Building Enterprise Apps with React

https://olsensoft.com/enterprise-react

Andy Olsen

andyo@olsensoft.com

Pearson

Course Contents

1. Creating a React Application
2. Single Page Applications and Routing
3. React-Redux
4. Redux Saga
5. Testing React Applications

Appendix

- TypeScript Essentials

Full Materials Online

- You can get full materials for this course online
  - https://olsensoft.com/enterprise-react/

- What's there?
  - All the slides for this course
  - Comprehensive demos in TypeScript

Creating a React Application

1. Creating an application
2. Running the application

Section 1: Creating an Application

- Overview
- Creating a React TypeScript app
- Reviewing the application
- Application home page
- Source code entry point
- Functional components

Overview

- We'll use TypeScript for our React demos today

- We're going to use **Vite** to create the apps
  - Generates a template React app, config, etc.
  - Can also be used to create apps in Vue, Svelt, 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 the options indicated below:

```
npm create vite@latest
```

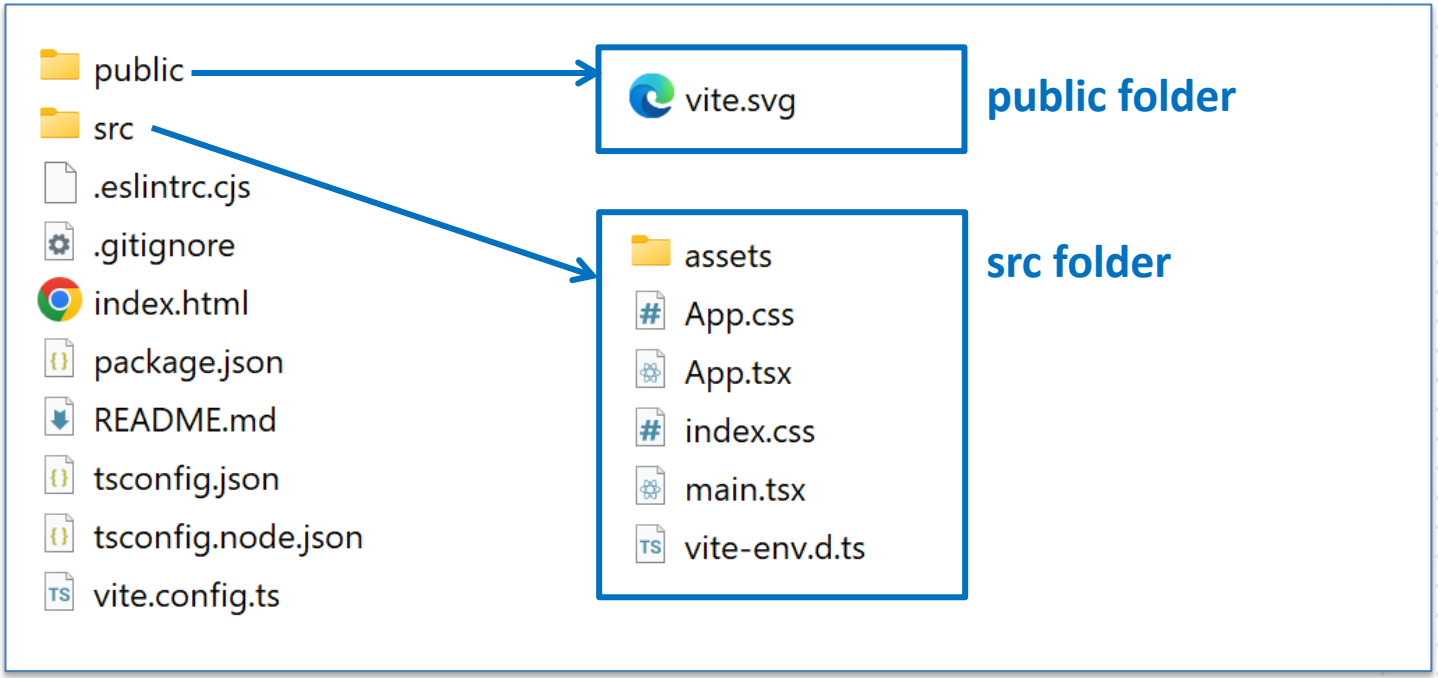
  - Project name: **demo-app**
  - Framework: **React**
  - Variant: **TypeScript**

- 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 React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.tsx'
import './index.css'

ReactDOM.createRoot(document.getElementById('root')!).render(
  <React.StrictMode>
    <App />
  </React.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:

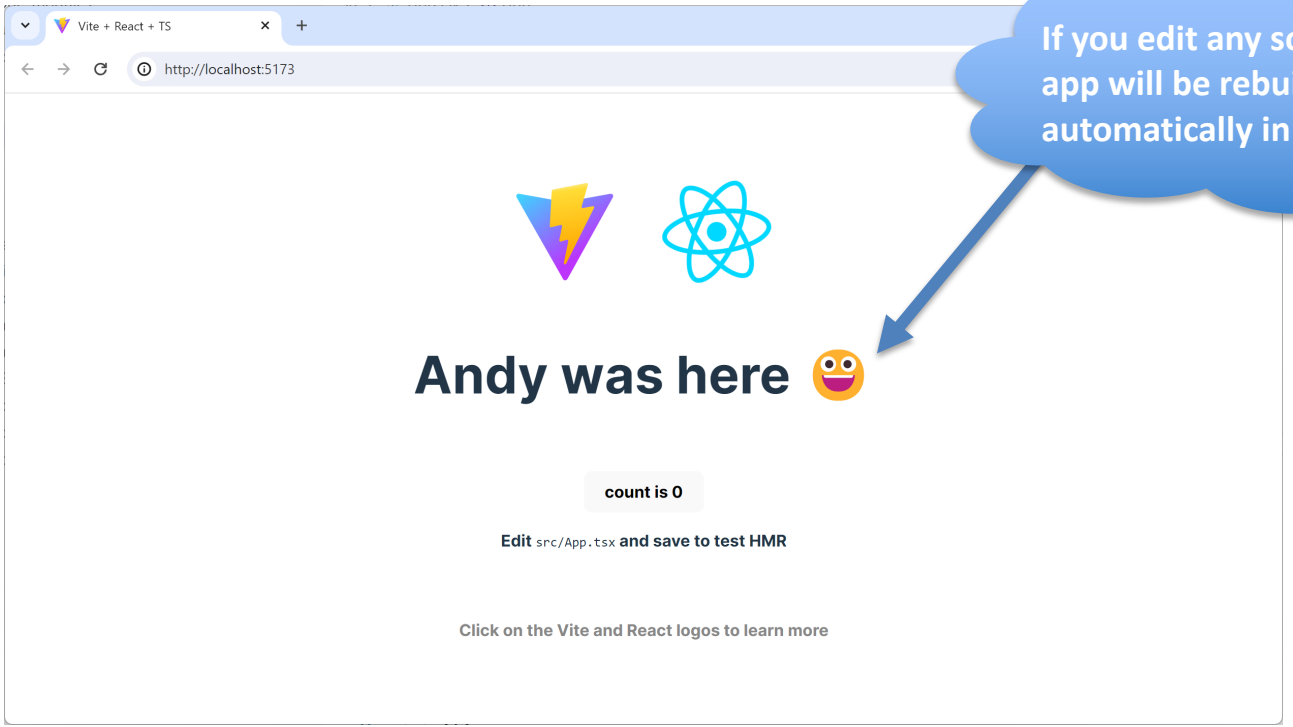
```
VITE v5.2.12  ready in 871 ms

➜  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



If you edit any source files, the app will be rebuilt and refreshed automatically in the browser

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
```

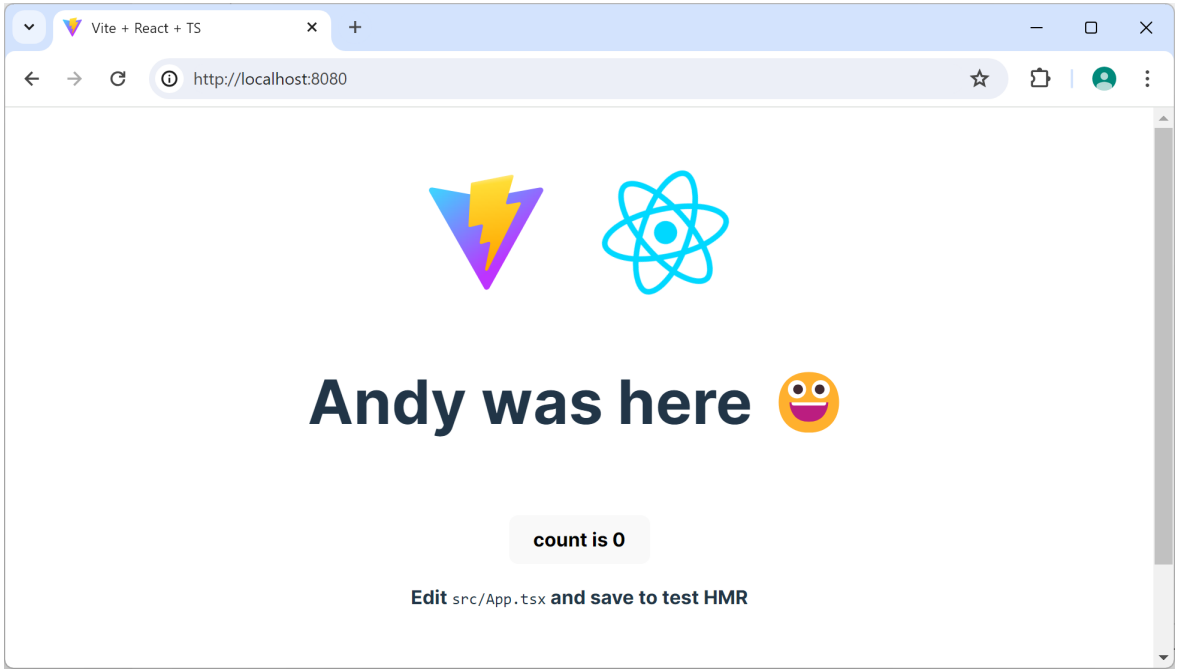
- Then serve the production build of the app as follows:

```
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

Single Page Applications and Routing

1. Getting started with SPAs and routing
2. Parameterized routes
3. Nested routes
4. Loading data for a route

Section 1: Getting Started with SPAs and Routing

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

Overview

- 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

Dependencies for React Router

- Add these dependencies in `package.json`:

