The beginners

GUIDE TO REACT & REDUX

The Complete React Web Development Course (with Redux)

Section 1

REACT 101

- Install Visual Studio Code
- Install Node.js
- Setting up a Web Server
- Setting Up Babel
- Exploring JSX & JSX Expressions
- ES6 Variables & Arrow Functions
- Manual Data Binding
- Forms & Input
- Arrays in JSX

In this section we will dive into the very basic fundamentals of React and setting up your own web developer environment to write and execute your own code. We will go through installing a very popular text editor release by Microsoft called Visual Studio Code to installing Node.js on your machine and then onto setting up your very own local web server and installing Babel.

Once we have setup your developer environment we will continue to learn the fundamentals of JavaScript (ES6) and JSX syntax to build our foundational skills on both languages and technologies on which the React Library is built upon.

After learning the basics we will move onto the next sections where we will start learning more about the React component architecture and will build our own applications.

It will soon become apparent why React is considered a very popular frontend JavaScript library as you are not required to learn a new language but rather learn an application architecture based on components.

INSTALLING VISUAL STUDIO CODE

There are many free and paid text editors out there available to download and use to write your own code, popular editors include Atom, Sublime Text, Notepad++ and Visual Studio Code to name a few. We will be using Visual Studio code as our text editor of choice to write our code as it allows us to install some useful packages to help write our code.

To install Visual Studio code simply visit the following website and download and install the application onto your machine: https://code.visualstudio.com/

Visual Studio Code has some useful extensions which you can install:

- Bracket Pair Colorizer
- ES7 React/Redux/GraphQL/React-Native Snippet
- Liver Server
- Prettier - Code Formatter
- Babel ES6/ES7

INSTALLING NODE.JS

Node is JavaScript on the server. You can visit the following website to download node onto your machine: https://nodejs.org/en/

Download the latest version of node that is available on their website.

To check that you have node.js installed on your machine, simply open up your terminal and enter the following command:

```
$ node –v
```

This will allow us to double check that node was installed onto our machine as we now have this new command and it also shows us what version of node you have installed on your machine. When installing node we also got NPM (node package manager) which allows us to install various dependencies/packages such as React or Yarn and other libraries. NPM and Yarn aims to do the same job. To check that you have npm installed enter the following command in your terminal:

```
$ nom –v
```

To install yarn on your machine globally run the following command in your terminal:

```
$ npm install –g yarn
```

On windows machines you will need to restart your machine to complete the installation. To check that yarn has installed successfully, run the following code in your terminal:

```
$ yarn ––version
```

SETTING UP A WEB SERVER

To setup a developer web server we can achieve this in two ways using live-server.

Firstly you will need to create a directory (folder) for your application. This folder will act as a place for all your project code. This folder can be called anything for example 'indecision-app'. In this example we will create a sub-folder called public and store our basic HTML file.

If we open VS Code and have installed live-server extension we can simply open up our html document and right-click to open the file with live-server. Every-time we update our project files in the folder, the live-server will refresh the browser which will update our application with the changes automatically.

Alternatively, we can use npm or yarn to install live-server globally onto our machines using either command in the terminal:

```
$ npm install –g live-server    or    $ yarn global add live-server
```

To check that we have installed live-server on our machine correctly we would run the command:

```
$ live-server –v
```

To run live server from the terminal, navigate to your file directory using cd and the file path. Note: you can use **cd** to change directory, **ls** (or **dir** on windows) to list all the files within the folder. You can use **cd ~** to navigate back to your user folder. Once you have navigated to your project directory run the following code:

```
$ live-server public
```

Note: you would run live-server followed by the folder name in the directory you wish to serve through the live web server (in our example we had a sub-folder called public which contained our HTML file). Any changes made in the folder will automatically update in the browser.

SETTING UP BABEL

Babel is a compiler and allows you to write for example JSX, ES6, ES7 code and have it compile down to regular ES5 code, allowing your code to work on browsers which only support ES5 syntax.

https://babeljs.io/

Babel on its own has no functionality. Babel is a compiler but its not going to compile anything by default. We have to add various plugins and presets in order to get any sort of change in our code (e.g. taking JSX code and converting it into ES5 createElement calls). A preset is just a group of plugins.

