JavaScript - Frameworks - React

# Overview

Quick js front end framework

# Setup

## Base HTML

React single page web apps render client HTML dynamically upon load of the javascript bundle. Therefore, the server can serve a very simple base HTML response, which loads the javascript bundle and contains a root element for the React HTML to be rendered into. There can be as many root elements as required for a project, which can be particuarly useful for integrating react into existing projects.

A typical react HTML page would look like:

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

Then the element would be passed into the React DOM using:

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

ReactDOM.render(element, document.getElementById('root'));

## Sass

# Concepts

## JSX

JSX is a syntax extension of javascript which allows HTML templating to be combined directly into javascript without the need for string interpolation. While normal web apps separate concerns by putting the markup and logic in separate files, react separates the concerns using components for each website feature, where inside the component the HTML template and logic are seperated.

JSX HTML is interpreted automatically when assigned to variables, variables and any valid javascript can be inserted into JSX using curly brackets:

const name = 'Josh Perez';

const element = <h1>Hello, {name}</h1>; << JSX

Attributes can be specified in JSX using string literals or curly brackets:

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

Empty tags can be closed immeditaley with a />:

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

By default, React DOM escapes any values embedded into JSX before rendering, therefore XSS attacks will be mitigated for user content.

Notes:

* Use either curly brackets or quotes for element attributes, not both
* JSX uses a camelCase naming convention

## Elements

React elements are plain javascript objects, meaning they are cheap to create. For an element to be rendered on the page, it is passed to the React DOM, which updates the browser DOM to match the elements. A simple react element would be:

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

ReactDOM.render(element, document.getElementById('root'));

React elements are immutable, meaning once they are created, they and their children cannot be changed. Therefore, the only way to update elements on the page is to re-render the element and pass it into the React DOM again.

When updating the React DOM compared the new elements to the previous, and only updates what has changed, bringing in the new state.

## Components

Components are the building blocks of a React app, allowing elements on a web app to be split into independent, reusable pieces, each with their own state.

The most basic level of a component is a function componet which accepts the argument "prop" (properties) and return a react element:

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

ES6 classes can also be used to define a class component, which allows for state management. Class components contain a render function and props is coverted to this.props:

class Welcome extends React.Component {

render() {

return <h1>Hello, {this.props.name}</h1>;

}

}

Inside components, elements can be rendered in JSX using their function name, any attributed added to the element are passed as props:

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

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

Typically React apps have a single App component at the top of the hierachy which is built up of other components. Using SRP components should be as small as possible while keeping their core functionality.

In respect to props, React defines that props should never be mutated or re-assinged.

## State

State can be added to a class component using the construction function to define the state, the base constructor should always be passed props:

constructor(props) {

super(props);

this.state = {

date: new Date()

};

}

State can then be accessed with this.state.<variable>, the component will then trigger a re-render each time the state changes due to a lifecycle method. State can be set using the setState method:

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

Due to asychronous processing, state should not be directly updated using previous state values, instead a function passing state and props should be used:

this.setState((state, props) => ({

counter: state.counter + props.increment

}));

State is kept entirely in the model it is defined in, parent or child componet should not know whether a certain component is stateful. However, state can be passed down to child components in props.

Upon state change via .setState() the component will re-run the render() method, generating new child components if they have changed, updating the page.

The first time a component is rendered to the DOM it is called 'mounting', when a component is removed or replaced in the DOM it is called 'unmounting'. The function componentDidMount() can be used to setup functions on component upon render, the function componentWillUnmount() can be used to trigger tear down functions to remove event listeners, intervals, and others. One use of the componentDidMount function is to async load an api request into state upon page load:

componentDidMount() {

fetchPosts().then(response => {

this.setState({

posts: response.posts

});

});

}

## Handling Events

In JSX events such as onClick are handled in camelCase and by passing a defined function, instead of a string as in standard HTML, preventDefaults must be explictly defined. Commonly in class components, events are defined as class functions, however if the event is passed into JSX HTML, 'this' will not be bound by default, therefore to ensure the correct context either use arrow functions defined in the component or bind(this) to the function:

constructor(props) {

super(props);

this.handleClick2 = this.handleClick2.bind(this);

}

handleClick = () => {

console.log('this is:', this);

}

handleClick2() {

console.log('this is:', this);

}

render() {

return (

<button onClick={this.handleClick}>

Click me

</button>

<button onClick={this.handleClick2}>

Click me

</button>

);

}

To avoid creating a new function each render, binding should be performed outside of the render section of the component.

## Conditional Rendering

Conditional rendering is used define what is rendered on the page based on component state. Conditional rendering is simply performed in the component using logical operators such as 'if'.

