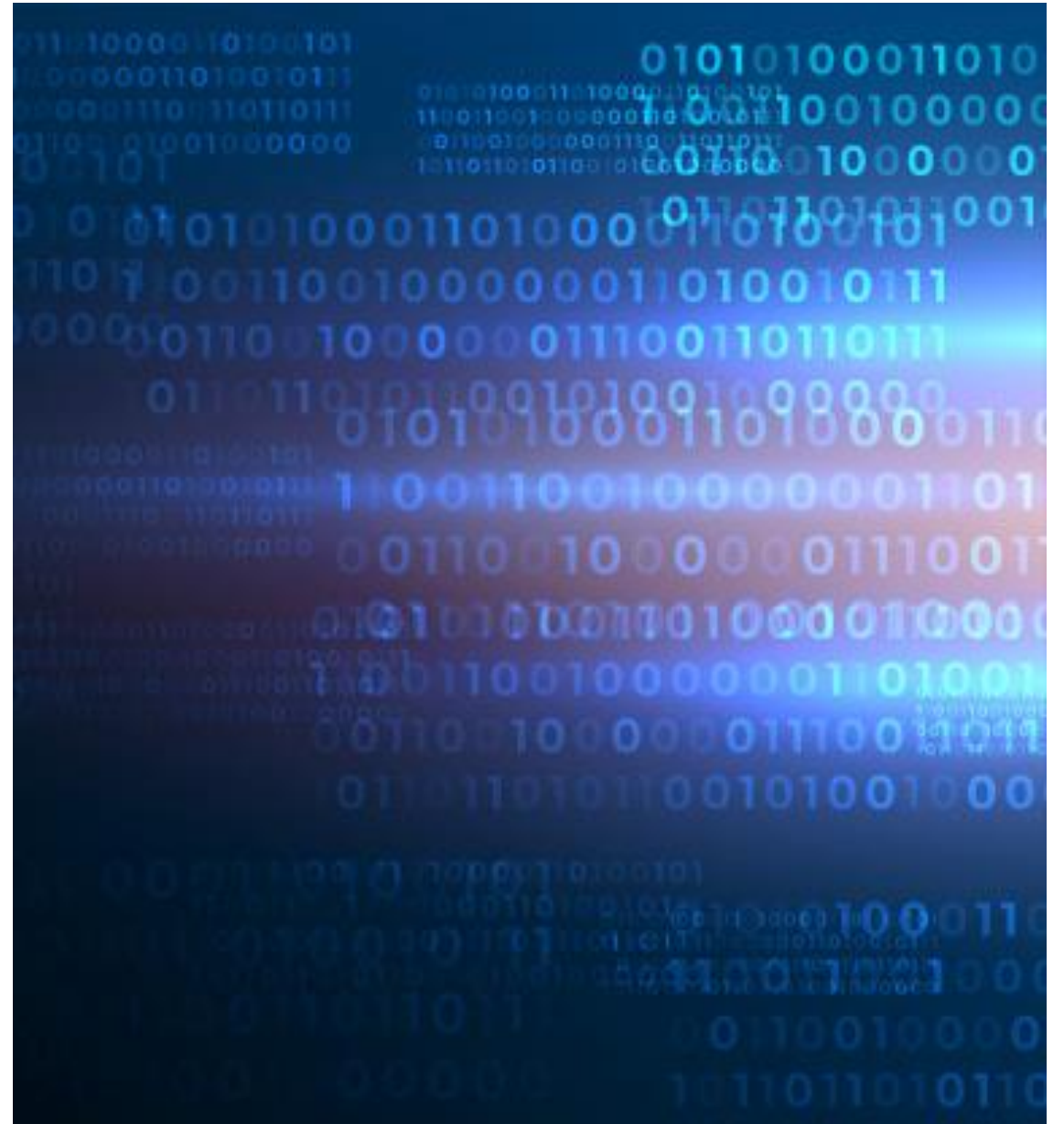


# WHAT IS REACTJS?

00\_01



# AGENDA

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

# ACTIVITY – CREATE A REACT APP

- Go to your workspace
- Run the command

```
npx create-react-app hello-world
```

# WHAT IS REACT?

- React is a set of 3<sup>rd</sup> party open source Javascript libraries that help us to build 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"

# 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 provides the full user interface.
- These can provide a better user experience as we avoid the server round trip.
- In React, the page we get from the browser is very simple... the complexity is in the Javascript
- Building an application like this from scratch would be very difficult + time consuming. React makes it easier.



# TRANSPILING

- We will create code in React which can contain other libraries, custom complex, and be structured over lots of files.
- When we are ready to build the application, the react transpiler converts our code to plain Javascript that the browser can understand
- This minimises the code size, and ensures that the browser has everything it 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?

