# Components Handout

## What are Components?

Components can be used to split your UI into independent, reusable pieces. In a way they are similar to functions, in that they take in data, and return values. In our case they will take in data from our application, and in turn return a series of react elements describing what should appear on the screen.

## How do we create a Component?

There are two main ways to create a component, the first method, that may be familiar is to define a class for your component. The way to do this is using the standard JavaScript keyword **class**, followed by what you want your component to be called, followed by ‘**extends Component**’ to specify that this class must contain everything a React Component does when created.

This method should look familiar, as it is how the default App Component is created in the default example when your project is created. An example is shown below of a subcomponent being declared using this method is shown below:

import React, { Component } from 'react';

export class SubComponentClassMethod extends Component {

render() {

return (

<table>

<tr>

<th>heading 1</th>

<th>heading 2</th>

<th>heading 3</th>

</tr>

<tr>

<td>value 1</td>

<td>value 2</td>

<td>value 3</td>

</tr>

</table>

)

}

}

The export keyword must be included to allow the class to be accessed from another file. We will be using this class in our App.js file in order to show the result. The modified App.js file is shown below:

import React, { Component } from 'react';

import { SubComponentClassMethod } from './components/SubComponentClassMethod';

class App extends Component {

render() {

return (

<div className="App">

<div>

Class method component declaration:

<SubComponentClassMethod />

</div>

</div>

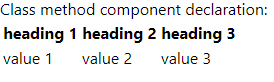
);

}

}

export default App;

We have modified this App.js file to remove all default contents so we can more clearly see our new component. To use our newly created Component, at the top of the file we must first import the class we created. We do this by using the keyword **import** followed by the name of the class we wish to import in curly braces, we then use the **from** keyword and provide a string, which is a relative path to the file the class is declared in, note that the file extension is not required on the end of this path. Using this class in the JSX section of App.js means we can now see the result on the screen. The result of the above code looks like this:



We have the Div in the App.js file, shown, with the next within it, and then the contents of the **SubComponentClassMethod** Component we declared, which is a simple table.

As mentioned earlier, there is however another method of declaring components. This method is to simply define them as a function. We do this in the traditional JavaScript method, using the keyword **function** followed by the name of the component we wish to define. Traditionally the name of a function would start with a lower case letter, however it is convention to have this function name start with an uppercase letter if we are defining a component.

The method we define as a component must have an argument called props, and return a portion of JSX. Below is the same example as the Component above, but declaring using this new method to show the difference:

import React from 'react';

export function SubComponentFunctionMethod(props) {

return (

<table>

<tr>

<th>heading 1</th>

<th>heading 2</th>

<th>heading 3</th>

</tr>

<tr>

<td>value 1</td>

<td>value 2</td>

<td>value 3</td>

</tr>

</table>

)

}

You implement this into another component in an identical manner, here is our updated App.js:

import React, { Component } from 'react';

import { SubComponentClassMethod } from './components/SubComponentClassMethod';

import { SubComponentFunctionMethod } from './components/SubComponentFunctionMethod';

class App extends Component {

render() {

return (

<div className="App">

<div>

Class method component declaration:

<SubComponentClassMethod />

</div>

<div>

Function method component declaration:

<SubComponentFunctionMethod />

</div>

</div>

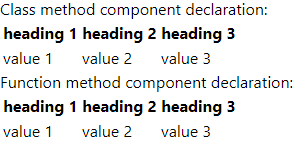
);

}

}

export default App;

And our final result looks like this:



As you can see both components appear on the page in an identical matter with no difference at all being shown to the end user.

## Why use one method of declaration over another?

Both methods can become very useful, and have their own unique cases where they should be used.

Function declarations are very short and concise in comparison to class declarations, and can help keep code to a minimum, and better organised and structured, as well as maintaining readability. When your component requires very little, or no logic, the function declaration is the better method to use.

