React

Introduction

THIS DOCUMENT COVERS

- Introduction

Overview

Why

- UI state is hard to manage in basic JavaScript.
- Highly performant.
- Build highly scalable web applications.

Cheating

Basic Component

FUNCTIONAL

```
function FunctionalComponent() : ReactElement {
  return <h1>FunctionalComponent</h1>;
}
```

CLASS

```
class ClassBasedComponent extends Component {
  render(): ReactElement {
    return <h1>ClassBasedComponent</h1>;
  }
}
```

Props

FUNCTIONAL

```
interface PropsShape {
  name: string;
}

function FunctionalComponent(props: PropsShape) : ReactElement {
  return <h1>Functional Component {props.name}</h1>
}
```

CLASS

```
interface PropsShape {
  name: string;
}

class ClassBasedComponent extends Component<PropsShape,{}> {

  constructor(props: PropsShape) {
    super(props);
  }

  render() : ReactElement {
    return <h1>ClassBasedComponent {this.props.name}</h1>
  }
}
```

Props and State

Let us now consider a stateful component. First as a class as it is easier. Note how we also deal with event handling in this sample.

CLASS

```
interface PropsShape {
   initialNumber: number;
}

interface StateShape {
   current: number;
}

export default class ClassBasedComponent extends
Component<PropsShape,StateShape> {

     constructor(props:PropsShape)
     {
         super(props);
         this.state = {current:props.initialNumber};

         this.handleClick = this.handleClick.bind(this);
     }

     render() : ReactElement {
         return <button onClick={this.handleClick}>
           {this.state.current}
           </button>;
     }

     handleClick() {

         this.setState((state:StateShape) => {
             return ({current:state.current*1.1});
         });
     }
}
```

FUNCTIONAL

And now maybe we can use a functional component. Note we must use hooks for this.

```
  interface PropsShape {
    initialNumber: number;
  }
  export default function FunctionBasedComponent(props:PropsShape)
  : ReactElement
  {
      const [value, setValue] = useState(props.initialNumber);

      return <button onClick={()=>setValue(value*1.1)}>{value}</button>;
  }
```

Lifecycle

Simple Class Based Example

This section shows how react renders a simple class-based component.