```
"dependencies": {
  "react-router-dom": "^6.3.0",
  …
},

"devDependencies": {
  "@types/react-router-dom": "^5.3.3",
  …
}
```

React Router

React Router TypeScript declarations

package.json

Creating a Browser Router

- The preferred way to define routes in React nowadays is via the `createBrowserRouter()` function
  - 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

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

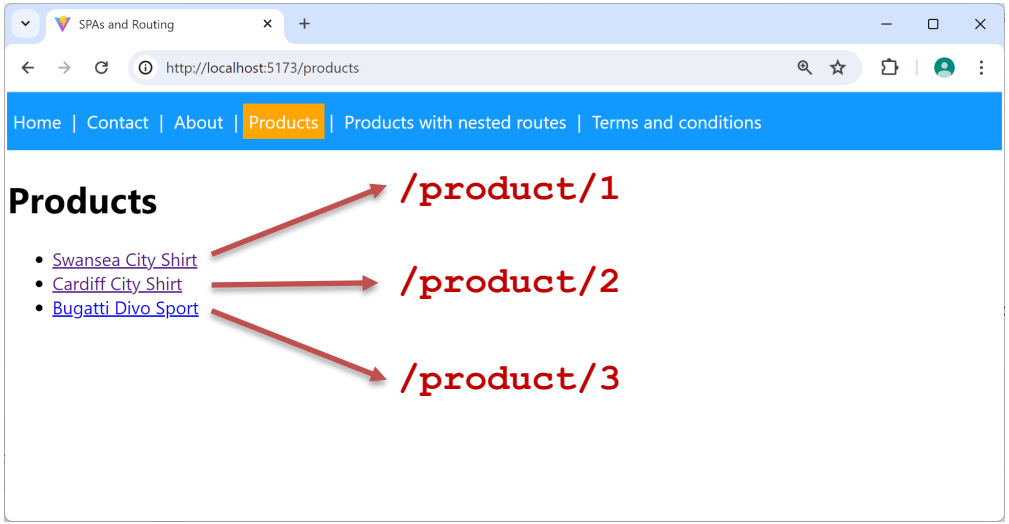
Section 2: Parameterized Routes

- 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.getProductItemById(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 3: 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 4: 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

- Getting started with SPAs and routing
- Parameterized routes
- Nested routes
- Loading data for a route

React-Redux

