neueda

What is React?

Objectives

- What is React?
- Why do we need it?

Activity – create a react app

Before we start…

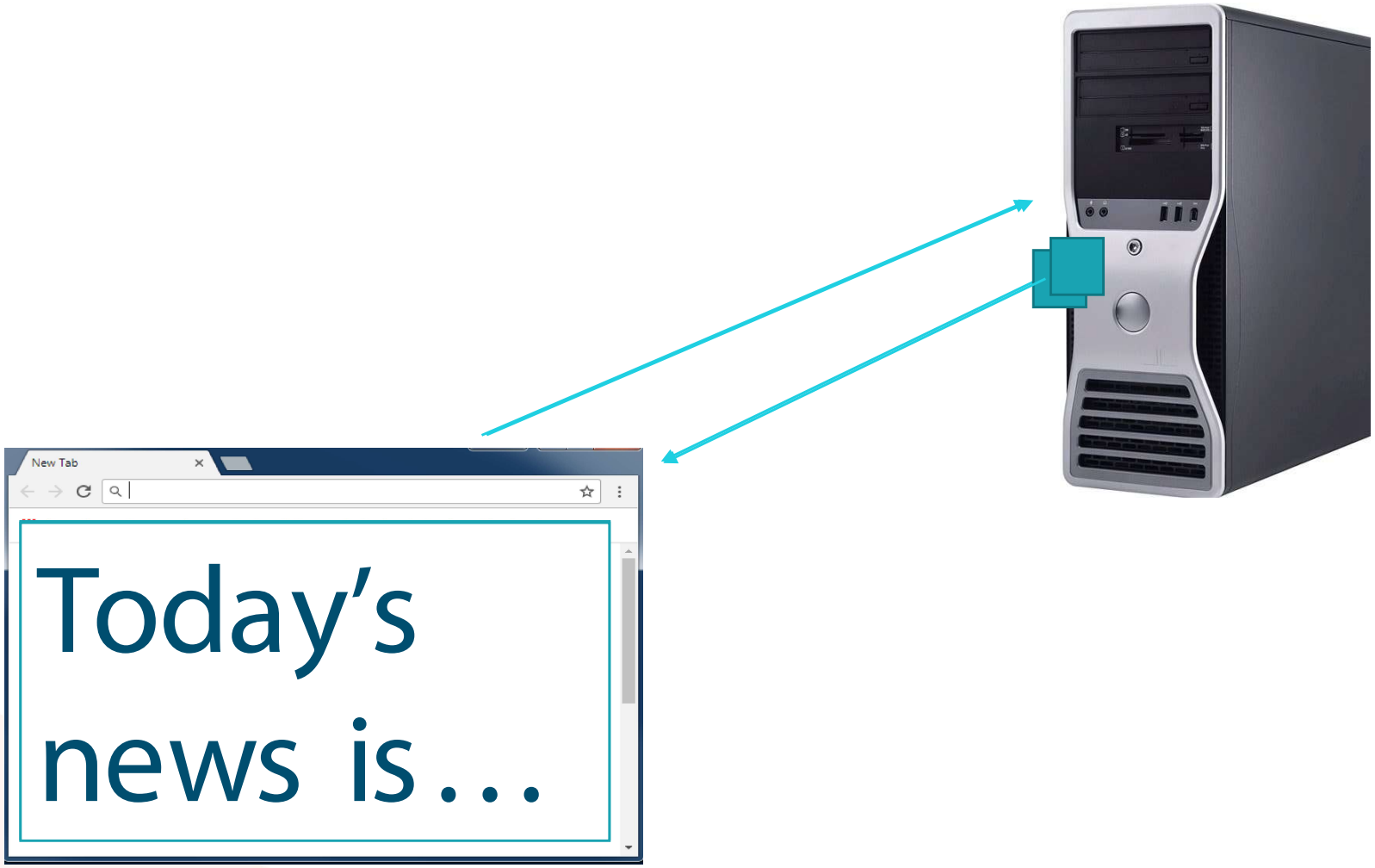
- Go to your workspace
- Run the command:

```
npx create-react-app hello-word --template typescript
```

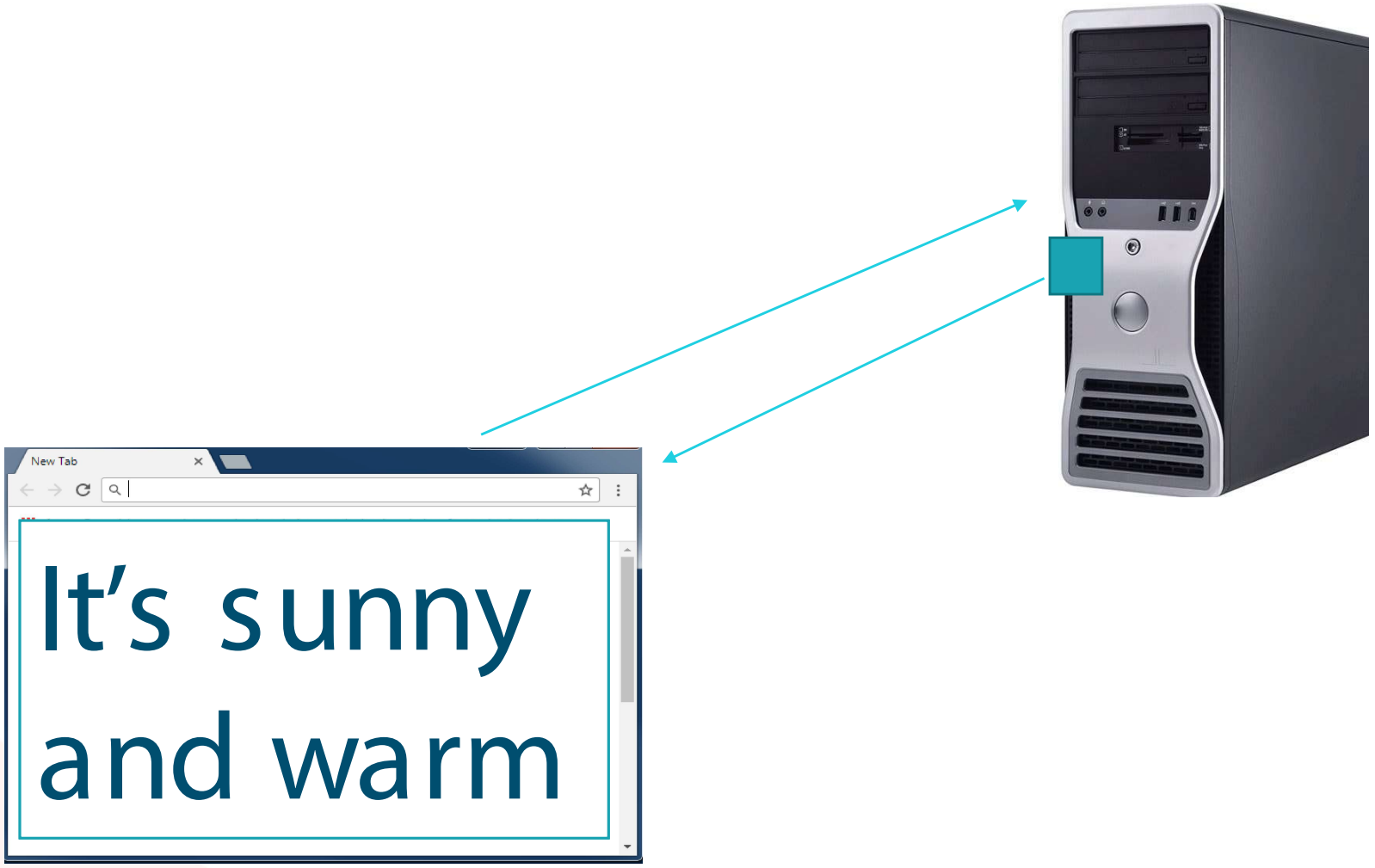
What is React?

- React is a set of 3rd party open source Javascript libraries that help interactive, fast, browser based user interfaces
- It was developed by Facebook.
- It's used by Netflix, AirBnB, Uber, Dropbox …
- Other similar frameworks include Angular and Vue.js
- It is a "single page application"

Traditional websites



Single page applications



Single page applications

- The browser will get one page only from the server (index.html).
- This page will reference other resources, including javascript files
- The javascript files contain code that simulates multiple pages and the full user interface.
- These can provide a better user experience as we avoid the server r
- In React, the page we get from the browser is very simple… the cor in the Javascript
- Building an application like this from scratch would be very difficult consuming. React makes it easier.

Transpiling

- We will create code in React which can contain other libraries, custo complex code (in typescript), and be structured over lots of files.
- When we are ready to build the application, the react transpiler cor code to plain Javascript that the browser can understand
- This minimises the code size, and ensures that the browser has eve needs straight away.

Activity – view a react app

- Open a React website.
- View the page source
- Inspect the page using the developer tools

What have we just done?

TLDR: Use `npx create-react-app hello-word --template typescrip` to create a new application, and `npm start` to run it.