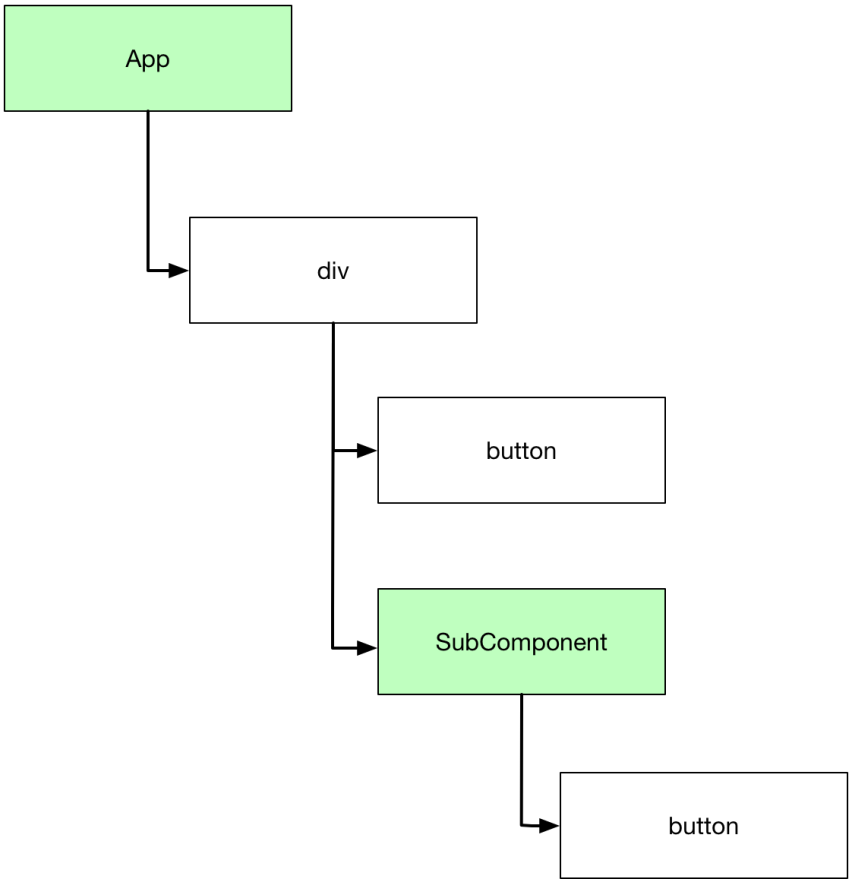
INITIAL RENDER

React Lifecycle

```
  export class SubComponent extends Component<{}, any>
  {
    constructor(props: any) {
      super(props);
      console.log("SubComponent()")
      this.state = { ctr: 0 }
    }

    handleClick = () => this.setState({ ctr: this.state.ctr + 1 });

    render = () => {
      console.log("SubComponent.render()");

      return <button onClick={this.handleClick}>{this.state.ctr}</button>
    };
  }

  export default class App extends Component<{}, any>
  {
    constructor(props: any) {
      super(props);
      console.log("App()");
      this.state = { ctr: 0 }
    }

    incrementCounter = () => this.setState({ ctr: this.state.ctr + 1 });

    render(): ReactNode {
      console.log("App.render()");

      return (
        <div>
          <button onClick={this.handleClick}>{this.state.ctr}</button>
          <SubComponent></SubComponent>
        </div>
      )
    };

    handleClick = () => this.setState({ ctr: this.state.ctr + 1 });
}
```

The following diagram shows the logical structure of our two react components and the DOM elements they create.

React Components and DOM Elements



When the application starts, React asks each component to render its content. We can see the order of events by looking at the log statements output in the browser console.

```
App()
App.render()
SubComponent()
SubComponent.render()
```

UPDATES

After rendering we say the application is in a reconciled state which means the rendered content is consistent with component state. React keeps a mapping between components and the DOM elements the render

Mapping between React Components and DOM elements

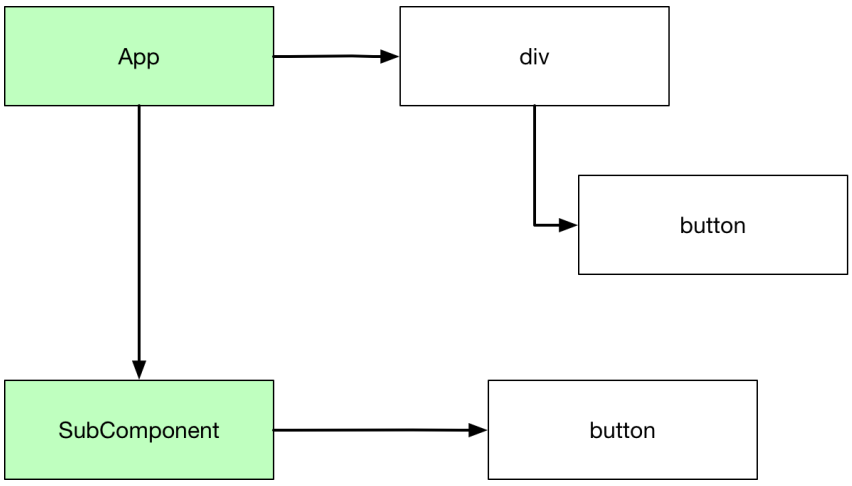
Once in a reconciled state the app waits for change . Change is most likely caused by a call to setState which updates the state. Once state is updated it is possibly inconsistent with the rendered DOM so setState marks the component and any child components as stale. So If we click the button on the App to change its state then both the App and the SubComponent render methods are invoked.

```
App.render()
SubComponent.render()
```

If we click the SubComponent button so only its state is changed then its render method is called. The parent App component is not rendered.

```
SubComponent.render()
```

In order to decide whether to update the DOM react compares the content produced by components with cache of previous results knows as the virtual DOM. This mechanism prevents React from having to query the DOM to determine if anything has changed. This improves performance.



FULL LIFECYCLE EVENTS

We add some other methods and logging.

```
export class SubComponent extends Component<{}, any>
{
  constructor(props: any) {
    super(props);
    console.log("SubComponent()")
    this.state = { ctr: 0 }
  }

  handleClick = () => this.setState({ ctr: this.state.ctr + 1 });
  componentDidMount = () =>
    console.log("SubComponent.componentDidMount()");
  componentDidUpdate = () =>
    console.log("SubComponent.componentDidUpdate()");


  render = () => {
    console.log("SubComponent.render()");

    return <button onClick={this.handleClick}>{this.state.ctr}</button>
  };
}

export default class App extends Component<{}, any>
{
  constructor(props: any) {
    super(props);
    console.log("App()");
    this.state = { ctr: 0 }
  }

  incrementCounter = () => this.setState({ ctr: this.state.ctr + 1 });
  componentDidMount = () => console.log("App.componentDidMount()");
  componentDidUpdate = () => console.log("App.componentDidUpdate()");

  render(): ReactNode {
    console.log("App.render()");

    return (
      <div>
        <button onClick={this.handleClick}>{this.state.ctr}</button>
        <SubComponent></SubComponent>
      </div>
    )
  };

  handleClick = () => this.setState({ ctr: this.state.ctr + 1 });
}
```