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

Section 1: Redux Concepts

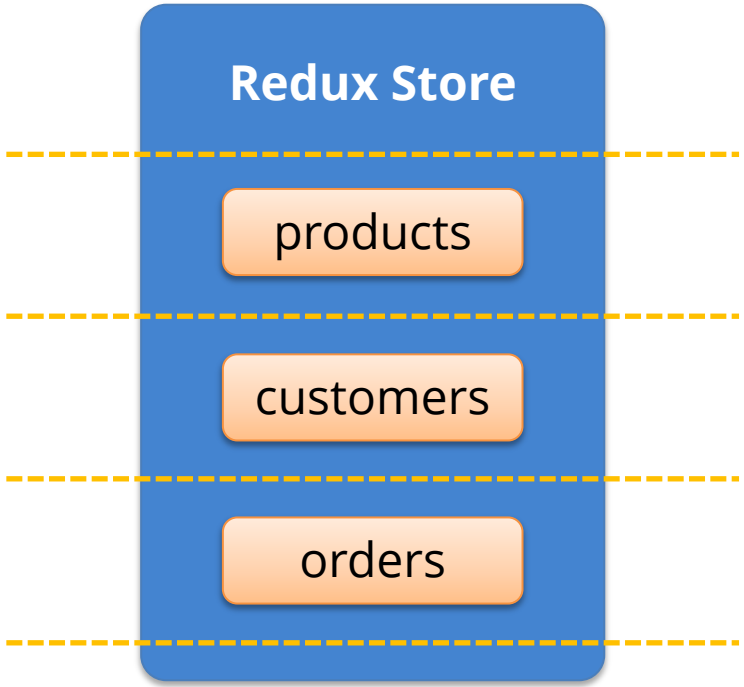
- Overview
- Organizing state into slices
- Identifying actions
- Defining reducer functions
- How it all fits together

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*



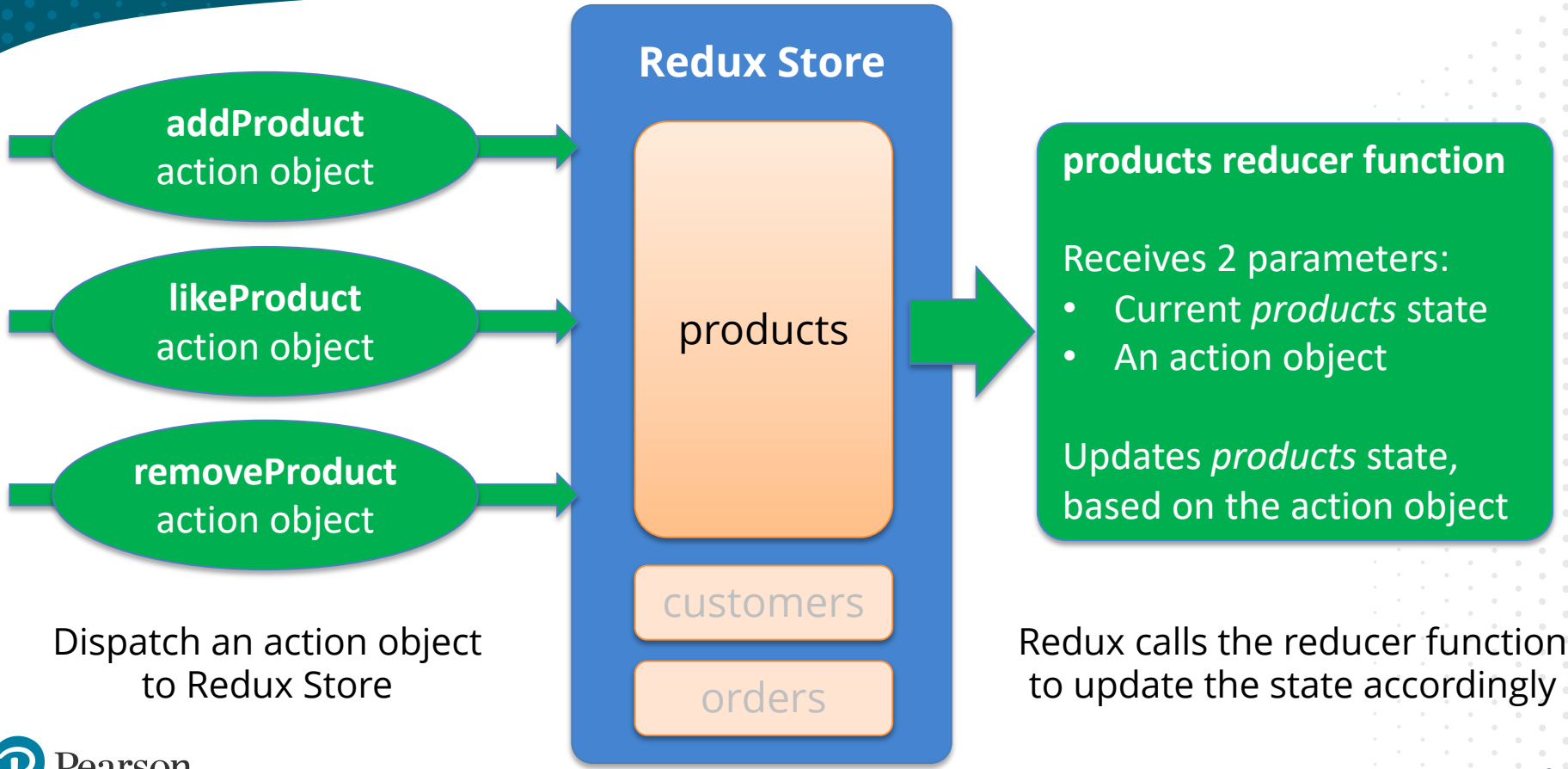
Identifying Actions

- 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



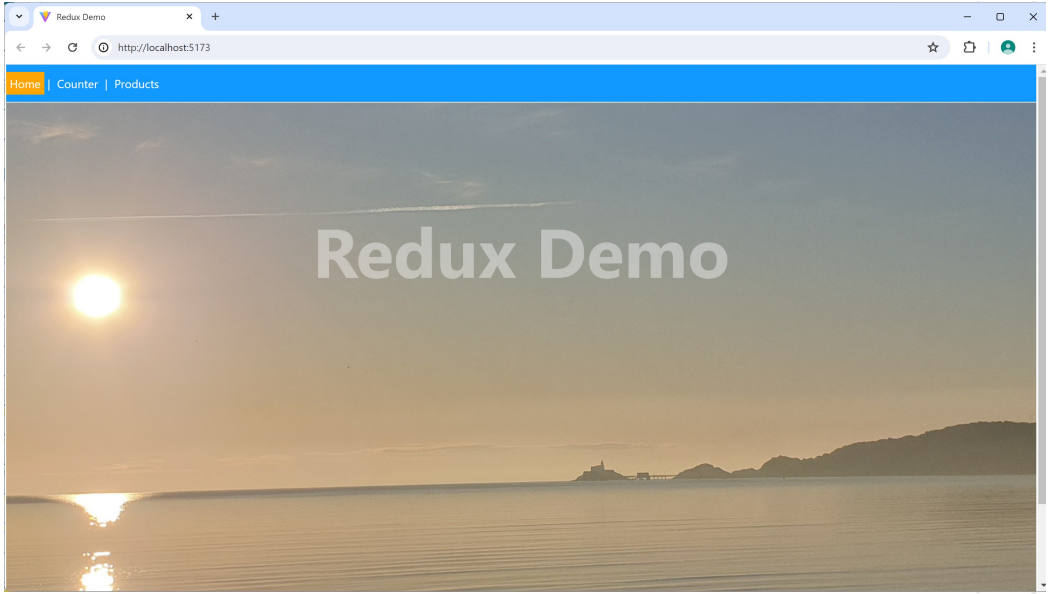
Section 2: Example Application

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

Overview

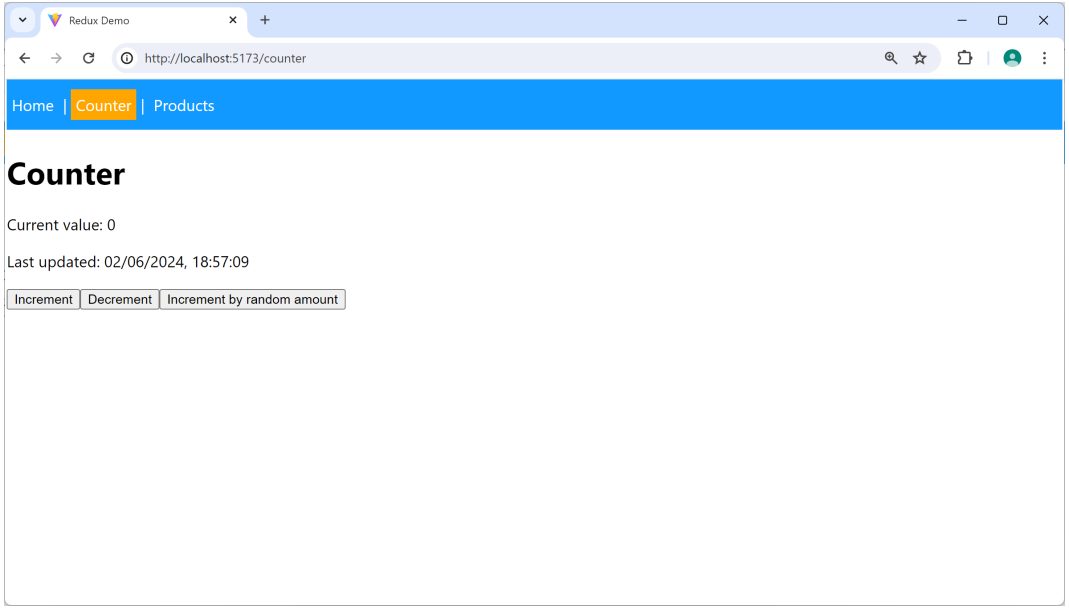
- In the `demo-app` folder, run `npm install` and `npm start`

- The application has 2 slices of state:
  - *counter*
  - *products*



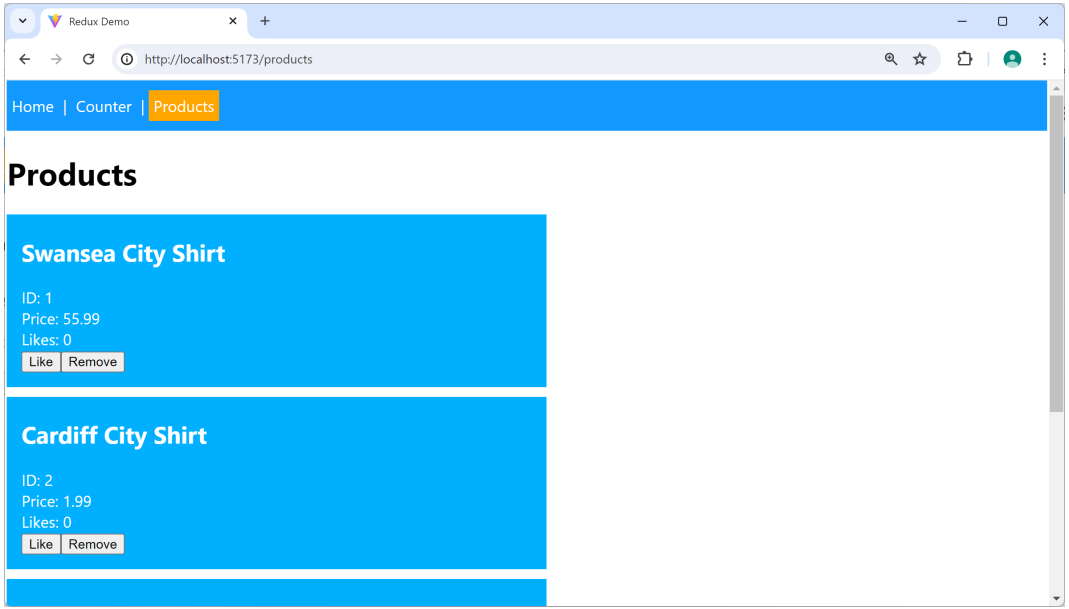
Viewing and Updating the *counter* State Slice

- 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) => {…}
  }
})
```