In functional components, conditional logic can be added anywhere. In classical components conditional logic is used in the render function of the component, since this will be called each time the a new component is rendered.

Conditional rendering can also be performed inline in JSX when placed in curly brackets.

To prevent rendering from a component, simply return null from the function or render function. By returning null, the component is still rendered, firing all hooks but no values are added to the page.

## Lists and Keys

In React collections of elements in arrays can be rendered in JSX by inlcuding the collection reference in curly brackets. React will then render each component in the collection one by one.

Rendering collections allow loops such as .map() to quickly generate lists and other items in components.

For React to refresh lists correctly, keys need to be added to each item during creation. Keys are commonly the id of the item being created, however if unique ids are not available, it is possible to use the array index instead (indexes are used by default if no key is given). For example:

const todoItems = todos.map((todo) =>

<li key={todo.id}>

{todo.text}

</li>

);

List keys should not be passed through as props, instead they should be applied to the element during the map() function.

## Forms

Since HTML forms keep state and redirect a user to a new page by default, the submission of HTML forms in SPAs should be handled by an event listener. HTML form elements such as input and textarea have self contained state, however it is useful to combine this with React component state to allow for functionality on user input such as validations.

An example of setting component state from form state is:

class NameForm extends React.Component {

constructor(props) {

super(props)

this.state = {value: ''}

this.handleChange = this.handleChange.bind(this)

this.handleSubmit = this.handleSubmit.bind(this)

}

handleChange(event) {

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

}

handleSubmit(event) {

alert('A name was submitted: ' + this.state.value)

// async send the form using the fetch() api

event.preventDefault()

}

render() {

return (

<form onSubmit={this.handleSubmit}>

<label>

Name:

<input type="text" value={this.state.value} onChange={this.handleChange} />

</label>

<input type="submit" value="Submit" />

</form>

);

}

}

In react some form items are modified to allow for clearer writing:

* textarea - uses value attribute for input instead of its children
* select - uses value attribute for selected input instead of selected, arrays can be used for multiple selection:

<select value={this.state.value} onChange={this.handleChange}>

<option value="grapefruit">Grapefruit</option>

<option value="lime">Lime</option>

</select>

Multiple inputs can be handled either using the same event handler function deciding by event.target.name, or by induvidual event handlers for each input. Computed property names can be particularly useful for setting state in these cases:

this.setState({

[name]: value

});

Specifiying the value prop on a controlled component such as input, will prevent a user from changing the input unless you desire so, i.e. update the value with an event listner. If the specified value is undefined or null then the user will be able to edit the prop.

## Lifting State

Lifting state is used to share state between two components by lifting it to their closest ancestor.

To lift state the ancestor passes down a function as a prop to its children, the children can then call this function on local state change. The functionc call allows the ancestor to change its own state and in turn rerender its child components with new props, sharing the state between all the children which receive the prop.

In state there should be a single 'source of truth' for any data which changes. Normally state is added to any components which require it, then if the state needs to be shared it is lifted up to their nearest ancestor. This helps react rely on the top down flow of data.

If a value can be derived from the state or props, it likely should be kept in state. Instead it should calculate dynamically as required in the render() functions of the components.

## Containment

The children prop can be used to yield to the parent allowing a element such as a sidebar to be defined without knowing that will be inside it. For example:

function FancyBorder(props) {

return (

<div className={'FancyBorder FancyBorder-' + props.color}>

{props.children}

</div>

);

}

function WelcomeDialog() {

return (

<FancyBorder color="blue">

<h1 className="Dialog-title">

Welcome

</h1>

<p className="Dialog-message">

Thank you for visiting our spacecraft!

</p>

</FancyBorder>

);

}

While children can be used for one insertion, multiple insertions can be manually defined by passing the elements in as props. For example:

function App() {

return (

<SplitPane

left={

<Contacts />

}

right={

<Chat />

} />

);

}

## Specialization

Specialization is where a more specific component renders a more generic one, adding further configuration with props. Specialization allows for better encapsulation and reuseability in code, since the more generic components can be used elsewhere:

function Dialog(props) {

return (

<FancyBorder color="blue">

<h1 className="Dialog-title">

{props.title}

</h1>

<p className="Dialog-message">

{props.message}

</p>

</FancyBorder>

);

}

function WelcomeDialog() {

return (

<Dialog

title="Welcome"

message="Thank you for visiting our spacecraft!" />

);

}

# React Features

## Fragments

Fragments can be used to return groups of components without added extra DOM nodes to the HTML, potentially breaking lists etc. Create a fragment by enclosing the child elements to be returned in a React.Fragment:

class Columns extends React.Component {

render() {

return (

<React.Fragment>

<td>Hello</td>

<td>World</td>

</React.Fragment>

);

}

}

Shorthand for a fragment is simply <> nd </>

## Context

React context can be used to assign data to a global context, in which any component below the defintion can access the data. While it should be avoided where possible, it can be useful for props which would require passing through almost all components, for example user login status and application theme.