On startup the logged events become.

```
App()
App.tsx:37 App.render()
App.tsx:8 SubComponent()
App.tsx:18 SubComponent.render()
```

```
App.tsx:6 SubComponent.componentDidMount()
App.tsx:26 App.componentDidMount()
```

And if we click a button to change the App state we see

```
App.render()
App.tsx:18 SubComponent.render()
App.tsx:6 SubComponent.componentDidUpdate()
App.tsx:26 App.componentDidUpdate()
```

Unmounting

Now we show how unmounting works. We modify our code to conditionally show the SubComponent and add to SubComponent more logging to show what happens

```
export class SubComponent extends Component<{}, any>
{
  constructor(props: any) {
    super(props);
    console.log("SubComponent()")
    this.state = { ctr: 0 }
  }

  handleClick = () => this.setState({ ctr: this.state.ctr + 1 });
  componentDidMount = () =>
    console.log("SubComponent.componentDidMount()");
  componentDidUpdate = () =>
    console.log("SubComponent.componentDidUpdate()");
  componentWillUnmount = () =>
    console.log("SubComponent.componentWillUnmount()");


  render = () => {
    console.log("SubComponent.render()");

    return <button onClick={this.handleClick}>{this.state.ctr}</button>
  };
}

export default class App extends Component<{}, any>
{
  constructor(props: any) {
    super(props);
    console.log("App()");
    this.state = { show: false }
  }

  componentDidMount = () => console.log("App.componentDidMount()");
  componentDidUpdate = () => console.log("App.componentDidUpdate()");

  render(): ReactNode {
    console.log("App.render()");

    return (
      <div>
        <button onClick={this.handleClick}>{this.state.show.toString()}</
button>

        {this.state.show ?
          <SubComponent></SubComponent> : ""
        }
      </div>
    )
  };

  handleClick = () => this.setState({ show: !this.state.show});
}
```

STARTUP

So now on startup the subcomponent is not displayed. Our messages are then

```
App()
App.tsx:39 App.render()
App.tsx:29 App.componentDidMount()
```

Now we click the button to add the subcomponent

```
App.render()
App.tsx:7 SubComponent()
App.tsx:21 SubComponent.render()
App.tsx:5 SubComponent.componentDidMount()
App.tsx:29 App.componentDidUpdate()
```

Now we click the subcomponent button to modify the subcomponent

```
SubComponent.render()
App.tsx:5 SubComponent.componentDidUpdate()
```

Now click the App button to remove the subcomponent

```
App.render()
App.tsx:5 SubComponent.componentWillUnmount()
App.tsx:29 App.componentDidUpdate()
```

Functional Components

BASICS

The lifecycle is a little more complex with functional component. Our code is rewritten as follows

```
  export function SubComponent() : ReactElement {
    console.log("SubComponent()");
    const [ctr, setCtr] = useState(-0);
    return <button onClick={() => setCtr(ctr + 1)}>{ctr}</button>
  }

  export default function App() : ReactElement {
    console.log("App()");
    const [showSub, setShowSub] = useState(false);

    return (
      <div >
        <button onClick={() => setShowSub(!showSub)}>
          {showSub.toString()}</button>

        {showSub ?
          <SubComponent></SubComponent> : ""
        }
      </div>
    );
}
```

On startup there is no subcomponent and so we see the following

```
  App()
}
```

If we click the App button the subcomponent becomes visible.

```
  App()
  App.tsx:4 SubComponent()
```

Now if we click the subcomponent button, we see

```
  SubComponent()
```

Now we click the App button to remove the subcomponent and see

```
  App()
```

EFFECTS

We do not have the same lifecycle events in function components that we have in classes. We do have effects though. The useEffect feature registers a function that is invoked when a component is

- Mounted
- Updated
- Unmounted

We modify out code as follows.

```
export function SubComponent() : ReactElement {
  console.log("SubComponent()");
  const [ctr, setCtr] = useState(-0);

  useEffect(
      () => {
         console.log("Subcomponent.useEffect()");

         return () => {
          console.log("Subcomponent.effectTeardown()");
         }
      }
  );
  return <button onClick={() => setCtr(ctr + 1)}>{ctr}</button>
}

export default function App() : ReactElement {
  console.log("App()");
  const [showSub, setShowSub] = useState(false);

  useEffect(
    () => {
       console.log("App.useEffect()");

       return () => {
        console.log("App.effectTeardown()");
       }
    }
);

  return (
    <div >
      <button onClick={() => setShowSub(!showSub)}>
        {showSub.toString()}</button>

      {showSub ?
        <SubComponent></SubComponent> : ""
      }
    </div>
  );
}
```

Now when we start the app we see.

```
App()
App.tsx:26 App.useEffect()
```

If we click the button to add the subcomponent we see.

```
App()
App.tsx:4 SubComponent()
App.tsx:9 Subcomponent.useEffect()
App.tsx:29 App.effectTeardown()
App.tsx:26 App.useEffect()
```

