React: Getting Started

The basics

- **Why React?**
  - It's a ***small*** incomplete ***library*** - you'll likely need to use other stuff in addition to React
    - Is therefore **more flexible** than a framework
  - Takes the ***declarative nature of HTML and extends this to dynamic data*** (rather than simply static content. It enabled developers to **declaratively describe their UIs** and **model the state of the interfaces**
  - Things that have made React popular
    - The **Virtual DOM** is easier than dealing with the DOM API
    - The strong **corporate backing** increases **trust** in the library
    - It provides a **declarative** language to model UI and state
- **Resources for learning**
  - Common problems faced by React developers are described in: https://jscomplete.com/learn/react-beyond-basics/react-cfp

Basic concepts

- **Components** - are just like functions. They ***take input and return a UI***
  - Input and output
    - **Input**: *props* and *state*
      - **Props** are **explicit**, and similar to HTML attributes. They are **immutable** and cannot be changed by the component. But they ***can be changed by the component's parent and for the component to be re-rendered***
      - **State**: components control and ***can change their internal state***
    - **Output:** UI (declared as JSX, shipped to the browser as React API calls)

```
: <div className="welcome">Hello!</div>
```

```
"use strict";
/*#__PURE__*/
React.createElement("div", {
  className: "welcome"
}, "Hello!");
```

      - JSX is not HTML. It will be **compiled to pure JavaScript** that is sent to the browser
      - Rather than using JSX, you *could* equally just write the raw API calls in your component
- **Reusable and composable**
  - Here's a gist I've created which demonstrates and explains some concepts of component composition and reuse: https://gist.github.com/gtvj/aea026d92dc6fabed0391f379d58fadc
- **Types of component** (both can have state and create side-effects)
  - ***Function components*** - are simpler and preferred
  - ***Class components*** - less simple but more powerful
- **Naming**
  - Must be capitalised - to prevent React thinking you're referencing a HTML element rather than a React component. Sound confusing? It isn't - just imagine having a React component called button. When passing this to ReactDOM.render(), it would appear to be an HTML element unless capitalised.
- The ***one way flow*** of data
  - Parent components can flow their data and behaviour down to their child components.
    - Props can pass data and behaviour down to children components
- **Hooks**
  - Allow you to use state and other features without writing a class
  - Our first hook is useState: https://reactjs.org/docs/hooks-state.html
- **Reactive updates / tree reconciliation / Virtual DOM**
  - In React, you generate HTML using JavaScript, rather than having a HTML template language
  - This allows React's Virtual DOM which uses tree reconciliation to update only what has changed
    - While it may be easier to generate a DOM more easily using the native DOM API, **the React Tree Reconciliation makes it far easier to update the DOM** - this is partly because React will only update those elements which need to change, rather than everything and partly because you do this in a declarative way when using React
      - **You can, of course, also do this with the DOM - but that requires imperative logic. React's Virtual DOM is delarative.**

Modern JavaScript: crash course

- Since 2015 we've had yearly releases of ECMAScript named ES[year] made by TC39
- New features
  - Variables and **block scopes**: let and const
    - In most cases, const is preferred because it guarantees the value will not have changed.
  - **Arrow functions**
    - **Work better than traditional functions for closures**. Why? *Because they do not implicitly bind 'this' to the caller*. Instead, Arrow function **close over the value of 'this' at the time it was defined**. This makes it great for delayed execution functions like listeners because it provides easy access to the defining environment https://gist.github.com/gtvj/dc4bdb8f9094d9ddf7cb1879de7e4107
    - *This is a great benefit for working with listeners and event handlers*
  - **Object literal improvements**
    - Shorthand property and method names
    - Computed property names
  - **Destructuring and rest/spread**
    - A few things to note:
      - You can use destructuring in function definitions and, when doing so, you can still set defaults https://gist.github.com/gtvj/30db8bfa523d35b2aaac3ea950c4e40c
      - You use the rest operator to create a new array or object with the 'rest' of the items https://gist.github.com/gtvj/db72f395a1dcaf168aead64abffcaca2
  - **Template strings**
  - **Classes**
  - **Promises and async/await**
    - Async/await syntax is a way to consume promises without having to nest .then calls
      - Remember that, once you await on anything in a function, the function itself becomes asynchronous and it will return a promise object https://gist.github.com/gtvj/bcc57454d8bad15c5a1009ecabcbcbec

React Class Components: the GitHub cards app

- The first decision to make in a React app is the component structure: how many components to use, and what each component should describe.
  - **You can mix and match** functional and class components in the same app
- Example of using a Class component https://gist.github.com/gtvj/4e35673537abc3c182dd313895584cbf
  - The only method **Class** components must have is a **render()** method
- Styling React components
  - There are several ways to style React components
    - Inline CSS (using JavaScript object within the 'style' property of your components - and camel casing where necessary
    - Styled components
- Working with data https://gist.github.com/gtvj/0f510f8af80a94493969b4cd1384ded1
- Taking input from users https://gist.github.com/gtvj/582f3b63cded2cff96bd939514d6ef88
- **Using custom hooks**
  - Naming: It's a good practice to name any custom hooks your create starting with 'use'

React Function Components: the star match game

- **Reuse** - when extracting components for re-use, you're looking for balance. **Too few or too many are both bad design**. When considering candidates a helpful pointer can be this: if you have many items that share data or behaviour, they could be a candidate for a component
- **Minimise state:** general advice when working with stateful components is to minimise state. Compute values where possible, rather than storing them.
  - Additionally, you want a components props to be exact. **Only what's needed to render the component** (and nothing else)
  - You should also seek to ensure that the structure of your components is consistent.
    - Start with any hooks into state and side effects
    - Follow this with an computations based on state (i.e. practicelsDone if remainingSongs.length == 0)
- **Using Side Effect Hooks**
  - **useEffect()**
    - From the React docs: "If you're familiar with React class lifecycle methods, **you can think of useEffect Hook as componentDidMount, componentDidUpdate, and componentWillUnmount combined.**"
    - There are two common kinds of side effects in React components. Those that require cleanup and those that don't.
      - Effects without cleanup https://gist.github.com/gtvj/464c5bcae76546bc87842e5aa324adc6
      - Effects with cleanup. Simple. **If your effect returns a function, that function will be used to cleanup when the component unmounts**
- **Unmounting and re-mounting components**
  - Changing a component's 'key' attribute will result in the component being unmounted and re-mounted. See this gist: https://gist.github.com/gtvj/2411b621e661030adad45026f20dcea2

Setting up a development environment

- Setting up a React development environment isn't exactly straightforward
  - Requires making many different tools (which have different APIs, configurations, release cycles etc.)
  - Development vs Production
    - You'll need to be able to run both locally for testing
- **Quick start with: create-react-app**
  - An NPM package
  - Provides many more tools that what is necessary
  - Use `npm run eject` to extract configuration to files you can edit. This operation is one way - there's no going back.
- **Configuring your own environment**
  - Requires a number of tools such as Babel, Webpack etc. Many guides are available and best practices change over time
- **Reactful**
  - This is an NPM package which functions similarly to create-react-app (in that you can use it to create a working environment). The difference is that the is bare bones and ejected by default (i.e. the configuration is flat and editable).