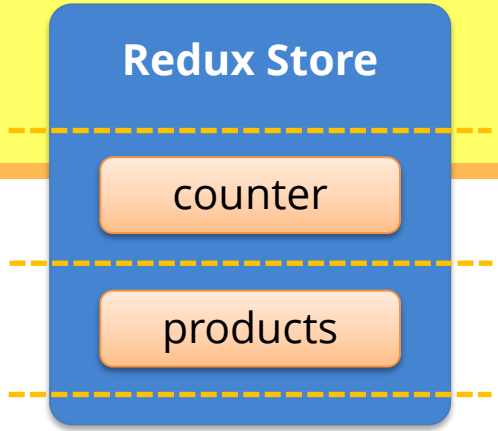
```
counterSlice.reducer(state, action) {

  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>,
)
```

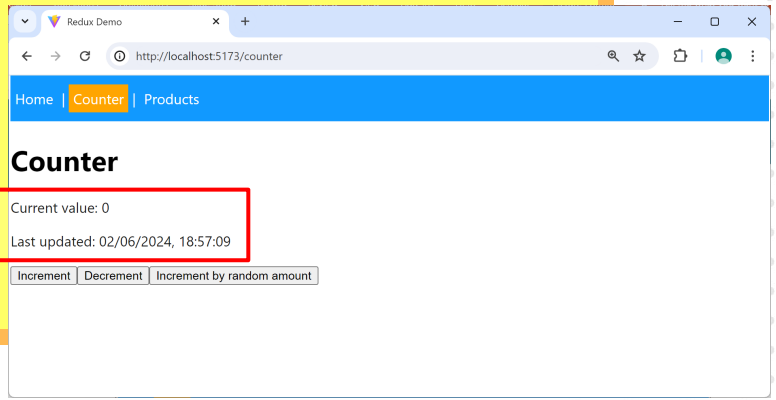
main.tsx

Accessing State in a Component

- A component can access Redux Store state as follows:

Counter.tsx

```
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:

Counter.tsx