Basics

JSX

WHAT IS JSX

JSX is a syntax extension to JavaScript that enables one to create React elements which can be subsequently rendered to the HTML DOM. It enables one to create expressions such as

```
const element = <h1 className=”myClass”>Hello World</h1>;
```

Upon compilation the compiler creates something like this

```
const element = React.createElement(
  'h1',
  {className: 'myClass'},
  'Hello World'
)
```

The resulting element looks something like this

```
const element = {
  type: 'h1',
  props: {
    className: 'myClass',
    children: 'Hello, world!'
  }
};
```

The generated elements can be thought of as rendering instructions. React uses these instructions to construct and update the DOM. The react elements are lightweight, immutable objects.

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

DETAILS

Because JSX generates JavaScript objects we can do things like this.

```
var flag = true;

var jsx = flag ?
    <h1>Hello World</h1> :
    <h1>Bye World</h1>;
```

We can put any valid java script inside JSX if we wrap it with curls braces.

```
var myName =
    {
      First: "Kenny",
      Second: "Wilson"
    };
```

```
const element = (
   <div>
       <h1>Name</h1>
       <h2>{myName.First}</h2>
       <h2>{myName.Second}</h2>
    </div>
);
```

React Components

PROPS

React components take a collection of properties called props and produce a react element

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

❷ const element = <Hello name="Wilson"/>;


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

❶ We create a component called Hello. Note that the component's render method uses JSX to return a React element.

❷ We now use the component in another piece of JSX to create a higher-level element

❸ We render the element

Notice that our component names must begin with an uppercase letter. This is so JSX can distinguish between React components and HTML elements. JSX assumes anything beginning with a lower-case letter is a HTML element and anything beginning with an uppercase letter is a React Component.

Components can refer to other components in their render methods, encouraging code reuse.

> **PROPS**
>
> A component must **never** modify its own props collection.

STATE

If we want to update state in a component, rather than update the props collection, we update the **state** collection.

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

  render() {
    return (
    <h1>{this.state.date.toLocaleTimeString()}</h1>
    );
  }

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

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

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

> **SETTING STATE**
>
> State should never be directly updated. Instead it needs to be updated inside a call to the **setState** method. We don't use `this.state.date = new Date();` Instead we use `this.setState({date: new Date()});`

Multiple updates can be batched together or executed asynchronously for performance. For this reason, we should use the following form when using current state to calculate the next value.

```
this.setState(function(❶state, ❷props) {
  return {
    counter: state.counter + props.increment
  };
});
```

❶ previous state

❷ props at the time the update is applied.

Where we use separate calls to independently update different parts of state the updates are merged.

EVENTS

In HTML we pass a string that contains a JavaScript statement when we register an event handler with an event. Also note the event itself is all lowercase.

```
<button onclick="handleEvent()">Click Me</button>
```

In React the event itself is camel case and the event handler is an actual java script function

```
<button onClick={handleEvent}>Click Me</button>
```

With HTML we generally use addEventListener if we want to add a handler to an event after the DOM element is created. In React we can just provide a handler when the element is rendered. If a component is created as a class, typically an event handler is a method on the class. Because java script methods are not bound by default, we need to make sure we bind to ensure this is available in the handler.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);

    // Make this available in the handler
    this.handleEvent = this.handleEvent.bind(this);
  }

  handleEvent() {
    /* Do Something */
  }
}
```

Keys

When creating lists of elements, we need to give each a special string key attribute. The keys enable react to know when list items are added, removed or changed. The value of the key must uniquely identify the element among its siblings in the list. Keys must only be unique within the containing list. They do not need to be unique among all lists. It is not recommended to use the index of an element itself as a key.

Forms and controlled components

HTML form elements have their own state. The react component generating the form elements also has its own state in its state property. A controlled react component is a react component whose state property is the single source of truth. In a controlled component every state mutation has an associated handler function. We can use this to modify or validate user input.

Questions – Basics

Overview

What is the philosophy of React?

Couple together rendering and logic.

What is the React abstraction that facilitates this?

React Components contain both markup and logic.

JSX

What is JSX?

A Syntax extension to JavaScript

What is JSX used for?

To create react elements.

The elements can be rendered to the DOM.

Given the following JSX show what JS a compiler might generate?