We are going to install babel locally on our machines so that when we write our code in JSX/ES6/ES7 it will compile locally to our ES5 code (i.e. we want to write our code locally on our machine and update without using Babel on the web).

The babel website has a docs page which provide documentation on plugins available to you to install. We will install two presets: react preset and env preset. These presets have all the necessary plugins we require and we would not need to install the plugins individually by ourselves (*as this could get out of hand very quickly*). The env preset will include ES2015, ES2016 and ES2017 plugins which will give us access to the new JavaScript features (e.g. arrow functions, const and let variables, spread operator etc).

In our local environment we are going to install three things:
Babel itself, env preset and react preset.

In your terminal run the following command to install babel @v6.24.1 globally on your machine.

$ npm install –g babel-cli@6.24.1 or $yarn global add babel-cli@6.24.1

This will give us a new command in our terminal which we can run while in our project directory. Run the following code to check if Babel has been installed successfully. This should print out the help output in your terminal and will indicate if Babel has been installed successfully on your local machine.

```
$ babel ––help
```

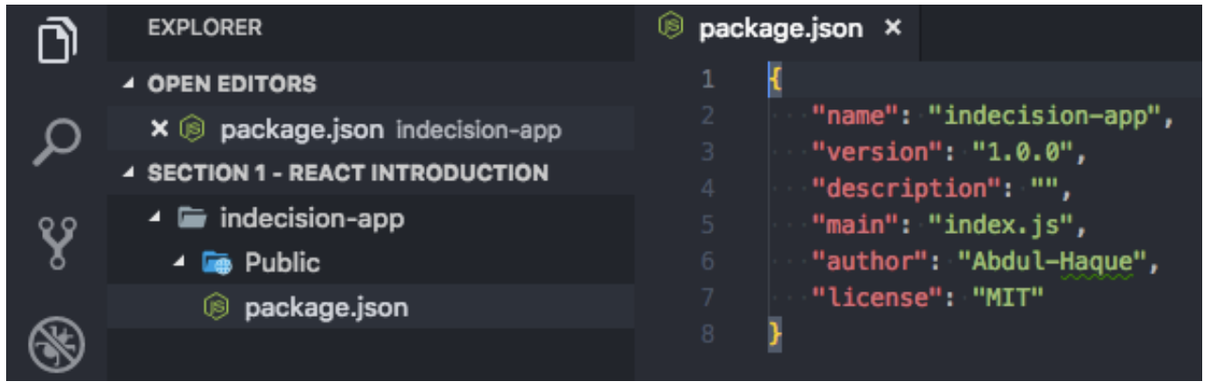
To clear the terminal enter the command line:

```
$ clear
```

The react and env presets will be installed locally in our projects (i.e. these codes will live in our projects so that babel CLI can take advantage of these codes to transform our code down to ES5 syntax). Within the project directory enter the following command:

```
$ npm init      or    $ yarn init
```

This command will setup our project to use node/yarn and specify the local dependencies. This will walk us through series of questions such as the app name, version, description, entry point, repository, author and license (MIT).
All this does is, it generates a brand new file in our project called package.json (*note we could have created this file ourselves without the npm/yarn init command*). The whole point of package.json is to outline all the dependencies that the project needs in order to run. This will make it easy to install everything.