```
import { useDispatch } from 'react-redux'
…

export default function Counter() {

  const dispatch = useDispatch()
  …
  dispatch(anActionObject)
  …
}
```

- 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

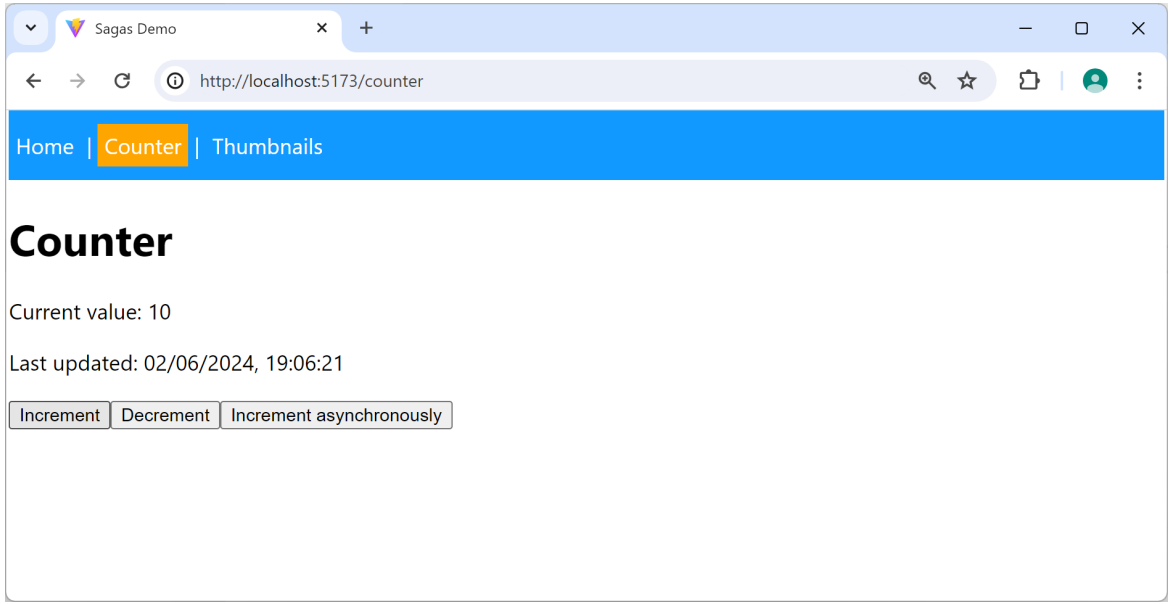
Overview

- 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`

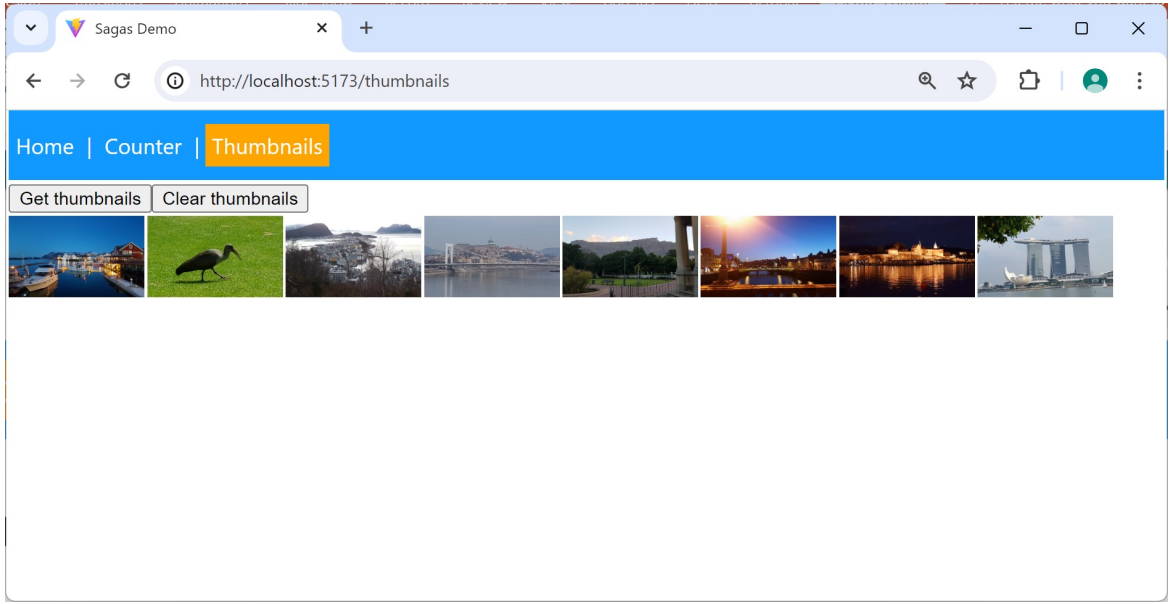
Using Sagas to Perform a Simple Async Task

- 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 dependencies in your `package.json` file

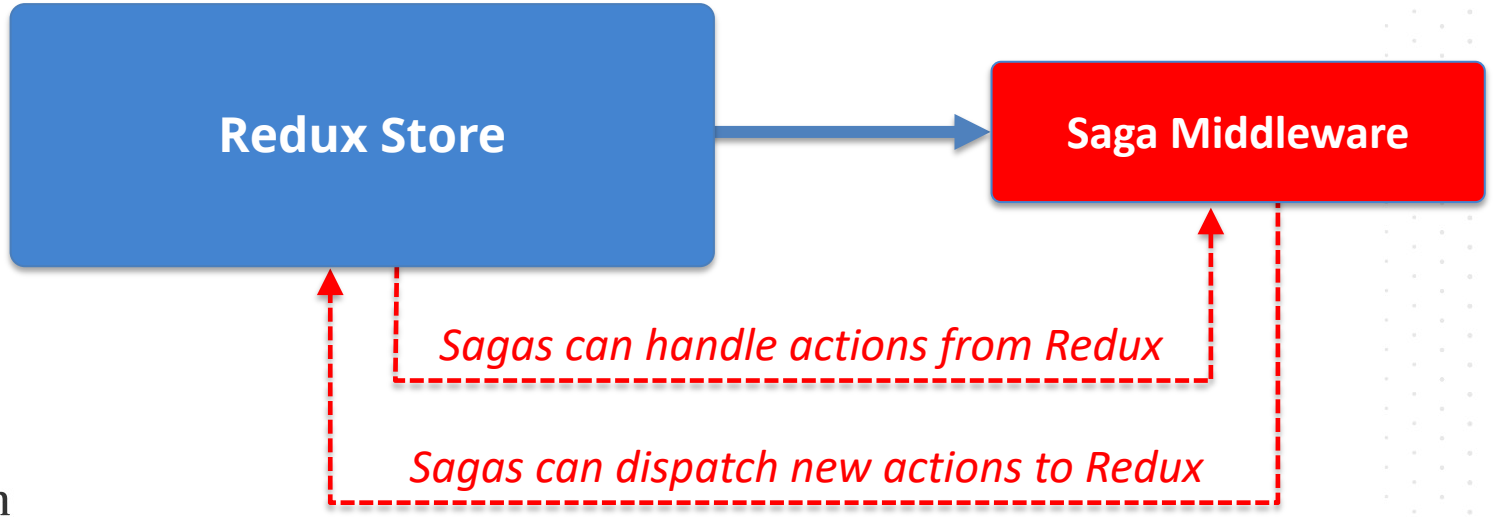
```
"dependencies": {
  "redux-saga": "^1.1.3",
  …
},