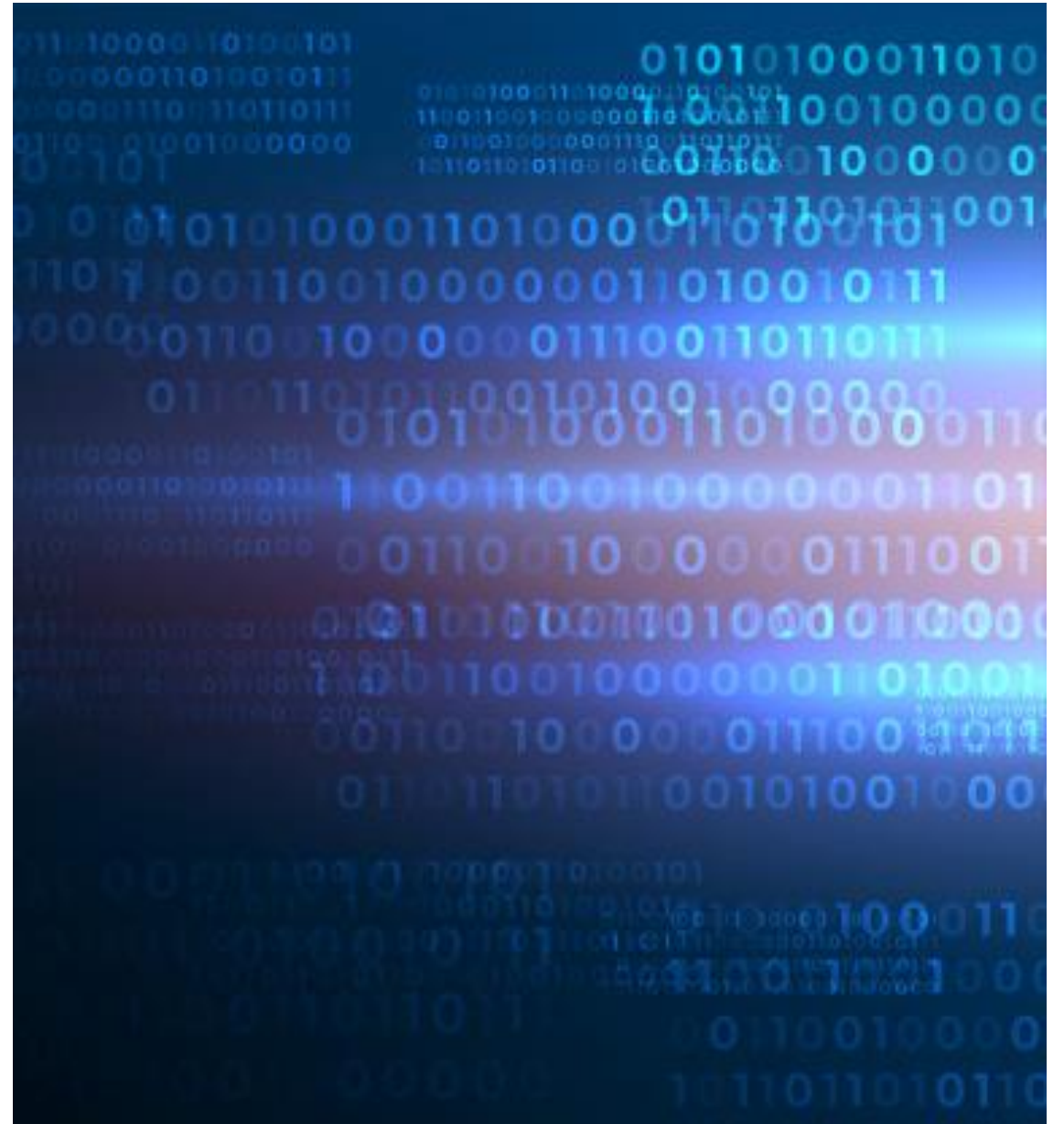
TLDR: Use `npx create-react-app app-name` to create a new application, and `npm start` to run it.

- To create and build a react application we need a set of tools that are written in Node.js.
- Node.js comes with a package manager (npm) and we used that to start the application (`npm start`) – that is to spin up a web server containing our code
- To create the new application we used `npx` – this is another tool that comes with Node.js and it's used to execute packages.
- It works best if you create the application outside of an IDE but you can do it within.



# MODERN JAVASCRIPT

01\_01



# AGENDA

- Basic HTML, CSS + Javascript
- let & const
- Arrow functions
- Template literals
- Spread operator
- Destructuring
- Imports and exports

# BASIC HTML, JS & JAVASCRIPT

For this course you should be able to:

- Construct a basic HTML page, containing elements such as div, span, p, input, button
- Apply basic styling using CSS rules
- Create a Javascript function that can find elements on the page and read / write their values

# LET & CONST

- let and const are replacements for var
- const = immutable variable
- let = mutable variable
- When working in functional programming, variables should be immutable, so we should use const to declare all variables unless we have a specific need to make it mutable within a limited scope.

# ARROW FUNCTIONS

- This is an alternative syntax for the Javascript function.... Using it we write less code!
- If you have one parameter only you can omit the brackets around the parameters (like Java)
- If you are returning a value and it is a single line you can use the short form version (like Java)

```
function sayHello(firstname,surname) {  
    alert('Hello ' + firstname + ' ' + surname);  
}
```

```
const sayHello = (firstname, surname) => {  
    alert('Hello ' + firstname + ' ' + surname);  
}
```

```
const sayHello = (firstname, surname) => alert('Hello ' + firstname + ' ' + surname)
```

# TEMPLATE LITERALS

- You can embed a variable or expression within a string if you declare the string using backticks.
- The variable / expression goes inside `${ }`

```
content.innerHTML = `Hello ${firstname} ${surname}`;
```

# SPREAD OPERATOR

- The spread operator will split an array into a list of its elements, or will pull properties out of an object.
- This is useful if you want to add to an array

```
const smallArray = [1,2,3];  
const largerArray = [...smallArray, 4, 5];
```

- Or change part of an object

```
let customer = {id:3, name:"Matt", totalSpend: 331.22, active: true};  
Customer = {...customer, totalSpend: 388.19};
```

# DESTRUCTURING

- Destructuring lets you assign the values of an array to variables

```
const smallArray = [1,2,3];  
const [one,two,three] = [smallArray];
```

- or to extract parts of an object as variables

```
const customer={name:"matt", age:21, active:true};  
const {age, name} = customer;
```



# IMPORTS AND EXPORTS

- In modern Javascript projects we can split our code across multiple files (called modules)
- To make a function available in a different module you must export it
- There are named exports, and a default export