- To create and build a react application we need a set of tools that are writt Node.js.
- Node.js comes with a package manager (npm) and we used that to start th application (npm start) – that is to spin up a web server containing our cod
- To create the new application we used npx – this is another tool that come Node.js and it's used to execute packages without first installing them.
- It works best if you create the application outside of an IDE but you can do

Summary

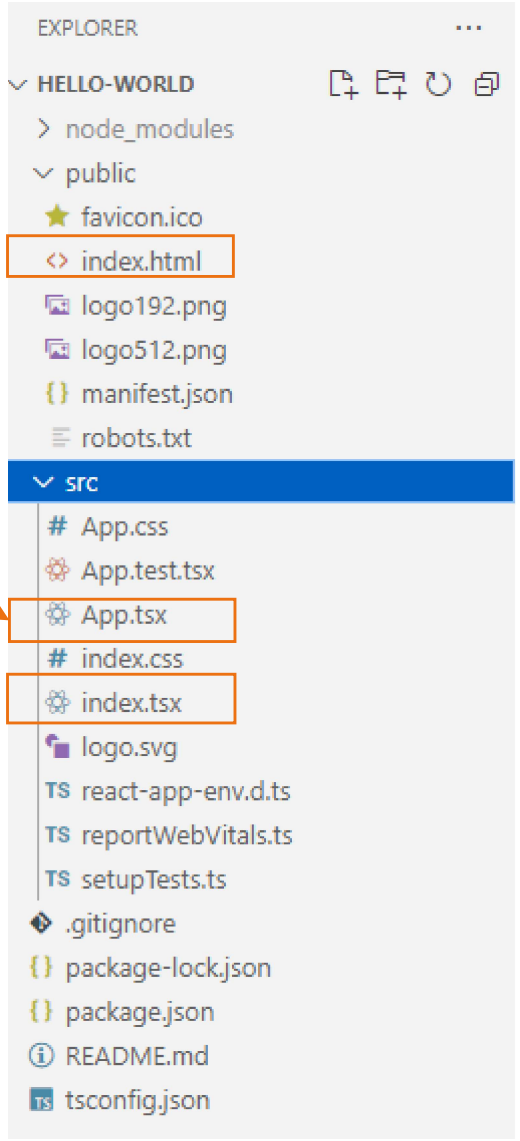
- What is React?
- Why do we need it?

neueda

React File Structure

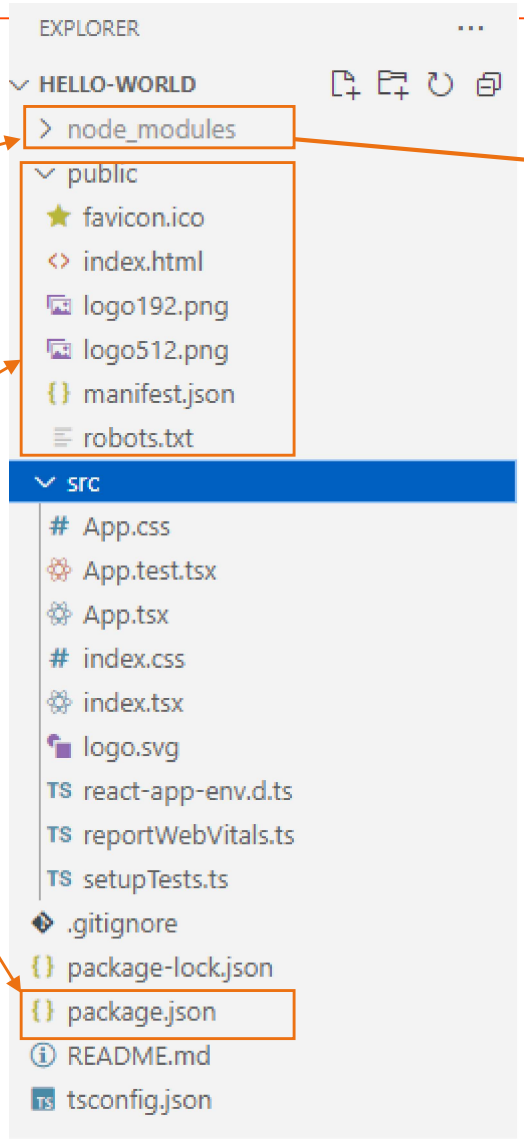
ReactJS File Structure

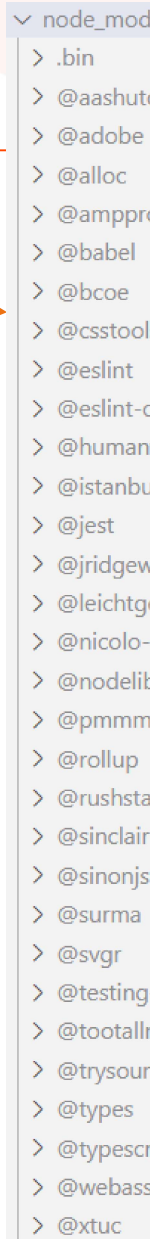
- Index.html is the entry point for the application
- The javascript that runs in the browser will execute the file index.tsx
- This file loads App.tsx which is where we start writing **our** code
- We can edit the App.tsx file, and add more files that can be referenced from here



ReactJS File Structure

- The libraries needed to create and build a React application are contained in the node_modules folder
- Static files (images, 3rd party JS libraries) go in the public folder
- The package.json file contains the JS library dependencies our application uses





neueda

Elements & JSX/TSX

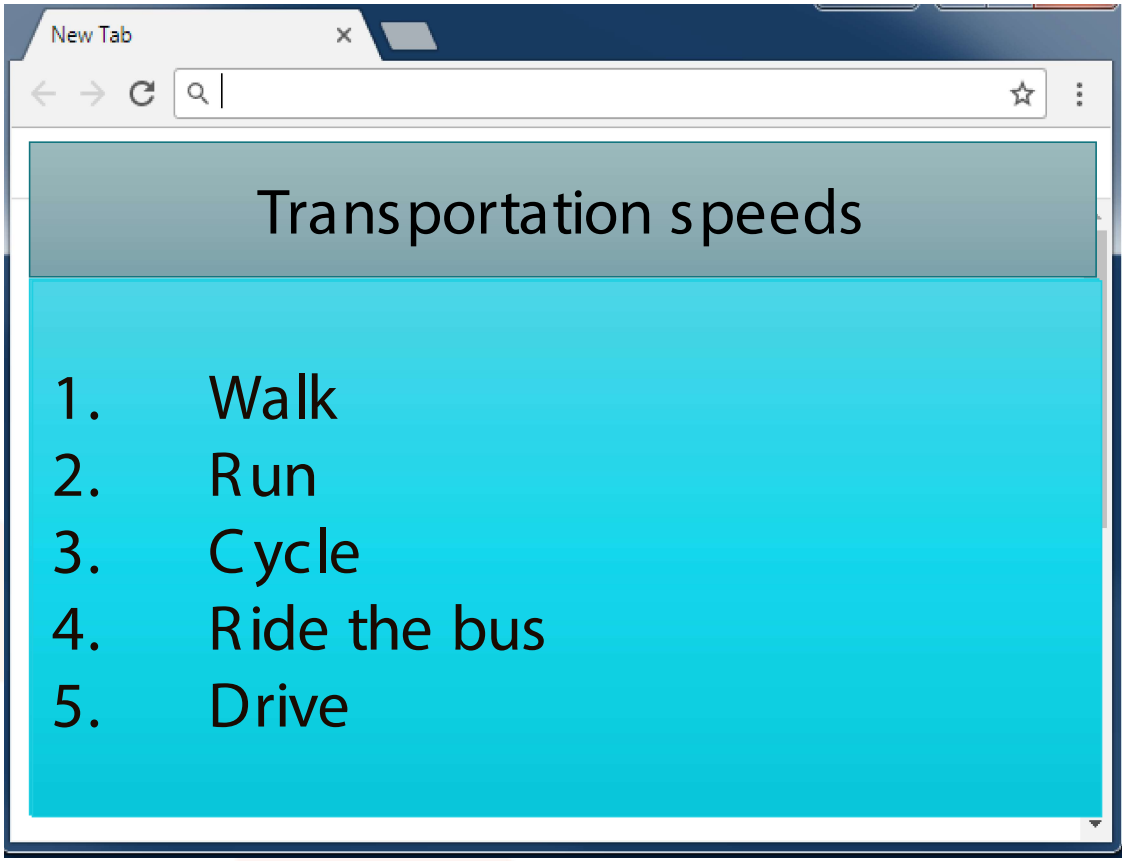
Objectives

- The Virtual DOM
- Elements
- JSX/TSX

The Virtual DOM

- React keeps an in memory version of the DOM called the Virtual D
- The browser's actual Dom and the virtual Dom are then synchronise
- We code against the Virtual DOM
- The entire React website is built as JavaScript code, not as HTML
- In this section we'll see the fully programmatic way that React work and then we'll introduce a much easier way to code this

The Virtual DOM



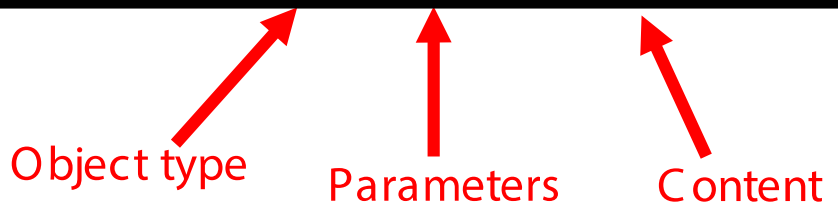
Transportation spee

1. Walk
2. Run
3. Cycle
4. Ride the bus
5. Drive

Elements

- The core building block of a web page is the element – think <p> <c
<button> etc
- To get these into our virtual DOM we need to create them as React
objects
- Creating an element uses the createElement method:

```
const myPara: React.ReactElement = React.createElement("p", null, "Hello World");
```



