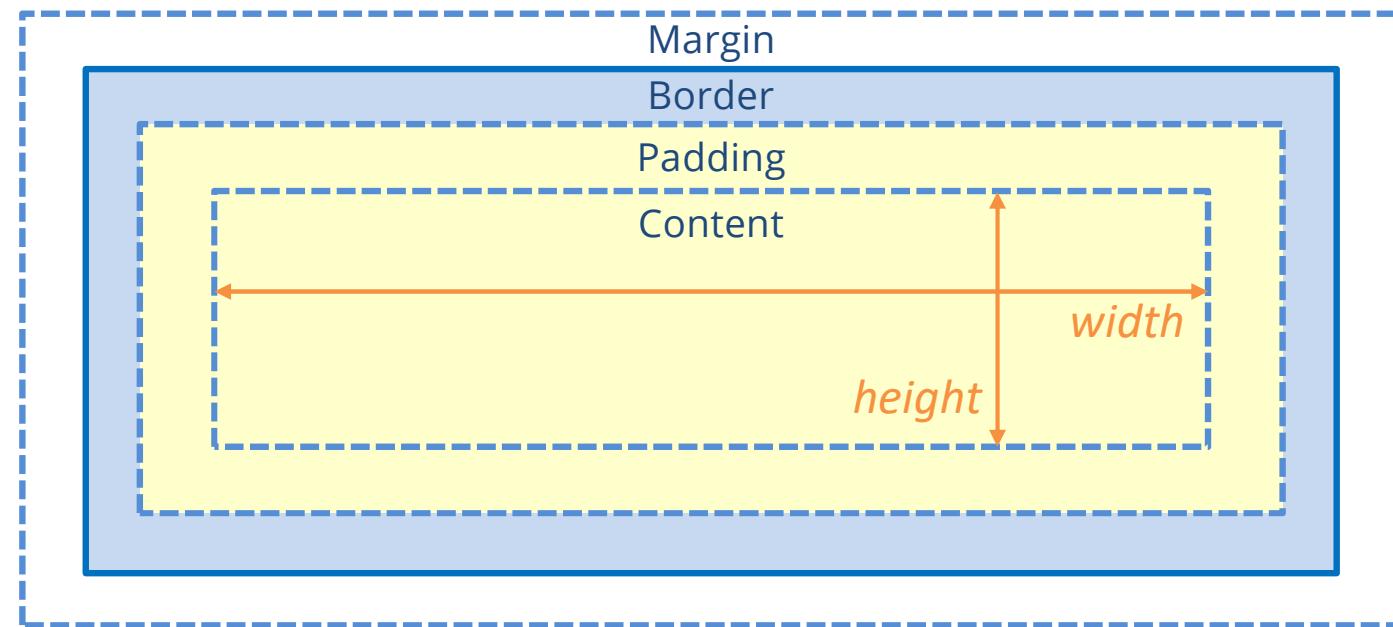
- See 14-FontItalicization.html

Section 4: Understanding the CSS Box Model

- Overview
- Setting the margin and padding
- Setting the border

Overview

- You can consider all HTML elements as boxes
 - Useful for designing layout & relative positioning of elements



Setting the Margin and Padding

- To set the margin and padding sizes:

```
aSelector {  
    margin: aMarginSize;  
    padding: aPaddingSize;  
}
```

- 1 value => applies to all boundaries
- 2 values => top/bottom and left/right
- 3 values => top, left/right, and bottom
- 4 values => top, right, bottom, left

- Valid sizes:
 - Length as a fixed value, e.g. 5px, 2em
 - % relative to width of containing block
 - auto (for margin), browser decides
- See 15-MarginPadding.html

Setting the Border

- You can set the following properties to define a border:

```
aSelector {  
    border-style: aBorderStyle;  
    border-width: aBorderWidth;  
    border-color: aColor;  
}
```

- 1 value => applies to all boundaries
- 2 values => top/bottom and left/right
- 3 values => top, left/right, and bottom
- 4 values => top, right, bottom, left

Style

none, hidden, solid, dotted, dashed, double,
groove, ridge, inset, outset

Width

thin, medium, thick (or numeric, e.g. 10px)

Color

(same as for color and background-color)

- See 16-Border.html

Summary

- Introduction to CSS
- Understanding CSS selectors
- Setting common style properties
- Understanding the CSS box model

JavaScript Essentials

1. Introduction to JavaScript
2. The JavaScript Language
3. The Document Object Model (DOM)

Section 1: Introduction to JavaScript

- Overview of JavaScript
- Defining JavaScript code
- Where to put the <script> tag

Overview of JavaScript

- JavaScript is a scripting language
 - Can run in a browser or at the server (Node.js)
- Characteristics of JavaScript:
 - Interpreted
 - Dynamically typed
- Typical uses of JavaScript in a web page:
 - To generate dynamic HTML content
 - To invoke REST services that are running on the server

Defining JavaScript Code

- You can define JavaScript inline in an HTML page
 - Add a `<script>` tag, with JavaScript code inside

```
<script>
    console.log('Hello world');
</script>
```

SomePage.html

- Alternatively, put JavaScript code in a separate JS file
 - Then add into an HTML page as follows:

```
<script src="SomeScript.js"></script>
```

SomePage.html

```
console.log('Hello world');
```

SomeScript.js

Where to put the <script> Tag

- You typically put the <script> tag in the <head>
 - The browser downloads the script first (handy for libraries that don't depend on the body content)
 - Then it parses the <body> content afterwards
- Or put the <script> tag at the bottom of the <body>
 - The browser parses the <body> content first
 - Then it downloads the script afterwards (handy if the script needs to access body content)
 - (Note: the `defer` property has the same effect)

Section 2: The JavaScript Language

- Core syntax
- Functions
- Literal objects
- Handling events

Core Syntax

- JavaScript core syntax is very similar to other languages
 - E.g., Java, C++, C#
- See examples in this demo folder:
 - 03-JsEssentials/CoreSyntax/

Functions

- To define a function:
 - Use the `function` keyword
 - Define parameters (i.e. inputs) in brackets, `()`
 - Implement the function body in braces, `{ }`

```
function add(num1, num2) {  
    return num1 + num2;  
}
```

- To call a function:

```
let result = add(100, 200);
```

[01-Functions.html](#)

Literal Objects

- You can define a literal object
 - The object can contain data properties and functions

```
let person1 = {
    name: 'Ola',
    age: 21,
    nationality: 'Norsk',

    toString() {
        return `${this.name}, ${this.age}, ${this.nationality}`
    }
};
```

- To use a literal object:

```
person1.age++;
person1.nationality = person1.nationality.toUpperCase();
console.log(person1.toString());
```

[03-Functions.html](#)

Handling Events (1 of 2)

- Every element on a web page has certain events that can trigger JavaScript code
- For example:
 - click
 - keydown, keyup
 - change
 - load, unload
 - submit, reset

Handling Events (2 of 2)

- The easiest way to handle an event is as follows:

```
<script>
    function clickHandler() {
        console.log("You clicked the button");
    }
</script>

<button onclick="clickHandler()">Click me</button>
```

[03-HandlingEvents.html](#)

Section 3: The Document Object Model (DOM)

- Overview
- Understanding the DOM
- Accessing elements in the DOM
- Accessing content
- Navigating the DOM tree
- Modifying the DOM tree

Overview

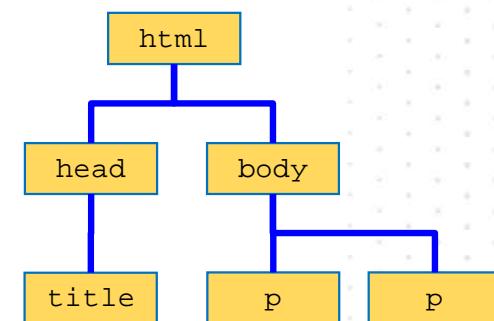
- JavaScript provides a set of standard objects that allow you to manipulate windows and documents
- HTML Document Object Model (DOM) objects:
 - Provide access to the current document (i.e. Web page)
- Browser objects:
 - Provide access to the browser environment

Understanding the DOM

- Consider this simple HTML document

```
<html>
  <head>
    <title>This is my simple page</title>
  </head>
  <body>
    <p>This is the first para</p>
    <p>This is another para</p>
  </body>
</html>
```

- The browser parses the HTML and creates an object tree in memory
 - The Document Object Model (DOM)



Accessing Elements in the DOM

- There are several ways to access elements in the DOM...
- Get an element by id:

```
let elem = document.getElementById(id);
```

- Get the first element that matches a CSS selector:

```
let elem = document.querySelector(cssSelector);
```

- Get all elements that match a CSS selector:

```
let elem = document.querySelectorAll(cssSelector);
```

Accessing Content

- You can get/set the text content of an element:

```
anElem.textContent = "Some text";
console.log(anElem.textContent);
```

- You can get/set the value of an input field:

```
anInputField.value = "Some value";
console.log(anInputField.value);
```

- Example:
 - [04-AccessingContent.html](#)

Navigating the DOM Tree

- All DOM objects provide a set of properties and methods that allow you to navigate DOM like a tree

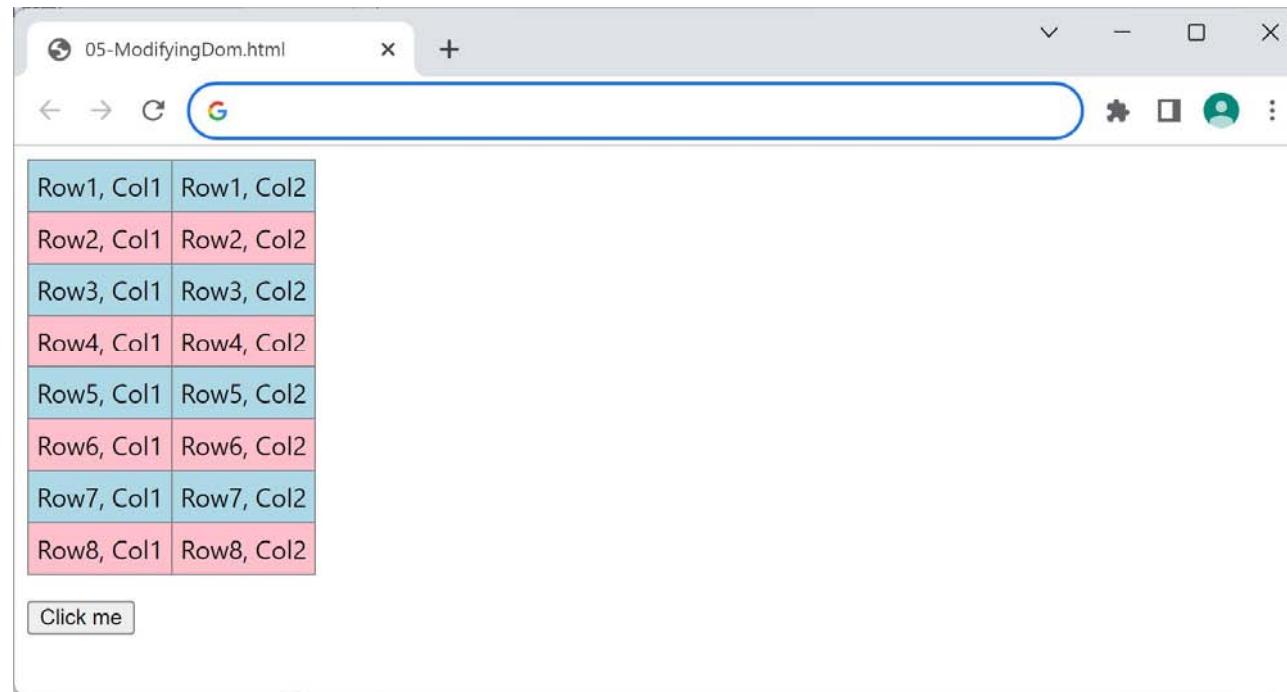
Property	Description
childNodes	Returns the set of child elements
firstChild	Returns the first child element
hasChildNodes()	Returns true if the current element has child elements
lastChild	Returns the last child element
nextSibling	Returns the sibling element defined after the current element
parentNode	Returns the parent element
previousSibling	Returns the sibling element defined before the current element

Modifying the DOM Tree (1 of 2)

- To create a new element in a document:
 - Call `document.createElement()`
- To add/remove nodes in a document:
 - Call `insertBefore()` to insert a node before current node
 - Call `appendChild()` to append a child node to current node
 - Call `removeChild()` to remove a child from current node
- To get/set the HTML text for a node:
 - Use `innerHTML` to get/set the inner HTML of an element
 - Use `outerHTML` to get/set the outer HTML of an element

Modifying the DOM Tree (2 of 2)

- For an example of how to modify the DOM tree, see:
 - 05-ModifyingDom.html



Summary

- Introduction to JavaScript
- The JavaScript Language
- The Document Object Model (DOM)

Getting Started with React

1. Overview of React
2. Creating a simple React app
3. Creating many elements
4. Data-driven UIs

Section 1: Overview of React

- What is React?
- Characteristics of React
- Tooling up

What is React?

- React is a lightweight client-side library from Facebook, to help you develop large-scale web apps
 - Gives you a logical way to construct your code
 - Simplifies UI management
 - Simplifies state management
- React allows you to create apps for:
 - Web browsers (we'll cover this)
 - iOS and Android native apps (via React Native)

Characteristics of React

- React is a relatively small library
 - E.g. much smaller and simpler than Angular
- React is very lean and flexible - it supports features such as:
 - Custom widget libraries
 - Modular CSS stylesheets
 - Testing
 - Etc.

Tooling Up

- You can use any IDE you like to develop React apps
 - E.g. Visual Studio Code
 - E.g. JetBrains WebStorm
 - E.g. React Studio (primarily for graphics designers)
- Most browsers also include handy extensions, to help you view React elements in your web page
 - E.g. React Developer Tools for Chrome
 - E.g. React Developer Tools for Firefox

Managing React Packages

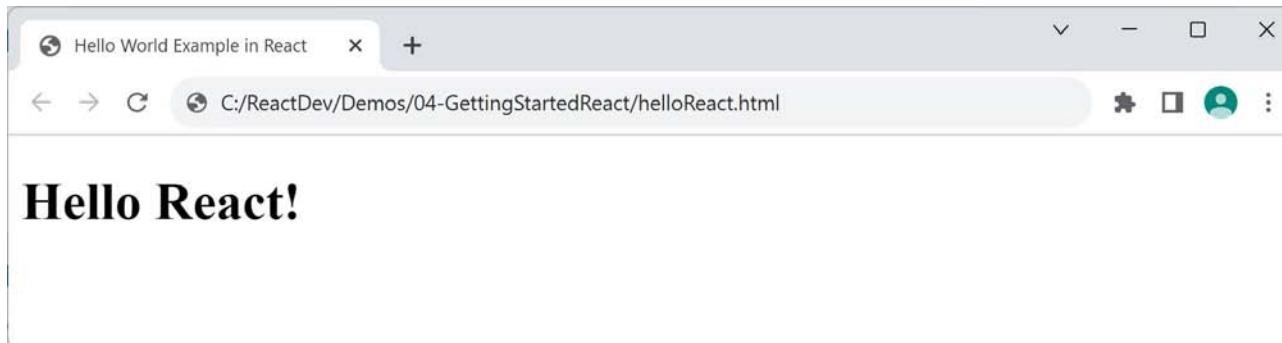
- React has quite a small number of libraries
 - You need to manage these libraries in your application
 - E.g. using Node Package Manager (npm) or Yarn
- To get started, we'll download React libraries directly from <https://unpkg.com/>
 - This is the CDN site for Node Package Manager libraries

Section 2: Creating a Simple App

- Scenario
- Defining an HTML target element
- Including React libraries
- The virtual DOM
- Creating React elements
- Rendering React elements

Scenario

- In this chapter we show how to create simple React apps in pure React
 - See folder `ReactDev/Demos/04-GettingStarted`
- Our first example is `helloReact.html`
 - It's a simple "Hello World" React app 😊



Defining an HTML Target Element

- A React web app has a single top-level HTML element into which React will render the UI
- You typically define it like this

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

 - Give it a suitable id
 - You'll refer to this id when you render content (see later)

Including React Libraries

- To use React in a web page, you need 2 libraries:
 - React - Creates views
 - ReactDOM - Renders views in the web browser
- The following code downloads these libraries directly:

```
<script src="https://unpkg.com/react@18/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>
```

The Virtual DOM

- In a web app, you can manipulate elements using DOM
 - Create elements, append child element, etc.
 - Low-level, tedious, and quite slow rendering
- React introduces the concept of the **virtual DOM**
 - You create React elements (lightweight JS objects)
 - You manipulate these lightweight JS objects
 - React renders the appropriate HTML very efficiently

Creating React Elements

- You can create a React element programmatically by calling `React.createElement()`
 - 1st argument specifies the type of element to create
 - 2nd argument specifies the element's properties
 - 3rd argument specifies the element's children

```
<script>
  const obj = React.createElement(
    'h1',
    {id: 'my-msg', title: 'This is my message'},
    'Hello React!'
  )
</script>
```

Rendering React Elements

- Render your top-level React element into a target location on the web page, as follows:

```
<script>
  ...
  const root = ReactDOM.createRoot(document.getElementById('root'))
  root.render(obj)

</script>
```

- `ReactDOM.createRoot()`
 - Identifies the target location where to render content
- `root.render()`
 - Tells React what to render

Section 3: Creating Many Elements

- Overview
- Hierarchy of React elements
- View the page in the browser
- View the virtual DOM

Overview

- In this section we'll see how to create a more ambitious virtual DOM tree, containing nested React elements
 - Then we'll render the root React element to the DOM
- Actually that's an important point...
 - You only ever render the root React element to the DOM

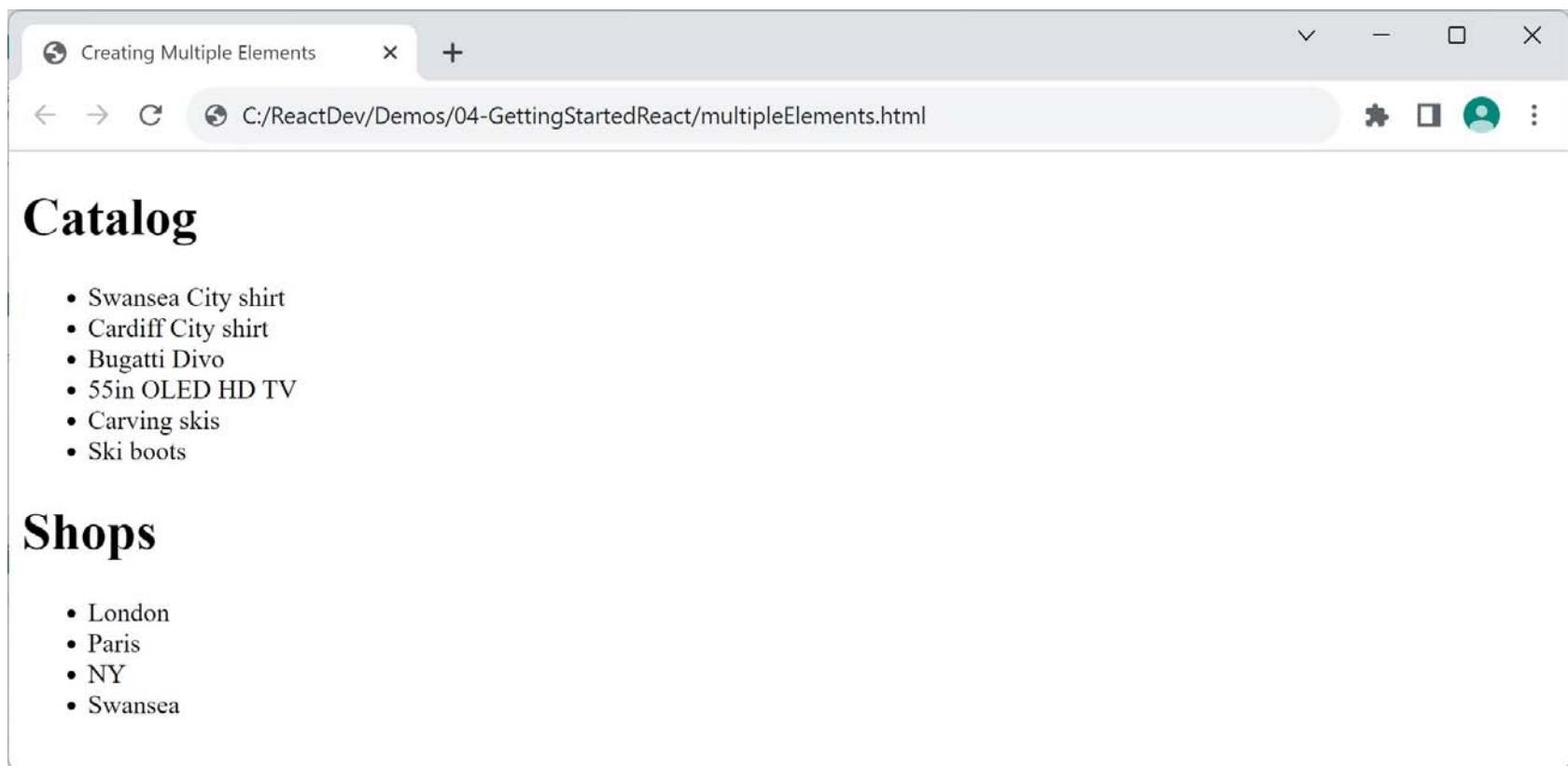
Hierarchy of React Elements

- `createElement()` can take a variadic list of child elements, so you can create a hierarchy of elements

```
const ul = React.createElement('ul', null,
    React.createElement('li', null, 'Item1'),
    React.createElement('li', null, 'Item2'),
    React.createElement('li', null, 'Item3'))
```