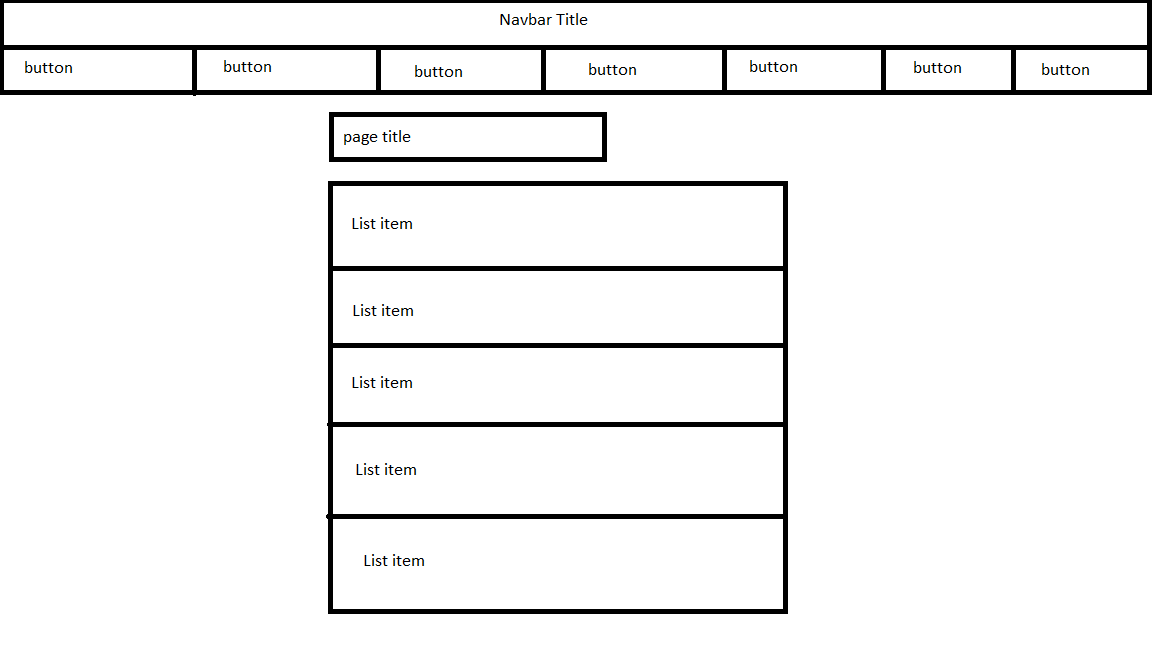
On the other hand, while class declarations can be longer, they also offer many advantages over function declarations. They can implement much more JavaScript logic, due to the ability to implement functions to manipulate the data provided to it in a more organised manner. A further advantage is the ability to implement ‘**lifecycle hooks**’ these are functions that are not called directly, but by the React library itself. Lifecycle hooks are events in react that occur at specific times such as when a component has finished loading onto the page, when the data the component is receiving updates, and when the component is about to be removed from the page. Overall this means that class declarations can keep our code much more organised and help our code exactly when it needs to.

Overall as you can see, declaring a component as a function is much better if it is a simple component that only needs to show data with very little logic in the background. However if the component requires more complex logic, and code that executes when a specific event occurs a class declaration is a much better method to use.

## Constructing a page using components

### Breaking down a design

To practice creating a page using components, let’s look at a page design and break down what would be a component, and what would be contents of a component. Take the following page design:



Let’s separate this into a possible react solution, splitting this into components and items within the components:



Looking at the diagram you can see 3 what are described as outer components, these are the navbar, page, and list container. This is because we can make the navbar static on every page, as this does not need to be reloaded, or its entire appearance changed on each page and can therefore be re-used to save resources. We then encapsulate the current page displayed in a component so this can be easily removed and added as needed and navigation occurs. Finally we have wrapped the list in a component, as the list could also be re-used and fed new data on other pages.

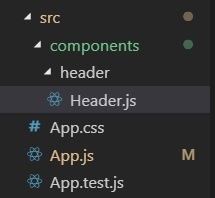
This is only one solution to this, there are many, many ways to separate this page, and none are correct or incorrect, there is no right answer for how to break down a page. For now however we will be working off this example.

### Creating the components

Let’s start creating this page

The first thing to start with is removing all of the contents from the App.js JSX so we can start fresh, and now we will begin creating the header:

The first step in creating the header is creating its containing file. Here a folder called Components was made, and a header folder placed within this with the Header.js file placed inside

  
Work then begun on the header itself, firstly making a class declaration of the component, placing a render function inside, and creating a div on the page of the correct size.