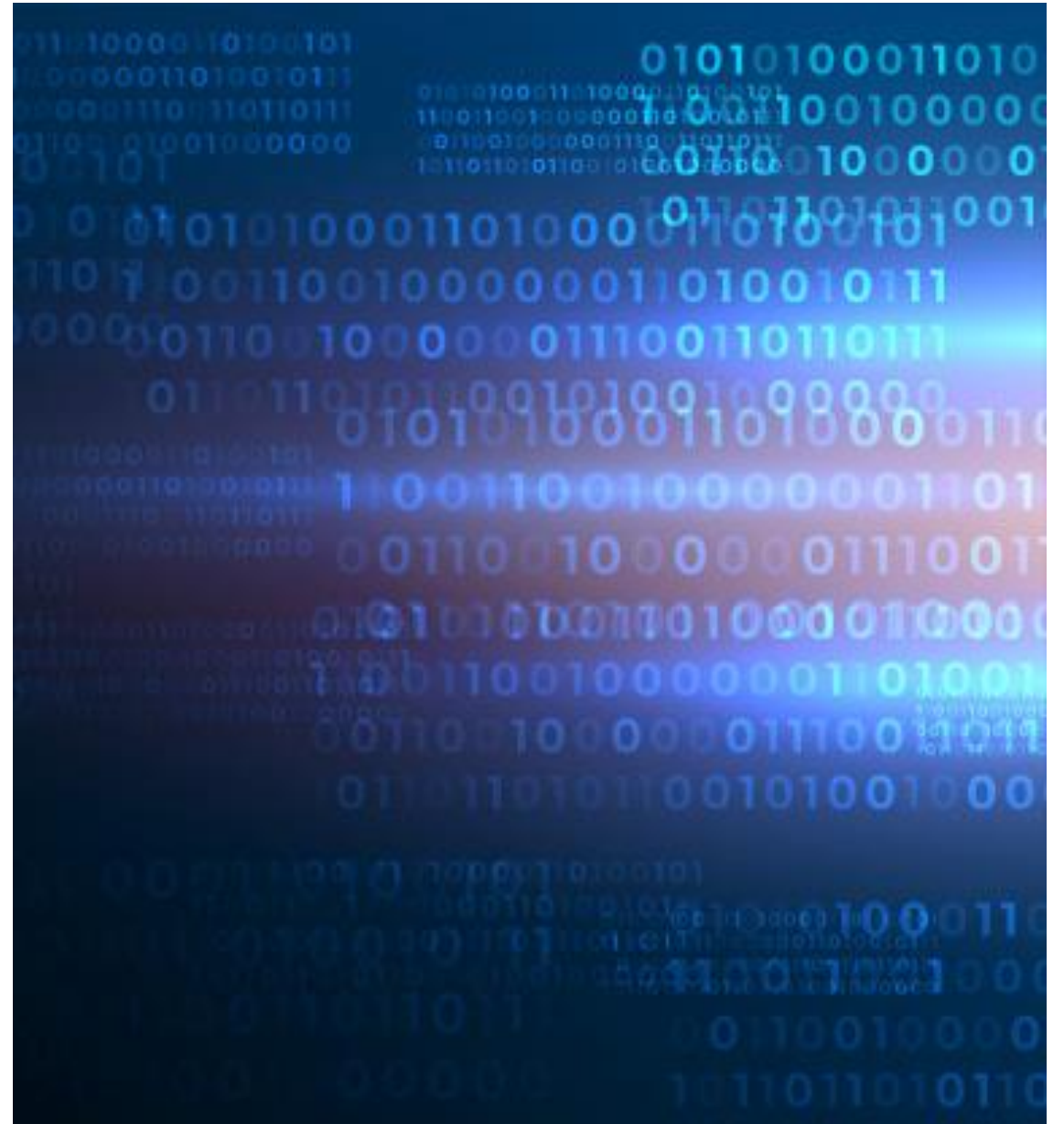
```
export const addingUp = (a,b) => a+b;  
const subtracting = (a,b) => a-b;  
export default subtracting;
```

- To use a function in a different module you must import it:

```
import subtracting, {addingUp} from './otherFile';
```