```
const element = <h1 className=”myClass”>Hello World</h1>;

const element = React.createElement(
  'h1',
  {className: 'myClass'},
  'Hello World'
)
```

What are react elements?

Descriptions of what we want rendered to screen.

React uses them to construct the DOM and keep it up to date

How do we modify react elements?

We cannot. They are immutable once created.

How do react elements differ from DOM elements?

They are lightweight and cheap to create.

How does React optimise the rendering process?

React only applies DOM updates needed to bring DOM to desired state.

How does one set literal attributes for HTML elements?

Using quotes

```
const element = (
    <input type="text" value="word">
    </input>
);
```

How does one specify expression to the value of attributes?

Using curly braces. No quotes

```
var myName = "John";
const element = (
    <input type="text" value={myName}/>
);
```

What are the HTML attributes class and tabindex called in React DOM?

className and tabIndex (Note the camel case)

Components, Props and State

What does a React Component do?

Take a collection of properties and produce a React element.

What two ways are components created?

As a function or as a class

What restrictions are there on React component names?

They must start with an uppercase letter

Why?

React treats components starting with lower case letters as DOM tags

What is the top-level component in a standalone React app called by convention?

App

When should we factor out components?

If part of the UI is used several times it is a good candidate for a component.

Props

What golden rule must all components respect?

They must behave like pure functions with respect to their props.

State

How does state differ from props?

It is private and fully controlled by the component?

How do we update state?

Always using the setState method. We only directly set the state collection in the constructor.

How do we update state?

Life Cycle

What life-cycle method runs after a component has been rendered to the DOM?

componenetDidMount

What life-cycle method is for tear down?

componnentWillUnmount

How do we update the state of a react component?

Using a set of lifecycle methods

Event Handling

TypeScript

This section covers the same material as basics but uses TypeScript

JSX

WHAT IS JSX

JSX is a syntax extension to JavaScript that enables one to create React elements which can be subsequently rendered to the HTML DOM. It enables one to create expressions such as

```
let element: ReactElement = <h1>Heading One</h1>;
```

Upon compilation the compiler creates something like this

```
const element = React.createElement(
    'h1',
    {},
    'Hello World'
  )
};
```

The generated elements can be thought of as rendering instructions. React uses these instructions to construct and update the DOM. The react elements are lightweight, immutable objects.

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

React Components

STATELESS

React components can be functional or class based. We start by looking at a simple stateless component. The functional form is as follows.

```
export default function FunctionalComponent() : ReactElement {
    return <h1>FunctionalComponent</h1>;
}
```

And the class-based form is then.

```
export default class ClassBasedComponent extends Component {

    render() : ReactElement {
        return <h1>ClassBasedComponent</h1>;
    }
}
```

PROPS

Now let's look at some props. First the functional form

```
export default function FunctionalComponent(props:PropsShape) :
    ReactElement {
    return <h1>FunctionalComponent {props.name}</h1>;
}
```

And a class-based form.

```
export default class ClassBasedComponent extends Component<PropsShape> {

    constructor(props:PropsShape)
    {
        super(props);
    }

    render() : ReactElement {
        return <h1>ClassBasedComponent {this.props.name}</h1>;
    }
}
```

STATE

Let us now consider a stateful component. First as a class as it is easier. Note how we also deal with event handling in this sample.

```
export default class ClassBasedComponent extends
    Component<PropsShape,StateShape> {

    constructor(props:PropsShape)
    {
        super(props);
        this.state = {current:props.initialNumber};

        this.handleClick = this.handleClick.bind(this);
    }

    render() : ReactElement {
        return <button onClick={this.handleClick}>{this.state.current}</b
utton>;
    }

    handleClick() {

        this.setState((state:StateShape) => {
            return ({current:state.current*1.1});
        });
    }
}
```

And now maybe we can use a functional component. Note we must use hooks for this.

```
export default function FunctionBasedComponent(props:PropsShape)
: ReactElement
{
    const [value, setValue] = useState(props.initialNumber);

    return <button onClick={()=>setValue(value*1.1)}>{value}</button>;
}
```

Template Projects

React, TypeScript and Jest – From Scratch

In this section we show how to setup a template project with the following support

- React
- TypeScript
- Unit Testing with Jest
- Debugging Unit Tests with Jest
- Debugging React in the browser

First create a new empty directory for our project and move to it.

Project Structure

We will add the following folders to give our project structure.

Folder	Details
`src`	The location of our source files
`src/components`	Where we store our components
`public`	Where we store any static content
`.vscode`	Store the `launch.config` we can use to start our application and/or tests in debug mode

```
mkdir src/components
mkdir public
mkdir .vscode
```

Package Dependencies (Runtime)

We will need the following packages

Package	Details
`react`	The code necessary to define react components
`react-dom`	A renderer that can render to Web Pages. We would use react-native to render for native
`react-scripts`	Scripts used for development. These are used by create react app.
`@types/react`	TypeScript definitions for react
`@types/react-dom`	TypeScript definitions for `react-dom`

Package Dependencies (Development)

We will require the following packages for development

Package	Details
`jest`	The types necessary Jest JavaScript testing library
`typescript`	The TypeScript compiler
`ts-jest`	TypeScript pre-processor for Jest that lets one use Jest to test code written in TypeScript
`react-test-renderer`	Render react components to pure JavaScript objects that represented the DOM tree without requiring a browser or DOM.
`@types/jest`	TypeScript definitions for `jest`
`@types/react-test-renderer`	TypeScript definitions `react-test-renderer`

We set up package.json with the dependencies for dev and runtime

Listing 1Package.json