- For an example, see `multipleElements.html`

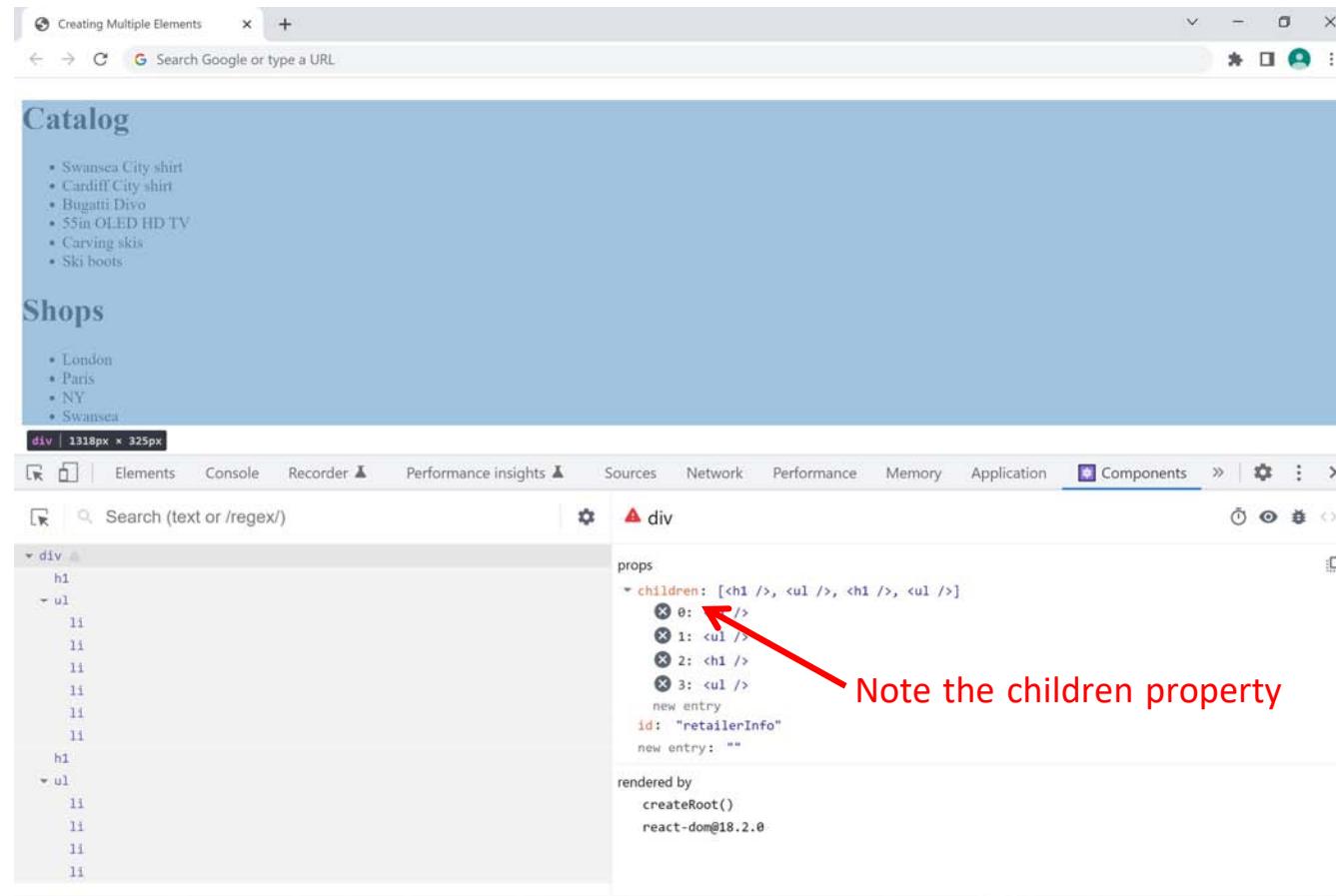
View the Page in the Browser



View the Virtual DOM (1 of 2)

- To view components in the virtual DOM, e.g. using React Developer Tools in Chrome:
 - Show the DevTools window (F12)
 - Click the Components tab
 - In the Search window, click the Settings icon
 - In the popup window, select the Components tab
 - Deselect the "Hide components where" option
 - You should now see a list of the components

View the Virtual DOM (2 of 2)



Section 4: Data-Driven UIs

- Overview
- Defining data
- Mapping data to elements
- Example
- View the page in the browser

Overview

- The previous section created a hard-coded hierarchy of React elements
- In a real app, you'll adopt a data-driven approach
 - The elements you create will depend on data
 - We'll see how to create a data-driven UI in this section
 - See `multipleElementsViaData.html`

Defining Data

- We'll have an array of products and an array of shops

```
const products = [  
  'Swansea City shirt',  
  'Cardiff City shirt',  
  ...  
]
```

```
const shops = [  
  'London',  
  'Paris',  
  ...  
]
```

Mapping Data to Elements

- You can use `map()` to map array item to React elem

```
someArray.map((arrayItem, idx) => React.createElement(htmlElem,  
                                              htmlProps,  
                                              htmlContent)  
)
```

- `map()` takes an arrow function
 - The arrow function receives 2 args - (array item, index)
 - The arrow function creates and returns a React element

Example (1 of 2)

- Let's map products array into an collection

```
let prodList = React.createElement(
  'ul',
  null,
  products.map((p, i) => React.createElement('li', {key: i}, p))
)
```

- Also map shops array into an collection

```
let shopList = React.createElement(
  'ul',
  null,
  shops.map((s, i) => React.createElement('li', {key: i}, s))
)
```

Example (2 of 2)

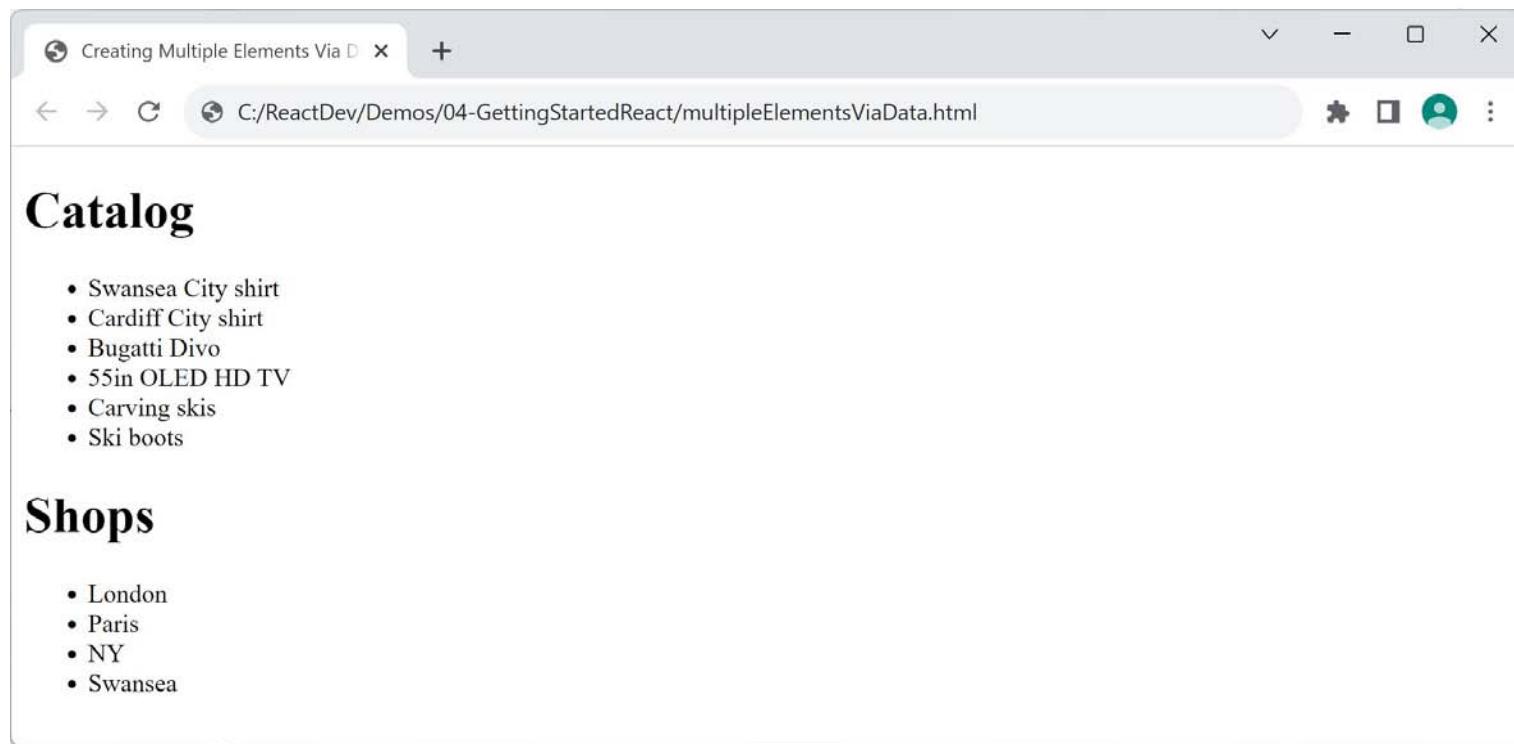
- Now let's put it all together

```
const retailer = React.createElement('div', null,
  React.createElement('h1', null, 'Catalog'),
  prodList,
  React.createElement('h1', null, 'Shops'),
  shopList
)

const root = ReactDOM.createRoot(document.getElementById('root'))
root.render(retailer)
```

Viewing the Page in the Browser

- Here's how the page looks in the browser



Summary

- Overview of React
- Creating a simple React app
- Creating many elements
- Data-driven UIs

Components

1. Overview
2. Class components
3. Functional components

Section 1: Overview

- The story so far...
- A more modular approach...
- Defining components in React

The Story So Far...

- The examples so far have created a monolithic dollop of React elements in one giant block of code

```
let prodList = React.createElement('ul', ... ... ... )
let shopList = React.createElement('ul', ... ... ... )

const retailer = React.createElement('div', null,
  React.createElement('h1', null, 'Catalog'),
  prodList,
  React.createElement('h1', null, 'Shops'),
  shopList
)

const root = createRoot(document.getElementById('root'))
root.render(retailer)
```

- Not feasible in a real application - too much content!

A More Modular Approach...

- Divide-and-conquer
 - Partition the UI into a bunch of components
 - Each component is responsible for one part of the UI
- Benefits of the component approach
 - Each component is relatively small and focussed
 - Easier to develop
 - Potential reuse
 - Easier to test

How to Define Components in React

- There are several ways to develop components in React
 - Via classes and inheritance - see Section 2
 - Via functional components - see Section 3
- In earlier versions of React, you could also create a component using `React.createClass()`
 - But this is deprecated nowadays

Section 2: Class Components

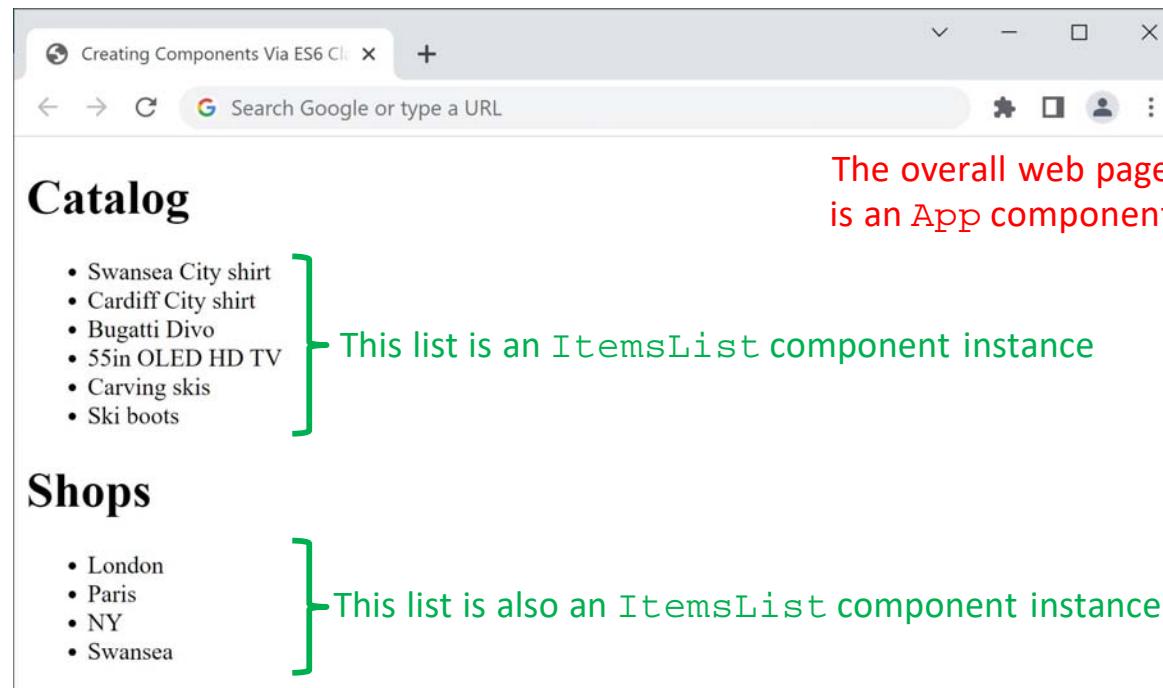
- Overview
- Example scenario
- Example data
- Component classes in our example
- Creating/rendering the App component

Overview

- React has a class named `React.Component`
 - Has common capabilities needed by all components
 - E.g. render elements for a component
- To define your own component as a class:
 - Define a class that inherits from `React.Component`
 - Override `render()` to render component's elements
 - Use `this.props` to access component's properties

Example Scenario

- ClassComponents.html has several components



Example Data

- Here's the familiar data for our components

```
const products = [
  'Swansea City shirt',
  'Cardiff City shirt',
  ...
]

const shops = [
  'London',
  'Paris',
  ...
]
```

Component Classes in our Example

```
class App extends React.Component {
  render() {
    return React.createElement('div', null,
      React.createElement('h1', null, 'Catalog'),
      React.createElement(ItemsList, {items: products}, null),
      React.createElement('h1', null, 'Shops'),
      React.createElement(ItemsList, {items: shops}, null)
    )
  }
}
```

```
class ItemsList extends React.Component {
  render() {
    return React.createElement('ul', null,
      this.props.items.map((item, i) => React.createElement('li', {key:i}, item))
    )
  }
}
```

Creating/Rendering the App Component

- We create/render the App component as the root React element as follows:

```
const app = React.createElement(App, null, null)

const root = createRoot(document.getElementById('root'))
root.render(app)
```

Section 3: Functional Components

- Overview
- Functional components in our example
- Creating/rendering the App component
- Recommendations

Overview

- Now we're going to see how to define components as functional components
 - This is usually simpler than defining class components
 - See `FunctionalComponents.html`
- A functional component:
 - Is just a function (i.e. not a class)
 - Receives properties as a function parameter (an object)
 - Creates/returns a React element, which React will render

Components in our Example

```
function App() {
  return (
    React.createElement('div', null,
      React.createElement('h1', null, 'Catalog'),
      React.createElement(ItemsList, {items: products}, null),
      React.createElement('h1', null, 'Shops'),
      React.createElement(ItemsList, {items: shops}, null),
    )
  )
}
```

```
function ItemsList(props) {
  return (
    React.createElement(
      "ul",
      null,
      props.items.map((item, i) => React.createElement("li", { key: i }, item))
    )
  )
}
```

Creating/Rendering the App Component

- We create/render the App component as the root React element as follows (same as before ☺):

```
const app = React.createElement(App, null, null)

const root = createRoot(document.getElementById('root'))
root.render(app)
```

Recommendations

- Most modern React code uses functional components
 - Easier than class components
 - Support several new techniques (e.g. effect hooks)
- So we'll focus on functional components from now on
 - We'll also show how to achieve the same effects using class components, for the sake of completeness

Summary

- Overview
- Class components
- Functional components

JSX

1. Overview of JSX
2. JSX syntax
3. JSX gotchas

Section 1: Overview of JSX

- The story so far...
- Introducing JSX
- Transpiling JSX

The Story So Far...

- In all the examples so far, we've created elements programmatically using `React.createElement()`

```
const prodList = React.createElement('ul', null,
  React.createElement('li', null, 'Swansea City shirt'),
  React.createElement('li', null, 'Cardiff City shirt'),
  React.createElement('li', null, 'Bugatti Divo'),
  React.createElement('li', null, '55in OLED HD TV'),
  React.createElement('li', null, 'Carving skis'),
  React.createElement('li', null, 'Ski boots')
);
```

- This is very verbose!