# ELEMENTS & JSX

01\_02



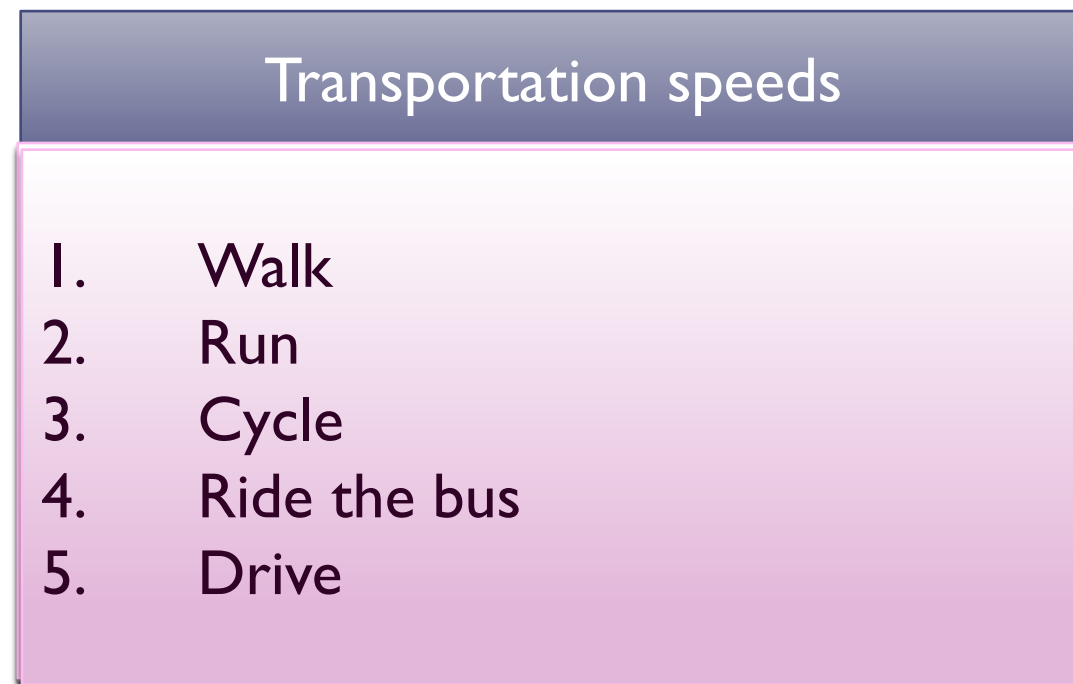
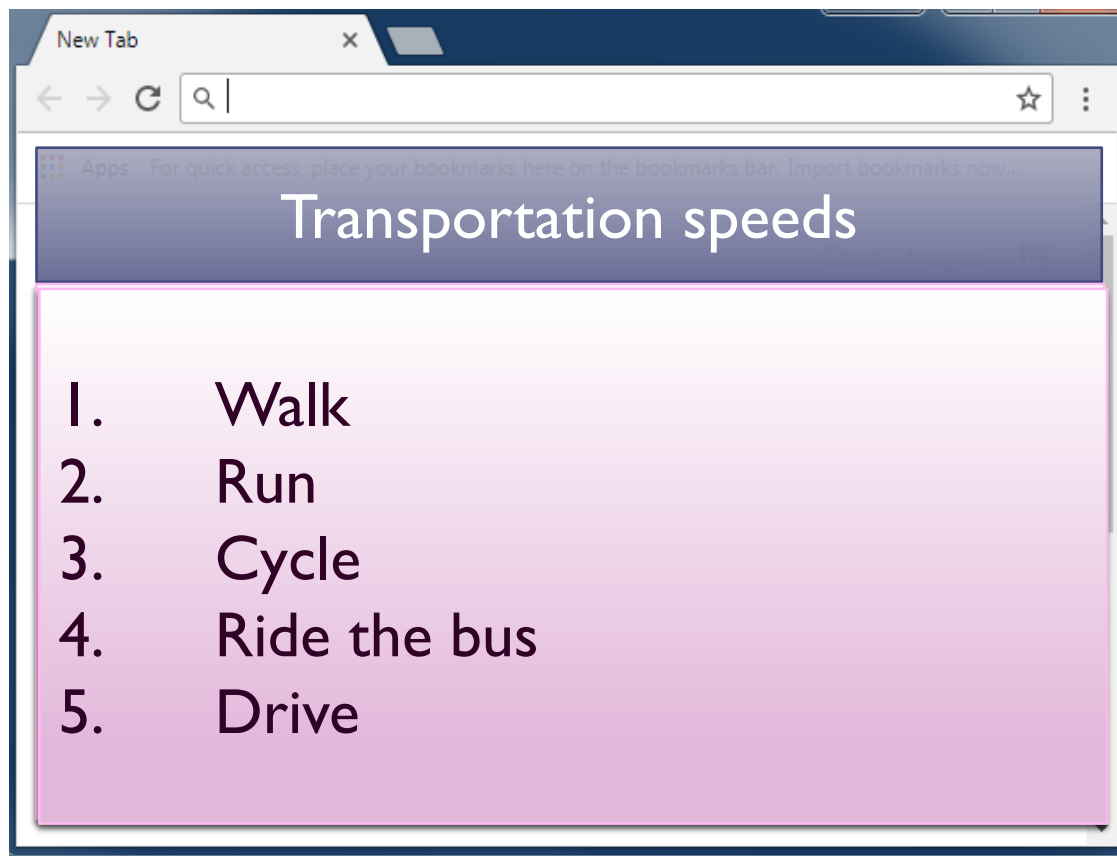
# AGENDA

- The Virtual DOM
- Elements
- JSX

# THE VIRTUAL DOM

- React keeps an in memory version of the DOM called the Virtual Dom
- The browser's actual Dom and the virtual Dom are then synchronised
- 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 works first, and then we'll introduce a much easier way to code this

# THE VIRTUAL DOM



# ELEMENTS

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

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

Object type

Parameters

Content

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

```
ReactDOM.render(myPara, document.getElementById("root"));
```

## ACTIVITY - ELEMENTS

- In the index.js file, comment out the existing ReactDOM.render method
  - Create an element called p1, which will be an HTML paragraph, containing the text "this is paragraph 1". It won't have any properties
  - Render the paragraph in the virtual dom
  - View the HTML page to check it has worked
- (to run the application use "npm start")

# PROPERTIES

- The middle parameter of the createElement method is used to define HTML tag's attributes.
- These are provided as a java script object

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



Properties



Child



# CHILD ELEMENTS

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

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

↑  
Parent

↑  
Properties

↑  
Child

- You can provide an array of children elements.
- The impact of this is we must define the children before we can define the parents
- NOTE: the render method can only render a single parent object.

## ACTIVITY - ELEMENTS

- Create a second element called p2, which will be an HTML paragraph, containing the text "this is paragraph 2". It won't have any properties
- Create a third element, which will be a button, give it a class of "myButton"
- Render the 2 paragraphs and the button in the virtual dom
- View the HTML page to check it has worked

# CHILD ELEMENTS

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

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

# JSX

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

JSX →

```
const myPara = <p>this is paragraph 1</p>;
```

React →

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

# JSX

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

```
const myInput = <input id="something" />
```

- Remember that the ReactDOM.render method can only take a single object, 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 style attribute, we need to provide these as a javascript object, but then we 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 expression by using curly brackets