- We can construct the virtual Dom with the render method:

```
const root: ReactDOM.Root = ReactDOM.createRoot(document.getElementById('ro
root.render(myPara);
```

Properties

- The middle parameter of the createElement method is used to defi
  tag's attributes.
- These are provided as a JavaScript (JSON) object

```
const myPara: React.ReactElement = React.createElement("p", {class: "myClass"}, "Hello World");
```

Properties

Child elements

- To create an element within an element, (eg a paragraph within a d need to provide the child element as the content of the parent ele

```
const myDiv: React.ReactElement = React.createElement("div", {class: "firstDiv"}, myPara);
```

- You can provide an array of children elements.
- The impact of this is we must define the children before we can def parents

- **NOTE: the render method can only render a single parent object.**

Child elements

- You could nest child elements to create a more complex UI

```
const myList: React.ReactElement = React.createElement( "ul", null, [
    React.createElement("li", null, "first"),
    React.createElement("li", null, "second"),
    React.createElement("li", null, "third")
]);
```

JSX/TSX

- What we have just seen is the real code that is needed to create ele
  React.
- However there is a syntax extension called JSX which allows us to c
  React elements as HTML.
- We will write what looks like HTML (actually TSX) and a transpiler c
  will convert this to standard React

TSX →

```
const myPara: React.ReactElement = <p>this is paragraph 1</p>;
```

React →

```
const myPara : React.ReactElement = React.createElement("p", {class: "firstDiv"
```

JSX

- You can nest elements in JSX like you would if you were writing HT
- You can provide attributes like you would in HTML, but watch out f
  class attribute – this is className in JSX. Also for becomes htmlFor
- JSX attributes must always be closed

```
const myInput: React.ReactElement =<input id="something" />
```

- Remember that the ReactDOM.render method can only take a sing
  so you may need to wrap multiple objects in a parent tag.

JSX

- Where the attribute would be a set of key value pairs, such as the s attribute, we need to provide these as a javascript object, but then enclose that in { } so that we can bind to it…

```
<p style={{'color' : '#f00' , 'max-width' : '200px'}}>

   ...

</p>
```

- You can insert variables or javascript expressions within a JSX expre using curly brackets

```
const myDiv: React.ReactElement = <div>Your name is {name}</div>;
```

Summary

- The Virtual DOM
- Elements
- JSX / TSX

neueda

Introducing Component

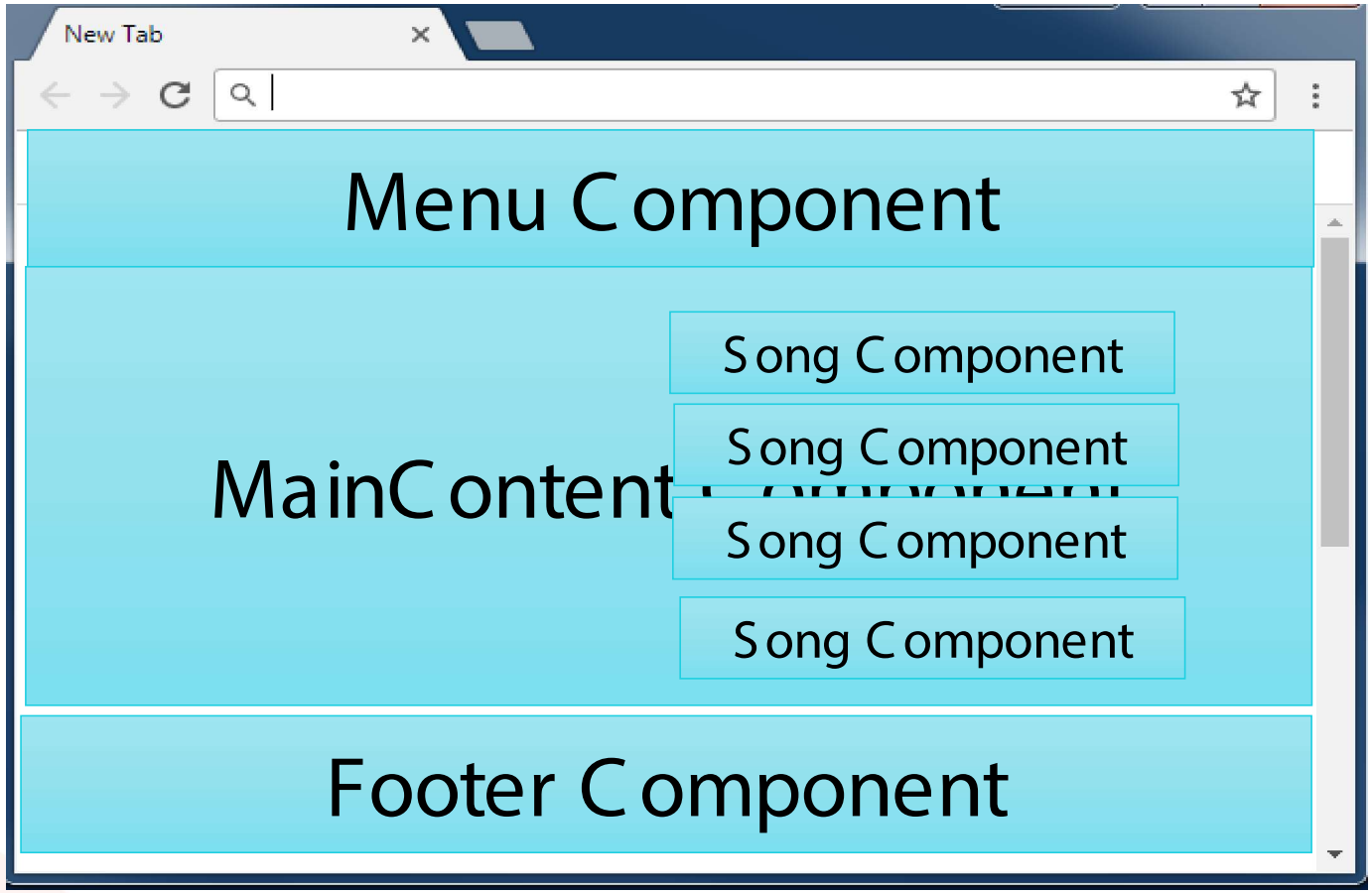
Objectives

- What are Components?
- Creating a component
- Using a component
- Component properties
- Component styling
- Fragments

What are components?

- Elements are not reactive – they don't respond to a mouse click
- To combine elements with functionality, we need to use componen
- Components are a grouping of elements, optionally with some func
- We can build pages out of elements
- Components can be re-used
- Components do not make up the Virtual DOM, but they are rendere elements that do.

What are components?



Creating a component

- A component is just a Typescript file containing a function that returns either a R
  or a JSX.Element
- What you place within the ( ) of the return statement is what will be used in the ReactDOM.render method
- Component names MUST begin with a capital letter
- There is an alternative way to create a component as a class – this is now outdated
- It used to be necessary to have an extra line at the top importing the react library but this is not needed now

```
const ExampleComponent = () : JSX.E

          return (<p>Hello world!</p
}

export default ExampleComponent;
```

Using a component

- We can place a component within another component by using its name as an ht
- We must import the component at the top of the file.
- The App component is the top level component
- This allows us to build up a page structure – the page is made of components which return elements

```
import Example from './components/Example

function App() {
  return (
    <div>
        <Example/>
    </div>
  );
}

export default App;
```

Component properties

- Component functions can take properties. These are provided as a javascript object
- It is convention to call the object “props”
- We define the structure of these props as a typescript Type

```
const Example = (props: ExampleProps) :
...
}

type ExampleProps = {status: string};
```

```
<Example status="ok"/>
```

- We pass properties into the component within the tag where we use it.

```
<p>The status is:
   {props.status}</p>
```

- The properties appear in the component as though they were variables. We can use variables in JSX by enclosing them in curly brackets.