Introducing JSX

- React supports a lightweight syntax called JSX
 - Create React elements concisely and directly
 - Use XML to specify the elements you want to create

```
const prodList =  
  <ul>  
    <li>Swansea City shirt</li>  
    <li>Cardiff City shirt</li>  
    <li>Bugatti Divo</li>  
    <li>55in OLED HD TV</li>  
    <li>Carving skis</li>  
    <li>Ski boots</li>  
  </ul>
```

Transpiling JSX

- Browsers don't understand JSX syntax
 - JSX syntax must be transpiled into "pure" React
- You can use the Babel transpiler to do this
 - Add a `<script>` to download the Babel transpiler
 - Embed JSX inside `<script type="text/babel">`

```
<script src="...URL for Babel on unpkg.com..."></script>

<script type="text/babel" ... >
  ... Put your JSX code here ...
</script>
```

Section 2: JSX Syntax

- JSX content
- Using JSX for components
- Simple example of JSX
- Evaluating JavaScript expressions in JSX
- Data-driven JSX
- Passing properties to a component in JSX
- Complete example

JSX Content

- JSX elements can contain plain text

```
const productsList =  
  <ul>  
    <li>Swansea City shirt</li>  
    <li>Cardiff City shirt</li>  
  </ul>
```

- JSX elements can contain JS expressions in {} braces

```
const data = ["Swansea City shirt", "Cardiff City shirt"]  
  
const summaryElem = <div>There are {data.length} items</div>
```

Using JSX for Components

- JSX can be used in a component:

```
function ItemsList() {  
  return (  
    <div>  
      <h1>JSX Example 1</h1>  
      <ul>  
        <li>Swansea shirt</li>  
        <li>Cardiff shirt</li>  
        <li>Lamborghini</li>  
      </ul>  
    </div>  
  )  
}  
  
const root = createRoot(document.getElementById('root'))  
root.render(<ItemsList/>)
```

example1.html

Evaluating JavaScript Expressions in JSX

- JSX can contain JavaScript expressions:

```
function ItemsList() {  
  const timestamp = new Date().toLocaleTimeString()  
  return (  
    <div>  
      <h1>JSX Example 2</h1>  
      <ul>  
        <li>Swansea shirt</li>  
        <li>Cardiff shirt</li>  
        <li>Lamborghini</li>  
      </ul>  
      <small>Page generated at {timestamp}</small>  
    </div>  
  )  
}  
  
const root = createRoot(document.getElementById('root'))  
root.render(<ItemsList/>)
```

example2.html

Data-Driven JSX

- JSX can contain data-driven content:

```
const data = ["Swansea shirt", ... ... ]                                example3.html

function ItemsList() {
  const timestamp = new Date().toLocaleTimeString()
  return (
    <div>
      <h1>JSX Example 3</h1>
      <ul>
        { data.map((item, i) => <li key={i}>{item}</li>) }
      </ul>
      <hr/>
      <small>Page generated at {timestamp}</small>
    </div>
  )
}

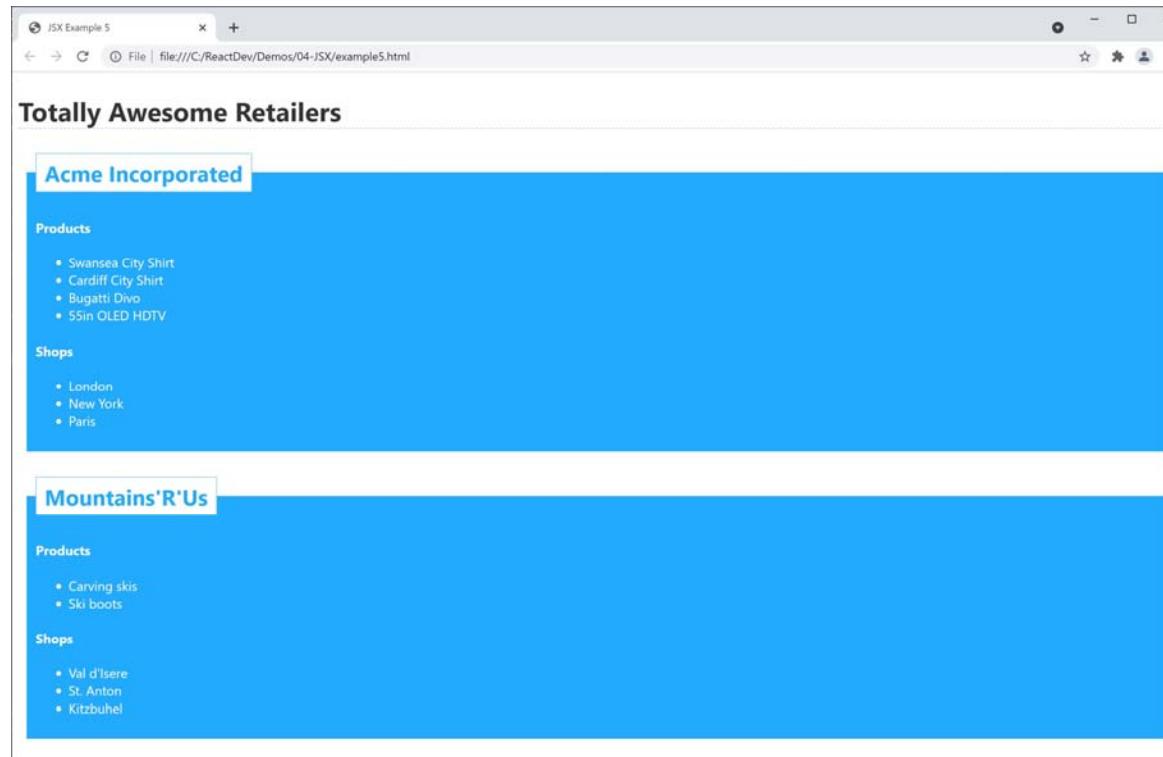
const root = createRoot(document.getElementById('root'))
root.render(<ItemsList/>)
```

Passing Properties to a Component in JSX

```
function ItemsList(props) {  
    return (  
        <div>  
            <h1>{props.heading}</h1>  
            <ul>  
                {props.items.map((item, i) => <li key={i}>{item}</li>)}  
            </ul>  
            <hr/>  
            <small>Page generated at {props.timestamp}</small>  
        </div>  
    )  
}  
  
const root = createRoot(document.getElementById('root'))  
  
root.render(  
    <ItemsList  
        heading={'JSX Example 4'}  
        items={data}  
        timestamp={new Date().toLocaleTimeString()}  
    />  
)
```

Complete Example

- See complete example in example5.html



Section 3: JSX Gotchas

- JSX gotchas - 1
- JSX gotchas - 2

JSX Gotchas - 1

- JSX is case-sensitive

```
const badElem1 = <DIV>oops</div>
```

- JSX tags must be closed

```
const badElem2 = <input type="text">
```

JSX Gotchas - 2

- Adjacent JSX elements must be wrapped inside an enclosing tag

```
const badElem3 =  
  <h1>Greetings</h1>  
  <div>This won't work. Sorry!</div>
```

- To assign a CSS class, use `className` (not `class`)

```
const badElem4 =  
  <div className="emphasis">Won't work!</div>
```

Summary

- Overview of JSX
- JSX syntax
- JSX gotchas

State Management

1. State in functional components
2. State in class components

Section 1: State in Functional Components

- Recap component properties
- Fixed vs. mutable state
- State in a functional component
- Complete example

Recap Component Properties

- We've seen how to pass properties into a component

```
function Person({name, age, isWelsh, skills}) {  
    return ( ... some elements ... )  
}
```

```
<Person name="John Evans"  
        age={21}  
        isWelsh={true}  
        skills={[...]} />
```

- (Ditto for class components)

Fixed Properties vs. Mutable State

- Properties are immutable
 - You can't change their values
- What if the component needs to hold mutable state?
 - E.g. add items to an array, update a timestamp, etc.
- You can achieve mutable state in a component
 - See following slides for details
 - (Different techniques for class vs functional components)

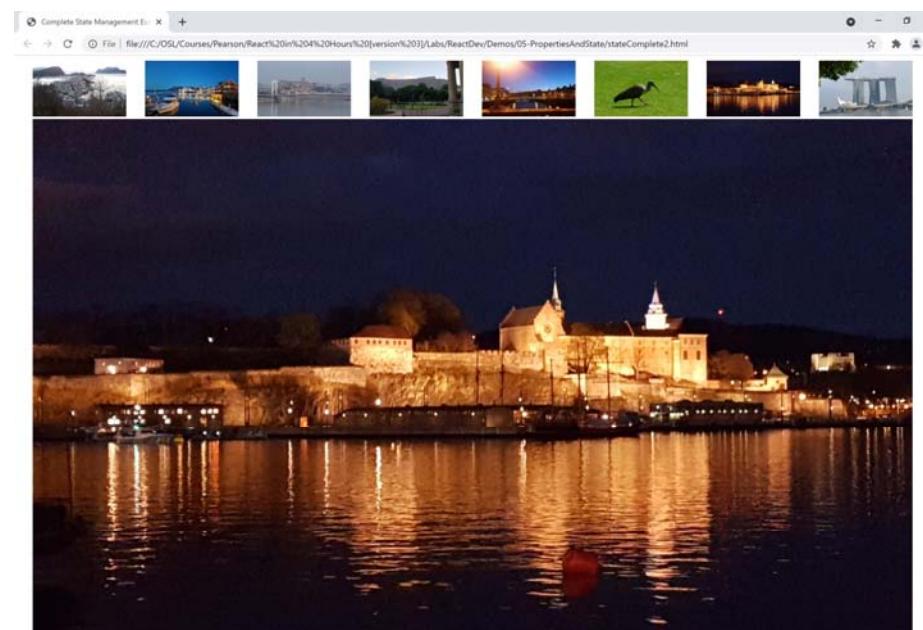
State in a Functional Component

- In a functional component, you use a "state hook"
- To initialize state:
 - Call `React.useState(initState)`,
 - Returns `[stateVariable, updateFunc]`
- To access state:
 - Use `stateVariable`
- To modify state
 - Call `updateFunc(newState)`

See example in
`stateSimple1.html`

Complete Example

- For a complete example of state management in a functional component, see:
 - `stateComplete1.html`



Section 2: State in Class Components

- Properties in a class component
- State in a class component
- Complete example

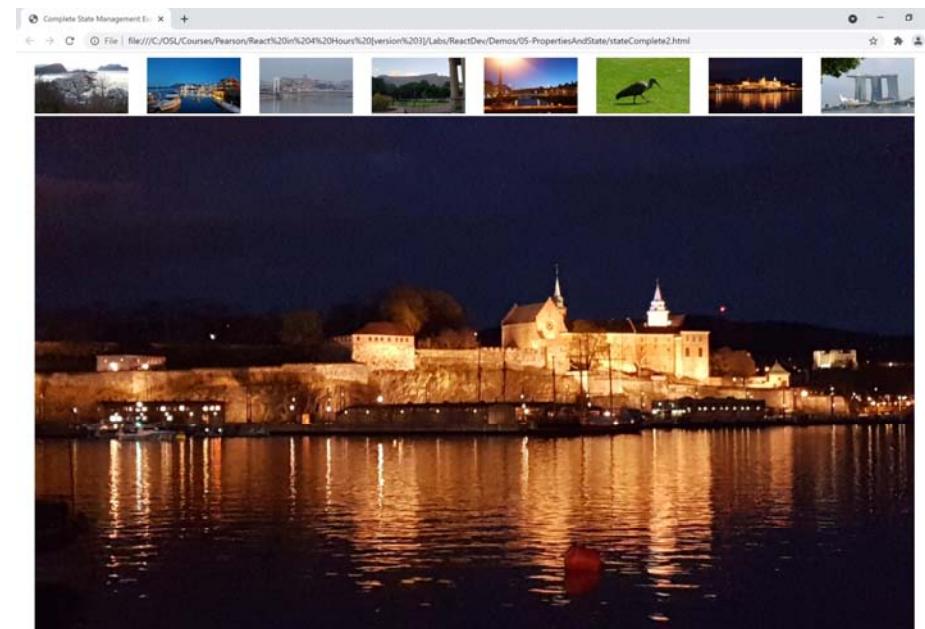
State in a Class Component

- In a class component, state is available via `this.state`
- To initialize state:
 - In the constructor, set `this.state`
- To access state:
 - Use `this.state`
- To modify state:
 - Call `this.setState(state)`

See example in
`stateSimple2.html`

Complete Example

- For a complete example of state management in a class component, see:
 - `stateComplete2.html`



Summary

- State in functional components
- State in class components

Creating a Complete React Application

1. Creating an application
2. Running the application

Section 1: Creating an Application

- Recap of our progress so far
- A better approach
- Creating a React TypeScript app
- Reviewing the application
- Application home page
- Source code entry point
- Functional components

Recap of our Progress so Far

- So far, we've seen how to create relatively simple React applications...
- We've put all our code/HTML in a single web page
- We've used `<script>` tags to download React libraries in the browser at run-time
- We've used Babel to transpile JSX into pure JS at run-time

A Better Approach...

- A better approach...
 - Put each component in a separate file (for modularity)
 - Transpile JSX into pure JS at dev-time (for faster run-time)
- We're going to use a tool called **Vite** to help us do this
 - Generates a template React app, config, etc.
 - See <https://vitejs.dev/>
- Install Vite as follows (requires Node.js 18 or above):

```
npm install -g vite@latest
```

Creating a React TypeScript Application

- Run Vite and specify these options, to create a simple app:

```
npm create vite@latest
```

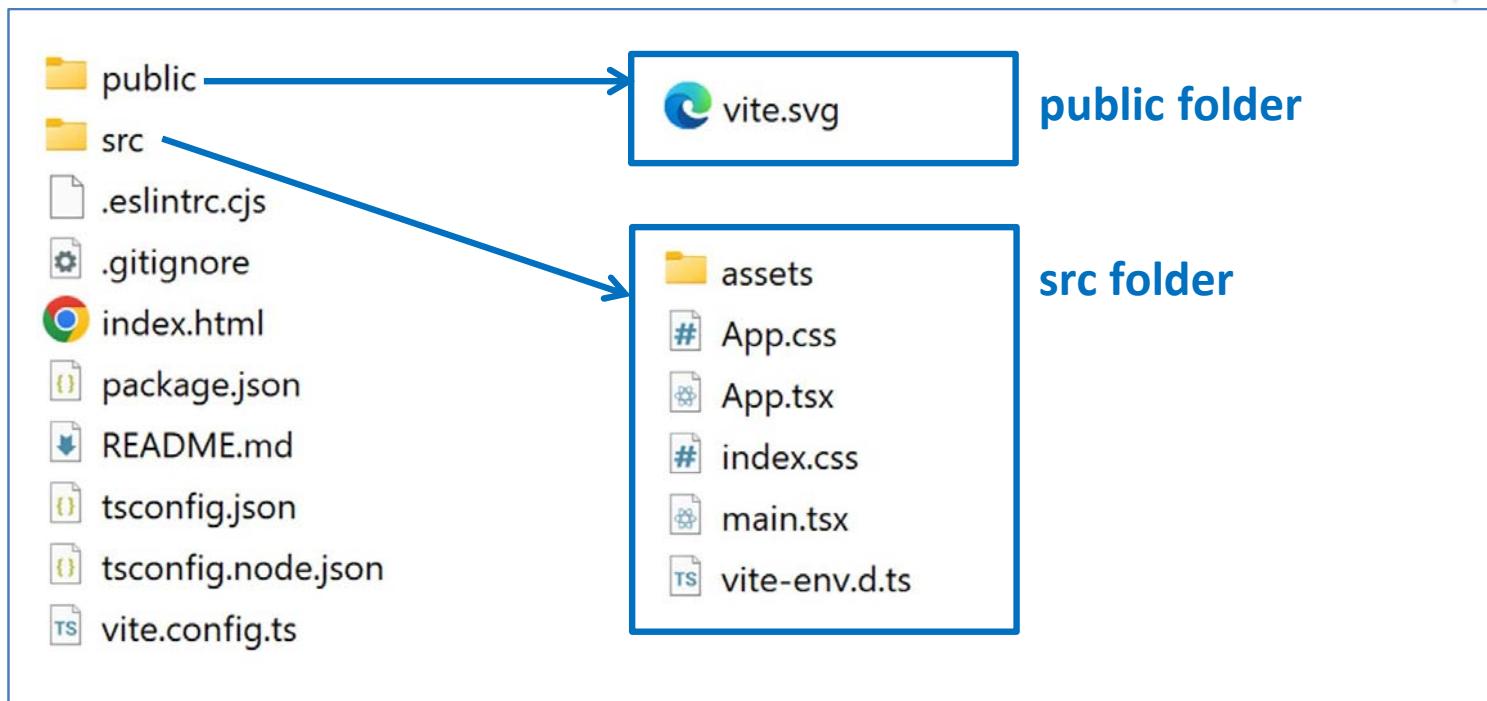
- Project name: **demo-app**
- Framework: **React**
- Variant: **TypeScript**

- Then install packages for your application, as follows:

```
cd demo-app  
npm install
```

Reviewing the Application

- Here's the structure of the generated application:



Application Home Page

- The application home page is /index.html

```
<!doctype html>
<html lang="en">
  <head>
    <title>Vite + React + TS</title>
    ...
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.tsx"></script>
  </body>
</html>
```

/index.html

Source Code Entry Point

- The source code entry point is `/src/main.tsx`

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.tsx'

createRoot(document.getElementById('root')!).render(
  <StrictMode>
    <App />
  </StrictMode>,
)

```

`/src/main.tsx`

- Aside: For info about *React strict mode*, see:
 - <https://react.dev/reference/react/StrictMode>

Functional Components

- In the generated code, App is a *functional component*

```
import { useState } from 'react'  
...  
  
function App() {  
  
  const [count, setCount] = useState(0)  
  
  return (  
    <>  
    ...  
    <button onClick={() => setCount((count) => count + 1)}>  
      count is {count}  
    </button>  
    ...  
    </>  
  )  
}  
export default App
```

/src/App.tsx

Section 2: Running the Application

- Running the app in dev mode
- Hot reloading
- Building the app for production
- Serving the production application
- Pinging the production application

Running the App in Dev Mode (1 of 3)

- You can run the app in dev mode as follows:

```
npm run dev
```

- What this does:
 - Transpiles TS code into ES (see `tsconfig.json` for version)
 - Transpiles JSX/TSX files into ES
 - Builds the application in memory
 - Starts a dev server to host the application
 - The dev server is on `http://localhost:5173`

Running the App in Dev Mode (2 of 3)

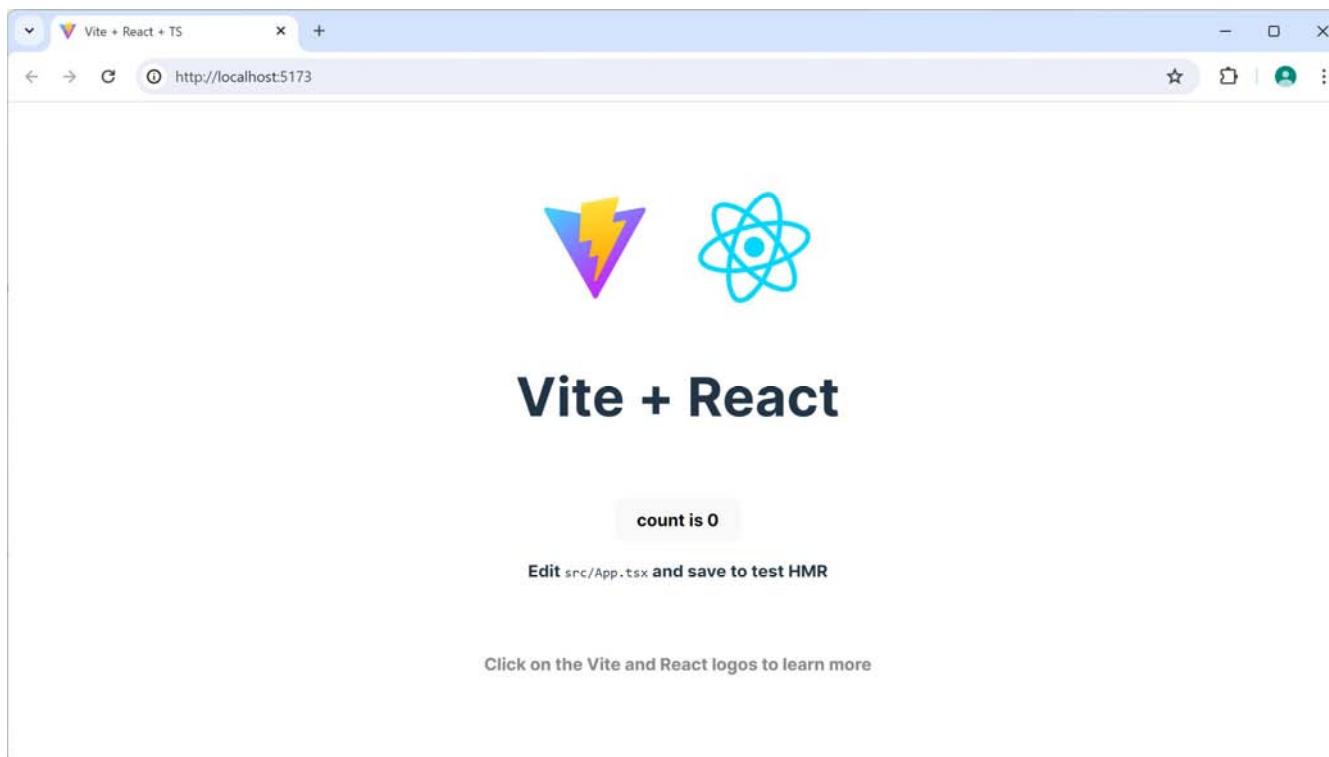
- Output from the command on the previous slide:

```
VITE v5.2.12  ready in 871 ms

→ Local:  http://localhost:5173/
→ Network: use --host to expose
→ press h + enter to show help
```

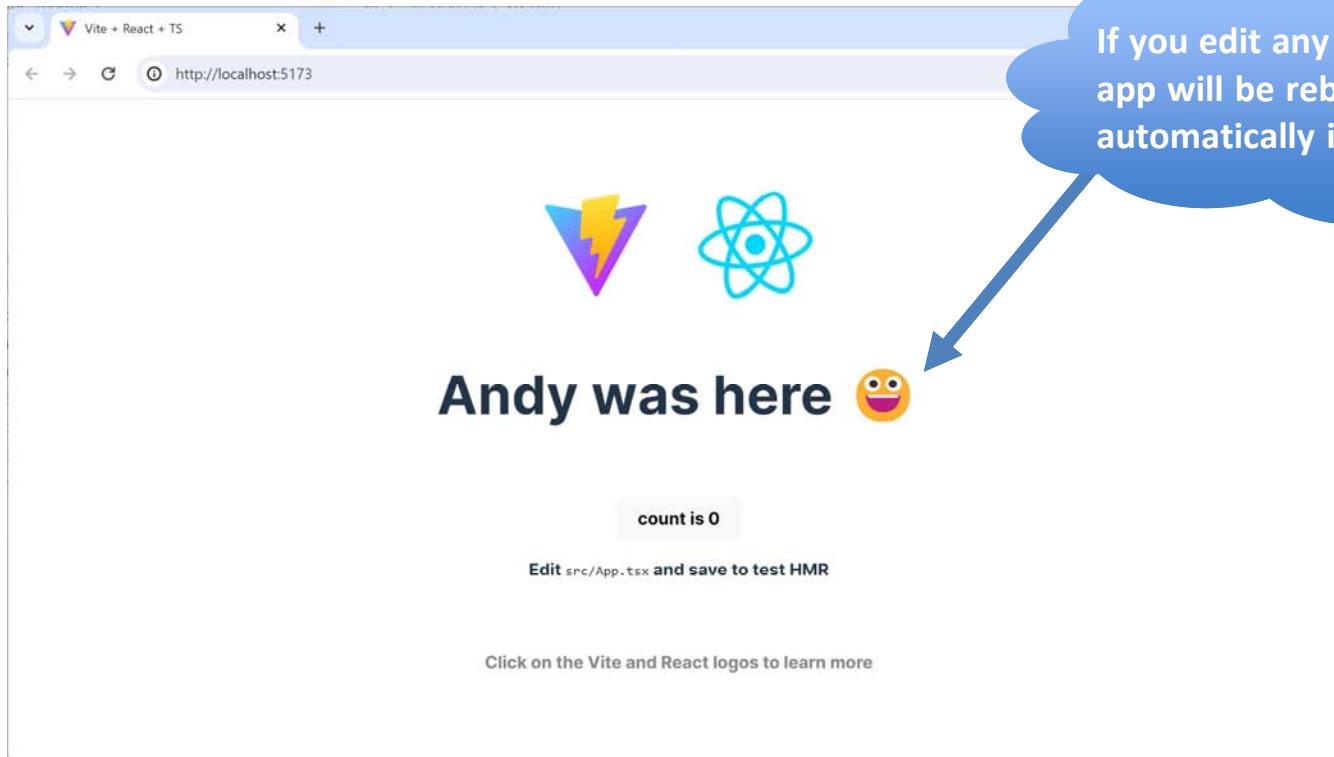
Running the App in Dev Mode (3 of 3)

- Ping the app at `http://localhost:5173`



Hot Reloading

- Hot reloading is supported



Building the Application for Production

- You can build the application for production as follows:

```
npm run build
```

- This creates a `dist` folder that contains a production build of your application
 - `dist/index.html` - Home page
 - `dist/assets/*.js` - Minified JS chunk files
 - `dist/assets/*.css` - Minified CSS files

Serving the Production Application

- You can now run the application on a production server
- E.g., install the Node "serve" server:

```
npm install -g serve
```

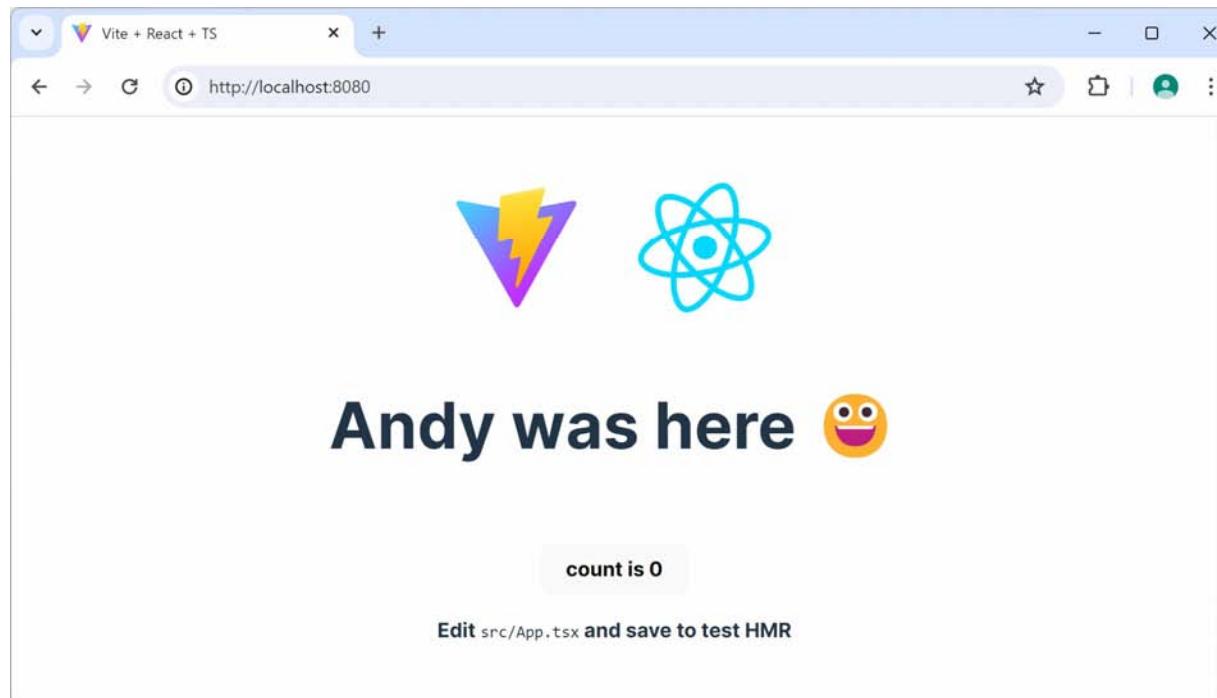
- Then serve the production build of the app as follows:

```
serve -s dist -l 8080
```

 - -s option - Location of app (i.e., the `dist` folder)
 - -l option - Port to listen on (default is 5000)

Pinging the Production Application

- Open a browser and navigate to the following URL:
 - `http://localhost:8080/`



Summary

- Creating an application
- Running the application

Introduction to Routing

1. Overview of SPAs and routing
2. Implementing routing in React

Section 1: Overview of SPAs and Routing

- What is an SPA?
- SPAs and React
- Demo application

What is an SPA?

- According to Wiki:



A **single-page application** is a web app that fits on a single web page with the goal of providing a more fluent user experience similar to a desktop application.

In an SPA, either all necessary code – HTML, JavaScript, and CSS – is retrieved with a single page load, or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions. The page does not reload at any point in the process, nor does control transfer to another page.

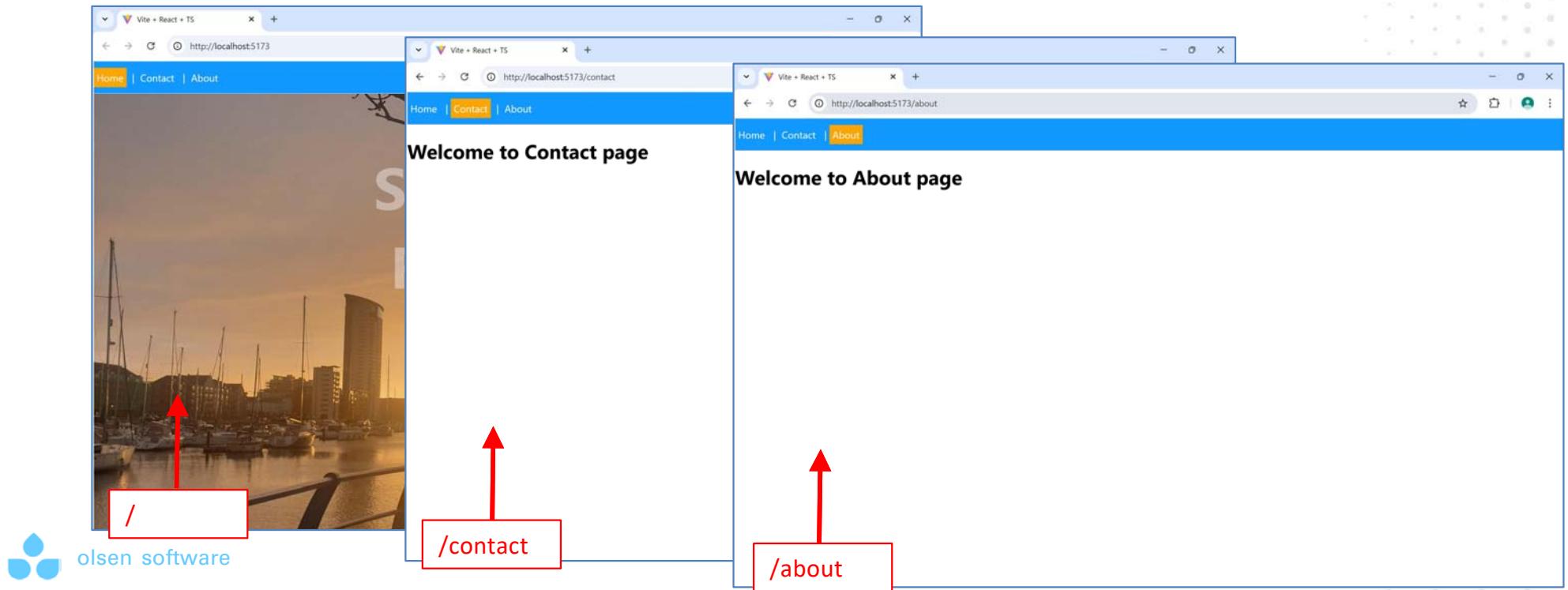
Interaction with the SPA often involves dynamic communication with the web server behind the scenes.

SPAs and React

- React has excellent support for implementing SPAs
 - Define an `App` component that always remains resident
 - Define multiple sub-components, which can be swapped in and out of the `App` component
- Each sub-component maps to a logical URL
 - This is called *routing*
 - To display a different sub-component in the browser, simply navigate to the URL for that sub-component

Demo Application (1 of 2)

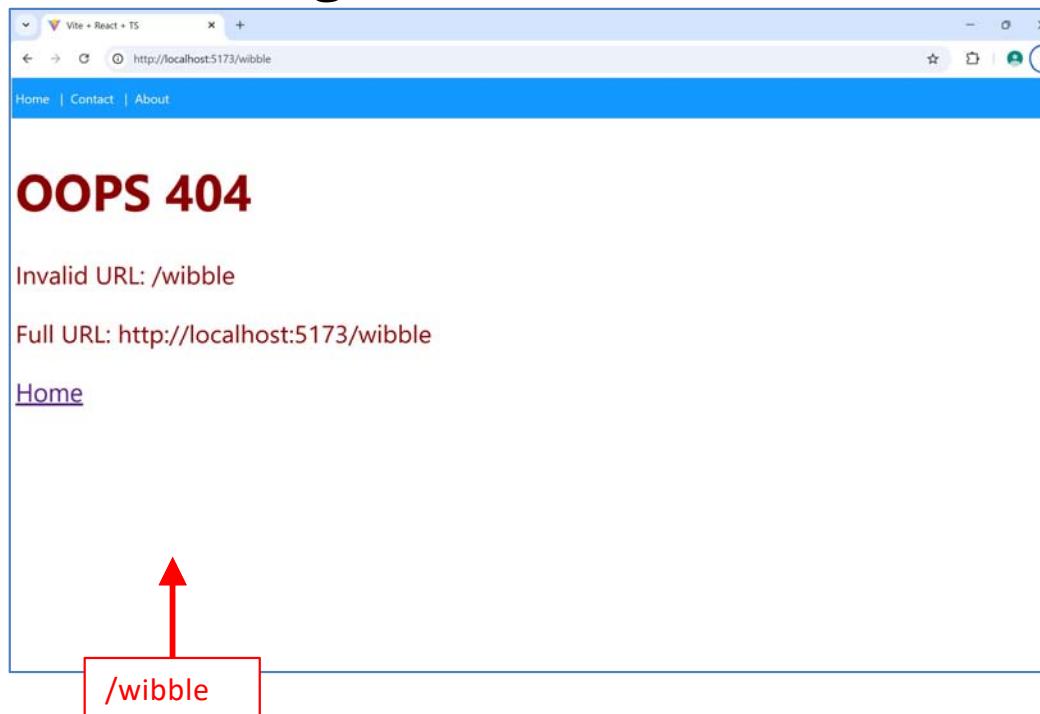
- For an example of React routing and SPAs, see this demo:
 - `ReactDev/Demos/09-SinglePageApps/demo-app`



olsen software

Demo Application (2 of 2)

- The application also has a "page not found" component for unrecognised URLs



Section 2: Implementing Routing in React

- Dependencies for React Router
- Creating a browser router
- Defining a common layout for all pages
- Simple example routes

Dependencies for React Router

- Add these dependencies in package.json:

```
"dependencies": {  
  "react-router-dom": "^6.3.0", ← React Router  
  ...  
},  
  
"devDependencies": {  
  "@types/react-router-dom": "^5.3.3", ← React Router TypeScript declarations  
  ...  
}
```

package.json

Creating a Browser Router

- You define routes via `createBrowserRouter()`
 - Enables you to specify routes up-front in your app

```
import { createBrowserRouter, RouterProvider } from 'react-router-dom'  
...  
  
const router = createBrowserRouter([
  { path: '/somePath1', element: <SomeComponent1 /> },
  { path: '/somePath2', element: <SomeComponent2 /> },
  ...
])
```

- Use the router in your App component as follows:

```
export default function App() {
  return <RouterProvider router={router} />
}
```

[App.tsx](#)

Defining a Common Layout for all Pages

- You can define a common layout for all pages:

```
import { Outlet } from 'react-router-dom'
...

function AppLayout() {
  return (
    <>
      <MyMenu />      { /* Always display my common menu here (for example) */ }
      <Outlet />     { /* Display the component for the current route here */ }
    </>
  )
}

const router = createBrowserRouter([
  {
    element: <AppLayout />,

    children: [
      { path: '/somePath1', element: <SomeComponent1 /> },
      { path: '/somePath2', element: <SomeComponent2 /> },
      ...
    ]
  }
])
```

App.tsx



olsen software

Simple Example Routes

- Here are some simple example routes to get us started:

```
import Home from './Home'
import About from './About'
import Contact from './Contact'
import PageNotFound from './PageNotFound'
...

const router = createBrowserRouter([
  {
    element: <AppLayout />,

    children: [
      { path: '/', element: <Home /> },
      { path: '/about', element: <About /> },
      { path: '/contact', element: <Contact /> },
      { path: '*', element: <PageNotFound /> },
    ]
  }
])
```

App.tsx

Summary

- Overview of SPAs and routing
- Implementing routing in React

User Input Techniques

- Overview
- Accessing elements by using DOM
- Using uncontrolled components
- Using controlled components

Overview

- A React app often has to access elements on the web page
 - Typically to get/set the value of input fields
- There are several ways to do this:
 - Using DOM
 - Using uncontrolled components
 - Using controlled components
- We'll explore all these techniques in this chapter
 - See demo-app

Accessing Elements by using DOM

- This is the most basic approach
 - Assign an id to an element
 - Access the element programmatically via its id
- Example:
 - See `DemoUsingDom.tsx`
- Pros and cons:
 - Simple and familiar
 - Low-level, cumbersome, possible conflicting element id's

Using Uncontrolled Components

- This is a React-specific approach
 - Store a reference to an element in React memory
 - Access the element programmatically via that reference
- Example:
 - See `DemoUncontrolledComponent.tsx`
- Pros and cons:
 - More React-orientated than using DOM
 - Gives full access to the actual element in the DOM tree
 - Still quite hard work and low-level

Using Controlled Components

- This is another React-specific approach
 - Store a value in React mutable state
 - Two-way bind the value directly to an input field
- Example:
 - See `DemoControlledComponent.tsx`
- Pros and cons:
 - The easiest way to get/set the value of an input field
 - Facilitates key-by-key validation
 - Causes a re-render for every key (could be slow)

Summary

- Overview
- Accessing elements by using DOM
- Using uncontrolled components
- Using controlled components

Routing Techniques

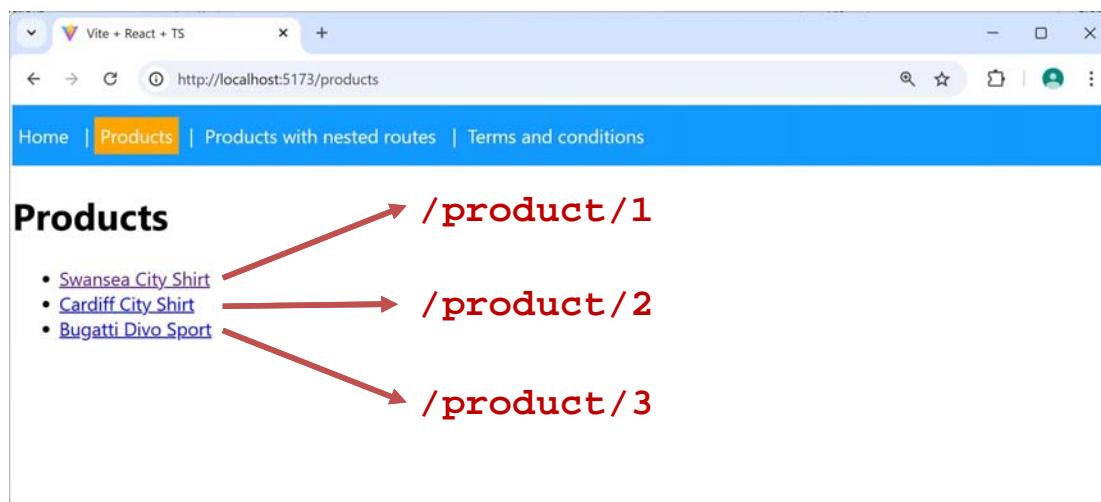
1. Parameterized routes
2. Nested routes
3. Loading data for a route

Section 1: Parameterized Routes

- Overview
- Understanding the data model
- Defining parameterized routes
- Rendering parameterized links
- Obtaining path parameters
- Programmatic navigation

Overview

- React router lets you define parameterized routes
 - E.g., /product/:id
- The demo app illustrates parameterized routes



Understanding the Data Model

- To understand the demo, the first step is to consider the data model
- See `Catalog.ts`, which defines 2 simple classes:
 - `ProductItem` – Represents a simple product item
 - `Catalog` – Contains hard-coded product items

Defining Parameterized Routes

- Here are the relevant routes:
 - /products - Display links for all product items
 - /product/:id - Display a product item with specified id

```
{ path: 'products', element: <Products /> },
{ path: 'product/:id', element: <Product /> },
```

App.tsx

Rendering Parameterized Links

- `Products.tsx` renders parameterized links like so:

```
export default function Products() {  
  
  const products = Catalog.getAllProductItems()  
  
  return (  
    <>  
      <h1>Products</h1>  
      <ul>  
        {  
          products.map((p, i) =>  
            <li key={i}>  
              <Link to={`/product/${p.id}`}>{p.description}</Link>  
            </li>  
          )  
        }  
      </ul>  
    </>  
  )  
}
```

`Products.tsx`

Obtaining Path Parameters

- ProductV1.tsx obtains path parameter(s) like so:

```
import { useParams } from 'react-router-dom'
...
export default function Product() {
  const { id } : any = useParams()
  const prod = Catalog.getProductById(id)
  if (!prod) {
    ...
  } else {
    return (
      <>
        <h1>Product details</h1>
        <div>Description: {prod.description}</div>
        <div>Price: {prod.price}</div>
        <div>Likes: {prod.likes}</div>
        <div>Most recent like: {prod.mostRecentLike}</div>
      </>
    )
  }
}
```

ProductV1.tsx

Programmatic Navigation

- We've seen how to navigate to a route via a <Link>

```
<Link to='somePath'>Click me</Link>
```

- You can also navigate to a route programmatically

```
import { useNavigate } from 'react-router-dom'  
...  
  
function SomeComponent() {  
  const navigate = useNavigate()  
  navigate(somePath)  
  ...  
}
```

- For an example, see ProductV2.tsx

Section 2: Nested Routes

- Overview
- Implementing the outer route
- Implementing an index route
- Implementing a nested route
- Implementing another nested route

Overview

- React router lets you define nested routes