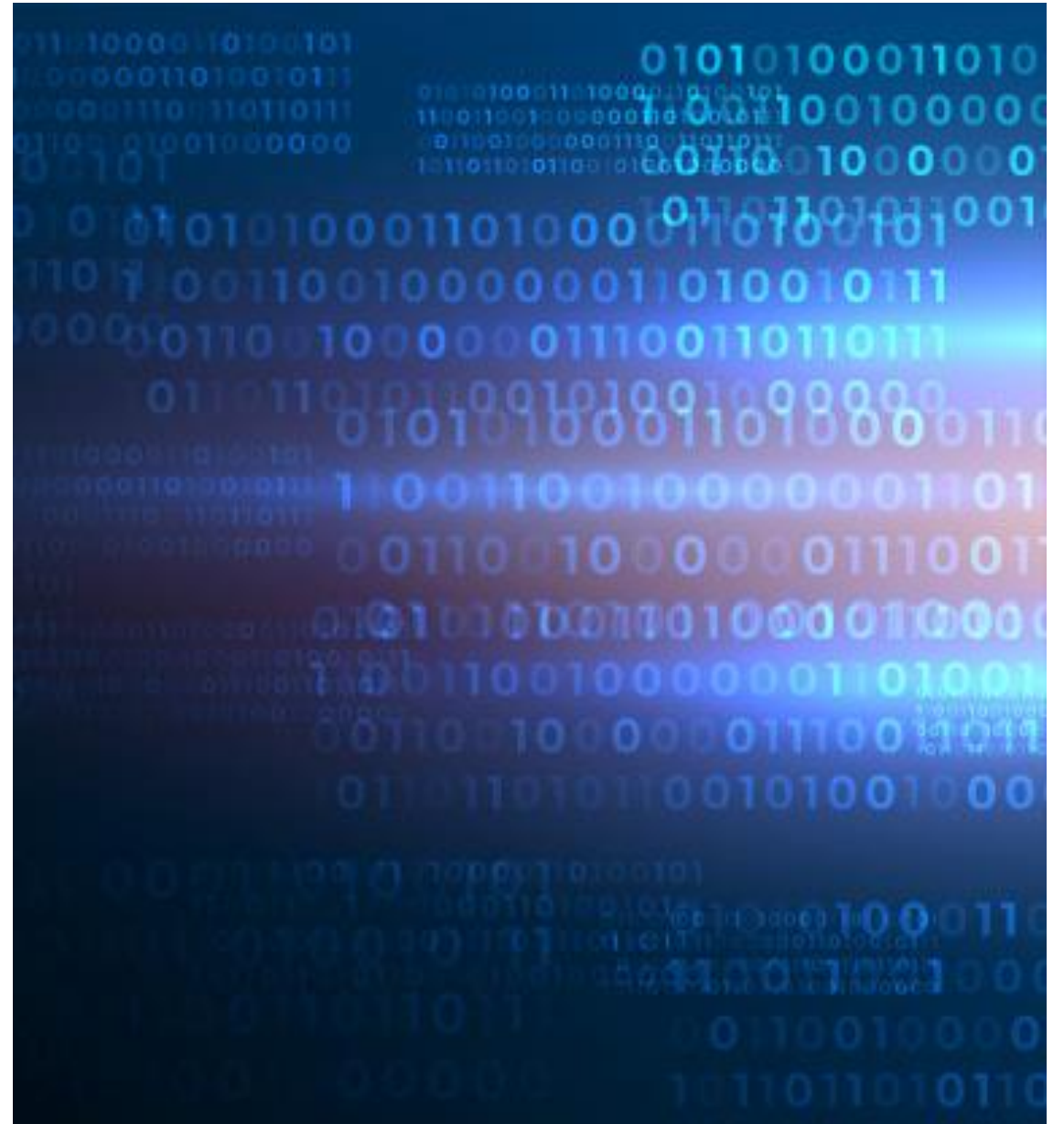
```
const myDiv = <div>Your name is {name}</div>;
```

## ACTIVITY - JSX

- Convert the code we wrote earlier to use JSX

# INTRODUCING COMPONENTS

01\_03





# AGENDA

- Creating a component
- Using a component
- Component properties
- Component styling
- Class based components



# INTRODUCING COMPONENTS

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

# INTRODUCING COMPONENTS



# CREATING A COMPONENT

- A component is just a Javascript file containing a function that returns an 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 – we'll see this later
- 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 = () => {  
    return (<p>Hello world!</div>);  
}  
  
export default ExampleComponent;
```

# ACTIVITY - COMPONENTS

Let's convert the code we wrote in index.js into a component:

- Create a folder called components and then a folder in there called Greeting.
- Create a new Javascript file called Greeting.js
- Create a const, set it equal to a function and export it as the default function
- Return the string "hello"
- Put the index.js file back to the original version

# USING A COMPONENT

- We can place a component within another component by using its name as an html style tag
- 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/Example'  
  
function App() {  
  return (  
    <div>  
      <Example/>  
    </div>  
  );  
}  
  
export default App;
```

## ACTIVITY - USING THE COMPONENT

- Remove all the code in the App.js file within the return block
- Import the Greeting component
- In the return block, insert a div, and within the div use the `<Greeting />` tag
- View the page in the browser

# COMPONENT PROPERTIES

- Component functions can take properties. These are provided as a javascript object (a comma delimited list of property names in curly brackets)
- We pass properties into the component within the tag where we use it.
- The properties appear in the component as though they were variables. We can use variables in JSX by enclosing them in curly brackets.

```
const Example = (action) => {  
  ...  
}
```

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

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



# COMPONENT PROPERTIES

- Where we need to pass more than 1 property into a component, these can be created either as a single variable (which we would normally call props) or as a comma delimited list in a javascript object
- We pass properties into the component as usual
- The way we then refer to the properties within the component depends on how they were defined

```
const Example = (props) => {  
  ...  
}
```

```
const Example = ({status, count}) => {  
  ...  
}
```

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

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

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

# ACTIVITY - COMPONENT PROPERTIES

- Pass 2 properties into the greeting component, name and age.
- Display the values within the output of the greeting component.

# COMPONENT STYLING

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

## ACTIVITY - CSS

- Create a css file called Greeting.css in the Greeting folder
- Define a css class that sets the text colour to red
- Import the css file into the greeting component
- Apply the css class to the paragraph in the greeting component.