Component properties

- It is possible to destructure the properties into separate variables rather than using a single props variable.

```
const Example = (props: ExampleProp
...
}
```

```
const Example = ({status, count} :E
...
}
```

- We pass properties into the component as usual

```
<Example status="ok" count="1"
```

- The way we then refer to the properties within the component depends on how they were defined

```
<p>The status is: {props.status
```

```
<p>The status is: {status} </p>
```

Component styling

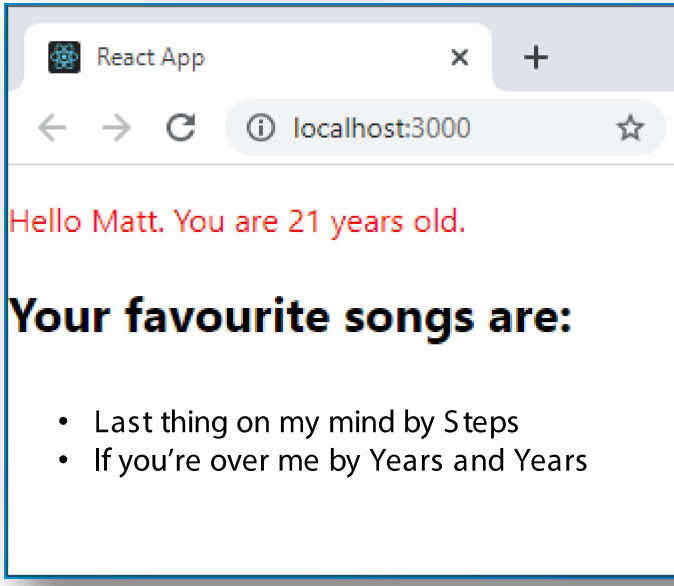
- Components will use the styling defined in App.css
- You can optionally create a css file for a specific component and im
  this styling will apply throughout the application, not just to 1 comp
  Use this structure for maintainability.
- Don't forget to use className if you are using class based css stylin

Component reusability

- You can re-use components by placing them anywhere inside other components.
- Components can appear more than once within the same compone

ACTIVITY - build a page

- Create a component called Song that takes 2 properties: title artist. The component should render these in an <li> tag.
- Create a component called SongList that will output some so within a <ul> tag. Put some kind of header text in this compo
- Place the SongList component into the App component.
- The output should look like the example shown. (Don't worry about styling)



Fragments

- Fragments allow us to return multiple JSX elements without
   or other rendering element to the DOM
- This is useful when an additional div might have an impact o
  or create invalid HTML

```
<React.Fragment>
   <p>...</p>
   <p>...</p>
</React.Fragment>
```

```
<>
   <p>...</p>
   <p>...</p>
</>
```

Summary

- What are Components?
- Creating a component
- Using a component
- Component properties
- Component styling
- Fragments

neueda

Component Events & Sta

Objectives

- Component Events
- The need for State
- Stateful variables in React
- The useState hook

Component events

- We can create events for our elements, such as onclick, onchange – work similarly to HTML
- The event names use camel case (eg onClick, onChange)
- We create arrow functions for each event, and then bind the event function with curly brackets

```
const someFunction = () : void => {
    //do something here!
}
```

```
<button onClick={someFunction} >send</button>
```

Component events

- The function that we bind an event to will receive a parameter of ty
  ClickEvent, ChangeEvent etc. This is a generic type so we can specif
  of object that generates the event.
- We do not need to supply the parameter to bind to this function – i
  automatically.
- We can use the parameter's **target** property to get access to the un
  object

```
<button onClick={someFunction} >send</button>
```

```
const someFunction = (event : ClickEvent<HTMLButton>) : void => {
    console.log(event.target.id);
}
```

The need for state

- We have seen that we can create variables and then bind to those v
  JSX. However if the value of a variable changes, then this will not g
  reflected on the DOM automatically.
- For the DOM to be refreshed the component must be re-executed
  render function must be run again)
- But if the component is re-executed, then the value of the property
  lost!
- The solution is that we need to create the variable as a stateful vari

What is a stateful variable?

- We can declare a variable within a component, but if we make it a s variable then the actual value is stored outside of the component. T it will survive a refresh of the component.
- To create a stateful variable, we need to use the useState hook.
- A hook is a way for us to do something as part of the regular React process. It lets us hook into the system and insert our own function

```
import {useState} from 'react';
```

How does useState work?

- We can define a stateful variable using this syntax:

```
let [variableName, setterMethod] = useState<datatype>(initialValue);
```

- For example, if we wanted to create a variable called status with an initial value o would write:

```
let [status, setStatus] = useState<string>("ok");
```

- This creates the variable status, which we can reference in our JSX as {status}
- We can change the value of the status variable by calling the setStatus method.
- A change to the value in this way **will** cause the component to be re-rendered, an we have set will be preserved.

How does useState work?

When the useState line of code is run, React does the following:

```
let [variableName, setterMethod] = useState<dataType>(initialValue);
```

- Create a variable called variableName
- Check to see if we already have this variable in our separate state storage outside of the component. If we do we set its value from there. If not, use value provided, and set this to be the value in the separate state area.

```
setterMethod(newValue);
```

When the setter method is called, React does the following:

- Update the value of this variable in the separate state area
- Re-render the component (re-evaluate the component function)

The rules for using hooks

- useState is an example of a React hook – we'll see more later on.
- Hooks are special functions which "hook into" React's component li
  methods and state management systems.
- Hooks can only be called inside functional components
- They can only be called at the top level – never in a function or a co

How does useState work?

Important:

Always change a stateful variable's va
by calling its setter method.

Never set its value directly.

Summary

- Component Events
- The need for State
- Stateful variables in React
- The useState hook

neueda

Communicating Betwee Components

Objectives

- Passing state to children
- Changing a part of a stateful object
- Changing a parent’s state

Passing state to children

- The value of a stateful variable can only be read or set within the co
  that it is defined.
- However the value can be passed to a child component through its
  properties. If the parent is re-rendered (the function is re-evaluated
  children will be re-rendered also.
- This means that if you pass a stateful variable to a child component
  its property, when the stateful variable's value changes, the child w
  "updated"

Changing part of an object state variable

- When we declare a stateful variable as a Javascript Object, we ofte want to change a part of the variable.
- Remember we can expand a Javascript object into its separate valu using the spread operator
- This allows us to more easy change a part of an object – the change come at the end of the new value

```
let [customer1, setCustomer1] = useState<Customer>(
{title: 'Mr', firstname: 'Simon', surname : 'Green', age: 36});

...

setCustomer1({...customer1, age : 37});
```

Changing a parent's state

- A component's properties are immutable (can't be changed)
- It is therefore not possible for a child component to directly change variable declared in a parent object and passed to it via its properti
- The only way to change a stateful variable is to call the setter meth isn't visible to the child by default

Changing a parent's state

Instead, for a child to change the state of a parent object, what we do is:

- Create a function in the parent object which will call the setter method

```
const changeSomething = (newValue : dataType) : void => {
    setVariable(newValue);
}
```

- Pass this function as a property to the child

```
<ChildComponent changeValue={changeSomething}/>
```

- The child can then call the function.

```
const changeValue = () : void => {
   props.changeValue ();
}
```

- This is functional programming – the properties of a component need not just be containing data, they could be variables which are pointers to functions

Defining functions as data types

When creating a type, you can specify that the data type for a parameter as a funct
this format:

```
type ComponentProps = { name : string,
  function1 : () => void,
  function2 : (a : string, b: number) => string
}
```

Be careful with the brackets

- Be careful when you bind a function, you should NOT include the br
- If you include the brackets this will EXECUTE the function when the component is RENDERED, even if that is within an event..

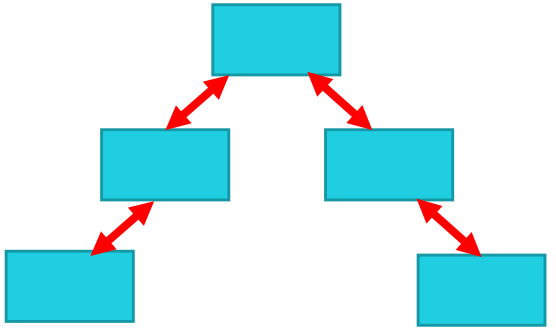
```
<ChildComponent changeValue={changeSomething()}/>
```

```
<ChildComponent changeValue={changeSomething}/>
```

```
<ChildComponent changeValue={ () => changeSomething(someValue)}/>
```

The State Triangle

- Stateful variables are only available in the component in which they declared and any child components to which they are passed via pr
- The “state triangle” can occur when you need to have access to stat variables in components that do not have a direct shared parent
- When this happens, create the stateful variable at the nearest com parent



Summary

- Passing state to children
- Changing a part of a stateful object
- Changing a parent’s state

neueda

Loops and Conditional Rendering

Objectives

- Loops in JSX
- Conditional rendering

Working with loops in JSX

When we wish to create components by looping through data in an a
there are a two rules to follow:

- JSX will render an array of well-formed elements
- Each entry must have a unique key

```
const cars = [
{make:'Fiat', rating: 3},
{make : 'Volvo', rating : 5},
{make : 'Ford', rating: 4}
];
```

```
const displaycars : JSX.Element[] = [
    <Car key="1" car={{make:'Fiat', rating:
    <Car key="2" car={{make:'Volvo', rating
    <Car key="3" car={{make:'Ford', rating:
    ];
```

```
return (<div>{displaycars}</div>);
```

Working with loops in JSX

There are two approaches to achieve this:

1 Create the array of elements as a property in regular JavaScri
before the returned JSX, then bind to the property within the

2 Within the returned JSX, use the map function to transform ea
element in the array to an array of well formed elements

Approach 1

- The first approach is to convert the array of elements and store the
  as an object, before the returned JSX, then bind to it within the JSX

```
const displayCars : JSX.Element[] =
cars.map ( (car, index) => {
    return (<Car key={index} car={car} />);
  }
);
```

Approach 2

- The second approach is to use functional syntax directly within the take care to get the brackets right!)

```
<div>
  {cars.map ( (car, index) => <Car key={index} car={car} /> )}
</div>
```

Loops and Fragments

- If your loop will create an array of JSX fragments, you must use the syntax for the Fragment (and provide the key!)

```
<div>
  {cars.map ( (car, index) => {
    return (
       <React.Fragment key={index}>
          <li>{car.make}</li>
          <li>{car.model}</li>
       </React.Fragment>
     )}
   )}
</div>
```