```
{
    "dependencies": {
      "react": "*",
      "react-dom": "*",
      "react-scripts": "*",

      "@types/react": "*",
      "@types/react-dom": "*"
    },

    "devDependencies": {
      "jest": "*",
      "typescript": "*",
      "ts-jest": "*",
      "react-test-renderer": "*",
      "@types/jest": "*",
      "@types/react-test-renderer": "*"
    }
}
```

And then run the command `npm install` to install all the packages.

SCRIPTS

The react scripts provide a set of useful commands. We add them to `package.json` so we can call them from `npm start`

We will need the following packages

Script	Purpose
`react-scripts start`	Run the app in development mode. Open http:localhost:3000 to view it in the browsser

```
{
    "dependencies": {
      "react": "*",
      "react-dom": "*",
      "react-scripts": "*",

      "@types/react": "*",
      "@types/react-dom": "*"
    },

    "devDependencies": {
      "jest": "*",
      "typescript": "*",
      "ts-jest": "*",
      "react-test-renderer": "*",
      "@types/jest": "*",
      "@types/react-test-renderer": "*"
    },

    "scripts": {
        "start": "react-scripts start",
        "build": "react-scripts build",
        "test": "react-scripts test",
        "eject": "react-scripts eject"
      }
}
```

BROWSER LIST

We add the following canned browserlist

```
{
  "dependencies": {
    "react": "*",
    "react-dom": "*",
    "react-scripts": "*",

    "@types/react": "*",
    "@types/react-dom": "*"
  },

  "devDependencies": {
    "jest": "*",
    "typescript": "*",
    "ts-jest": "^25.3.1",
    "react-test-renderer": "^16.13.1",
    "@types/jest": "^25.2.1",
    "@types/react-test-renderer": "^16.9.2"
  },

  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
```

TYPE-SCRIPT COMPILER

We use the following canned type-script compiler options. Add these in a file called `tsconfig.json` at the root level of the project

```
{
  "compilerOptions": {
    "target": "es5",
    "lib": [
      "dom",
      "dom.iterable",
      "esnext"
    ],
    "allowJs": true,
    "skipLibCheck": true,
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true,
    "strict": true,
    "forceConsistentCasingInFileNames": true,
    "module": "esnext",
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "noEmit": true,
    "jsx": "react",
    "sourceMap": true,
    // "isolatedModules": true
  },
  "include": [
    "src"
  ]
}
```

REACT APPLICATION FILES

public/index.html

This is the static html file, into which react will render its top-level component. Note the bold div into which react will render.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1" />
    <title>Hello World React Application</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

/src/components/App.tsx

The top-level Application react component. We define it using TypeScript. Note that the file extension is hence `.tsx`. Add the file to `src/components`

```
import * as React from "react";

// 'HelloProps' describes the shape of props.
// State is never set so we use the '{}' type.
export class Hello extends React.Component<{},{}> {
    render() {
        return ( <h1>Hello {new MyType().getValue()}</h1> );
    }
}

class MyType {

    public getValue() : number {
        return 14.55;
    }
}
```

/src/index.tsx

The entry point to react. Note it contains code to render into the div we specified in `index.html`

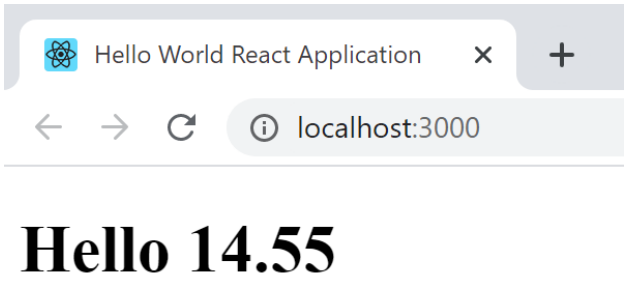
```
import * as React from "react";
import * as ReactDOM from "react-dom";

import { Hello } from "./components/App";