"devDependencies": {
  "@types/redux-saga": "^0.10.5",
  …
}
```

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

```
"scripts": {
    "test": "vitest",
    …
}
```
package.json

Writing Tests

- Here are some simple tests using Vitest:

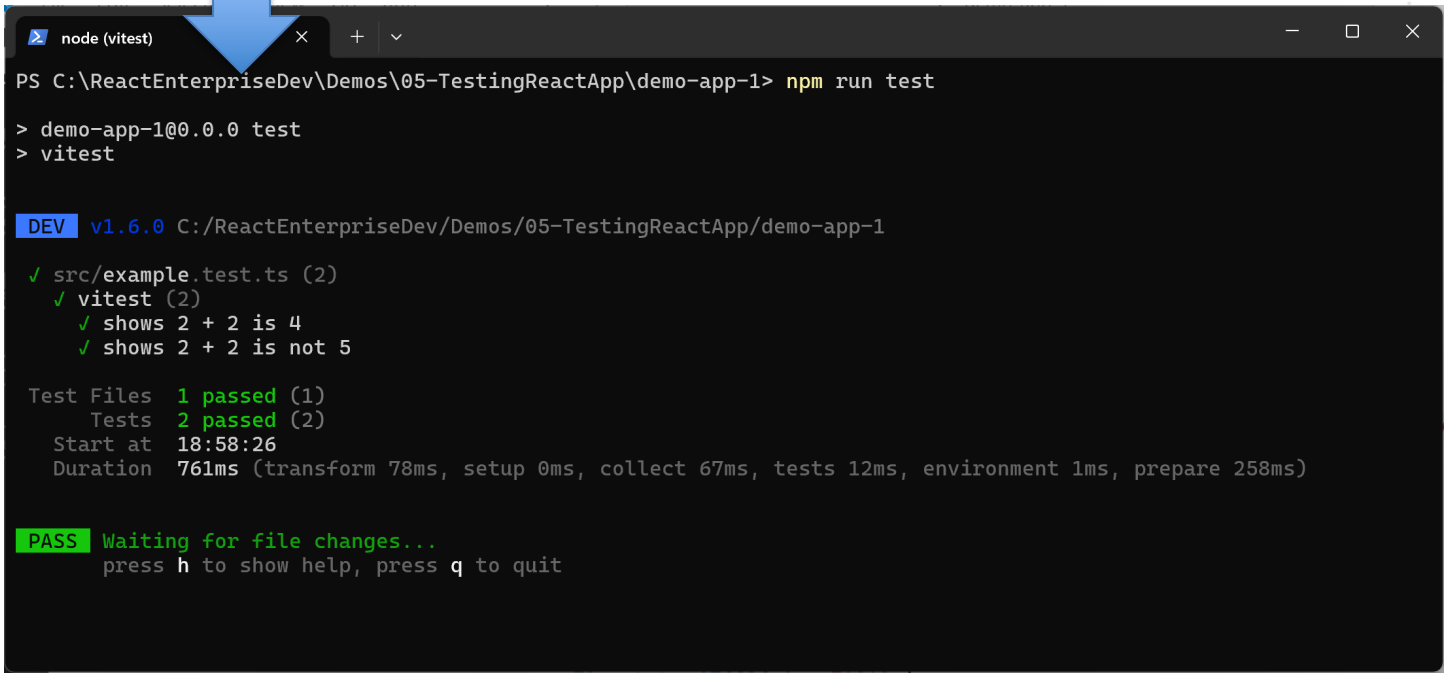
```
import { describe, it, expect } from 'vitest'

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
```



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