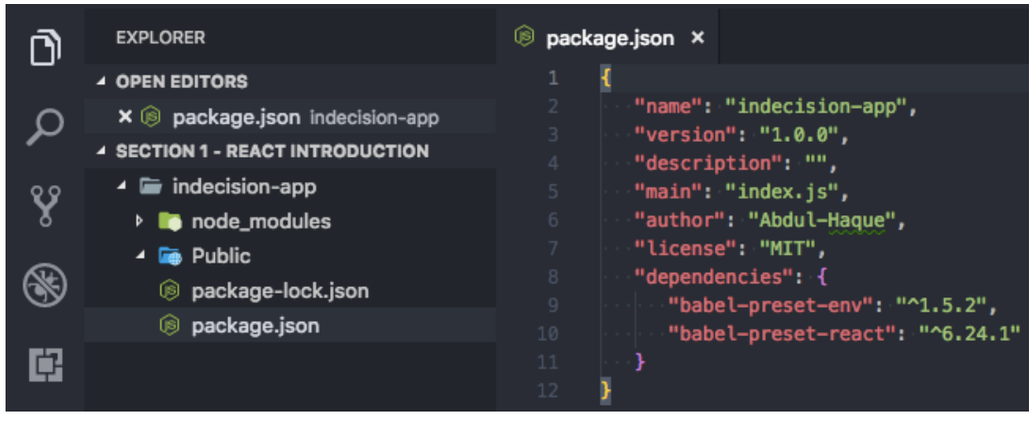


We are going to add our dependencies to this package.json file using the following commands in the terminal:

```
$ npm install babel-preset-react@6.24.1 babel-preset-env@1.5.2      or
```

```
$ yarn add babel-preset-react@6.24.1 babel-preset-env@1.5.2
```

This will install the two dependencies and will update the package.json file in our app folder which will now list our dependencies for our app. Notice that a new folder has been created called node_modules. This is where all the modules and dependencies will live.



The dependencies will have their own package.json file which will list the dependencies they require to run. This will install the all the sub-dependencies of their own in the node_modules folder. The package.json file makes it easy to install all the dependencies and sub-dependencies your application will need.

We will never need to go into the node_modules folder to change any of the code. This folder is a auto-generated folder from the package.json file and we can always delete and reinstall this folder/modules using the dependencies information from the package.json file (*we would use the command $ npm install or $ yarn install to install the node_modules folder again*).

This command has also generated a package-lock.json file (*if you used yarn this file will be called yarn.lock*) in our project folder. We will not need to edit/manually change this file. This file lists out all the dependencies in the node_module folder, the version used and where exactly it got that package from. This helps npm/yarn behind the scenes.

We are now able to use Babel locally on our machines within our project directory to compile our React JSX code into regular ES2015 code. In the terminal we will need to run the following command with a few arguments:

```
$ babel src/app.js ––out–file=public/scripts/app.js ––presets=env, react ––watch
```

The first argument specifies the path to our code we wan to compile (*in our example above it lives in the src folder and the file is called app.js*).
The second argument specifies the output file (in our example above it lives in the public/scripts folder and is also called app.js). This file will always be overridden by Babel.
The third argument specifies the presets we would like to use (this is a *comma separated list of the presets we wish to use*).
Finally, the last argument will watch for any changes in the first specified file to update (compile) the second specified file automatically.

Babel will compile the JSX/ES6/ES7 code in one file into regular ES2015 code the browser will understand in the other file.

EXPLORING JSX

In JSX you can only have a single root element. If you want to add more elements side by side we will need to wrap it within a single root element for example:

```
var template = <div><h1 id="someid">Header Text</h1><p>Paragraph Text</p></div>;

var appRoot = document.getElementById('app');

ReactDOM.render(template, appRoot);
```

When we are creating JSX expressions, we can get really complex expressions to include a lot of information/nested elements - however, we must have a single root element.

We can make our JSX expression more readable by formatting the elements on separate lines i.e. format as we would do in our html code for example:

```
var template = (
    <div>
        <h1 id="someid">Header Text</h1>
        <p>Paragraph Text</p>
    </div>
);
```

The above would still be seen as valid JSX expression and table will successfully render the expression down to ES2015.

The above is more readable to the eye and easily understood. We can make the expression even more complex and as long as there is a single parent root element (the wrapper) then this will be seen by Babel as valid JSX. Below is another example of a complex JSX expression with nested elements.

```
var template = (
    <div>
        <h1 id="someid">Header Text</h1>
        <p>Paragraph Text</p>
        <ol>
            <li>Item One</li>
            <li>Item Two</li>
        </ol>
    </div>
);
```

All this is doing is creating nested React.createElement() functions calls to create each element in the ES2015 syntax.

```
    React.createElement('ol', null,
        React.createElement('li', null, 'Item One'),
        React.createElement('li', null, 'Item Two')
    )
```

This is why we do not write our React code in React.createElement calls as it is difficult to read and write, instead we use JSX.