Conditional rendering

- We can use an if statement in the Javascript part of our code as you expect
- You can use an if statement within the JSX, but it's not good practic messy).
  The better options are:
  - The ternary statement
  - The logical && operator
- Using these methods means that the JSX elements will not be inser the DOM

Summary

- Loops in JSX
- Conditional rendering

neueda

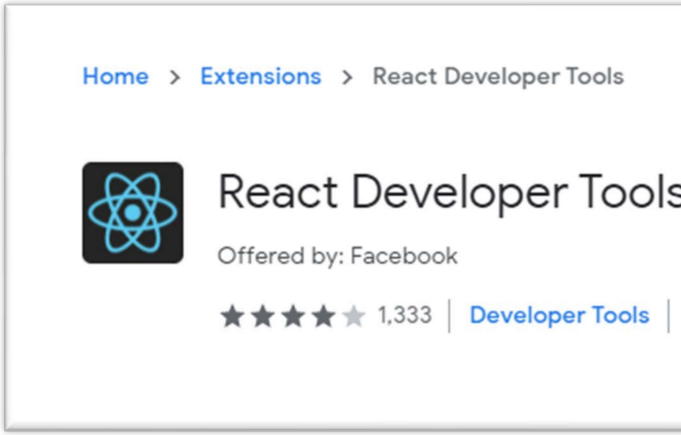
Debugging

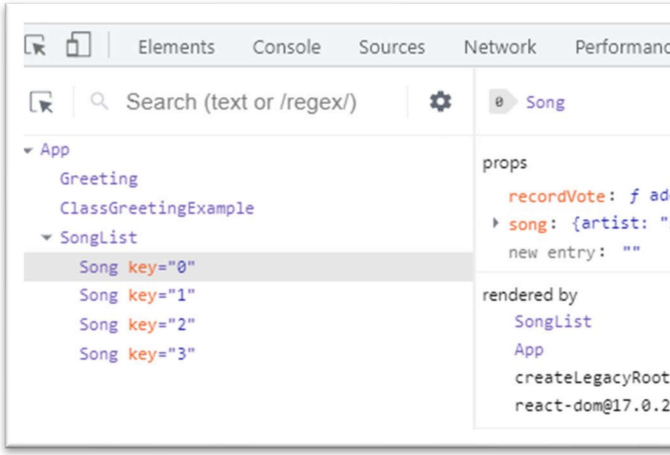
Objectives

- The react developer tools browser plugin
- Using breakpoints

React developer tools plugin

- There is a plugin for the Chrome and Firefox browsers called React Developer tools
- This tool creates an extra tab called components in the browser's developer tools window
- The components tab lists the components on the left hand side of the window. Clicking on a component shows you its properties on the right hand side. You can edit these properties if you wish.





ACTIVITY - React dev tools

- Install the react developer tools plugin if you don't have it al
- Open the developer tools window and find the components ta
- Navigate through the components and select one of the song
- Review its' properties
- See how voting for a song changes its properties

Using breakpoints

- To place a breakpoint in the code use the debugger keyword

```
debugger;
```

- This will make the browser go into debugging mode, **but only when dev tools are visible**
- In debugging mode you can:
  - Hover over variables to see their values
  - Step through the code
  - Add extra breakpoints
- The debugger command is not included in the final transpiled versi application you would create for deployment

ACTIVITY - breakpoints

- Create a breakpoint by placing the debugger keyword at the addVote function.
- Run the code and explore the debugger view.

Summary

- The react developer tools browser plugin
- Using breakpoints

neueda

Getting Ready for REST

Objectives

- What is a webservice?
- What is REST?
- Rest principles

Http verbs

- Before we start learning about webservices, there are 2 useful term understand, Verbs and Status codes.
- When you send a request to a server over HTTP, such as a request t web page, the request always includes a verb. The main verbs are:

  GET POST PUT DELETE OPTIONS

- In standard Html we use GET and POST.
- For example, GET is used when we visit a URL in a browser or click on a lir page.
- POST is often used when we send data to a server using a form.
- Technically the difference is that GET is **idempotent** - it won't change any the server, so it's safe to repeat. POST is not safe to repeat

Http Status codes

- When a server responds to a client over HTTP it will send back a sta
- You might have come across the idea of a 404 error page – 404 is a code.
- The most common status codes are:

200 (OK)	301 (URL has changed)	400 (bad r
401 (unauthorised)	(403) forbidden	404 (not fo
500 (server error)	(504) timeout	

- View the full list at https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

What is a webservice?

- A webservice is a way of communicating between systems using a s format (such as XML or JSON) over an internet protocol
- Clients can send an HttpRequest to a URL (the “endpoint”) , which w made up of:
  - An Http verb, such as GET, POST, PUT or DELETE
  - Headers, containing information such as security details
  - Optional Parameters contained within the URL
  - Optional data contained within the body of the request
- The server will send back an HttpResponse, consisting of a status code ar optionally some data.

SOAP

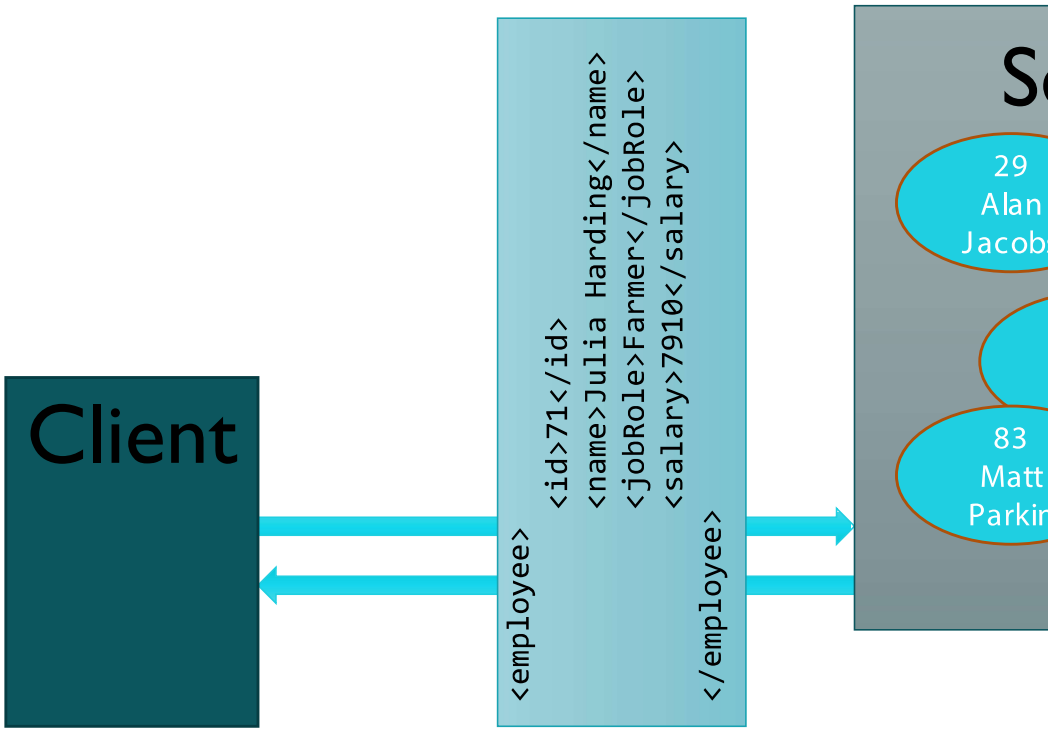
- SOAP was the first generally accepted webservice standard, origina
  released in 1998. It uses XML as the format for data. Many systems
  SOAP.
- One of the principles of SOAP is that for users of the webservice to
  what the different endpoints are, and how to use them, they first v
  known URL where they are provided with a list of options – this is in
  known as WSDL (Web Service Definition Language).
- SOAP requests are always sent as POSTs and because they use XML
  data mechanism are often unwieldly

REST

- Most modern systems now use REST instead of SOAP
- The principles of REST was developed by Roy Fielding and publishe doctoral thesis in 2000. The thesis contained a number of best prac were followed in designing the World Wide Web.
- Roy's thesis doesn't set out to design what a good web service shou like – he is trying to define what a good architecture of any distribu system should be.
- It has however become the most successful network architecture o
- Rest is based on 4 key principles, but to understand them we first n know what a resource and a representation is.

Resources and representations

- A key concept in REST is that of the resource.
- A resource is something that the server cares about – normally this will be our er
- Representations are a serialised form of a resource. – we convert our resources into a format that can be sent across the internet such as XML or JSON – this is a representation of the resource.
- Rest stands for REpresentational State Transfer – the transferring of the state of our representations from one system to another.



Principle 1 : Unique identifiers

- The first key feature of a REST based architecture is that we're goi unique identifiers for each of our resources
- The identifier is the resource's URL (or more correctly its' URI)
- For example, Employee Julia Harding, with employee number 71 m URL of http://myserver.com/webservice/employee/71. The URI, the resource's unique identifier is employee/71
- This is a very significant idea, and it is very different from SOAP – if expose the entities in our system to the outside world, each entity own URL.

Principle 2 : resources are manipulated through represent

- The second key feature of a REST based architecture is that when w send the content of a resource to a client, the client receives a repr of this resource.
- This representation should give the client sufficient information to change the resource's state.
- For example for a client to update an employee's name (and theref change its state), the representation of the employee resource mus sufficient information to allow the client to request that change.