Header.js now contains:

import React, { Component } from 'react';

export class Header extends Component {

headerStyle = {

width : '100vw',

height: '15vh'

}

render() {

return (

<div id='header' style={this.headerStyle} >Header contents</div>

)

}

}

Which creates a div and makes it take up 100% of the width of the page, and 15% of the height, enough for both the title and the buttons.

We also alter App.js so we can see this on the page, this now contains:

import React, { Component } from 'react';

import { Header } from './components/header/Header';

class App extends Component {

render() {

return (

<div className="App">

<Header />

</div>

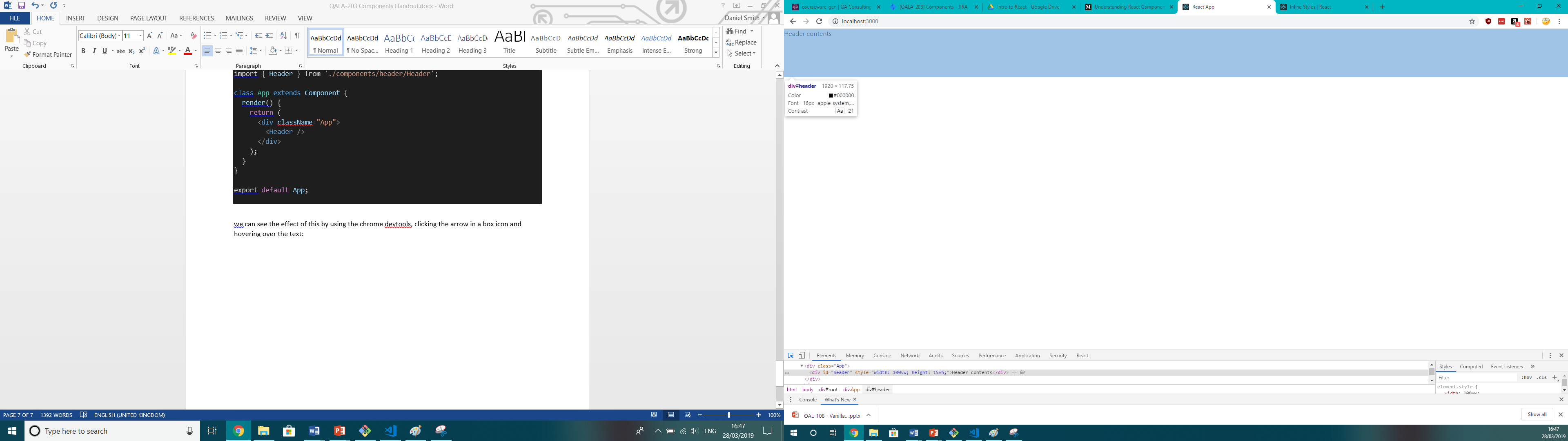
);

}

}

export default App;

we can see the effect of this by using the chrome devtools, clicking the arrow in a box icon and hovering over the text:



This demonstrates our styles working, and shows us that this is roughly the size we need.

We then add the rest of the elements on to the page and style them to match our desired appearance. This includes a h1 tag for the page title and buttons for the navigation elements. When the header has been finished the code looks like this:

import React, { Component } from 'react';

export class Header extends Component {

headerStyle = {

width: '100vw',

height: '15vh'

};

websiteTitleStyle = {

width: '100vw',

height: '50%'

};

buttonContainerStyle = {

width: '100vw',

height: '50%',

};

buttonsStyle = {

width: '20%',

height: '100%'

}

render() {

return (

<div id='header' style={this.headerStyle} >

<div id='website-title' style={this.websiteTitleStyle}>

<h1 style={{ margin: 0 }}>Website title</h1>

</div>

<div id='buttons' style={this.buttonContainerStyle}>

<button style={this.buttonsStyle}>Button 1</button>

<button style={this.buttonsStyle}>Button 2</button>

<button style={this.buttonsStyle}>Button 3</button>

<button style={this.buttonsStyle}>Button 4</button>

<button style={this.buttonsStyle}>Button 5</button>

</div>

</div>

)

}

}

For now this means our Header Component is complete. We will now focus our attention on the actual content of the page. We will wrap the page contents in another component I will call ListPage. Which is created in an identical manner to the Header Component:

import React, { Component } from 'react';

export class ListPage extends Component {

render() {

return (

<div>

<h2>List Page</h2>

</div>

)

}

}

And is implemented in the App.js file also in an identical manner:

import React, { Component } from 'react';

import { Header } from './components/header/Header';

import { ListPage } from './components/list-page/ListPage';

class App extends Component {

render() {

return (

<div className="App">

<Header />

<ListPage />

</div>

);

}

}

export default App;

This results in the following appearance:



This is all that will be put onto the ListPage Component for now, as the other item on this page will be a list, which will be a component in itself.