We can make our JSX code dynamic by using variables to store data and then referencing these variables within our JSX expressions. For Example:

```
var userName = 'John Doe';

var template = (
    <div>
        <h1> { userName .toUpperCase()} </h1>
    </div>
```

We use the curly brackets to write any JavaScript expressions. Therefore we can enter the JavaScript variable within the curly braces to output the variable value in our JSX code. This allows our JSX to be dynamic rather than static.

In our JavaScript expressions we can use different types such as strings, numbers, operators, arrays, objects, functions etc.

To render an object in React we must specify the object property, for example:

```
var user { name: 'John Doe',   age: 28,   location: 'Derby' } ;

var template = (
    <div>
        <h1> Name: { user.name } </h1>
        <h1> Age: { user.age } </h1>
        <h1> Location: { user.location } </h1>
    </div>
);
```

We have now dynamically injected our data from our variables into our JSX expression.

We should now have knowledge on how to use JavaScript expression in React, how to render JavaScript strings and number types and that we cannot render object but we can render the object properties.

There are still many more other JavaScript types. We are going to continue to look at JavaScript expressions and how we can make truly dynamic and useful JSX.

Conditional Rendering and conditional logic in general is at the very core of software development. The same will be true for our React interfaces for example, is the user logged into the app? If true, then show the logout button, else show the login button.

To perform Conditional rendering in our JSX we will need to use JavaScript expressions. We can continue using the regular conditional statements we have been using in vanilla JavaScript and do not need to learn any weird syntax.

The conditional JavaScript tools that are available are:
1) If Statements.
2) Ternary operators.
3) Logical and operator.

IF STATEMENTS:
A if statement is not an expression and therefore cannot live inside the curly brackets (i.e. JavaScript expression). However, calling a function is an expression. We can add whatever we want in our function including if statements. The return value from the function will show up in the JSX. For example:

```
var user = {
    user: 'John Doe',
    age: 28,
    location: 'Derby'
}
```

```
function getLocation(){
        if (location) {
                return location;
        } else {
                return ‘unknown’;
        }
};
var template = (
        <div> <h1> { getLocation(user.location) } </h1> </div>
) ;
```

We can use the function to operate the if statement and whatever is returned from our function can be used in our JavaScript expression to pass the data into our JSX to display.

We can use other JSX expression such as {123} and {<h3>Heading 3</h3>} and this will render in our browser. This is equivalent to writing JSX expression outside of the curly brackets. This is useful as it allows us to setup getLocation() to return a separate JSX expression. For example:

```
function getLocation(){
        if (location) {
                <h1> { getLocation(user.location) } </h1>
        } else {
```

```
            return undefined;
        }
};
var template = (
        <div> { getLocation(user.location) } </div>
) ;
```

Important Note: undefined is implicitly returned if you do not explicitly return it. Therefore the above getLocation() function can be simplified without returning the else statements:

```
function getLocation(){
        if (location) {
            <h1> { getLocation(user.location) } </h1>
        }
};
```

This will implicitly return undefined when the statement returns false. If the Statement returns true this will return the JSX expression value within the other JSX expression curly brackets. This will then render in our browser.

TERNARY OPERATORS:
A ternary operator is a expression and not a statement and therefore we do not need to add it to a function. A ternary operator allows us to write if statements more concisely. For example:

```
True ? 'Return True' :  ' Return False';          [This will return 'Return True' as the statement is true]
```

False ? ‘Return True’ : ' Return False’; [This will return ‘Return False’ as the statement is false]

These two ternary operator will check whether the statement is true or false. If true it will return the first value else it will return the false value. Therefore the first example will return ‘Return True’ while the second example will return ‘Return False’. We use the ? to return something when the statement equates to true while we use the : to return something when the statement equates to false.

The first part of the ternary operator is the statement we wish to check the value i.e. whether it will return true or false. Depending on the answer we want to return the first value else we return the second value. For example:

{ app.option.length >0 ? ’Here are your options’ : ’No options’ }

LOGICAL AND OPERATOR:

Much like the undefined, null and the boolean values true and false are all ignored by JSX. Therefore we can write a JSX expression of {true}, {false}, {null} or {undefined} and all will be ignored by JSX and will not be rendered on the screen. This is a very useful feature.