Principle 3 : messages should be self-desc

- The third key feature of a REST based architecture is that messages include information about how to process its data.
- This simply means that here should be some meta-data attached to message (an HTTP header) that tells us the format that the messag such as JSON or XML.
- We sometimes describe this as “content negotiation” - the agreeme between the client and server of the format to be used.

Principle 4 : HATEOAS

- The fourth key feature of a REST based architecture is that a messa
include links to other URLs, to tell the client what else it might wan
- For example if a request to http://myserver.com/webservice/emplo
returns a list of all the employees, it should also include informatio
the URLs for each employee, or the URL to be used to add a new em
- HATEOAS stands for Hypermedia as the Engine of Application State
- Very few REST APIs actually implement Hateoas!

REST example

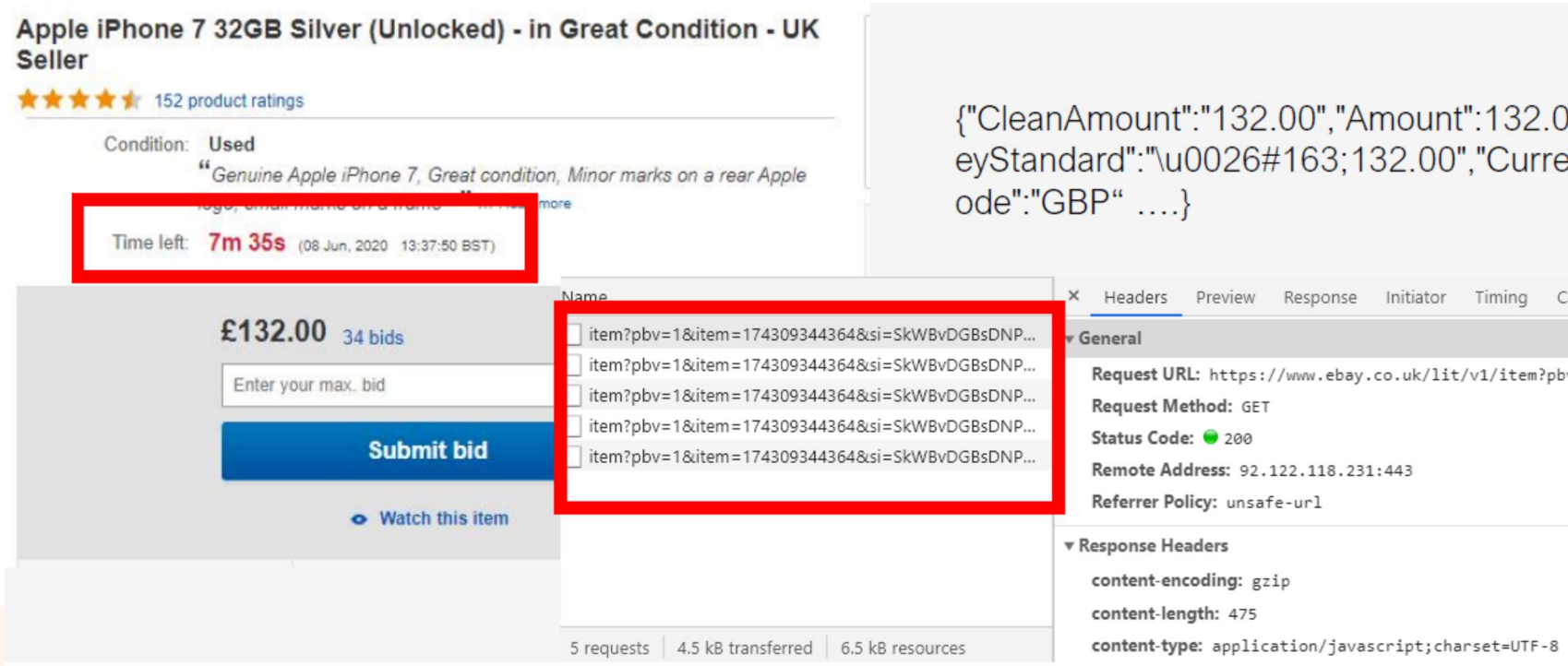
- A good example of a REST api is that provided by Mailchimp - the d
  be found at: https://mailchimp.com/developer/marketing/api/
- Mailchimp is an email marketing platform. A user of Mailchimp will
  mailing lists, each containing subscribers, and can send email messa
  their lists (called campaigns). The key resources are mailing list, sub
  and campaign.
- This is a good example as they have mostly followed the REST rules
  URLs are logical.

Designing a rest api

- Use verbs appropriately. GET to retrieve resources, POST to add res
  PUSH to update resources, DELETE to delete resources
- Use logical URIs…
  - GET /api/employee should retrieve a list of all employee re
  - GET /api/employee/7 should retrieve the employee resource
  - GET /api/employee/7/contract should retrieve the list of contracts fo
  - GET /api/employee?name=Smith should retrieve all employees with a na
  - POST /api/employee should add a new employee
  - PUT /api/employee/7 should update employee no 7
- Return appropriate status codes to provide a meaningful response

REST in action

- EBAY use rest to provide an update to the browser on the current s auction



Accessing rest as a client

- To test a rest server we need to have a client which can make the re
- You can only do GET requests from a browser’s URL bar
- If you have the Ultimate version of IntellIJ you can create rest files.
- You can create rest requests with Javascript and run them in the br
- Most developers use the command line tool CURL.

Activity – Access a REST API

- Visit the website: https://payments.multicode.uk in your browser (o
  run this project locally – see the resources/payments folder)
- This is a project that provides a sample rest API.
- Explore the API
- Open the curlGenerator.html file in the resources/curl generator fo
  try to create a rest request to the payments server.
- View the Curl command that has been generated and try running th
  the command line

Note that this project has a couple of features in that are not part of th
and the javascript in the html page is also not part of this course. It has
necessarily been written to production standards or tested well!

Summary

- What is a webservice?
- What is REST?
- Rest principles

neueda

Making REST calls

Objectives

- Fetch
- Promises
- useEffect
- Axios
- Error handling
- Using Environment variables

The fetch function

- Javascript contains a function called fetch – it's syntax is:

```
fetch(url, {method: "POST", headers: headers, body: body})
```

- If you are doing a GET and there are no additional attributes, the 2nd parameter can be removed.
- This function returns a Promise<Response> – it will return immedia execution immediately (it’s a non-blocking function) but the data al certainly won't be there at this time.
- We can use the .then() method of the promise, to execute code wh data arrives from the server.
- The promise will return a Response object. We need to extract its' c calling the .json() method… and this again returns a promise.

Using fetch

- Because getting data is a 2-step process, which can take some time
  experience we want to create is
  - Load the component, with a placeholder for the data, so that the user
    something straight away.
  - Indicate to the user that the data is loading (eg with a spinner)
  - Show the data when it is received.
- There are alternative ways of doing a fetch, which would avoid pro
  using async / await, but we'll stick to promises for now.

Using fetch

- To achieve this what we will do is:
  - Create a stateful variable in the component which can store the current stat (whether it is loading or not)
  - Use this stateful variable to determine what gets shown on screen (a spinner
  - Run the fetch and json methods, and when both of these are complete, chan loading variable's value
- It's not quite this straightforward…

Use effect

- The code we have just created isn't working because
  - The component renders and as part of the initial render calls the getAllPayments
  - When the setPayments() line is called, this changes state…
  - Which makes the component re-render, so the getAllPayments method is called
  - And we're in a loop!
- There is a react hook called useEffect which can be used to create c runs once only, the first time that a component is mounted.

```
useEffect( () => loadData(),[]);
```

- The second parameter of the useEffect hook can either be an empt this means run once only, or it can contain a list of stateful variable the variables in this list changes, the code block will re-run. If you d include the 2nd parameter then this code will run on every re-rende component.

Axios

- Some Rest developers choose to use an alternative to the built in fe
party library called axios.
- With axios, the retrieval and conversion to JSON are done as a single pro
code becomes simpler.
- The syntax for the method call is slightly different – it only takes a single

```
fetch(url, {method: "POST", headers: headers, body: body})
```

```
axios<data_return_type>({url: url, method: "POST", headers: headers, body: body
```

- The object returned from the promise (a Promise<AxiosResponse<data_t
contains a .data() method that contains the json data… this is not a prom

Error handling promises

- Promises might not always work. The server might not respond, or might not be valid JSON.
- The error handling concept is the same whether we use Fetch or Ax it's easier to do it in Axios as there's only one promise!
- After the .then() method on a promise, you can add a .catch() metho optionally a .finally() method.
- You can also check the status of your response (did you get a 200 st code?) within the then method.

```
.catch(error => {
    console.log("something went wrong", error);
});
```

Using environment variables

- We will often have a different url for development and production
- A good way to deal with this is to use environment variables. On th production server we define the URL of the server to be used…

```
let serverURL = "http://localhost:8080";

if (process.env.APP_SERVER_URL) {
    serverURL = process.env.APP_SERVER_URL;
}
```

Summary

- Fetch
- Promises
- useEffect
- Axios
- Error handling
- Using Environment variables

neueda

Forms

Objectives

- HTML or the Virtual DOM?
- Why we need state to use forms
- Creating forms
- Form validation