Beginning the List Component is identical to the other components, however it will implement another component for each list item. Therefore we will declare both components at the same time. As follows:

**ListContainer.js**

import React, { Component } from 'react';

import { listData } from './ListData';

import { ListItem } from './ListItem';

export class ListContainer extends Component {

render() {

return(

<div>

{this.generateListItems()}

</div>

)

}

generateListItems() {

const listItems = listData.map((item) =>

<ListItem item={item} />)

return listItems;

}

}

**ListItem.js**

import React, { Component } from 'react';

export class ListItem extends Component {

render() {

return(

<div>

<p>ID: {this.props.item.id}</p>

<p>Name: {this.props.item.name}</p>

</div>

)

}

}

As you can see, data is also provided for this list to use, typically this would come from a web service such as an API, however we will hard code this for now. This is simply an array defined in the usual fashion, and imported into the ListContainer Component in the same was as an exported class would be. The data is shown below:

export const listData = [

{

id: 1,

name: 'Dave'

},

{

id: 2,

name: 'John'

},

{

id: 3,

name: 'Daniel'

},

{

id: 4,

name: 'James'

}

]

In the ListContainer.js file you can see a new concept, this is using functions to determine the elements that will be placed in the JSX. In the function we take the data provided from the hardcoded file, and for each element in the array, create a ListElement component that gets passed the data it needs in what we call a **prop.** Do not worry about these for now as we will cover them in more detail in the future. The main concept to understand, is that we can use functions in our JSX, as long as they return JSX themselves. This can make our pages much more dynamic than hard coded elements. Our example here means that as the number of elements in the provided data array increases and increases, so will the number of JSX elements within the list.

We can then import the List Container into the List Page in the following fashion:

import React, { Component } from 'react';

import { ListContainer } from '../list/ListContainer';

export class ListPage extends Component {

render() {

return (

<div>

<h2>List Page</h2>

<ListContainer />

</div>

)

}

}

Our page should now be somewhat complete and give the following result:



Before we do a final code overview, we should follow best practice and remove the inline styles and place these in a separate file, in order to help keep our code more tidy and organised.

The first step is to create a new css file and move all of the css rules into this. This css file looks like this:

.header {

width: 100vw;

height: 15vh;

}

.websiteTitle {

width: 100vw;

height: 50%;

padding: 3px;

}

#buttonContainer {

width: 100vw;

height: 50%;

}

.buttonsStyle {

width: 20%;

height: 100%;

}

.headingText {

margin: 0;

}

And our modified header to use this file looks like this:

import React, { Component } from 'react';

import './header.css';

export class Header extends Component {

render() {

return (

<div id='header' className='header' >

<div className='websiteTitle'>

<h1 className='headingText'>Website title</h1>

</div>

<div id='buttonContainer' className='buttonContainer'>

<button className='buttonsStyle'>Button 1</button>

<button className='buttonsStyle'>Button 2</button>

<button className='buttonsStyle'>Button 3</button>

<button className='buttonsStyle'>Button 4</button>

<button className='buttonsStyle'>Button 5</button>

</div>

</div>

)

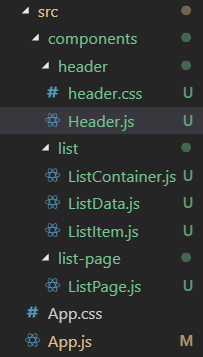
}

}

Before we go through the final review of the code we just created, let’s recap the process we took to get to this stage. We took a very basic design image and thought about what potential components this could be split into. We created a total of 4 new components, one to hold the website header, one to hold the page content, one which will create a list, and one which will repeat and become the items themselves. These Components were then connected to create our overall result, replicating our design in a useable format.

### Final code review

Our final project structure looks like:

This structure is used to keep our code organised and tidy. It keeps each component in its own folder so that the CSS and any other supporting files can be kept in the same location to keep it easy to find.