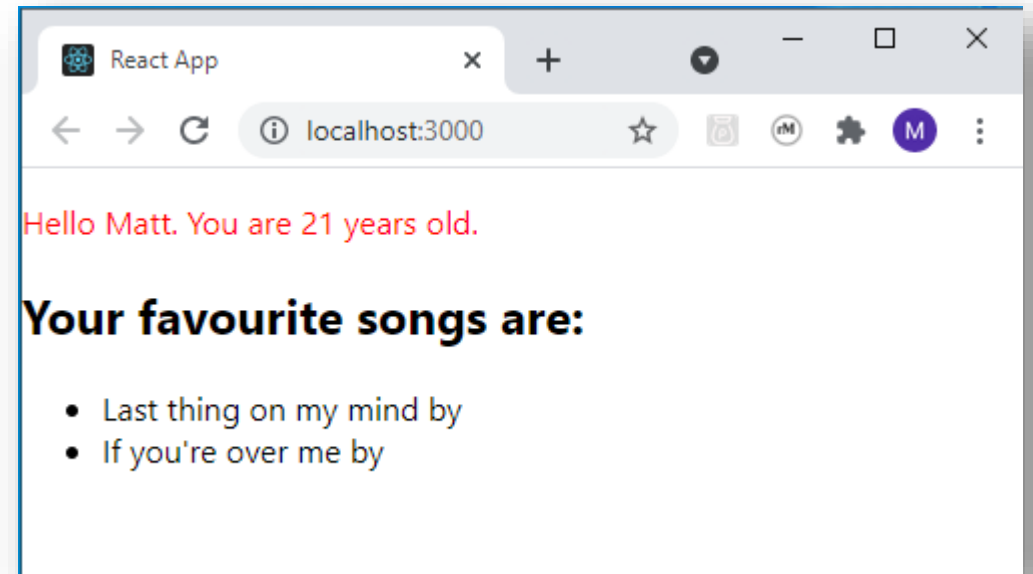


# COMPONENT REUSABILITY

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

# ACTIVITY - BUILD A PAGE

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

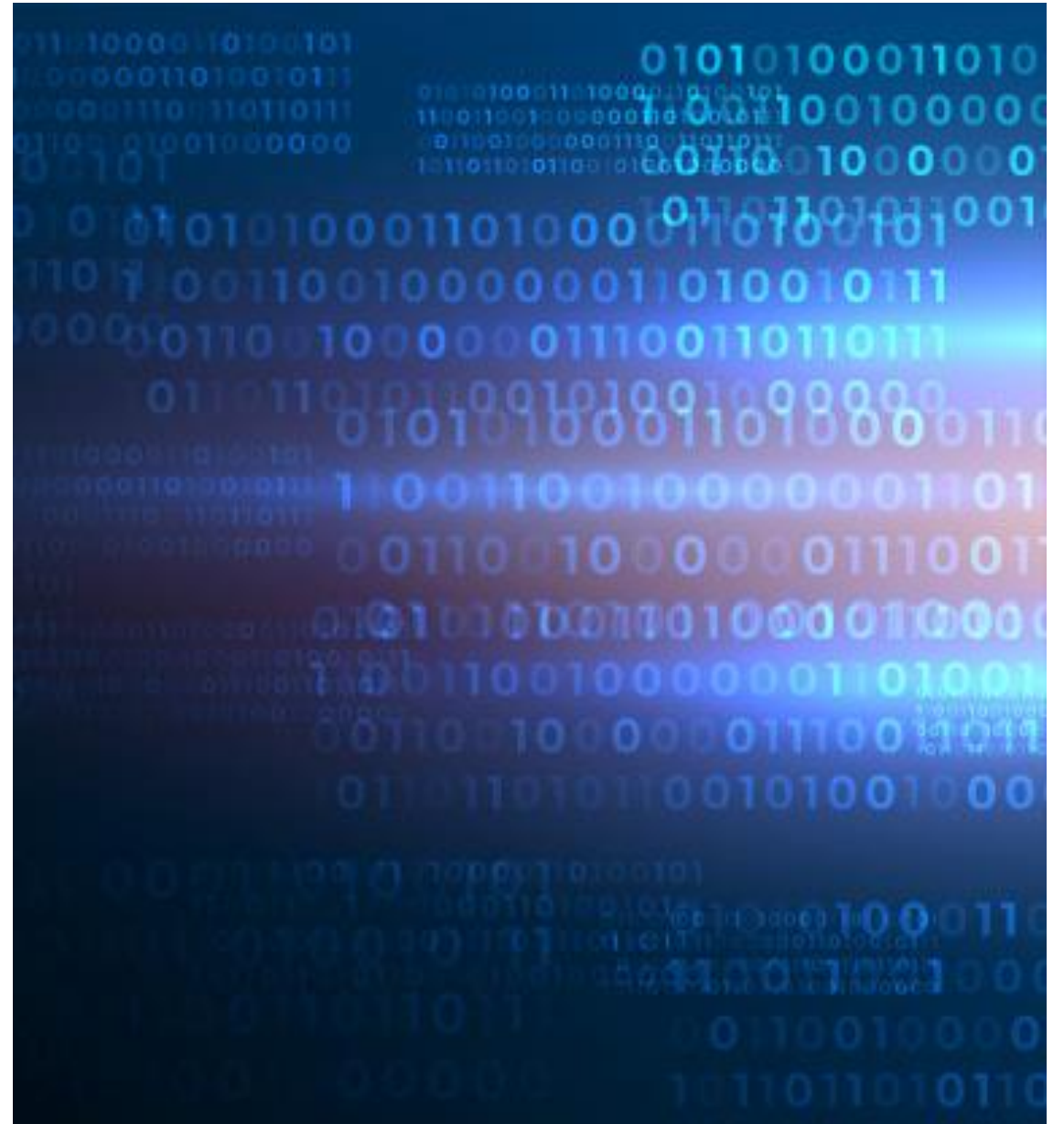


# CLASS BASED COMPONENTS

- We can create all components as functional components
- Prior to v16.8 of React (which was launched in early 2019) it used to be the case that some components had to be created as classes – these were components that had a need for state (we'll learn what this is soon)
- See the example in the closing workspace (ClassGreetingExample.js)
  - The class must extend `React.Component`
  - The class must contain a function called `render`, which must return the JSX
  - Props work a bit differently – but don't worry about that for now
- You probably won't create any new class based components but you may see some in older projects.

# COMPONENT EVENTS AND STATE

02\_01





# AGENDA

- 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 – these 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 to the function with curly brackets

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

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

## ACTIVITY - EVENTS

With the Greeting functional component

- Create a button with the text "change my name"
- When the button is clicked, make it run a function called `changeName`.
- Within the `changeName` function, print out to the console something to indicate the button was clicked (use `console.log`)

# THE NEED FOR STATE

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

# WHAT IS A STATEFUL VARIABLE?

- We can declare a variable within a component, but if we make it a stateful variable then the actual value is stored outside of the component. This means it will survive a refresh of the component.
- To create a stateful variable, we need to use the useState hook.