Html vs Virtual Dom

- When we create a form object (such as an <input>) in a component, created in the virtual dom, and the virtual dom is then rendered to
- We code against the virtual dom and when the component is re-ren screen is synchronised
- This means that HTML form elements work in a very different way i what you might expect
- Suppose you have an input and the user has typed a word in… how find out what the word is? We can only interact with the virtual DO word is in the HTML representation on screen… we can't access tha

State to the rescue

- Because we can't access the HTML, only the virtual DOM, what we n is:
  - Store the value of each form element in a stateful variable
  - Whenever the value changes on the screen, fire an event that will upd stateful variable.
  - When the form is then submitted, we can access the stateful variable the value of the element
  - For this to work, we also need to bind the display value of the input to stateful variable, or it will get lost when the form re-renders

Using forms

- Our simple example doesn't actually use a form – it's just an input a button
- If we do use a form we have to stop the form actually being submit server – for this we must call event.preventDefault() in the method run when the form is submitted…

```
const handleSubmit = (event : FormEvent<HTMLFormElement>) => {
    event.preventDefault();
     ...
}
```

Form validation

- We can validate the form in the submit handler method that we wri
  even in the onChange handler.
- You can use stateful variables to display meaningful errors to the u

Summary

- HTML or the Virtual DOM?
- Why we need state to use forms
- Creating forms
- Form validation

neueda

The reducer function

Objectives

- useReducer
- Reducer functions
- The dispatch function

Use Reducer

- When you have a lot of state variables in a single component, thing
  bit messy… it's possible to combine some of them together into a s
  object, but you need a special hook to do this, called useReducer.

ACTIVITY - Getting ready

- To give us an opportunity to see how reducer functions work, w
  create a form to allow us to add in a new transaction.
- As we don't yet know how to separate these into pages (that's
  soon) we'll just put it below the current page.
- Create a new component called AddTransaction and wrap it i
  AddTransactionPage component.
- Within this component return the required form – a sample fo
  provided in the resources folder.

Reducer functions

- We are going to work with a stateful variable which is an object, e.g

```
{name: "Matt", age: 21, country: "UK", style: "not trendy" }
```

- We first need to define a reducer function. The purpose of this func adjust the stateful variable – we will be giving the function, for exar data `{country : "FR"}`, and we want the reducer function to change jus of the stateful variable.
- This is a boilerplate reducer function which you can copy and paste!

```
const reducer = (state : dataType, data : { field: string; value: any }) => {
    return {...state, [data.field]: data.value}
}
```

Use reducer

- Instead of useState, we'll use a different hook called useReducer.
- The syntax for this is

```
const [variable, dispatch] = useReducer(reducer, initialState);
```

- This is similar to the useState function – the differences are:
  - We normally call the equivalent to the setter function "dispatch" – tha method we'll use when we want to change a value.
  - There's an extra parameter needed when we create the variables – th function we just created.
- You may get a warning that useReducer expects a 3rd parameter – t optional and you can ignore that warning

The dispatch function

- When we want to change part of the state, we need to call the disp function.
- This function expects 1 parameter, which is the new data. Its syntax

```
dispatch( {field : fieldName, value : newValue});
```

- When we call dispatch, react uses our reducer function to update th variable – we don’t call our reducer function directly.

Summary

- useReducer
- Reducer functions
- The dispatch function

neueda

Deploying

Objectives

- Building the application

Building the application

- So far we have been running our application in development mode, server (using **npm start**)
- When we are ready to build the application into a set of standalone use the command **npm run build**
- This will create the files in a folder called build, ready to be placed o server.
- The files can't just be opened in the browser directly from the file s they do need to be placed on a web server. You can install a web se this out with **npm install –g serve**. Then to run the application run th command **serve –s build**

Summary

- Building the application

neueda

Routing

Objectives

- The need for routing
- How routing works
- Routing concepts
- Links
- 404 pages
- Parameters & query strings
- Surviving a Browser Refresh

The need for routing

- React applications are described as a "single page" application – thi
  that there is only 1 HTML file, everything is created in Javascript.
- There are no round-trips to the server to generate new pages as we
  through the application
- We will however want to give the impression of navigating through
  the user – eg via a menu
- It can also be helpful to show the user meaningful URLs, such as
  myserver/myApp/customers?id=7
- For this we need to implement Routing. This is not part of the defa
  install – it needs to be added with:

```
npm install react-router-dom
```

The way routing works

The way routing works in React is:

- We will define Routes (URLs) and specify which components will be within each Route
- We will define Links to allow the user to navigate to the different U equivalent to the HTML anchor tag)
- We will need to ensure that as the components in each route are lo check the URL for any parameters, so that any data needed is loade appropriately if, for example, the URL was bookmarked

Routing Concepts

To implementing routing, we need to use 4 components:

BrowserRouter

This is the top level component. All the other routing components must be within a sin
We will normally define this at the App component level.

Route

This defines each URL, and its children will be the components that are rendered when
active

Routes

This acts as a wrapper for each of the Routes. The first matching route within the switc
given URL will be used.

(in older versions of React this object was called Switch)

Link

This is the replacement for the anchor tag – we'll use it to navigate to URLs which will b
on the client, without a visit to the server

V6 syntax changes

Before React-dom 6

```
<BrowserRouter>
  <Switch>
    <Route path="/p1">
      <Component1/>
    </Route>
    <Route path="/p2">
      <Component2 varibale={value}/>
    </Route>
    <Route path="/p3:/id">
      < Component3 />
    </Route>
    <Route path={["/p4","/p5"]} >
       < Component4 />
    </Route>
  </Switch>
</BrowserRouter>
```

React-dom 6+

```
<BrowserRouter>
  <Routes>
    <Route path="/p1" element={<Component1/>}
    <Route path="/p2"
        element ={<Component2 variable={value
    <Route path="/p3/:id" element ={<Componer
    <Route path="/p4" element ={<Component4/>
    <Route path="/p5" element ={<Component4/>
  </Routes>
</BrowserRouter>
```

Links

- Instead of anchor tags, we use the Link component – this prevents
  trip to the server taking place.
- The link component takes the target URL as a parameter called to

```
<Link to="newURL">click me</Link>
```

- An alternative is **NavLink** – this works like link but will automatically
  class called **active** added to it when the link is active (for CSS styling

```
<NavLink to="newURL">click me</Link>
```

404 pages

If we want to ensure that if the user attempts to visit a URL we haven
they are shown a 404 page (page not found), we need to:

- Create a component to display when there is no matching route
- Create an extra route with no path and display the page not found
  component in this route.
- Ensure this is the final route in the routes list

Url parameters

- As the users use the application we will want to consider appending parameters to a URL
- For example we might want to allow users to bookmarks URLs like:
  /customer/172
- We can change the url that is shown int the browser by using the **us** hook (see next slide)
- In case the user refreshes the page, we need to read in the URL wh page loads and check if there's a parameter there…
- We can read in parameter values from a URL using the **useParams** h

V6 syntax changes

Before React-dom 6

```
const history =useHistory();

...

history.push("/newURL");
```

React-dom 6+

```
const navigate =useNavigate();

...

navigate("/newURL");
```

Query strings

- As the users use the application we will also want to consider embe
  query strings to the URL.
- For example we might want to allow users to bookmarks URLs like:
  /customer?name=smith
- To achieve this we need to: (1) get the right URL into the browser, a
  (2) load the data specified in the query string.
- We move to the correct URL using the **useNavigate** hook
- We can read in parameter values the query string using the **useSear**
  hook. (for older versions of React-dom this was called URLSearchPa

Surviving a refresh

- One of the challenges when working with a single page framework is how do we survive if the user clicks on the refresh icon on their b
- By careful planning of URLs (which we now have) we can survive a r
- Note also that the user can click the back button on the browser an work too.
- Unfortunately if there is data that the browser needs to remember be put into the URL, e.g. a user's authroisation data, then this is a b complicated, and outside the scope of this course!

Summary

- The need for routing
- How routing works
- Routing concepts
- Links
- 404 pages
- Parameters & query strings
- Surviving a Browser Refresh

neueda

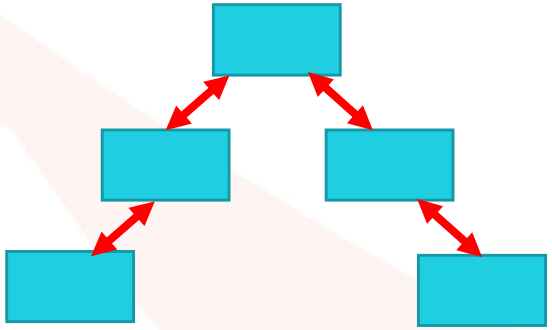
useContext

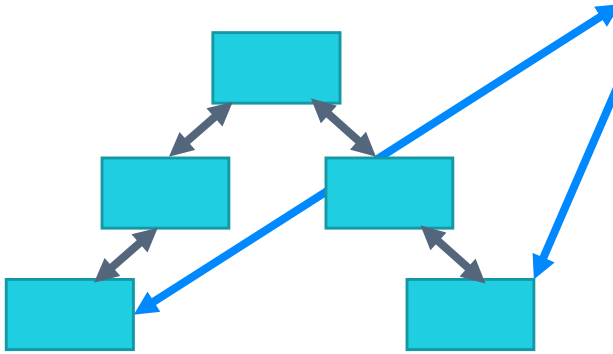
Objectives

- The need for global state
- The structure of a context object
- Creating the context object
- Providing the context object
- Using the context object