```
{  
  path: '/products-with-nested-routes', element: <ProductsWithNestedRoutes />,  
  children: [  
    { index: true, element: <ProductUnselected /> },  
    { path: ':id', element: <Product /> },  
    { path: 'summary', element: <ProductSummary /> }  
  ]  
},
```

App.tsx

- The user can navigate to the following routes:
 - /products-with-nested-routes
 - /products-with-nested-routes/1 (etc.)
 - /products-with-nested-routes/summary

Implementing the Outer Route

- Let's consider the outer route:

```
{  
  path: '/products-with-nested-routes', element: <ProductsWithNestedRoutes />,  
  children: [  
    { index: true, element: <ProductUnselected /> },  
    { path: ':id', element: <Product /> },  
    { path: 'summary', element: <ProductSummary /> }  
  ]  
},
```

App.tsx

- When we go to **/products-with-nested-routes**
 - ProductsWithNestedRoutes** component is rendered
 - Contains an `<Outlet/>`, to house a nested component

Implementing an Index Route

- We've defined an *index route* as follows:

```
{  
  path: '/products-with-nested-routes', element: <ProductsWithNestedRoutes />,  
  children: [  
    { index: true, element: <ProductUnselected /> },  
    { path: ':id', element: <Product /> },  
    { path: 'summary', element: <ProductSummary /> }  
  ]  
},
```

App.tsx

- Indicates the default component to render in `<Outlet>` if the user navigates just to the parent route
 - `ProductUnselected` component is rendered

Implementing a Nested Route

- We've defined a nested route to display one product:

```
{  
  path: '/products-with-nested-routes', element: <ProductsWithNestedRoutes />,  
  children: [  
    { index: true, element: <ProductUnselected /> },  
    { path: ':id', element: <Product /> },  
    { path: 'summary', element: <ProductSummary /> }  
  ]  
},
```

App.tsx

- Go to `/products-with-nested-routes/1`
 - `ProductsWithNestedRoutes` component is rendered
 - In its `<Outlet/>`, `Product` component is rendered

Implementing another Nested Route

- We've defined another nested route as follows:

```
{  
  path: '/products-with-nested-routes', element: <ProductsWithNestedRoutes />,  
  children: [  
    { index: true, element: <ProductUnselected /> },  
    { path: ':id', element: <Product /> },  
    { path: 'summary', element: <ProductSummary /> }  
  ]  
},
```

App.tsx

- Go to `/products-with-nested-routes/summary`
 - `ProductsWithNestedRoutes` component is rendered
 - In its `<Outlet/>`, `ProductSummary` component is rendered

Section 3: Loading Data for a Route

- Defining a data loader function for a route
- Accessing loader data in a component
- Example

Defining a Data Loader Function for a Route

- In this chapter we've shown how to define routes via the `createBrowserRouter()` function

```
const router = createBrowserRouter([
  { path: '/somePath1', element: <SomeComponent1 /> },
  { path: '/somePath2', element: <SomeComponent2 /> },
  ...
])
```

- You can also specify a *data loader function* for a route
 - Invoked asynchronously when user navigates to the route
 - Returns data that the component can utilize

```
const router = createBrowserRouter([
  { path: '/somePath1', element: <SomeComponent1 />, loader: someLoaderFunc1 },
  { path: '/somePath2', element: <SomeComponent2 />, loader: someLoaderFunc2 },
  ...
])
```

Accessing Loader Data in a Component

- A component can access the loader data via the `useLoaderData()` hook

```
import { useLoaderData } from 'react-router-dom'  
...  
  
function SomeComponent() {  
  
  const data: any = useLoaderData()  
  ...  
}
```

Example

- For an example of loading data and accessing it in a component, see the following files:
- `App.tsx`
 - See the route `/ts-and-cs/:id`
- `TsAndCs.tsx`
 - See `getDataForRegion()`, which loads data
 - See `TsAndCsForRegion()`, which accesses the data

Summary

- Parameterized routes
- Nested routes
- Loading data for a route

Component Techniques

1. Effect hooks
2. Calling REST services

Annex

- Memoization

Section 1: Effect Hooks

- Overview
- Defining a simple effect hook
- Effects with cleanup
- Effect dependencies

Overview

- A functional component can define *effect hooks*
 - An effect hook is effectively a call-back function
 - Called automatically by React, after each render
 - Enables the component to do some additional work
- We'll see how to define effect hooks in this section
 - See the project `demo-effect-hooks`

Defining a Simple Effect Hook (1 of 2)

- To define an effect hook:
 - Call `useEffect()` from the `react` module
- useEffect(() => { side-effect code... })
- React calls your effect(s) after each render, including after the first render
 - Gives you an opportunity to perform side-effect work

Defining a Simple Effect Hook (2 of 2)

- Example - see `Gallery1.tsx`
- Note:
 - You can define multiple effect hooks, which is good for separation of concerns
 - Effect hooks are invoked in the order defined

Effects with Cleanup (1 of 2)

- Some effects might require cleanup
 - E.g. cancel a timer/interval
 - E.g. cancel a subscription to an external data source
- To provide cleanup behaviour for an effect:
 - Return a function from the effect

```
useEffect(() => {
  side-effect code...
  return () => { cleanup code... }
})
```

Effects with Cleanup (2 of 2)

- Example - see `Gallery2.tsx`
- React invokes an effect cleanup function as follows:
 - Before the component re-renders itself
 - Just before the component unmounts itself

Effect Dependencies (1 of 2)

- `useEffect()` takes an optional 2nd argument
 - An array of dependencies
 - Effect will only be invoked if a dependency has changed

```
React.useEffect(() => {  
  side-effect code...  
  return () => { cleanup code... }  
}, [dependency1, dependency2, ...])
```

Effect Dependencies (2 of 2)

- Example - see `Gallery3.tsx`
- `Gallery` has 4 effects, illustrating different scenarios:
 - Effect 1 - Called on every re-render
 - Effect 2 - Called on every thumbnail click
 - Effect 3 - Called on a different thumbnail click
 - Effect 4 - Called on initial render, cleaned up on unmount

Section 2: Calling REST Services

- Overview
- Starting the REST service
- Testing the REST service
- Calling the REST service

Overview

- In this section we show a realistic example of why you'd implement an effect hook
- The example will call a REST service at the point when a component is first rendered
 - The component calls the REST service asynchronously
 - When the REST service returns with the data, the component will re-render itself with the result data

Starting the REST Service

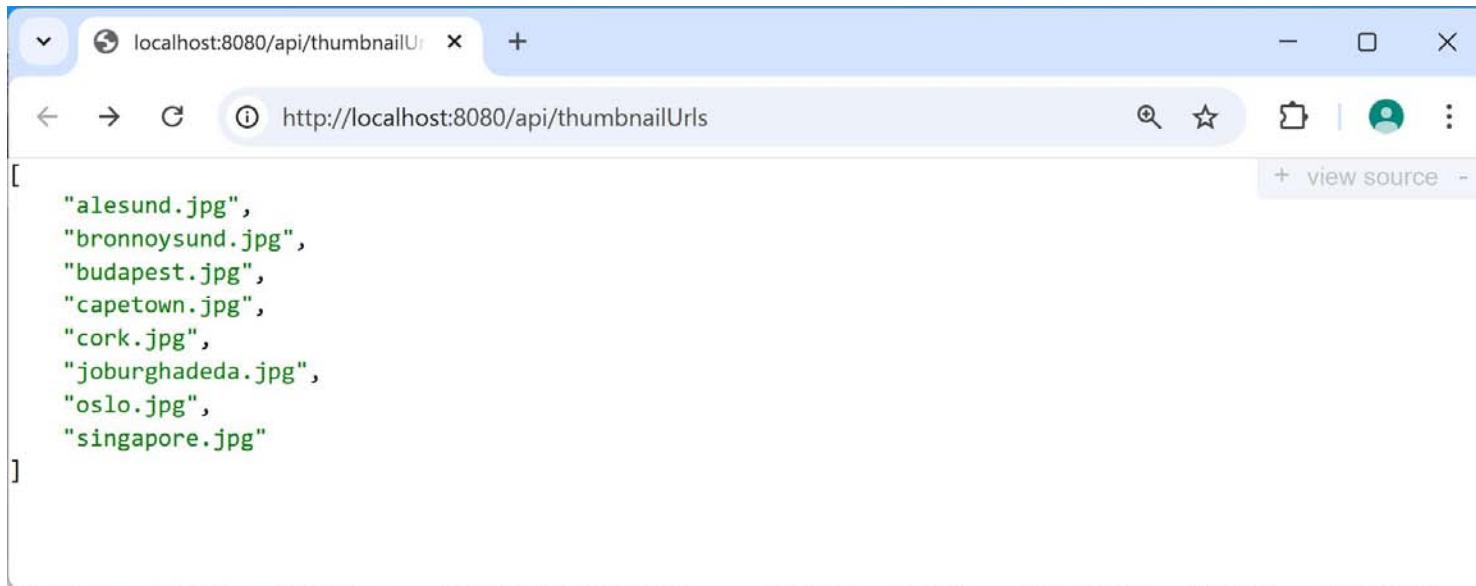
- We've implemented the REST service in Node.js
 - Open a command window in the `server` folder
 - Run the following commands

```
npm install
```

```
npm start
```
- The REST service starts up on port 8080

Testing the REST Service

- To test the REST service is working, browse here:
 - <http://localhost:8080/api/thumbnailUrls>



A screenshot of a web browser window. The address bar shows the URL <http://localhost:8080/api/thumbnailUrls>. The main content area displays a JSON array of file names:

```
[  
  "alesund.jpg",  
  "bronnoysund.jpg",  
  "budapest.jpg",  
  "capetown.jpg",  
  "cork.jpg",  
  "joburghadeda.jpg",  
  "oslo.jpg",  
  "singapore.jpg"  
]
```

Calling the REST Service

- Now see `Gallery4.tsx`
- After the `Gallery` component is first rendered:
 - Our effect hook makes an asynchronous REST call
 - When the REST call returns, we update component state
 - This causes the `Gallery` component to re-render
- The example also shows how to use:
 - `fetch()`, `async`, `await`

Summary

- Effect hooks
- Calling REST services

Annex: Memoization

- Overview
- Simple memoization
- Default memoization behaviour
- Custom comparison function
- Caching child content

Overview

- Memoization is a render optimization technique...
 - The ability to cache virtual DOM tree content
- Memoization is relevant if you have a pure component
 - i.e. a component that always generates the same virtual DOM tree for a given set of input properties
 - React caches the virtual DOM tree when the component is first rendered, then reuses the cached content later
- We'll see how to use memoization in this section
 - See the project demo-memoization

Simple Memoization (1 of 2)

- To memoize a component:
 - Wrap it in a call to `memo()` from the `react` module
- You can do it in 2 steps:

```
function SomeComponent() { ... }

const MemoizedComponent = memo( SomeComponent )
```

- Or you can do it in 1 step:

```
const MemoizedComponent = memo( () => { ... } )
```

Simple Memoization (2 of 2)

- Example - see `Page1.tsx`
- `Page1` component has 2 interesting child components:
 - A normal `Panel` component
 - A memoized `Panel` component
- Type in the textbox, to update the `text` state
 - Causes the `Page1` component (and children) to re-render
 - But the memoized `Panel` component isn't re-rendered

Default Memoization Behaviour (1 of 2)

- Imagine you have a memoized component...
 - ... and React might need to render the component
 - ... e.g. because state has changed in a parent component
- React does a shallow comparison of the component's current properties vs. previous properties
 - If all the properties are the same, cached content is used
 - If any property is different, component is re-rendered

Default Memoization Behaviour (2 of 2)

- Example - see `Page2.tsx`
- The `Panel` component has 2 properties:
 - `title` – this is invariant
 - `text` – this is the text from the text box
- The memoized version of the `Panel` component is now re-rendered on every keystroke
 - The `text` property is different, so no caching occurs!

Custom Comparison Function (1 of 2)

- You can pass a comparison function into `memo()`
 - The function takes previous and next properties
 - If the function returns `true`, cached content will be used

```
function func(prevProps, nextProps){ ...return true/false... }

const MemoizedComponent = memo(SomeComponent, func)
```

- Some reasons for defining a comparison function:
 - To compare only a subset of the properties
 - To do deep property comparisons (e.g. nested objects)
 - To use some global state in the comparisons

Custom Comparison Function (2 of 2)

- Example - see `Page3.tsx`
- A memoized `Panel` component has a comparison function
 - Just compares the prev/next length of `text` property
- When you type into the text box, this is what happens to that memoized `Panel` component:
 - If text is same length as before, cached content is used
 - If text is different length, component is re-rendered

Caching Child Content (1 of 2)

- We've seen how to use `memo()` to cache the entire virtual DOM tree for a component
- You can also cache a particular piece of child content within a component, using `useMemo()`

```
const cachedContent = useMemo(content, [dependencies])
```
- React will cache the specified content
 - If any dependency changes, React recomputes the cache

Caching Child Content (2 of 2)

- Example - see `Page4.tsx`
- The `Panel` component displays the date/time when the panel was initially displayed
 - This content is cached via `useMemo()`
- Note the dependency array is empty
 - So there's nothing in here that can change
 - So there's nothing to inhibit caching here ☺

React-Redux

1. Redux concepts
2. Example application
3. Understanding the example application

Section 1: Redux Concepts

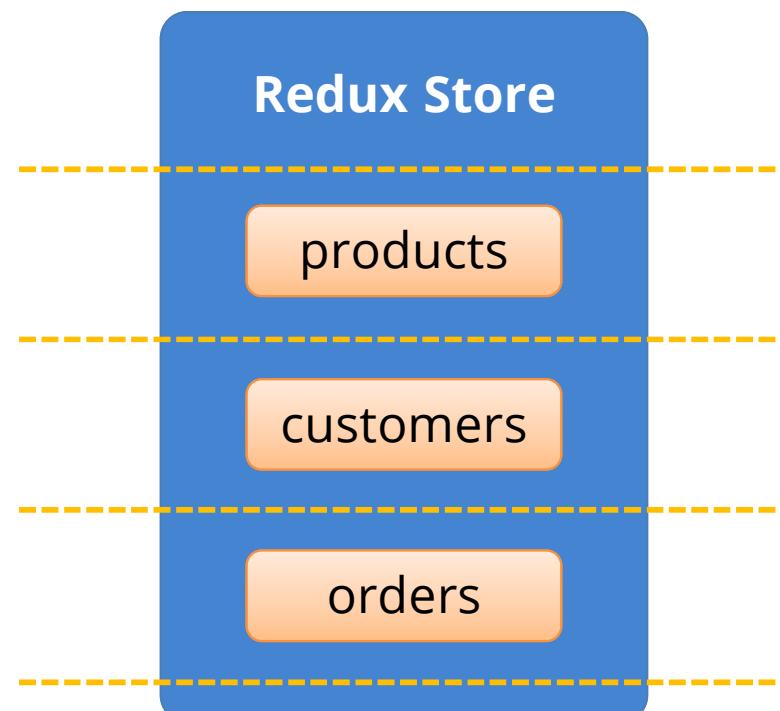
- Overview
- Organizing state into slices
- Identifying actions
- Defining reducer functions
- How it all fits together

Overview

- Redux helps you manage state in an application
- Redux has a *store*
 - Holds all the state for your application, globally
- Redux has *action objects*
 - Specify a change you want to make to the state
- Redux has *reducer functions*
 - Receives current state and action, updates state accordingly

Organizing State into Slices

- Redux stores your application state in *slices*



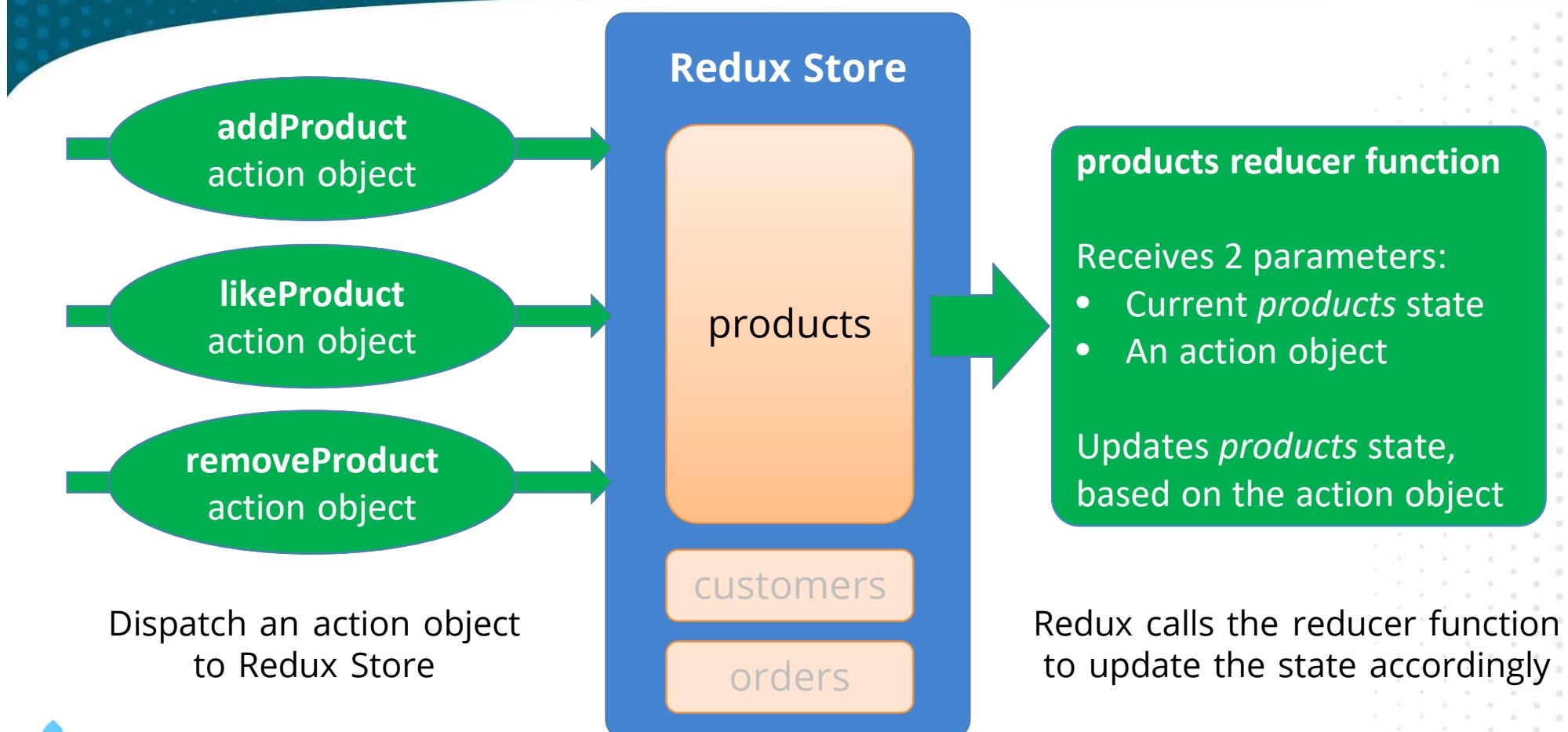
Identifying Actions

- For each slice of state:
 - Think about what *actions* can happen in the system, to cause that slice of state to be updated
- E.g. actions for the *products* slice:
 - Add a product, like a product, remove a product
- To cause an update:
 - You create an *action object*, describing a desired state change
 - You *dispatch* the action object to the Redux Store

Defining Reducer Functions

- Actions are carried out by *reducer functions*
 - You define a separate reducer function for each slice of state
- A reducer function receives 2 parameters:
 - The current slice of state (e.g. *products*)
 - An action object, indicating the change required
- A reducer function updates the state, as instructed by the action object

How it all Fits Together

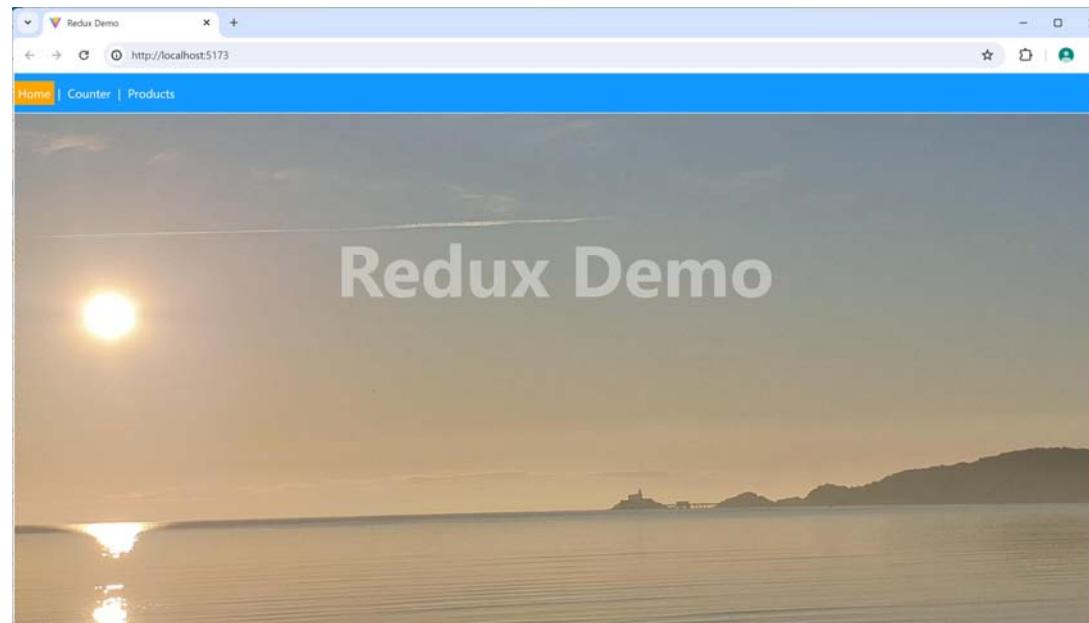


Section 2: Example Application

- Overview
- Viewing and updating the *counter* state slice
- Viewing and updating the *products* state slice

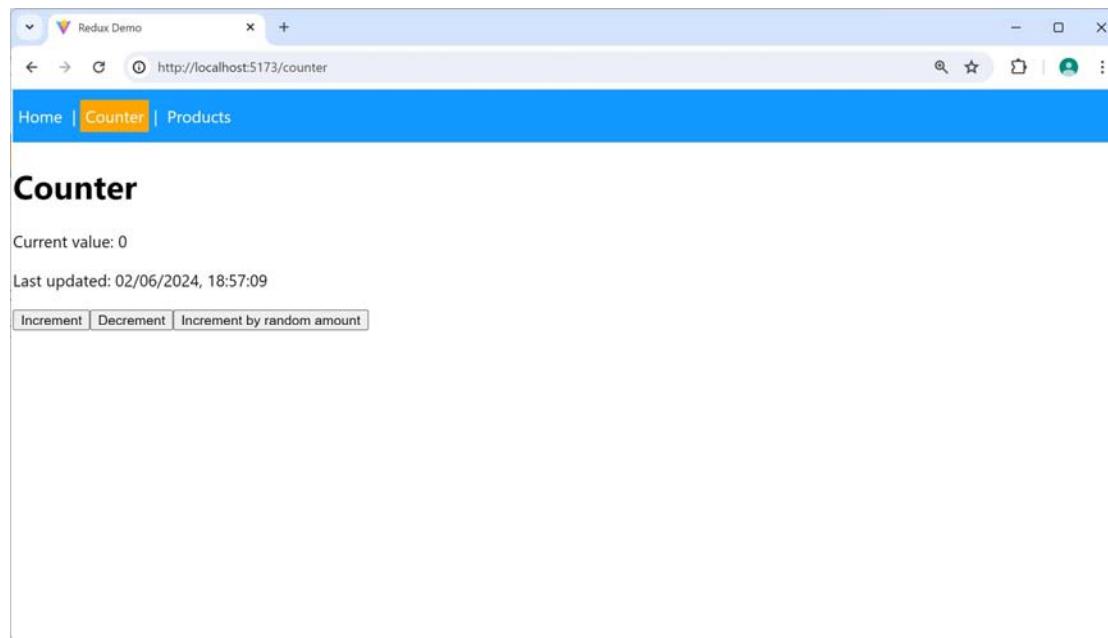
Overview

- In the demo-app folder, run `npm install` and `npm start`
- The application has 2 slices of state:
 - *counter*
 - *products*



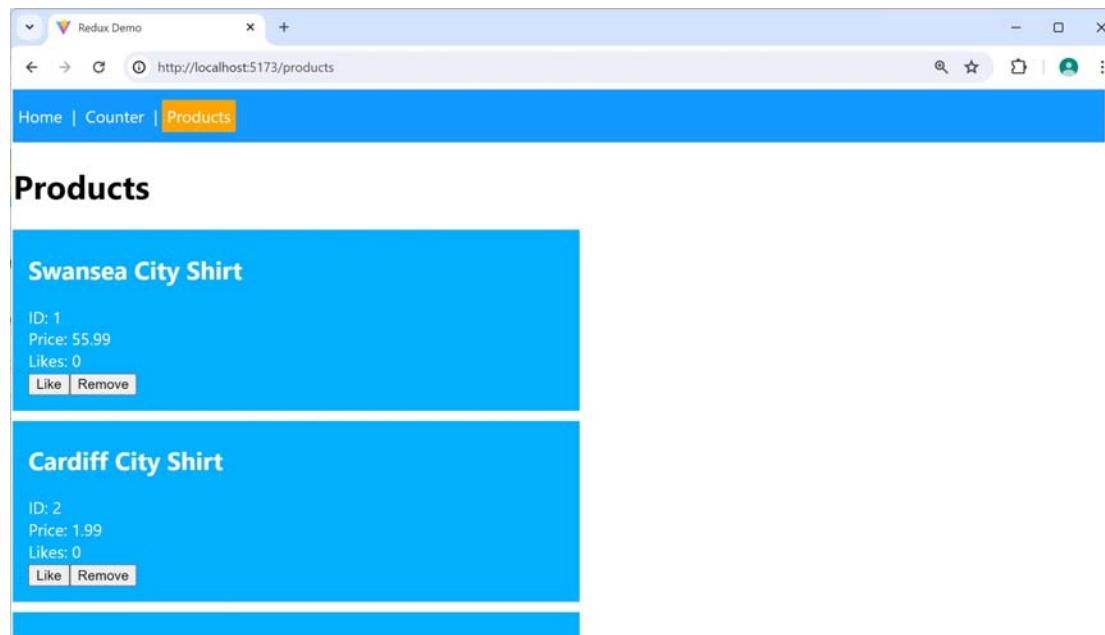
Viewing and Updating the *counter* State Slice

- Click the Counter menu item
 - Renders the Counter.tsx component
 - Views and updates the *counter* state slice



Viewing and Updating the *products* State Slice

- Click the Products menu item
 - Renders the `Products.tsx` component
 - Views and updates the *products* state slice



Section 3: Understanding the Example Application

- Dependencies for React Redux
- Creating slices
- Configuring the Redux Store
- Providing state to all components
- Accessing state in a component
- Creating and dispatching actions
- Understanding the *products* slice

Dependencies for React Redux

- To use React Redux, add the following dependencies in your package.json file:

```
"dependencies": {  
  "react-redux": "^8.0.2",  
  "@reduxjs/toolkit": "^1.8.3",  
  ...  
},  
  
"devDependencies": {  
  "@types/react-redux": "^7.1.24",  
  ...  
}
```

package.json

- Note:
 - Redux Toolkit (RTK) is optional, but strongly recommended
 - Simplifies many aspects of React Redux development

Creating Slices (1 of 3)

- RTK has a `createSlice()` function
 - Creates a slice of state for your application
 - The slice contains initial state, actions, and reducer logic
- We create 2 slices of state in our app, see:
 - `counterSlice.ts` – We'll discuss this first
 - `productsSlice.ts` – We'll discuss this later

Creating Slices (2 of 3)

- Here's how we create the *counter* slice:

```
import { createSlice } from '@reduxjs/toolkit'

const counterSlice = createSlice({
    name: 'counter',
    initialState: { value: 0, lastUpdated: new Date().toLocaleString() },
    reducers: {
        increment: (state) => {
            // Case-reducer for 'increment' action, increments value by 1.
        },
        decrement: (state) => {
            // Case-reducer for 'decrement' action, decrements value by 1.
        },
        incrementByAmount: (state, action) => {
            // Case-reducer for 'incrementByAmount' action, increments value by amount.
        }
    }
})

case-reducer functions
counterSlice.ts
```

Creating Slices (3 of 3)

- RTK combines our case-reducer functions into a single *reducer function*, like so:

```
const counterSlice = createSlice({  
  name: 'counter',  
  initialState: {...},  
  reducers: {  
    increment: (state) => {...},  
    decrement: (state) => {...},  
    incrementByAmount: (state, action) => {...}  
  }  
)
```

```
counterSlice.reducer(state, action) {  
  
  switch (action.type) {  
  
    case 'counter/increment':  
      increment(state)  
      break  
  
    case 'counter/decrement':  
      decrement(state)  
      break  
  
    case 'counter/incrementByAmount':  
      incrementByAmount(state, action)  
      break  
  }  
}
```

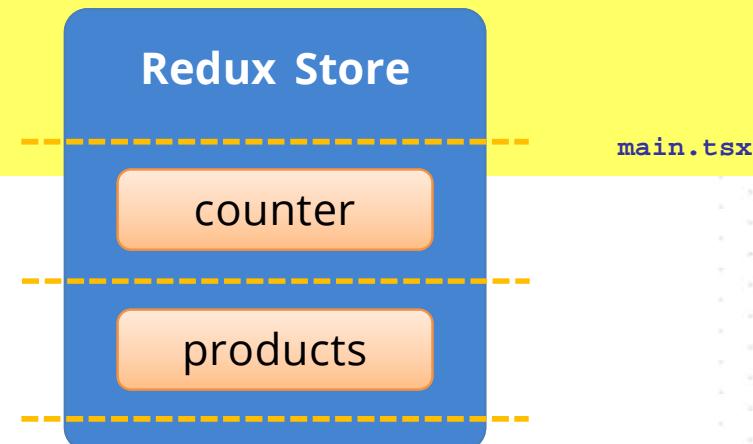


Configuring the Redux Store

- Once you've created your slices, the next step is to configure the Redux Store to house these slices

```
import { configureStore } from '@reduxjs/toolkit'
import counterSlice from './counterSlice'
import productsSlice from './productsSlice'

const store = configureStore({
  reducer: {
    counter: counterSlice.reducer,
    products: productsSlice.reducer
  }
})
```



main.tsx

Providing State to all Components

- Redux makes it easy to *provide* state to all components:

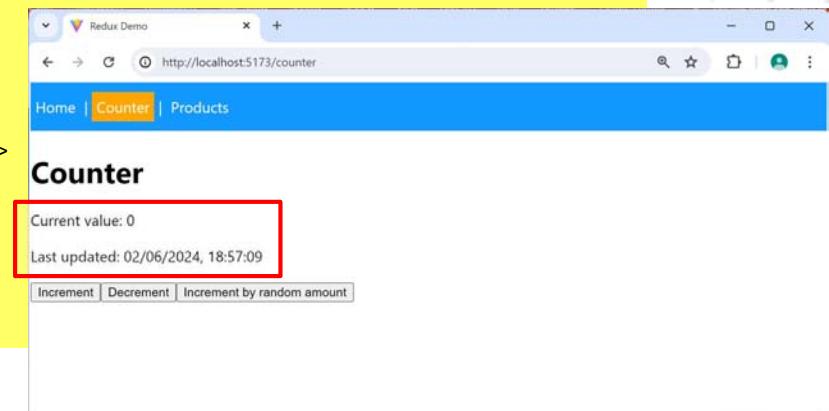
```
import { Provider } from 'react-redux'  
...  
  
ReactDOM.createRoot(document.getElementById('root')!).render(  
  <React.StrictMode>  
    <Provider store={store}>  
      <App />  
    </Provider>  
  </React.StrictMode>,  
)
```

main.tsx

Accessing State in a Component

- A component can access Redux Store state as follows:

```
import { useSelector } from 'react-redux'  
...  
  
export default function Counter() {  
  
  const counter: any = useSelector((store: any) => store.counter)  
  
  return (  
    <div>  
      <div>  
        <h1>Counter</h1>  
        <p>Current value: {counter.value}</p>  
        <p>Last updated: {counter.lastUpdated}</p>  
        ...  
      </div>  
    </div>  
  )  
}
```



Creating and Dispatching Actions (1 of 3)

- When you want to update state in a slice:
 - Create an action object
 - Dispatch the action object to Redux Store
 - (Redux Store invokes a reducer function, to do the work)
- Let's see how to do this...

Creating and Dispatching Actions (2 of 3)

- RTK generates *action-creator functions* for your slice
 - counterSlice.actions.increment()
 - counterSlice.actions.decrement()
 - counterSlice.actions.incrementByAmount()
- These functions create action objects for you:

```
import counterslice from './counterslice'  
...  
  
const num = Math.round(Math.random() * 10)  
const anActionObject = counterSlice.actions.incrementByAmount(num)
```

an action object

```
{  
  type: 'counter/incrementByAmount',  
  payload: num  
}
```

Creating and Dispatching Actions (3 of 3)

- To dispatch an action object to Redux Store:

```
import { useDispatch } from 'react-redux'  
...  
  
export default function Counter() {  
  
  const dispatch = useDispatch()  
  ...  
  dispatch(anActionObject)  
  ...  
}
```

Counter.tsx

- For a complete example
 - See Counter.tsx

Understanding the *products* Slice

- We've seen how the *counter* slice works
 - counterSlice.ts – Creates *counter* slice
 - Counter.tsx – Views and updates *counter* slice
- Now let's see how the *products* slice works
 - productsSlice.ts – Creates *products* slice
 - Products.tsx – Views and updates *products* slice

Summary

- Redux concepts
- Example application
- Understanding the example application

Redux Saga

1. Redux Saga concepts
2. Example application
3. Understanding the example application

Section 1: Redux Saga Concepts

- Overview
- A saga is a generator function
- Calling a generator
- Using a generator in a loop

Overview of Redux Saga

- Redux Saga is a library to help you run "side effects" in your React application, typically asynchronously
 - E.g., call a REST service
 - E.g., interact with local storage
 - E.g., perform a complex calculation
- Reasons for using Redux Saga to do this:
 - Easier to coordinate asynchronous tasks
 - Integrates very smoothly with Redux Store

A Saga is a Generator Function

- A saga is a *generator function* (an ES6 language feature)
 - The function signature has a *
 - Inside the function, use the **yield** keyword to yield control back to the client (optionally supplying a value)

```
function * ducklingGenerator() {  
  
  console.log( "\nGenerating duckling #1" )  
  yield "Huey"  
  
  console.log( "\nGenerating duckling #2" )  
  yield "Luey"  
  
  console.log( "\nGenerating duckling #3" )  
  yield "Duey"  
}
```

Calling a Generator

- Here's some client code, which shows how to use the generator function to get a series of values:

```
console.log("Before call to ducklingGenerator")
let genObj = ducklingGenerator()
console.log("After call to ducklingGenerator")

let res1 = genObj.next()
console.log(res1)

let res2 = genObj.next()
console.log(res2)

let res3 = genObj.next()
console.log(res3)
```

```
Before call to ducklingGenerator
After call to ducklingGenerator

Generating duckling #1
▶ {value: "Huey", done: false}

Generating duckling #2
▶ {value: "Luey", done: false}

Generating duckling #3
▶ {value: "Duey", done: false}
```

Using a Generator in a Loop

- You can use a generator with a `for-of` loop
 - Each iteration returns the next yielded value
 - When the generator is "done", the loop terminates

```
for (let res of ducklingGenerator())
  console.log(res)
```

Section 2: Example Application

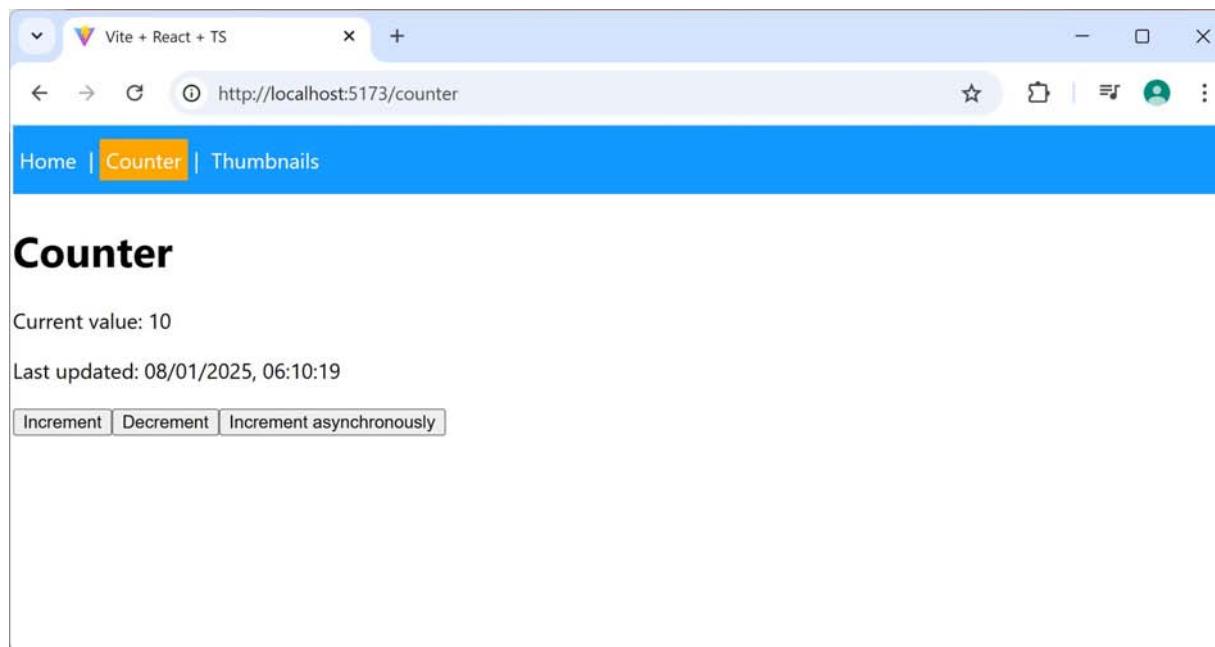
- Overview
- Using sagas to perform a simple async task
- Using sagas to call a REST service

Overview

- We've implemented a React app to demonstrate sagas:
 - Go to the `demo-app` folder
 - Run `npm install` and `npm run dev`
- You'll also need to run a Node.js REST server app:
 - Go to the `server` folder:
 - Run `npm install` and `npm start`

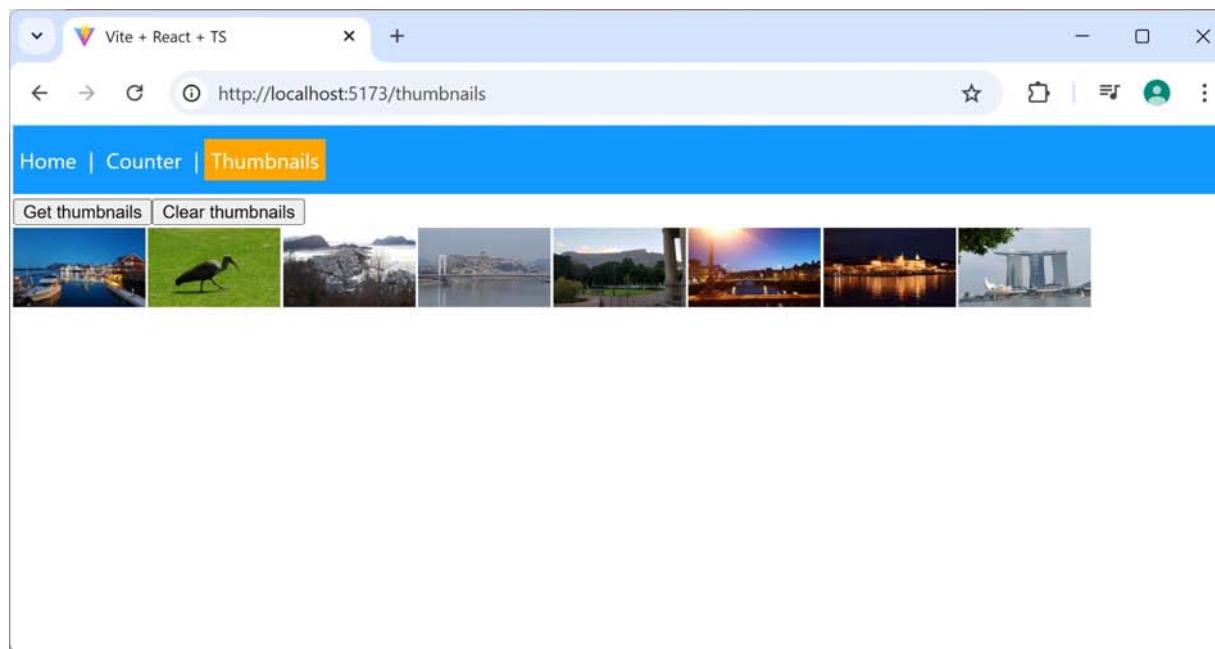
Using Sagas to Perform a Simple Async Task

- In the React demo app, click the Counter menu item
 - Then click the *Increment asynchronously* button
 - The app uses sagas to update a counter asynchronously



Using Sagas to Call a REST Service

- In the React demo app, click the Thumbnails menu item
 - Then click the *Get thumbnails* button
 - The app uses sagas to call a REST service asynchronously



Section 3: Understanding the Example Application

- Dependencies for Redux Saga
- Integrating Saga middleware into Redux
- Implementing the root saga
- Implementing watcher sagas
- Implementing worker sagas
- Putting it all together - counter component
- Putting it all together - thumbnails component

Dependencies for Redux Saga

- To use Redux Saga in a React app, add the following dependency in your package.json file

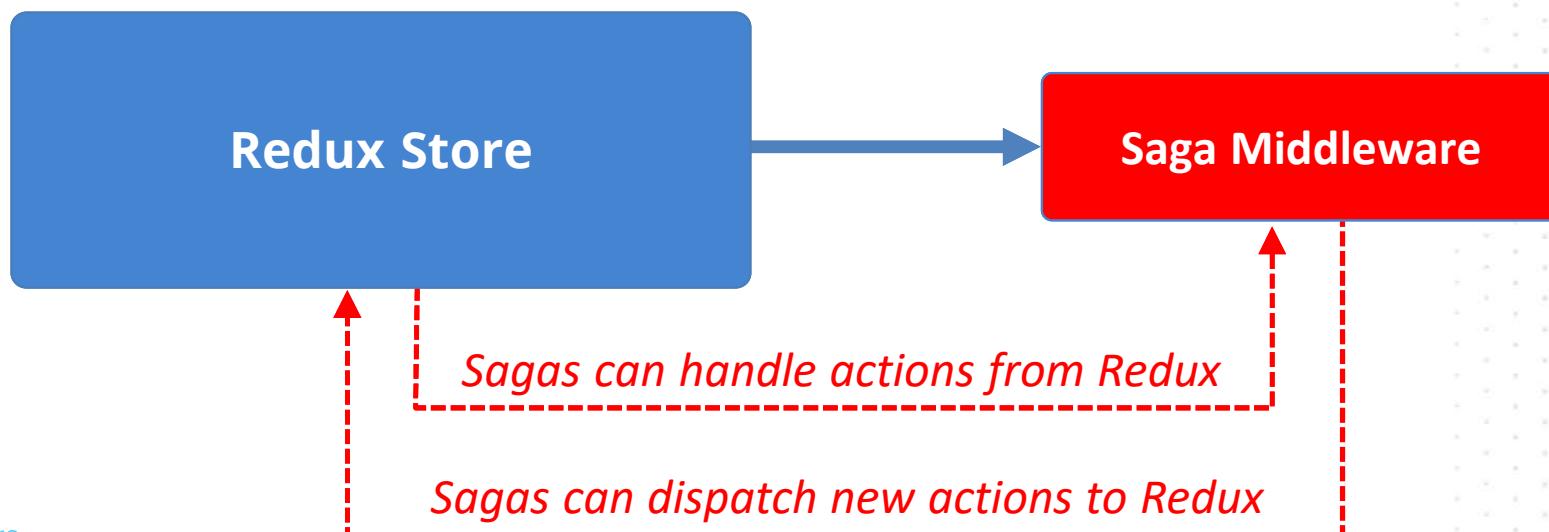
```
"dependencies": {  
  "redux-saga": "^1.1.3",  
  ...  
},
```

package.json

- The app also has the following dependencies...
 - React Redux (required by Redux Saga)
 - Redux Toolkit (to simplify React Redux code)
 - React Router (to make the app look pretty 😊)

Integrating Saga Middleware into Redux (1 of 2)

- When a React app starts, you must add *Saga middleware* into the Redux Store
 - Enables sagas to handle actions from Redux
 - Enables sagas to dispatch new actions to Redux



Integrating Saga Middleware into Redux (2 of 2)

- Here's how to add Saga middleware into the Redux Store:

```
import createSagaMiddleware from 'redux-saga'
import myRootSaga from './sagas'

...
const sagaMiddleware = createSagaMiddleware()

const store = configureStore({
  reducer: { ... },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(sagaMiddleware)
})

sagaMiddleware.run(myRootSaga)
```

main.tsx



Implementing the Root Saga

- On the previous slide, we ran the *root saga*:

```
sagaMiddleware.run(myRootSaga)
```

main.tsx

- What is the *root saga*?
 - It's a saga (i.e., a generator function)
 - It runs all *watcher sagas*, which watch for Redux actions

```
import { all } from 'redux-saga/effects'
...
export default function * myRootSaga() {
  yield all([
    watchIncrementAsync(),
    watchGetThumbnailUrlsAsync()
  ])
}
```

sagas.ts

Implementing Watcher Sagas

- What is a *watcher saga*?
 - It's a saga that takes an action from Redux Store, and passes it on to a *worker saga* to process it

```
import { takeEvery } from 'redux-saga/effects'  
...  
function * watchIncrementAsync() {  
    yield takeEvery('incrementAsync', doIncrementAsync)  
}
```

sagas.ts

Implementing Worker Sagas

- What is a *worker saga*?
 - It's a saga that performs an asynchronous operation
 - Typically sends an action back to Redux Store when done

```
import { put } from 'redux-saga/effects'
...
function * doIncrementAsync() {
  for (let i = 0; i < 10; i++) {
    yield delay(1000) // Delay for 1 second.
    yield put(counterSlice.actions.increment()) // Then send action to Redux.
  }
}
```

sagas.ts

- Note:
 - `delay()` returns a Promise, which we yield to Saga m/w
 - Saga m/w calls us back when the Promise is resolved

Putting it all Together - Counter Component

- `counterSlice.ts`
 - What state does the slice manage?
 - What actions does the slice reducer handle?
- `Counter.tsx`
 - How does the component display state?
 - What actions does the component dispatch?
- `sagas.ts`
 - What counter actions are handled by sagas, and how?

Putting it all Together - Thumbnails Component

- thumbnailsSlice.ts
 - What state does the slice manage?
 - What actions does the slice reducer handle?
- Thumbnails.tsx
 - How does the component display state?
 - What actions does the component dispatch?
- sagas.ts
 - What thumbnails actions are handled by sagas, and how?

Summary

- Redux Saga concepts
- Example application
- Understanding the example application

Testing React Applications

1. Introduction to web testing
2. Getting started with Vitest
3. Using Vitest to test React applications

Section 1: Introduction to Web Testing

- The traditional approach to testing web apps
- A better approach to testing
- Web testing frameworks

The Traditional Approach to Testing Web Apps

- Traditionally, developers used to take a manual approach to testing web apps
 - Look at the web page in a browser, to see if it looks right
 - Or call `alert()` many times, to display variable values
- What's wrong with this approach?

A Better Approach to Testing

- Here's a better approach to testing:
 - Write a formal test
 - Run the test to verify successful outcome
 - Keep test code as well as real code, so you can run the tests again and again as the code evolves

Web Testing Frameworks

- Many web testing frameworks have emerged and proved popular over the years, including:
 - Mocha
 - Jasmine
 - Jest
 - Vitest
- We're going to use Vitest
 - Fast, modern, Vite-native testing framework
 - See <https://vitest.dev/>

Section 2: Getting Started with Vitest

- Overview
- Minimal dependency for Vitest
- Writing tests
- Running tests

Overview

- In this section we'll see a bare-bones example of how to use Vitest to write and run tests
- We'll show:
 - How to install minimal Vitest dependencies
 - How to write simple tests
 - How to run tests
- See demo-app-1

Minimal Dependency for Vitest

- Here's the minimal dependency for using Vitest:

```
"devDependencies": {  
    "vitest": "^1.6.0",  
    ...  
}
```

package.json

- It's also convenient to define the following script entry:

```
"scripts": {  
    "test": "vitest",  
    ...  
}
```

package.json

Writing Tests

- Here are some simple tests using Vitest:

```
import { describe, it, expect } from 'vitest'

describe('vitest simple example', () => {

    it('shows 2 + 2 is 4', () => {
        expect(2 + 2).toBe(4);
    });

    it('shows 2 + 2 is not 5', () => {
        expect(2 + 2).not.toBe(5);
    });
});
```

`example.test.ts`

- Vitest is compatible with the Jest test framework
 - It uses `describe`, `it`, `expect`, etc. in the same way as Jest

Running Tests

```
npm run test
```



```
PS C:\ReactEnterpriseDev\Demos\05-TestingReactApp\demo-app-1> npm run test
> demo-app-1@0.0.0 test
> vitest

[DEV] v1.6.0 C:/ReactEnterpriseDev/Demos/05-TestingReactApp/demo-app-1

✓ src/example.test.ts (2)
  ✓ vitest (2)
    ✓ shows 2 + 2 is 4
    ✓ shows 2 + 2 is not 5

Test Files 1 passed (1)
Tests 2 passed (2)
Start at 18:58:26
Duration 761ms (transform 78ms, setup 0ms, collect 67ms, tests 12ms, environment 1ms, prepare 258ms)

PASS Waiting for file changes...
press h to show help, press q to quit
```

Section 3: Using Vitest to Test React Applications

- Overview
- Full dependencies for Vitest and React
- Vitest configuration
- Test setup
- Writing tests for React
- Running the tests

Overview

- In this section we'll see how to use Vitest to test React applications
- We'll show:
 - How to install full dependencies for Vitest and React
 - How to configure Vitest
 - How to define test setup
 - How to write and run tests
- See demo-app-2

Full Dependencies for Vitest and React (1 of 5)

- Here are the full dependencies for using Vitest to test a React application:

```
"devDependencies": {  
    "vitest": "^1.6.0",  
    "jsdom": "^24.1.0",  
    "@testing-library/react": "^15.0.7",  
    "@testing-library/jest-dom": "^6.4.5",  
    ...  
}
```

package.json

- The following slides describe these dependencies...

Full Dependencies for Vitest and React (2 of 5)

- Here are the full dependencies for using Vitest to test a React application:

```
"devDependencies": {
    "vitest": "^1.6.0",
    "jsdom": "^24.1.0",
    "@testing-library/react": "^15.0.7",
    "@testing-library/jest-dom": "^6.4.5",
    ...
}
```

package.json

- **vitest**
 - Vitest framework for defining and running tests
 - Defines `describe`, `it`, `expect`, etc. (as we saw earlier)

Full Dependencies for Vitest and React (3 of 5)

- Here are the full dependencies for using Vitest to test a React application:

```
"devDependencies": {  
    "vitest": "^1.6.0",  
    "jsdom": "^24.1.0",  
    "@testing-library/react": "^15.0.7",  
    "@testing-library/jest-dom": "^6.4.5",  
    ...  
}
```

package.json

- **jsdom**
 - Pure JavaScript implementation of a DOM tree
 - Emulates a web browser environment, for testing a web app

Full Dependencies for Vitest and React (4 of 5)

- Here are the full dependencies for using Vitest to test a React application:

```
"devDependencies": {  
    "vitest": "^1.6.0",  
    "jsdom": "^24.1.0",  
    "@testing-library/react": "^15.0.7",  
    "@testing-library/jest-dom": "^6.4.5",  
    ...  
}
```

package.json

- **@testing-library/react**
 - React Testing Library
 - Enables you to test React components

Full Dependencies for Vitest and React (5 of 5)

- Here are the full dependencies for using Vitest to test a React application:

```
"devDependencies": {  
    "vitest": "^1.6.0",  
    "jsdom": "^24.1.0",  
    "@testing-library/react": "^15.0.7",  
    "@testing-library/jest-dom": "^6.4.5",  
    ...  
}
```

package.json

- **@testing-library/jest-dom**
 - Defines a set of custom Jest matchers
 - Enables you to expressively test the content in the DOM tree

Vitest Configuration (1 of 5)

- Configure Vitest as follows, to facilitate web testing:

```
import react from '@vitejs/plugin-react';
import {defineConfig} from 'vitest/config';

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [react()],
  test: {
    globals: true,
    environment: 'jsdom',
    setupFiles: './tests/setup.ts',
  },
});
```

vite.config.ts

- The following slides describe the details...

Vitest Configuration (2 of 5)

- Configure Vitest as follows, to facilitate web testing:

```
import react from '@vitejs/plugin-react';
import {defineConfig} from 'vitest/config';

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [react()],
  test: {
    globals: true,
    environment: 'jsdom',
    setupFiles: './tests/setup.ts',
  },
});
```

vite.config.ts

- import {defineConfig} from 'vitest/config'
 - (Rather than import {defineConfig} from 'vite')
 - Enables you to specify test-related configuration

Vitest Configuration (3 of 5)

- Configure Vitest as follows, to facilitate web testing:

```
import react from '@vitejs/plugin-react';
import {defineConfig} from 'vitest/config';

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [react()],
  test: {
    globals: true,
    environment: 'jsdom',
    setupFiles: './tests/setup.ts',
  },
});
```

vite.config.ts

- `globals: true`
 - Makes Vitest functions (`describe`, `expect`, `it`) available globally, without the need to import manually in your tests

Vitest Configuration (4 of 5)

- Configure Vitest as follows, to facilitate web testing:

```
import react from '@vitejs/plugin-react';
import {defineConfig} from 'vitest/config';

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [react()],
  test: {
    globals: true,
    environment: 'jsdom',
    setupFiles: './tests/setup.ts',
  },
});
```

vite.config.ts

- environment: 'jsdom'
 - Tells Vitest to operate in *jsdom* mode
 - Simulates a browser environment (creates a document obj)

Vitest Configuration (5 of 5)

- Configure Vitest as follows, to facilitate web testing:

```
import react from '@vitejs/plugin-react';
import {defineConfig} from 'vitest/config';

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [react()],
  test: {
    globals: true,
    environment: 'jsdom',
    setupFiles: './tests/setup.ts',
  },
});
```

vite.config.ts

- setupFiles: './tests/setup.ts'
 - Tells Vitest where to find additional test setup info
 - See next slide...

Test Setup

- Here's the test setup :

```
import { afterEach } from 'vitest';
import { cleanup } from '@testing-library/react';

afterEach(() => {
  cleanup();
});
```

tests/setup.ts

- Ensures `cleanup()` is called automatically after each test
 - `cleanup()` is defined in the React Testing Library
 - Disposes test-related resources, to prevent memory leaks

Writing Tests for React (1 of 4)

- Here's a test for the App component:

```
import { render, screen } from '@testing-library/react'
import '@testing-library/jest-dom'

import App from './App'

describe('App', () => {

  it('renders heading', () => {
    render(<App />)
    const h1Element = screen.getByText(/This is my cool app/i)
    expect(h1Element).toBeInTheDocument()
  });
});
```

[App.test.tsx](#)

- See following slides for an explanation...

Writing Tests for React (2 of 4)

- Here's a test for the App component:

```
import { render, screen } from '@testing-library/react'
import '@testing-library/jest-dom'

import App from './App'

describe('App', () => {

  it('renders heading', () => {
    render(<App />)
    const h1Element = screen.getByText(/This is my cool app/i)
    expect(h1Element).toBeInTheDocument()
  });
});
```

App.test.tsx

- `render()` renders a component in the virtual browser

Writing Tests for React (3 of 4)

- Here's a test for the App component:

```
import { render, screen } from '@testing-library/react'
import '@testing-library/jest-dom'

import App from './App'

describe('App', () => {

  it('renders heading', () => {
    render(<App />)
    const h1Element = screen.getByText(/This is my cool app/i)
    expect(h1Element).toBeInTheDocument()
  });
});
```

App.test.tsx

- screen gives access to content in document.body in the virtual browser's DOM tree

Writing Tests for React (4 of 4)

- Here's a test for the App component:

```
import { render, screen } from '@testing-library/react'
import '@testing-library/jest-dom'

import App from './App'

describe('App', () => {

  it('renders heading', () => {
    render(<App />)
    const h1Element = screen.getByText(/This is my cool app/i)
    expect(h1Element).toBeInTheDocument()
  });
});
```

App.test.tsx

- `toBeInTheDocument()` is a jest-dom custom matcher
 - Enables you to write expressive tests for DOM content

Running the Tests

```
npm run test
```



```
PS C:\ReactEnterpriseDev\Demos\05-TestingReactApp\demo-app-2> npm run test
> demo-app-2@0.0.0 test
> vitest

[DEV] v1.6.0 C:/ReactEnterpriseDev/Demos/05-TestingReactApp/demo-app-2
✓ src/App.test.tsx (1)
  ✓ App (1)
    ✓ renders heading

Test Files 1 passed (1)
Tests 1 passed (1)
Start at 21:07:04
Duration 2.37s (transform 129ms, setup 351ms, collect 210ms, tests 103ms, environment 925ms, prepare 327ms)

PASS Waiting for file changes...
press h to show help, press q to quit
```

Summary

- Introduction to web testing
- Getting started with Vitest
- Using Vitest to test React applications

What's New in Modern ECMAScript

1. Miscellaneous language features
2. Destructuring
3. Modules

1. Miscellaneous Language Features

- Block-scope variables
- String interpolation
- Multi-line strings
- The spread operator
- Arrow functions

Block-Scope Variables

- ES5 doesn't support block-scope variables properly

- Local variables are hoisted to the top of the function
 - So they are accessible in the whole function!

```
if (true) {  
    var s = "Hi"  
}  
console.log(s) // Hi
```

ES5

- ES6 and above support block-scope variables properly

- Use the `let` keyword rather than `var`

```
if (true) {  
    let s = "Hi"  
}  
console.log(s) // Error!
```

ES6++

String Interpolation

- ECMAScript now supports string interpolation, via template literals
 - Enclose the string in `back-ticks`

```
let n = 'John'  
let a = 21  
  
let html = `<b>${n}</b> will be ${a+1} soon`  
console.log(html)
```

Multi-Line Strings

- ECMAScript also supports multi-line strings
 - Enclose the string in back-ticks

```
let p = { name: 'Jane', age: 21 }

const personHtml =
  `<dl>
    <dt>Person info</dt>
    <dd>Name: ${p.name}</dd>
    <dd>Age next birthday: ${p.age+1}</dd>
  </dl>` 

console.log(personHtml)
```

The Spread Operator

- You can use the spread operator . . . to expand an array into its separate values

```
let nums = [100, 200, 300]
console.log(...nums)
```

- You can use the spread operator to clone an array

```
let nums = [100, 200, 300]

let numsClone = [...nums]
numsClone.push(400)

console.log(nums)      // [100, 200, 300]
console.log(numsClone) // [100, 200, 300, 400]
```

Arrow Functions (1 of 2)

- ECMAScript now supports arrow functions
 - () encloses params - you can omit () if 1 param
 - => separates params from body
 - The function body is implicitly the return expression

```
let getFullName = (fn,ln) => `${ln}, ${fn}`
```

- You call an arrow function just like a regular function

```
console.log(getFullName('John','Smith'))
```

Arrow Functions (2 of 2)

- You can define an arrow function over multiple lines
 - Enclose function body in {} braces
 - Use an explicit return statement to return a value

```
let getFullName = (fn,ln) => {  
    let fullName = `${ln.toUpperCase()} ${fn}`;  
    return fullName  
}
```

2. Destructuring

- Overview of destructuring
- Destructuring an array
- Destructuring an object literal
- Specifying default values
- Destructuring parameters
- Destructuring a return value

Overview of Destructuring

- Destructuring allows you to unpack items in an array, object literal, or function arguments/returns
 - Similar-ish to tuples in other languages
- Destructuring looks a bit odd initially, but you soon get used to it
 - And it can simplify your code considerably!

Destructuring an Array (1 of 2)

- You can use destructuring with an array
 - Assigns array items to variables in left-hand-side array

```
let [a,b,c] = [1,2,3]      // a=1, b=2, c=3
```

- If surplus variables, they are undefined

```
let [a,b,c,d] = [1,2,3]  // d is undefined
```

- If surplus array items, they are not assigned

```
let [a,b] = [1,2,3]      // a=1, b=2
```

Destructuring an Array (2 of 2)

- You can use the destructure part of an array if you like

```
let [elem0, elem1, ...others] = [10, 20, 30, 40]

console.log(elem0)          // 10
console.log(elem1)          // 20
console.log(others.length)   // 2
console.log(others[0])       // 30
console.log(others[1])       // 40
```

Destructuring an Object Literal (1 of 2)

- You can use destructuring with an object literal
 - Assigns object properties to variable names in LHS object

```
let {x, y, z} = {x:10, y:20, z:30}
console.log(x, y, z)      // 10 20 30
```

- You can give different names for variables in LHS object
 - Use the syntax `keyName : newKeyName`

```
let {x:h, y:w, z:d} = {x:10, y:20, z:30}
console.log(h, w, d)      // 10 20 30
```

Destructuring an Object Literal (1 of 2)

- LHS object can specify a subset of properties

```
let {x, y} = {x:10, y:20, z:30}
console.log(x, y)           // 10 20
```

- LHS object can specify missing properties
 - Will be undefined

```
let {x, y, z} = {x:10, y:20}
console.log(x, y, z)        // 10 20 undefined
```

Specifying Default Values

- You can specify default values for array items:

```
let [http='80',https='443'] = ['8080']

console.log(http, https) // 8080 443
```

- You can specify default values for object properties:

```
let {x=0, y=0, z=0} = {x: 100, y: 200}

console.log(x, y, z) // 100 200 0
```

Destructuring Parameters (1 of 3)

- You can use destructuring with function parameters
 - In the function declaration, group parameters in { }
 - In the function call, pass in an object literal

```
function displayPoint({x, y, z}) {
  console.log(x, y, z)
}

displayPoint({x:10, y:20, z:30}) // 10 20 30
```

Destructuring Parameters (2 of 3)

- You can define default values for individual properties

```
function displayPoint({x=0, y=0, z=0}) {
  console.log(x, y, z)
}

displayPoint({x:10, z:30})    // 10 0 30
```

- You can rename properties, if you like

```
function displayPoint({x:h=0, y:w=0, z:d=0}) {
  console.log(h, w, d)
}

displayPoint({x:10, z:30})    // 10 0 30
```

Destructuring Parameters (3 of 3)

- You can define a default value for the parameter object
 - You can even combine this with individual defaults!