If we want to display a JSX expression e.g. tag for age, only if the user is 18yrs or older else we do not want to display the age of the use if they are under age. We could use the function technique, however, we are going to explore this new technique which is just as concise as the ternary operator.

The Logical And Operator uses the && to check if two arguments returns true to run the code. If both do not return true we do not want to run the code. We will examine the Logical AND Operator to understand the technique and how it can be used to solve the above problem without using to the function technique.

true && ‘Some Age’;

If we run the above, this will return the value ‘Some Age’. So in Logical And Operator if the first value is true it is not going to use/return that first value; however, if the second value is also true it will use/return the second value.

```
False && 'Some Age';
```

If we now run the above, this will return false. So in this instance, where the first value is false, that value is what actually gets used/returned and ignores the second value ever exists.

Therefore if we check for age and the user is below 18 a boolean (false) is returned and this is ignored by JSX. We can use this tools and technique to add this conditional rendering in our JSX.

The ternary function is great for when you want to do one of two things, while the Logical And Operator is useful for when you want a condition to do one thing else you want to do nothing at all.

We can make our Logical And Operator even more complex by checking two things in our first value to equate to true or false for example we can check if the age exists at all and is above 18 to return our JSX. For example:

```
{ (user.age && user.age >=18) && <p>Age: { user.age } </p> }
```

We have now explored the very basics of conditional rendering techniques we will use over and over again. Eventually, over time and lots of practice, we will eventually master all three strategies. However, we should now be confident of being able to make our JSX expressions more dynamic.

ES6 ASIDE: LET & CONST

ES6 introduces let and const variables keywords which are alternatives to the var variable keyword. Is there anything inherently broken with var? No. The only problem with var is that it can be easily misused creating unnecessarily weird situations. For example: not only can you reassign a var variable but you can also redefine the variable. This means if recreate a var variable without knowing that you have created it elsewhere in your code, you are essentially overriding the original variable value and therefore you will run into hard to debug issues.

```
var nameVar = 'John Doe';

var nameVar = 'Mike';
```

The let and const variables prevents you from reassigning/redefining existing variables.

The let variable allows you to reassign the variable however you cannot redefine the variable (i.e. you cannot redeclare the same variable. You can however reassign it to another type e.g. a string to a number).

```
let nameLet =  'John Doe';

let nameLet = 'Mike';          > Uncaught SyntaxError: Identifier 'nameLet' has already been declared at...
```

The const variable does not allow you to reassign nor redefine the variable again.

```
const nameConst = 'Frank';

const nameConst = 'Barry';     > Uncaught SyntaxError: Identifier 'nameLet' has already been declared at...

nameConst = 'Barry';           > Uncaught TypeError: Assignment to content variable at...
```

By default you should always use the const variable when creating a new variable, however, if you need to reassign the variable then you should switch to the let variable instead. You should stop using the var variable.

There are a few other differences between var, let and const variables i.e. the scoping of the variables are also different.

The var, let and const variable are all function–scoped meaning that the variable within the function cannot be accessed outside the function.

```
function getName(){
        name = 'Paul';
}
getName();
console.log(name);                    > Uncaught ReferenceError: name is not defined at…
```

Note: we could create the variable called name outside the function and set the value to the function which will work.

The let and const variables are all block-scoped whereas var is not. Bock Scoping is where the variable are also bound to the code block (e.g. the code block for an if statement or a code block for a for loop). For example: if we had a variable in an if statement this variable can be accessible outside this code block provided this variable uses var instead of let and const.

```
const fullName = 'John Doe';
If(fullName) {
        const firstName = fullName.split(' ');
}
console.log(firstName)                > Uncaught ReferenceError: name is not defined at…
```

Remember the rule: '*const first; if we need to reassign, let; … var never!*'

ES6 ASIDE: ARROW FUNCTIONS

Arrow functions is a new ES6 syntax and is used heavily throughout React, therefore, it is important to have a basic grasp of how arrow functions operate.