```
{
  "compilerOptions": {
    "types": ["vitest/globals"],
    …
  }
}
```

tsconfig.json

- `globals: true`
  - Makes Vitest functions (`describe`, `expect`, `it`) available globally, without the need to import manually in your tests
  - You must also make *type info* available globally too

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:

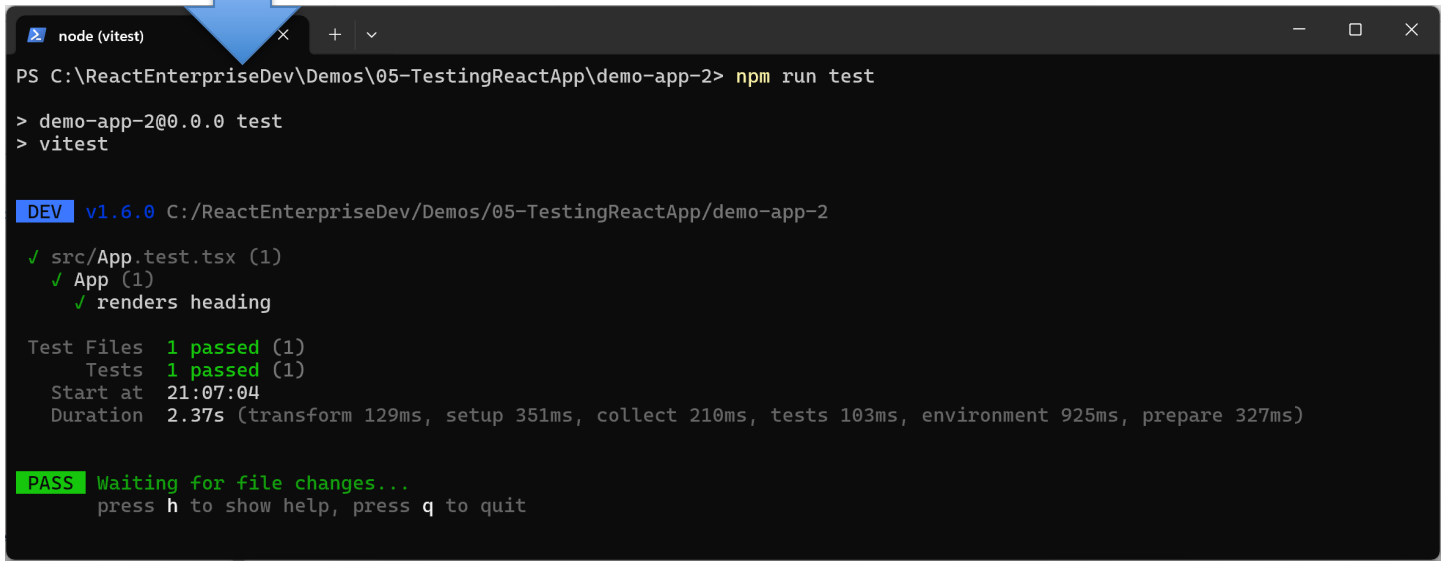
```
import { render, screen } from '@testing-library/react'
import '@testing-library/jest-dom'

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
```



Summary

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

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

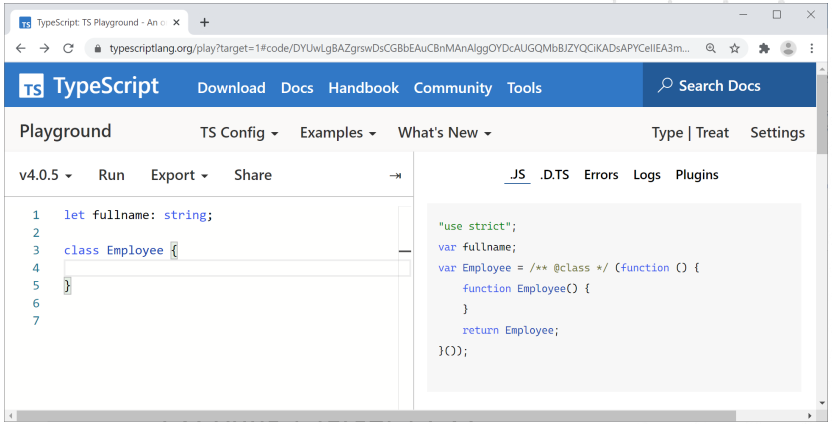
- You can use TypeScript to enhance the type-safety of your React applications
  - TypeScript supports ES6++ features, plus…
  - Data typing ☺
  - Interfaces
  - Decorators (similar to annotations in Java)
  - Class member variables (i.e. fields)
  - Generics
  - Keywords `public`, `private`, `protected`

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!



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


```
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)

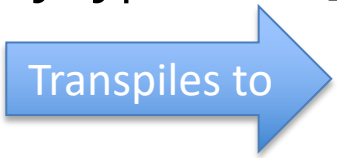
- `void` – no type (e.g. function with no return)
- `never` – function that never returns normally
- `any` – disables type-checks, e.g. legacy code
- `null` – data type of `null` value
- `undefined` – data type of `undefined` value

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




```
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


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

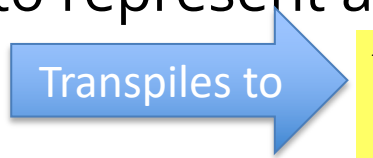
ES code

Enums

- TS supports enums, to represent a fixed set of

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

TS code




```
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 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

```
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

```
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

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

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

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 emp1 = new Employee()
emp1.name = "Paul"
emp1.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

```
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`

```
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

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

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

- A class can implement any number of interfaces

```
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

```
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

```
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

```
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