ReactDOM.render(
  <Hello/>,
  document.getElementById("root")
);
```

Test the build

We can now test the app by running

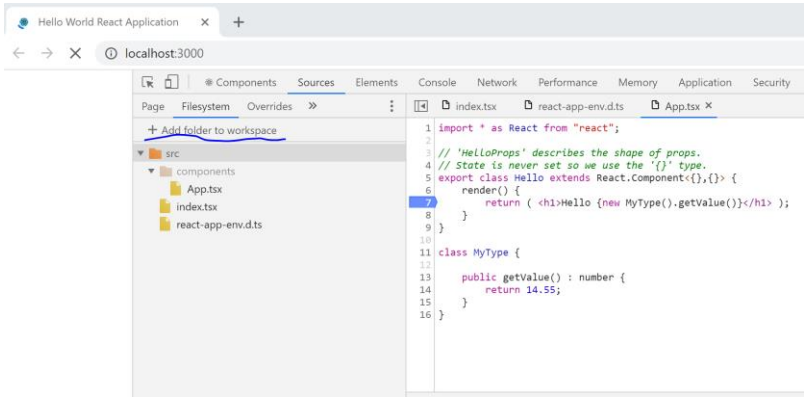
```
npm run start
```

We should see the following



DEBUGGING IN CHROME

Make sure you have installed the react debugging extension in Chrome. Hit F12. Because the `npm run start` command uses webpack we need to point the tools to the source code. Use the following "Add folder to workspace"



Then the "Sources" and "Components" commands become very useful.

ADDING TESTING

We use Jest for testing. In order to support TypeScript, add the following file to the root and call it `jest.config.js`

```
module.exports = {
    "roots": ["src"],
    "transform":{"^.+\\.tsx?$": "ts-jest"}
}
```

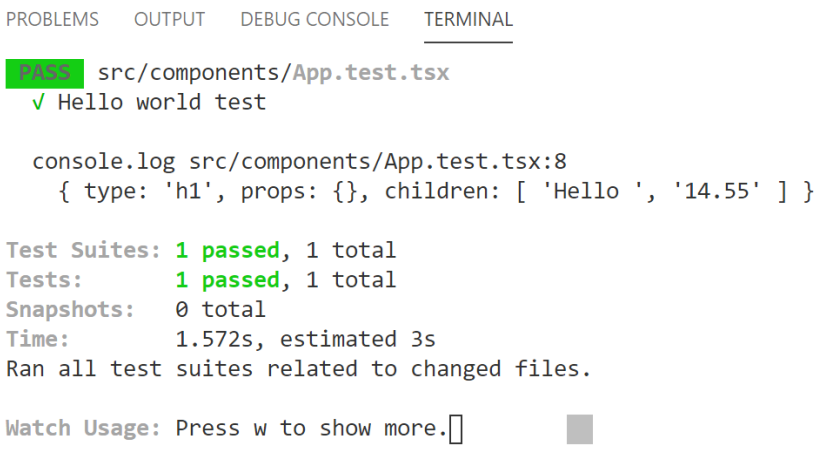
Now add a file to the src/components directory called `App.test.tsx` and enter the following code.

```
import {Hello} from "./App";
import * as React from "react";
import * as renderer from "react-test-renderer";

test("Hello world test", () => {
    let js = <Hello></Hello>;
    let res = renderer.create(js).toJSON();
    console.log(res);
});
```

We can now test the app by running

```
npm run test
```

Our terminal window should look like this



DEBUGGING TESTS

To debug our tests we add lanch.json to .vscode folder with the following

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Debug Jest Tests",
            "type": "node",
            "request": "launch",
            "runtimeArgs": [
              "--inspect-brk",
              "${workspaceRoot}/node_modules/jest/bin/jest.js",
              "--runInBand"
            ],
            "console": "integratedTerminal",
            "internalConsoleOptions": "neverOpen",
            "port": 9229
          }
    ]
}
```

We can add breakpoint to our test file and hit F5 to run the tests in debug mode.

DEBUGGING APP

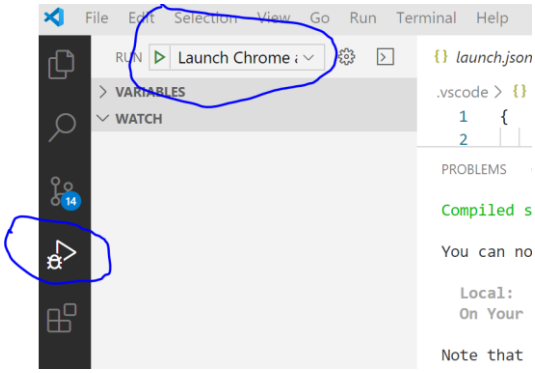
Add the following to `launch.json`

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Debug Jest Tests",
            "type": "node",
            "request": "launch",
            "runtimeArgs": [
              "--inspect-brk",
              "${workspaceRoot}/node_modules/jest/bin/jest.js",
              "--runInBand"
            ],
            "console": "integratedTerminal",
            "internalConsoleOptions": "neverOpen",
            "port": 9229
          },
          {
            "type": "chrome",
            "request": "launch",
            "name": "Launch Chrome against localhost",
            "url": "http://localhost:3000",
            "webRoot": "${workspaceFolder}"
          }
    ]
}
```

Make sure the app is running in the browser by first running