The need for global state

- Stateful variables are only available in the component in which they declared and any child components to which they are passed via pr
- Sometimes it is helpful to have stateful variable that is available th our application
- Examples include:
  - Current user information
  - Cached data fetched via REST
  - Time spent using the site





The structure of a context object

- React lets us define a javascript object, which can be accessed anyw within the application.
- It will use stateful variables that are defined within the App compo will make them available throught the application.
- This object needs to include:
  - Data
  - Methods to update its data

```
{value1: "", value2: 0,  updateValue1 : () => {}, updateValue2 : () => {
```

Creating the context object

- The createContext function is used to create the context obje
- We provide as a parameter the default value* of the object
- The context object then is exported

```
export const MyContext = createContext({...});
```

* This value will always be overridden, so it's better to think of tl
being the structure of the object

Providing the context object

- Within the App component we need to define stateful variables and functions that will be used within the context object
- We then use the Provider of the Context object that we created to these functions to the context

```
const [var1, setVar1] = useState("Hello world");
const [var2, setVar2] = useState(0);

return (
<MyContext.Provider value={{value1: var1, value2 : var2, updateValue1 : setVar1, updateValue2:
          <OtherComponents />
</MyContext.Provider>
```

Using the Context Object

- In any component that is used within the Provider component, we c
  useContext function to get a reference to the context object
- This hook requires a function that takes the state object, and extra
  part of the object we are interested in

```
const myContext = useContext(MyContext);

return (<p>Hello {{myContext.var1}}</p>)
```

Activity- create a user login context

Create a context object to store the currently logged in user:

- Create and export a UserContext object containing the fields:
  - id (number)
  - name (string)
  - role (string)
  - login (function)
  - logout (function)
- In the app component, create a stateful variable to store this c define the functions

```
const [user,setUser] = useState({id: 0, name : "", role : ""})

   const login = setUser;

   const logout = () => {
       setUser({id: 0, name : "", role : ""});
   }
```

Activity- create a user login context (conto

- Provide the UserContext object at the top of the App componer
- Create a Login component to simulate a user logging in
- Add the login page to the routes
- In the Menu component, display a login link if the user id is 0.
- In the PageHeader display the current logged in user's name + button if the user id is not 0.

Summary

- The need for global state
- The structure of a context object
- Creating the context object
- Providing the context object
- Using the context object

neueda

useRef

Interracting with the DOM

- The virtual in-memory DOM and the browser's DOM are not connec
- Sometimes we need to interact with the browser's DOM, e.g:
  - Set the focus to a specific element
  - Dynamically add an event listener to an element
- For this we use the useRef hook

```
const myInput = useRef<HTMLInputElement | null>(null)
```

```
<input type="text" ref={myInput} />
```

- This hook has other uses (mutable stateful data that doesn't trigge component re-render)

neueda

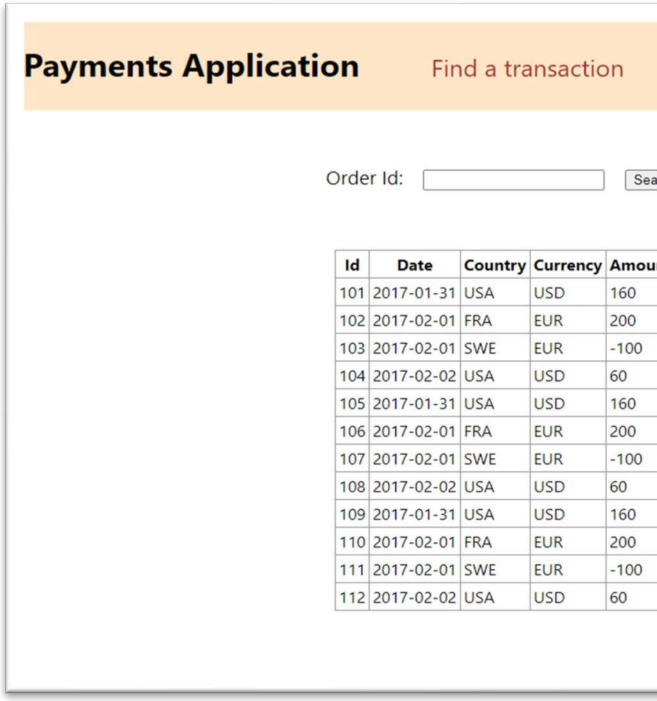
End of day exercises

Day 1

- Create a new project called payments-ui. This will be the front-en
  connect to the payments database rest application.
- Create a set of components that will come together to form the u
  interface for the application. This should contain:
  - The application name at the top of the page
  - A menu with the options: “find a transaction” and “new transactio
  - A search box which would ask for an order_id
  - A List of all the transactions which are found from the search
- Ensure that there are no errors when the code is run (check the co

Day 1

- Here is an example of what you might build.
- If you are not confident with css, do not worry about the design aspect.
- For now – just use dummy data to build up the interface (3 or 4 lines of data is sufficient).
- You will probably adjust what we build as we progress through the course, so don't worry about accuracy – the key purpose of this exercise is to create some different components, and then place them on the page.



Id	Date	Country	Currency	Amou
101	2017-01-31	USA	USD	160
102	2017-02-01	FRA	EUR	200
103	2017-02-01	SWE	EUR	-100
104	2017-02-02	USA	USD	60
105	2017-01-31	USA	USD	160
106	2017-02-01	FRA	EUR	200
107	2017-02-01	SWE	EUR	-100
108	2017-02-02	USA	USD	60
109	2017-01-31	USA	USD	160
110	2017-02-01	FRA	EUR	200
111	2017-02-01	SWE	EUR	-100
112	2017-02-02	USA	USD	60

Day 2

Part 1 - looping

- Import the provided DataFunctions.ts – this is a simple Javascript
(not a component!) which is designed to simulate the REST conne
build later on. We suggest you put it in a folder called data.
- Amend the table you created so that it will:
  - Include an extra field called orderId
  - Have content generated by calling the function in the dataFunction
- You should do this by creating a new component called PaymentT
This component which will take a PaymentType object as a param
it will generate a complete table row (ie a <tr> </tr> element).
- Ensure that there are no errors when the code is run (check the co

Day 2

Part 2 – looping

- Create a dropdown filter which lists all the countries in the payments lis
  need to extract the countries from the list of all payments, and then ma
  unique).

  Hint – the following are 2 examples of how to remove duplicates from a
  called countries:

```
const uniqueCountries : string[] = countries.filter((country, index) => countries.indexOf(coun
```

```
const uniqueCountries : string[] = Array.from(new Set(countries));
```

Day 2

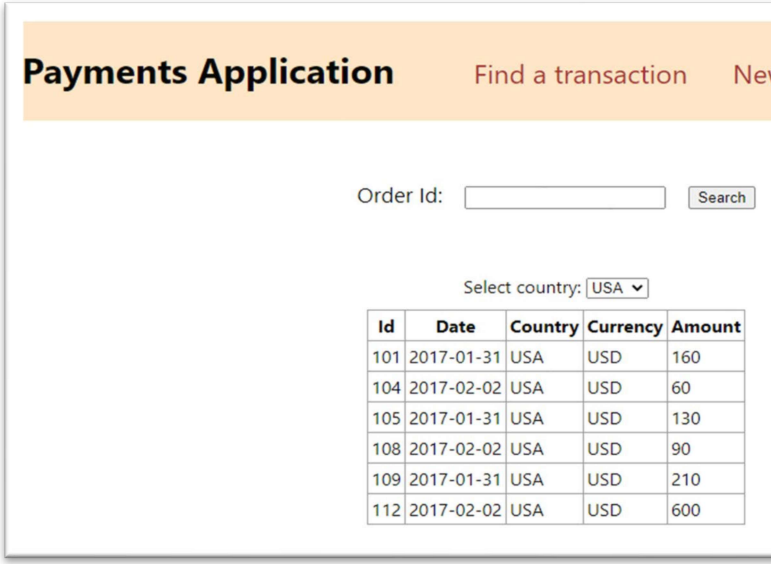
Part 3 – conditions

- Based on the value of the drop down you created, change the list of pa that only payments from the matching country are shown. To do this yo to:
  - Create a stateful variable which will store the currently selected cou
  - When the onChange event runs, use this to update the selected cou
- Hint – when you bind a function to an event like click or chan can map to a function which takes an object of type ClickEven ChangeEvent. For example:

```
const changeCountry = (e : ChangeEvent<HTMLSelectElement>) => {
       const option = e.target.options.selectedIndex;
       console.log("the item selected was "+ option);
   }
```

Day 2

- Use the value of selected country to determine
whether or not to show the payment line.
- Ensure that there are no errors when the code
is run (check the console).



Id	Date	Country	Currency	Amount
101	2017-01-31	USA	USD	160
104	2017-02-02	USA	USD	60
105	2017-01-31	USA	USD	130
108	2017-02-02	USA	USD	90
109	2017-01-31	USA	USD	210
112	2017-02-02	USA	USD	600

Day 3

- We now have a working application and have implemented the various features. You to tidy everything up:
- Use the routing feature to link to the new transaction page (the Add Transaction pa
- Review and organise the components into a more sensible file structure.
- Test every aspect of the application and make sure it works as expected- pay atten things like browser refreshses