We are going to compare and contrast a regular ES2015 function with a ES6 Function. The function we are going to create will square a number.

ES2015 Regular Function Syntax:

```
const square = function(x) {
    return x * x;
};
console.log(square(5));
```

ES6 Arrow Function Syntax:

```
const square =  (x) => {
    return x * x
};
```

Arrow functions do not need the function keyword. We start with the function argument(s) followed by an arrow and then the function body.

Arrow functions are always anonymous (*unlike ES2015 functions where we can give the function a name*). To give an arrow function a name we must always declare a const or let variable first and assign the arrow function to it as we did above.
Example of ES2015 Function with a Name:

```
function square (x) {

        return x* x;

};

console.log(square(5));
```

If an arrow function body returns only a single expression we can use the new syntax. We no longer require the curly brackets. The expression is implicitly returned and therefore we do not need to write the return keyword (compared to the ES2015 syntax where we need to explicitly return):

```
const square = (x) => x * x
```

The argument object s and this keywords are no longer bound to the arrow function. This means that if you try to access the arguments, it is not going to work.

ES2015 Example:

```
const add = function (a, b) {

        console.log(arguments);

        return a + b;

};

console.log(add(1, 2, 3);
```

The arguments will have access to the 100 object even though this has not been defined in the function and is outside the function.

However this has gone away with arrow functions as we cannot access objects outside the arrow function.

```
const add = (a, b) => {
        console.log(arguments);
        return a + b;
};
```

```
console.log(add(1, 2, 3)        > Uncaught ReferenceError: argument is not defined at…
```

In ES2015 when we use a regular functions and we define it on an object property, the this keyword is bound by that object e.g. we have access to the values for example:

```
const user =  {
        name: 'John',
        cities: ['Bristol', 'Bath', 'Birmingham']
        printPlacesLived: function ()  {
                this.names;
                this.cities.forEach(function(city) {
                        console.log(this.name + ' has lived in ' + city);
                });
        }
};
```

The forEach function takes in a single function as a parameter and will be called once for each item in the array with a single argument called city. However, this will return an error in the JavaScript console of Uncaught TypeError: Cannot read property 'name' of undefined at… This is because the this.name is not accessible inside the forEach function but is

accessible in the printPlacesLived function which is and object inside of the user object. The past workaround was to create a const variable called that and assign the this value to it and then in the nested function use the that.name which will make the function work without causing the error message.

With arrow functions, they no longer bind their own values, instead they use the value of the context they were created in i.e. it would just use its parent 'this' value. We no longer require the ES2015 workaround.

```
const user = {
    name: 'John',
    cities: ['Bristol', 'Bath', 'Birmingham']
    printPlacesLived: function () {
        this.names;
        this.cities.forEach((city) => {
            console.log(this.name + ' has lived in ' + city);
        });
    }
};
```

There is no need for the const that = this workaround as arrow functions will use the value from the parent this keyword.

There are places where you would not want to use the arrow function for the very reason above such as methods for example if we turned the printPlacesLived method into an arrow function this will return and error message of Uncaught TypeError: Cannot read property 'name, cities' of undefined at... because the arrow function is no longer equal to the user object and will look to its parent level which is the global scope and therefore undefined causing the error.

ES6 does introduce a new method syntax where we can remove the ES2015 function keyword and have the above work.

```
printPlacesLive() {...};
```

Map is an array method like the forEach() but it works a little differently. This function gets called one time for every item in the array. Just like the forEach we have access to the item via the first argument (e.g. we can call them city). The difference is that the forEach method just lets you do something each time e.g. print to the screen, whereas, the map allows you to actually transform each item and return a new item back. We can therefore transform our array and get a new array back.

```
const userMap = {

    name: 'Peter Parker',

    cities: ['New York', 'London', 'Rome']


    printPlacesLived() {

        return this.cities.map((city) => this.name + ' has live in ' + city);

    }
};

console.log(printPlacesLived());
```

This will return a new array object of ['Peter Parker has lived in New York', 'Peter Parker has lived in London', 'Peter Parker has lived in Rome']. Maps are used a lot in React development.