```
npm run start
```

Then make sure we select the correct run profile from debugger



Now put a breakpoint and hit F5 to attach.

React, TypeScript and Jest – React App

Move to where you want to create the app and run the following command

```
npx create-react-app my-react-app --typescript
```

To debug the app in VS code then add the following in bold

Add the following to `launch.json`

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Debug Jest Tests",
            "type": "node",
            "request": "launch",
            "runtimeArgs": [
              "--inspect-brk",
              "${workspaceRoot}/node_modules/jest/bin/jest.js",
              "--runInBand"
            ],
            "console": "integratedTerminal",
            "internalConsoleOptions": "neverOpen",
            "port": 9229
          },
          {
            "type": "chrome",
            "request": "launch",
            "name": "Launch Chrome against localhost",
            "url": "http://localhost:3000",
            "webRoot": "${workspaceFolder}"
          }
    ]
}
```

React, TypeScript and Ag-grid

Move to where you want to create the app and run the following command

```
npx create-react-app my-react-app --typescript
```

Move to my-react-app and add the following

```
npm install --save @ag-grid-community/all-modules
npm install --save ag-grid-community
npm install --save ag-grid-react
```

Replace index.tsx with the following

```
import React, { Component } from 'react';
import { render } from 'react-dom';
import { AgGridReact } from 'ag-grid-react';
import 'ag-grid-community/dist/styles/ag-grid.css';
import 'ag-grid-community/dist/styles/ag-theme-balham.css';

interface AppProps { }
interface AppState {
  name: string;
}

enum PutCall {
  Put,
  Call
}

const columnDefs =  [
    {headerName: 'Underlyling', field: 'ul'},
    {headerName: 'Expiry', field: 'expiry'},
    {headerName: 'Strike', field: 'strike'},
    {headerName: 'Put/Call', field: 'putCall'}
];
const rowData = [
    {ul: 'FTSE',expiry: 'Dec19', strike: 5400, putCall:PutCall.Call},
    {ul: 'SX5E',expiry: 'Dec19', strike: 2600, putCall:PutCall.Put},
    {ul: 'DAX',expiry: 'Dec19', strike: 5400, putCall:PutCall.Put},

]

class App extends Component<AppProps, AppState> {
  constructor(props: any) {
    super(props);
    this.state = {
      name: 'React'
    };
  }

  render() {
     return (
        <div
          className="ag-theme-balham"
          style={{ height: '200px', width: '800px' }}
        >
          <AgGridReact
            columnDefs={columnDefs}
            rowData={rowData}>
          </AgGridReact>
        </div>
      );
  }
}

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

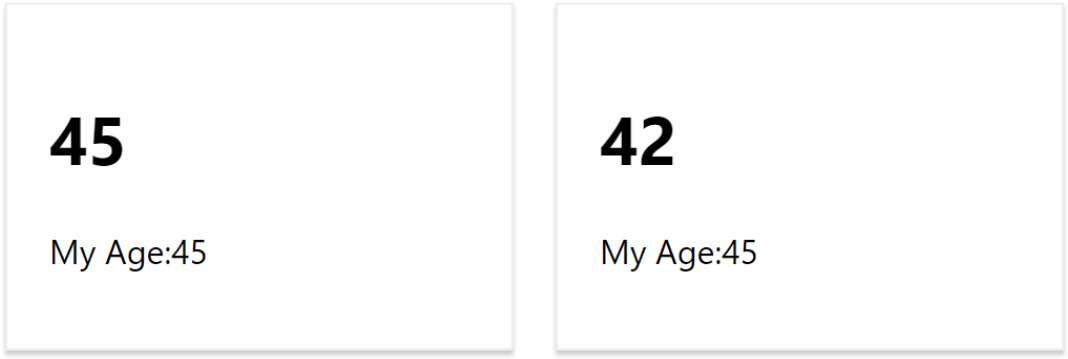
Tooling

A toolchain is a pipeline of tools that take compile and otherwise process source code and other artefacts ready for delivery and execution. With frameworks toolchains are opaque. The React toolchain uses WebPak and WebPack development to create bundles and send them to the browser. WebPack in turn uses Babel to compile JSX into JavaScript which it can then bundle.

TypeScript

Examples

Card



React

```
import React, { ReactElement } from "react";
import './Card.css';

export interface CardProps
{
    name: string;
    age: number;
}

export function Card(props: CardProps) : ReactElement
{
    return (
        <div className="person">
            <h1>{props.age}</h1>
            <p>My Age:{45}</p>
        </div>
    );
}
```

CSS

```
.person {
    display: inline-block;
    margin: 10px;
    border: 1px solid #eee;
    box-shadow: 0 2px 2px #ccc;
    width: 200px;
    padding: 20px;
  }
```