```
function displayPoint({x=0, y=0, z=0} = {x:1, y:2, z:3}) {
  console.log(x, y, z)
}

displayPoint({x:10, z:30})    // 10 0 30
displayPoint({})              // 0 0 0
displayPoint()                // 1 2 3
```

Destructuring a Return Value

- If you return an object from a function...

```
function convertEuros(euroAmount) {  
  return {  
    USD: euroAmount * 1.17,  
    GBP: euroAmount * 0.89,  
    NOK: euroAmount * 9.27  
  }  
}
```

- You can destructure the return value (no obligation ☺)

```
let allAmounts = convertEuros(100)  
let {USD, GBP, NOK} = convertEuros(100)
```

3. Modules

- Overview
- Modules in ES6 and above
- Exporting artifacts
- Importing artifacts

Overview

- A large web application might contain hundreds of classes and/or functions
 - It's infeasible to put all this code into a single source file
 - Instead, you split the code across separate source files
- In JavaScript:
 - A source file can be interpreted as a *module*
 - There can be only one module per file
 - A module is defined in 1 file (can't span multiple files)

Modules in ES6 and Above

- A *module* is an ECMAScript source file
 - Defines artifacts such as variables, functions, classes etc.
 - Artifacts are local and private to that module, by default
- You can export artifacts from one module
 - Other modules can import them

Exporting Artifacts

- Here's module1.js, it exports some artifacts

```
console.log("Start of module1.js")

let a = 100
function funcA() {console.log('funcA')}
class ClassA {}

export let b = 42
export function funcB() {console.log('funcB')}
export class ClassB {}

console.log("End of module1.js")
```

module1.js

Importing Artifacts

- Here's module2.js, it imports some artifacts

```
import { b }                  from './module1'
import { funcB, ClassB } from './module1'

console.log("Start of module2.js")

console.log(b)
funcB()
let obj = new ClassB()

console.log("End of module2.js")
```

module2.js

Summary

- Miscellaneous language features
- Destructuring
- Modules

TypeScript Essentials

1. Getting started with TypeScript
2. Functions
3. Classes
4. Inheritance and interfaces

1. Getting Started with TypeScript

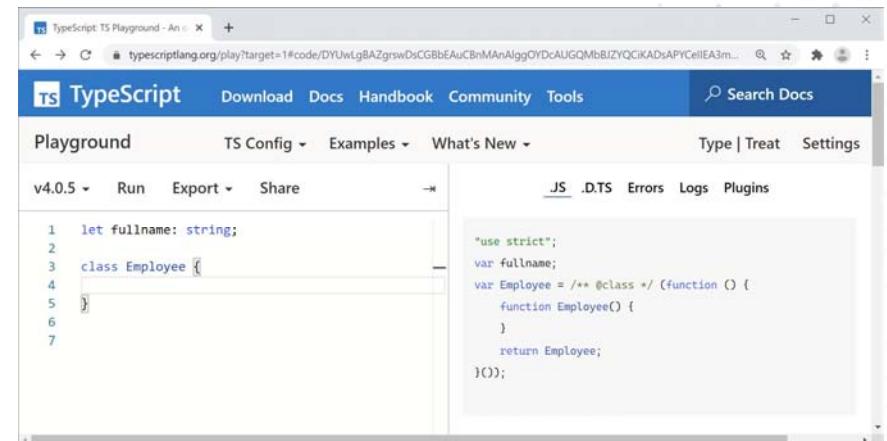
- Overview
- Using the TypeScript Playground
- Defining types in declarations
- TypeScript basic types
- Arrays
- Tuples
- Enums

Overview

- You can use TypeScript to enhance the type-safety of your React applications
 - TypeScript supports ES6++ features, plus...
 - Data typing ☺
 - Interfaces
 - Decorators (similar to annotations in Java)
 - Class member variables (i.e. fields)
 - Generics
 - Keywords `public`, `private`, `protected`

Using the TypeScript Playground

- There's a handy TypeScript transpiler available online, where you can practice your TypeScript skills
 - <http://www.typescriptlang.org/play/>
- Choose ESxxx as the target language, via the menu
TS Config | Target | ESxxx
- Then try out some TS!

A screenshot of the TypeScript playground interface. The top navigation bar includes links for 'TypeScript', 'Download', 'Docs', 'Handbook', 'Community', 'Tools', and a search bar. Below the navigation is a toolbar with buttons for 'Playground', 'TS Config', 'Examples', 'What's New', 'Run', 'Export', and 'Share'. The main area has tabs for 'JS' (selected), 'DTS', 'Errors', 'Logs', and 'Plugins'. On the left, there is a code editor containing the following TypeScript code:

```
1 let fullname: string;
2
3 class Employee {
4
5 }
6
7
```

On the right, the corresponding JavaScript output is shown:

```
"use strict";
var fullname;
var Employee = /** @class */ (function () {
    function Employee() {
    }
    return Employee;
}());
```

Defining Types in Declarations

- TS allows you to define types in declarations
 - Variables, parameters, and function return types
 - Use the syntax `variableName : type`

```
let name1 = 'Fred'  
let name2: string = 'Wilma'
```

TS code

Transpiles to

```
var name1 = 'Fred'  
var name2 = 'Wilma'
```

ES code

TypeScript Basic Types (1 of 2)

- number – floating point or integral number
- boolean – true or false
- string – literal text or template string `\\$ {x}`
- function – a function
- object – non-primitive type (e.g. object, array)

TypeScript Basic Types (2 of 2)

- void
 - no type (e.g. function with no return)
- never
 - function that never returns normally
- any
 - disables type-checks, e.g. legacy code
- null
 - data type of null value
- undefined
 - data type of undefined value

Arrays

- TS supports arrays
 - Use the type of the elements followed by []
 - Or use the generic array type `Array<elemType>`

```
let a: number[] = [1,2]
let b: Array<number> = [3,4]
```

TS code

Transpiles to

```
var a1 = [1,2]
var a2 = [3,4]
```

ES code

Tuples

- TS supports tuples
 - Effectively an array of mixed types

```
let bd: [number, string]  
  
bd = [3, 'December']  
  
let day: number = bd[0]  
let month: string = bd[1]
```

TS code

Transpiles to

```
var bd  
bd = [3, 'December']  
var day = bd[0]  
var month = bd[1]
```

ES code

Enums

- TS supports enums, to represent a fixed set of states

```
enum Color {R=1, G, B}  
let c: Color = Color.R
```

TS code

Transpiles to

```
var Color  
  
(function (Color) {  
  Color[Color["R"] = 1] = "R"  
  Color[Color["G"] = 2] = "G"  
  Color[Color["B"] = 3] = "B"  
})(Color || (Color = {}))  
  
var c = Color.R
```

ES code

- Enum mnemonics can be strings

```
enum Color {R="rouge", G="vert", B="bleu"}
```

2. Functions

- Typed parameters and returns
- Default parameters
- Optional parameters
- Rest parameters
- Lambda expressions

Typed Parameters and Returns

- Functions in TS are similar to JS, but...
 - TS allows you to declare parameter and return types
 - TS performs type-checking

```
function calcTotalSalary(basic: number,
                        bonus: number,
                        director: boolean) : number {
  var earnings: number = basic + bonus
  if (director) {
    earnings *= 2
  }
  return earnings
}
```

Default Parameters

- You can specify default values for parameters

```
function calcTotalSalary(basic: number,
                        bonus: number = 0.0,
                        director: boolean = false) : number {
  var earnings: number = basic + bonus
  if (director) {
    earnings *= 2
  }
  return earnings
}
```

- Note:
 - Default params don't have to appear after required params - you can pass `undefined` to use a default

Optional Parameters

- You can indicate parameter(s) are optional
 - Append question mark ? after the parameter name
 - Optional parameters must follow required parameters
 - In the function, check if the client passed in a value

```
function calcTotalSalary(basic: number,
                        bonus: number = 0.0,
                        director: boolean = false,
                        offshoreSlushFund?: number) : number {
  var earnings: number = basic + bonus
  if (offshoreSlushFund) {
    earnings += offshoreSlushFund
  }
  if (director) {
    earnings *= 2
  }
  return earnings
}
```

Rest Parameters

- You can define variadic functions via "rest" parameters
 - Define an array parameter, precede param name with ...
 - Must be at the end of the parameter list

```
function getFullName(fname: string, ...othernames: string[]) {  
    return fname + " " + othernames.join(" ")  
}
```

Lambda Expressions

- TS supports lambda expressions
 - () contain the params (you can omit () if only 1 param)
 - => separates params from the lambda body
 - The lambda body is implicitly the return expression

```
var getFullName = (fn: string, ln: string): string => fn + ' ' + ln
```

- You invoke a lambda expression like a regular function

```
console.log(getFullName('Peter', 'John'))
```

3. Classes

- Defining a simple class
- Constructors
- Read-only properties
- Encapsulation
- Constructor parameter properties
- Defining additional methods
- Defining static members

Defining a Simple Class

- TS makes it much easier to define classes
 - Use the class keyword
 - Define members using familiar OO syntax

```
class Employee {  
    name: string = ''  
    salary: number = -1  
}
```

- You can create objects using familiar JS syntax

```
let empl = new Employee()  
empl.name = "Paul"  
empl.salary = 42000
```

Constructors

- You can define a constructor in a class
 - Define a method named constructor

```
class Employee {  
    name: string  
    salary: number  
  
    constructor(name: string, salary: number) {  
        this.name = name  
        this.salary = salary  
    }  
}
```

```
let emp1 = new Employee("Lydia", 43000)
```

Read-Only Properties

- TypeScript has the concept of read-only fields
 - Declare a field with the `readonly` modifier
 - Must be initialized in constructor, can't be modified after

```
class Circle {  
  
    readonly radius: number  
  
    constructor(radius: number) {  
        this.radius = radius  
    }  
}  
  
let myCircle = new Circle(10)  
console.log(myCircle.radius)      // OK  
myCircle.radius = 42             // Error
```

Encapsulation (1 of 2)

- You can qualify members with access modifiers
 - `public` - accessible to anyone (this is the default)
 - `protected` - accessible to this class plus subclasses
 - `private` - accessible to this class only
- You can also define getters and setters to encapsulate access to member variables
 - `get xxx()`
 - `set xxx()`

Encapsulation (2 of 2)

```
class Employee {  
    private _name: string  
    private _salary: number  
  
    constructor(_name: string, _salary: number) {  
        this._name = _name  
        this._salary = _salary  
    }  
  
    get name(): string {  
        return this._name  
    }  
  
    set name(newName: string) {  
        this._name = newName  
    }  
  
    get salary(): number {  
        return this._salary  
    }  
}
```

```
let emp = new Employee("Thomas", 10000)  
emp.name = "Tom"  
console.log(`${emp.name} earns ${emp.salary}`)
```

Constructor Parameter Properties

- The examples on the previous slides declared instance variables and initialized them in the constructor
- This is such a common practice that TS provides a shortcut, "constructor parameter properties"
 - Define params as public/protected/private
 - TS automatically declares/initializes instance variables

```
class Employee {  
    constructor(private _name: string, private _salary: number) {}  
    ...  
}
```

Defining Additional Methods

- You can define additional methods as necessary
 - Encapsulate logic and business rules for your class

```
class Employee {  
    constructor(private _name: string, private _salary: number) {}  
  
    payRise(amount: number): void {  
        this._salary += amount  
    }  
  
    isHigherTax (): boolean {  
        return this._salary > 42000  
    }  
    ...  
}  
  
let emp = new Employee("Tom", 10000)  
emp.payRise(100000)  
console.log("Higher tax? " + emp.isHigherTax())
```

Defining Static Members (1 of 2)

- You can define class-wide members
 - Belong to the whole class, not to a particular instance
- To define a class wide member:
 - Prefix definition with `static`
 - Can also define as `public/protected/private`
 - Works for member variables and methods
- To access a static member, prefix with class name

Defining Static Members (2 of 2)

- Example

```
class Employee {  
  
    private static _taxThreshold: number = 42000  
  
    constructor(private _name: string, private _salary: number) {}  
  
    isHigherTaxPayer(): boolean {  
        return this._salary > Employee._taxThreshold  
    }  
  
    static get taxThreshold(): number {  
        return Employee._taxThreshold  
    }  
...  
}  
...  
    console.log("Tax threshold is " + Employee.taxThreshold)  
}
```

4. Inheritance and Interfaces

- Inheritance in TypeScript
- Additional inheritance techniques
- Using an interface to specify methods
- Using an interface as a property bag
- Using an interface as a func signature
- Using an interface as an array type

Inheritance in TypeScript

- A class can extend another class
 - The subclass uses the `extends` keyword
- The subclass can override superclass methods
 - The subclass can invoke superclass methods and constructors, via the `super` keyword
- Under the covers, TS inheritance is transpiled to prototypical inheritance in JavaScript

Additional Inheritance Techniques

- Additional techniques in the superclass:
 - Can be abstract (cannot instantiate)
 - Can have abstract methods (must override)
 - Can have protected items (accessible to subclasses)
- Additional techniques in client code
 - Downcast via instanceof and <type> typecasts

Using an Interface to Specify Methods

- TS allows you to define interfaces, to specify methods that must be defined in implementation classes

```
interface ILoggable {  
    log(msg: string) : void  
}
```

```
interface ISerializable {  
    serialize() : void  
}
```

- A class can implement any number of interfaces
 - Via the `implements` keyword

```
class MyClass implements ILoggable, ISerializable {  
    log(msg: string) : void {...}  
    serialize() : void {...}  
}
```

Using an Interface as a Property Bag

- An interface can specify a property bag

```
interface IShape {  
    cx: number    // Required.  
    cy: number    // Required  
    w: number     // Required.  
    h?: number    // Optional.  
}
```

- You can use the interface type in function parameters
 - Compiler ensures you pass in a compatible object

```
function useShape(shape: IShape) : void {  
    ...  
}
```

Using an Interface as a Func Signature

- An interface can specify a function signature
 - Define an anonymous function inside the interface

```
interface ISearchFunc {  
    (src: string, subStr: string): boolean  
}
```

- You can use the interface when you declare a variable
 - Variable will point to a function of that signature

```
let mySearchFunc: ISearchFunc  
  
mySearchFunc = function(sourceString: string, subString: string) {  
    return sourceString.search(subString) != -1  
}
```

Using an Interface as an Array Type (1)

- An interface can specify an array type
 - Define an anonymous array inside the interface
 - Specify data type, and index type (number/string)

```
interface IStringArray {  
  [index: number]: string  
}
```

- You can use the interface when you declare a variable
 - Indicates the variable is an array of the specified type

```
let cities: IStringArray  
cities = ["London", "Paris", "NY"]  
console.log(cities[0])
```

Using an Interface as an Array Type (2)

- This example shows how to use a string index type
 - Effectively, it's a key-value dictionary

```
interface IStringDictionary {  
    [index: string]: string  
}
```

- Usage:

```
let capitalCities: IStringDictionary = {}  
  
capitalCities[ "Norway" ] = "Oslo"  
capitalCities[ "UK" ] = "London"  
capitalCities[ "Romania" ] = "Bucharest"  
  
console.log(capitalCities[ "Norway" ])
```

Summary

- Getting started with TypeScript
- Functions
- Classes
- Inheritance and interfaces

Custom Hooks

1. Recap of built-in React hooks
2. Defining and using custom hooks

Section 1: Recap of Built-in React Hooks

- What is a hook?
- The `useState()` hook
- The `useRef()` hook
- The `useEffect()` hook
- The `useMemo()` hook

What is a Hook?

- A hook is a function that lets you “hook into” React, to do something interesting on behalf of your component
- Let's remind ourselves about some built-in hooks:
 - `useState()`
 - `useRef()`
 - `useEffect()`
 - `useMemo()`

The useState() Hook

- useState() maintains mutable state for a component, and triggers a re-render if you modify the state
- Example:
 - Solutions\10-UserInputTechniques\trilingo-app
 - LanguageTest_Dom.tsx

The useRef() Hook

- `useRef()` maintains a reference to an object (typically an input control in the DOM tree)
- Example:
 - `Solutions\10-UserInputTechniques\trilingo-app`
 - `LanguageTest_UncontrolledComponent.tsx`

The useEffect() Hook

- `useEffect()` does some side-effect work after each render (e.g., to post data to a REST API)
- Example:
 - `Demos\12-ComponentTechniques\demo-effect-hooks`
 - `Gallery1.tsx`

The useMemo() Hook

- useMemo() caches a piece of UI content (to avoid costly regeneration of the virtual DOM)
- Example:
 - Demos\12-ComponentTechniques\demo-memoization
 - Page4.tsx

Section 2: Defining and Using Custom Hooks

- Overview
- How to define and use a custom hook
- Example without a custom hook
- Example with a custom hook

Overview

- You can define your own custom hooks
 - To encapsulate non-UI tasks performed by a component
- Possible scenarios for defining a custom hook:
 - To interact with a network resource
 - To deal with local storage
 - To perform low-level browser interactions
- Benefits:
 - Keep the component simple
 - Move the hairy bits into a hook instead

How to Define and Use a Custom Hook

- To define a custom hook:
 - Just define a function named `useXXXX()`
- To use a custom hook:
 - Just call it at the start of the component
- The hook operates within the context of the component
 - A custom hook can use other hooks (e.g. `useState()` etc.)
 - Just as if you used the hooks directly in the component

Example without a Custom Hook

- See this demo app, which doesn't use a custom hook:
 - Demos\C-CustomHooks\demo-app-1
- See these components:
 - StatusBar.tsx
 - SaveButton.tsx
- Observations 😢 ...
 - There's a lot of duplicate low-level code in the components

Example with a Custom Hook

- Now see this demo app, which does use a custom hook:
 - Demos\C-CustomHooks\demo-app-2
- See these files:
 - CustomHooks.ts
 - StatusBar.tsx
 - SaveButton.tsx
- Observations 😎 ...
 - The components are much simpler, no duplicate code

Summary

- Recap of built-in React hooks
- Defining and using custom hooks