Create context using:

const ThemeContext = React.createContext('light')

Assign context at the top of a tree using:

return (

<ThemeContext.Provider value="dark">

<NextComponent />

</ThemeContext.Provider>

);

Any component can now access the context using:

static contextType = ThemeContext

## Refs

Refs provide a way to access DOM nodes or React elements created under the render method. While in a typical React app props are the only way that parents interact with their children, sometimes a child needs to be modified outside of the typical dataflow. Common uses of refs are:

* managing focus, text selection, or media playback
* triggering animations
* integrating with third-party DOM libraries

Refs are created using:

React.createRef()

and then attached to an element using the 'ref' attribute. Refs are then commonly assigned to an instance property when the component is constructed to they can be refereced throughout the component. For example:

class MyComponent extends React.Component {

constructor(props) {

super(props);

this.myRef = React.createRef();

}

render() {

return <div ref={this.myRef} />;

}

}

The ref will then be accessible from the .current property of the ref once the component has mounted. When the component unmounts, the ref will be assigned back to null.

ref forwarding can be used to pass refs down through components:

const FancyButton = React.forwardRef((props, ref) => (

<button ref={ref} className="FancyButton">

{props.children}

</button>

));

// You can now get a ref directly to the DOM button:

const ref = React.createRef();

<FancyButton ref={ref}>Click me!</FancyButton>;

## Uncontrolled Components

In a controlled component, form data is handled by the React component. An uncontrolled component is where the data is controlled by the DOM.

To handle an uncontrolled component a ref can be dded to get form values from the DOM. A common example of a forced uncontrolled component is the file input tag, since it is controlled by the FileAPI and cannot be have its value set by React. To handle it, add a ref:

class FileInput extends React.Component {

constructor(props) {

super(props);

this.handleSubmit = this.handleSubmit.bind(this);

this.fileInput = React.createRef();

}

handleSubmit(event) {

event.preventDefault();

alert(

`Selected file - ${this.fileInput.current.files[0].name}`

);

// AJAX submit form and handle response

}

render() {

return (

<form onSubmit={this.handleSubmit}>

<input type="file" ref={this.fileInput} />

<button type="submit">Submit</button>

</form>

);

}

}

## Render Props

A render prop is a technique to share code between react components using a prop whose value is a function. The function prop is passed into and used in a component, so it knows what to render. For example, the Mouse object can then pass its state into the render-prop and the Cat will be returned rendered in the correct place:

class MouseTracker extends React.Component {

render() {

return (

<div>

<h1>Move the mouse around!</h1>

<Mouse render={mouse => (

<Cat mouse={mouse} />

)}/>

</div>

);

}

}

# Designing React Webapps

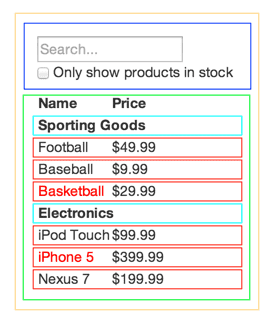
## Setup

Generally, before building the app, the backend API server will be developed, and a mock-up of the website design will be provided.

The API will provide the majority of the business logic and information processing. The mock-up will provide the base of which components are required for creating the app.

## Define Components

Go throught the mockup breaking it down into its hierarchy, to keep to the SRP each section of the hierarchy will likely be a component. For example:



## Create a Static App

Decouple the static and dynamic parts of the application by first creating all the components statically (no user interaction) first. The static version of the app will take all the data and render the components, there will be no state since state is reserved for interactivity.

## Identify the Mimimal Representation of UI State

To keep the code DRY it is useful to figure out the minimal representation of state the app requires to function. If the variable can be calculated from the props passed to the component or remains unchnged over time, it probably isn't state.

## Identify Which Components Own State

Identify which components should own state and which should mutate it. Since React is designed to have one way data flow down the hierarchy, it may not be clear which object should own or mutate state, particually if the state requires lifting into common ancestor components.

## Apply Inverse Data Flow

Create the methods to return modified state back up the hierarchy to where it is stored. For example, a search box being passed an onChange method through props to change state in a higher up ancestor.

# Performance Optimisation

## Production Build

Generating a production build will minify code, reducing the helpful warnings and many other features required on a dev machine.