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

- A hook is a way for us to do something as part of the regular react running process. It lets us hook into the system and insert our own functionality.

# HOW DOES USESTATE WORK?

- We can define a stateful variable using this syntax:

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

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

```
let [status, setStatus] = useState("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, and the value we have set will be preserved.

# ACTIVITY - USESTATE

With the Greeting functional component

- Import the useState hook
- In the body of the component function, before the button click method, set up a stateful variable called `currentName`, and give it the initial value of the name passed into the component
- In the `changeName` function, change the value of the `currentName` variable to "James"

# HOW DOES USESTATE WORK?

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

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

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

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

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



# 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 lifecycle 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 code block.

## HOW DOES USESTATE WORK?

Important:

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

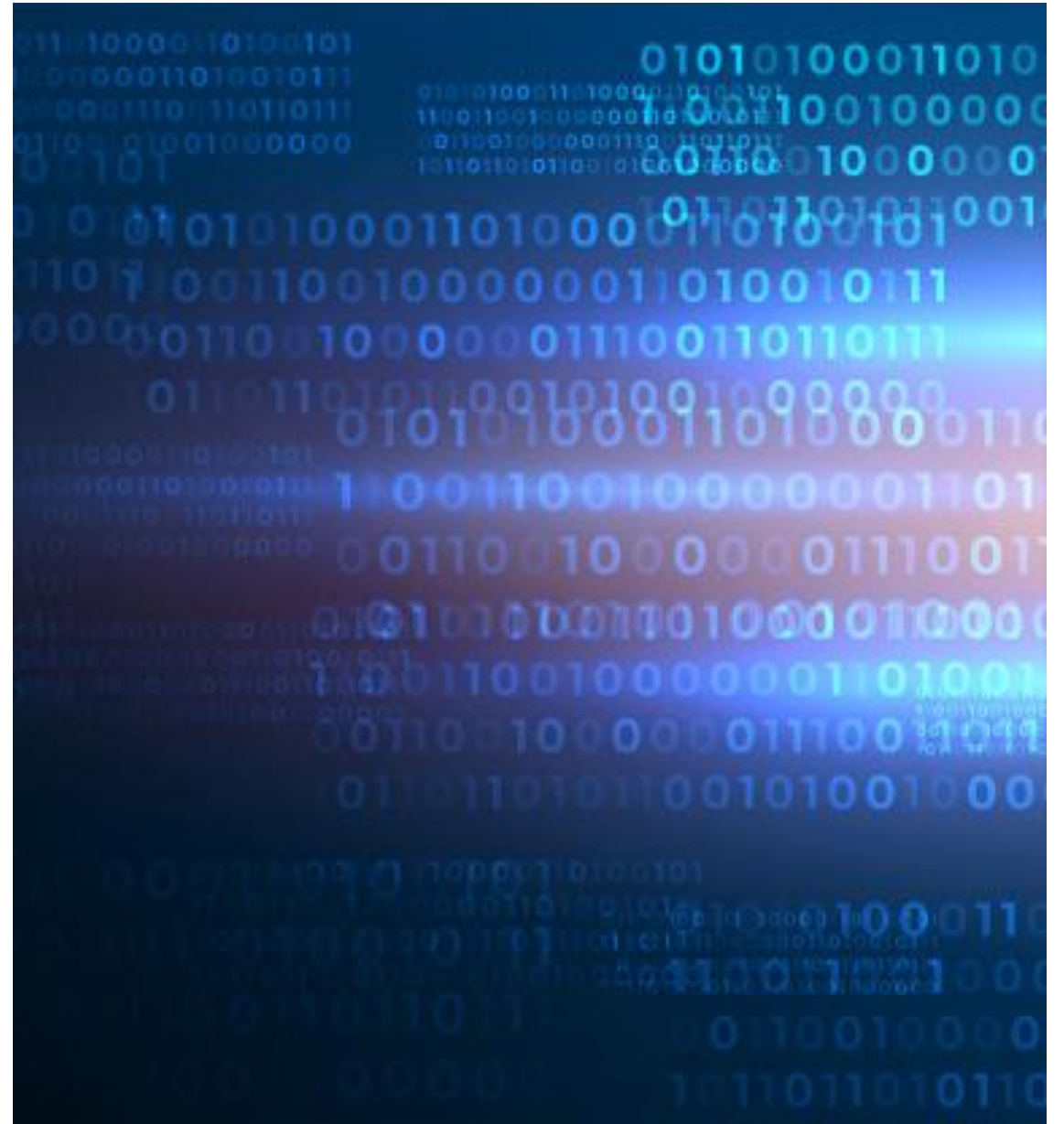
Never set its value directly.

## ACTIVITY - USING STATE CHALLENGE

- In the song list component, set up a button with the text "show songs", and set the UL to have a style of "display: none;"
- When the show songs button is clicked then the UL should be shown, and the button text should change to "hide songs".
- The button should then act as a toggle to show or hide the songs.

# COMMUNICATING BETWEEN COMPONENTS

02\_02



# AGENDA

- 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 component 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) then its children will be also.

# ACTIVITY - PASSING STATE

In the SongList component

- Create 2 local stateful javascript variables called song1 and song2, each containing the title and artist name of a song.
- Change the properties passed into the Song object to be a javascript object
- Check that the code still works
- Create a button in the SongList component that will change the name of each song to a different one by the same artist
- Check that when the button is clicked, the children are re-rendered

# CHANGING PART OF AN OBJECT STATE VARIABLE

- When we declare a variable as a Javascript Object, we can expand this into its separate values by using the spread operator
- This allows us to more easily change a part of an object – the changes must come at the end of the new value

```
let [customer1, setCustomer1] = useState(  
  {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 to change the state of a parent object – it would need to call the `setState` method which isn't visible

# 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 setState method

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

- Pass this function as a property to the child

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

- The child can then call the function.