The exception to this is the Components relating to the list, which I have also placed together, because they will always be used together rather than individually.

Our App.js file looks like this:

import React, { Component } from 'react';

import { Header } from './components/header/Header';

import { ListPage } from './components/list-page/ListPage';

class App extends Component {

render() {

return (

<div className="App">

<Header />

<ListPage />

</div>

);

}

}

export default App;

You can see here the advantage of using a library such as react that allows us to split elements on the page into separate files. While this is the homepage to our website there is still very little code in this file itself. This keeps code extremely clean and tidy, while still allowing us to easily understand which sections of the page come from which component. ReactJS does a great job of keeping files small, concise and understandable.

Our final Header.js file looks like this:

import React, { Component } from 'react';

import './header.css';

export class Header extends Component {

render() {

return (

<div id='header' className='header' >

<div className='websiteTitle'>

<h1 className='headingText'>Website title</h1>

</div>

<div id='buttonContainer' className='buttonContainer'>

<button className='buttonsStyle'>Button 1</button>

<button className='buttonsStyle'>Button 2</button>

<button className='buttonsStyle'>Button 3</button>

<button className='buttonsStyle'>Button 4</button>

<button className='buttonsStyle'>Button 5</button>

</div>

</div>

)

}

}

This is a very simple file, used in a similar way to a standard html file, it simply defines what html elements should appear on the page. This file is an example where the component could possibly have been declared as a function rather than a class as it does not currently require any logic in order to function. Notice that we use the JSX keyword className is used rather than just class which would be used in html, because class is a reserved name in JavaScript and therefore had to be changed.

The final version of the header.css file to go alongside this looks like this:

.header {

width: 100vw;

height: 15vh;

}

.websiteTitle {

width: 100vw;

height: 50%;

padding: 3px;

}

#buttonContainer {

width: 100vw;

height: 50%;

}

.buttonsStyle {

width: 20%;

height: 100%;

}

.headingText {

margin: 0;

}

This is identical to a traditional css file, this just applies some very simple styling to make the header appear as we want it to, filling the entire width of the view, and having the buttons fill the available space.

Our final version of the ListPage.js file looks like this:

import React, { Component } from 'react';

import { ListContainer } from '../list/ListContainer';

export class ListPage extends Component {

render() {

return (

<div>

<h2>List Page</h2>

<ListContainer />

</div>

)

}

}

This is another simple Component that could perhaps be declared as a function. It is also kept incredibly simple by having all the code relating to the list in another file, and only having the surrounding content on this page within this file.

Our ListContainer.js file looks like this:

import React, { Component } from 'react';

import { listData } from './ListData';

import { ListItem } from './ListItem';

export class ListContainer extends Component {

render() {

return(

<div>

{this.generateListItems()}

</div>

)

}

generateListItems() {

const listItems = listData.map((item) =>

<ListItem item={item} />)

return listItems;

}

}

While this is still a reasonably simple component it does require more logic in order to generate the correct number of list items. The function works because it is called every time the component is re-rendered, which is typically every time its data changes. We can also force this to happen by using the forceUpdate function. When a component is re-rendered its DOM structure is discarded, and then re-created, meaning if the number of objects in the data provided to the list changes, the list is discarded, and re-created with the new number of list items within it.

The final ListItem.js file looks like this:

import React, { Component } from 'react';

export class ListItem extends Component {

render() {

return(

<div>

<p>ID: {this.props.item.id}</p>

<p>Name: {this.props.item.name}</p>

</div>

)

}

}

Once again if we wished to shorten this code, this is a place in which we could do so by declaring this as a function rather than a class, this is an incredibly simple component which simply displays data that is passed to it on the screen. As mentioned earlier we will cover this in more detail later. This component is ideal for our purpose as it is easily repeatable and simple to create with very little data required.

Our ListData.js file looks like this:

export const listData = [

{

id: 1,

name: 'Dave'

},

{

id: 2,

name: 'John'

},

{

id: 3,

name: 'Daniel'

},

{

id: 4,

name: 'James'

}

]

You can easily modify this to include more or less data, and quickly see the results on the page by refreshing the page. You could even include more data in each record and show this on the page with a slight modification to ListItem.js