## Avoid Reconciliation

The React shouldComponentUpdate callback is called before a re-render and normally returns true to re-render the component. If for some reason there is no reason to re-render a component, the callback can be change to return false to prevent a render.

shouldComponentUpdate(nextProps, nextState) {

return true;

}

# React Router

## General

The react-router plugin turns a standard single page react webapp (SPA) into a SPA which feels likes a traditional mutlipage app, adding in browser history and conditional rendering based on brower route.

Install using a node package manager:

yarn add react-router-dom

Then import the required modules in the React components which require them:

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

There are three primary categories of components in React Router:

* routers, like <BrowserRouter> and <HashRouter>
* route matchers, like <Route> and <Switch>
* and navigation, like <Link>, <NavLink>, and <Redirect>

react-router uses dynamic routing, meaning routes are processed as the app is being rendered.

## Router

The router component uses the HTML5 history API to keep the UI in sync with the URL. Enclose any routing components within a router to keep everything working.

## Switch and Route

A switch component ensures only one component tree in the switch is rendered at one time. A switch component will iterate through its child Route components until it finds one which matches. Route tags define which component to load based on the current routes, for example the homepage of an app will generally load on the '/' route:

function App() {

return (

<Router>

<Switch>

<Route path="/" component={Home} exact />

<Route path="/about" component={About} />

<Route path="/shop" component={Shop} />

</Switch>

</Router>

)

}

Route paths match from the start, so shorter more general routes should be place at the bottom of the routes list. To match exact paths only, use the 'exact path="<path>"' syntax.

## Link

React-router Link elements create a link which allow for traditional style navigation while remaining in the SPA:

function Navbar() {

return (

<div>

<Link to="/">Home </Link>

<Link to="/about">About Us </Link>

<Link to="/shop">Shop Now </Link>

</div>

);

};

## Nested Routing

Nested routes can be conditionally rendered using route matching and params, for example with a topics resource:

import {

BrowserRouter as Router,

Switch,

Route,

Link,

useRouteMatch,

useParams

} from "react-router-dom";

function Topics() {

let match = useRouteMatch();

return (

<div>

<ul>

<li>

<Link to={`${match.url}/components`}>Components</Link>

</li>

</ul>

{/\* The Topics page has its own <Switch> with more routes

that build on the /topics URL path. You can think of the

2nd <Route> here as an "index" page for all topics, or

the page that is shown when no topic is selected \*/}

<Switch>

<Route path={`${match.path}/:topicId`}>

<Topic />

</Route>

<Route path={match.path}>

<h3>Index page</h3>

</Route>

</Switch>

</div>

);

}

function Topic() {

let { topicId } = useParams();

return <h3>Requested topic ID: {topicId}</h3>;

}

## Scroll Reset

In SPAs scroll height does not automatically reset on navigation, therefore if a document is long, navigating to a new page make lead to it being scrolled mid way down. To mitigate this effect, use a scroll to top component to automatically scroll the page back to the top on navigation:

import { useEffect } from "react";

import { useLocation } from "react-router-dom";

export default function ScrollToTop() {

const { pathname } = useLocation();

useEffect(() => {

window.scrollTo(0, 0);

}, [pathname]);

return null;

}

function App() {

return (

<Router>

<ScrollToTop />

<App />

</Router>

);

}

## Media Queries

Media queries can be used to dynamically render different routes based on screen size:

const Invoices = () => (

<Layout>

{/\* always show the nav \*/}

<InvoicesNav />

<Media query={PRETTY\_SMALL}>

{screenIsSmall =>

screenIsSmall ? (

// small screen has no redirect

<Switch>

<Route

exact

path="/invoices/dashboard"

component={Dashboard}

/>

<Route path="/invoices/:id" component={Invoice} />

</Switch>

) : (

// large screen does!

<Switch>

<Route

exact

path="/invoices/dashboard"

component={Dashboard}

/>

<Route path="/invoices/:id" component={Invoice} />

<Redirect from="/invoices" to="/invoices/dashboard" />

</Switch>

)

}

</Media>

</Layout>

);

# React Helmet

## General

Document head manager for React apps, allowing for SEO optimisation. Install using node package manager:

yarn add react-helmet

Import into required components:

import {Helmet} from "react-helmet";

## Supported Tags

Helmet supports all valid head tags:

* title
* base
* meta
* link
* script
* noscript
* style

Support attributes for:

* body
* html
* title

## Base Head

Add a base head section for the website in the app component:

<Helmet>

<meta charSet="utf-8" />

<title>My Title</title>

<link rel="canonical" href="http://mysite.com/example" />

</Helmet>

## Update Header in Children

Add new tags to overwrite the current lower down in the component tree. For example, specifiying meta tags for specific page descriptions:

<Helmet>

<title>Nested Title</title>

<meta name="description" content="Nested component" />

</Helmet>