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

## BE CAREFUL WITH THE BRACKETS

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

# ACTIVITY - PASSING TO PARENT STATE

Create the ability to vote for each song:

With the SongList component

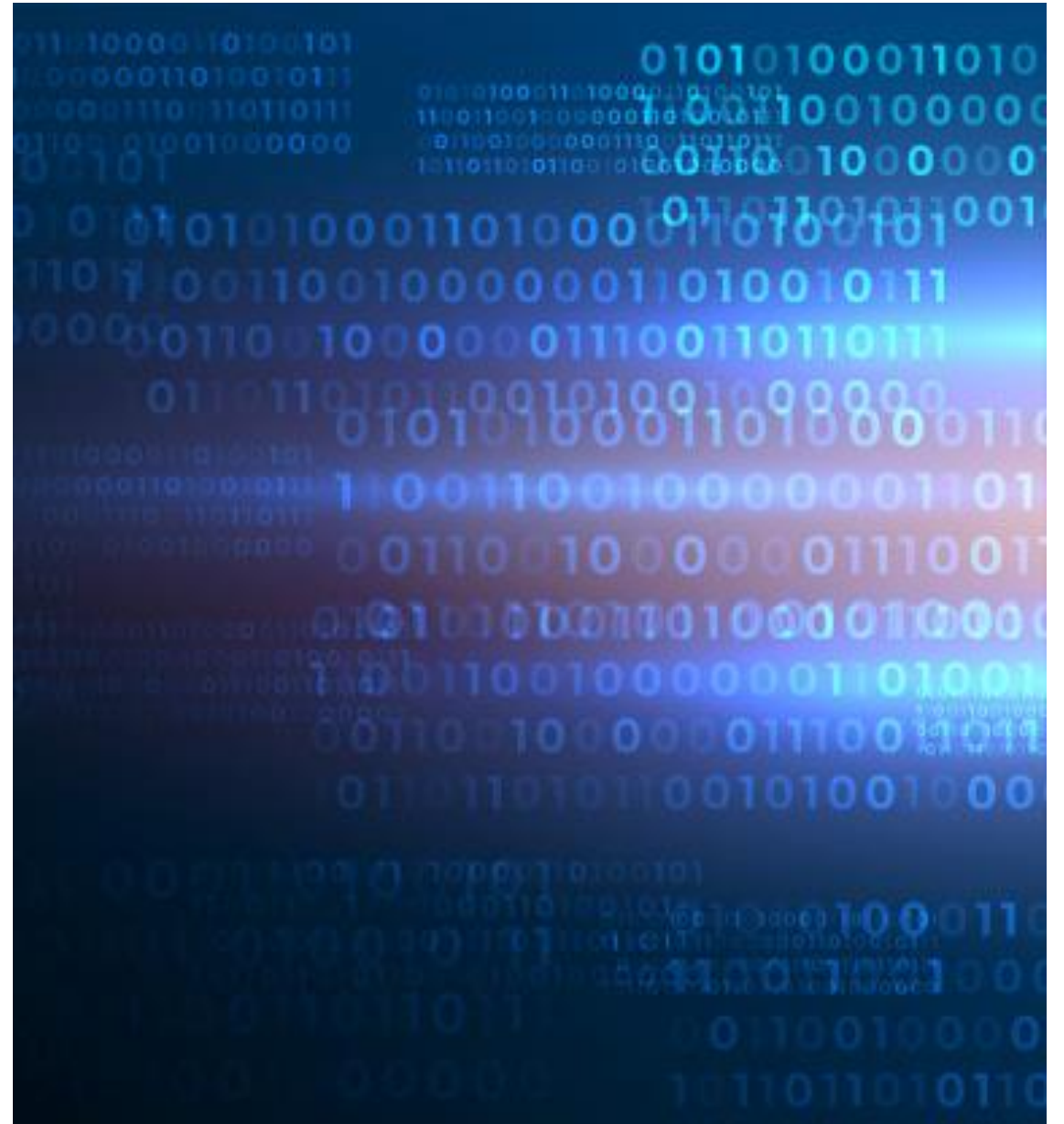
- Add an extra attribute to Song objects which will store a count of the number of votes.

In the song object

- display the number of votes
- add a button which when pressed will increment the number of votes for that object

# LOOPS AND CONDITIONS

02\_03



# AGENDA

- Loops in JSX
- Conditional rendering

# WORKING WITH LOOPS IN JSX

When we wish to create components by looping through data in an array in JSX there are 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 = [  
  <Car index="1" car={{make: 'Fiat', rating: 3}} />,  
  <Car index="2" car={{make: 'Volvo', rating: 5}} />,  
  <Car index="3" car={{make: 'Ford', rating: 4}} />  
];
```

# WORKING WITH LOOPS IN JSX

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

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

There are two approaches to achieve this:

- Create the array of elements as a property in regular JavaScript loop before the returned JSX, then bind to the property within the JSX
- Within the returned JSX, use the map function to transform each element in the array to an array of well formed elements



# APPROACH I

- The first approach is to convert the array of elements and store the new array as a property in regular JavaScript, before the returned JSX, then bind to it within the JSX

```
const displayCars = [];  
  
for (const [index, value] of cars.entries()) {  
  displayCars.push(<Car key={index} car={value}/>)  
}  
  
...  
  
return (<div>{displayCars}</div>);
```

# ACTIVITY - LOOPING

In the SongList component

- remove the changeSong method + associated button
- create a list of songs as an array, and make this a stateful variable
- remove the previous stateful song variables
- comment out the addvote methods for now
- create a variable which takes the existing array and converts it to an array of well formed elements, each with a unique key (ignore the recordVote property for now)
- Bind the array into the JSX

# MOVING TO FUNCTIONAL PROGRAMMING STYLE

- An alternative way of creating the loop is to use the functional programming style, with the map method:

```
const displayCars = songs.map ( (car, index) => {  
  return (<Car key={index} car={car} />);  
})
```

- This syntax will make it easier to write code to alter 1 of the entries if this is a stateful variable.

# ACTIVITY - LOOPING

In the SongList component

- Convert the loop to the functional programming style syntax
- Implement an addVote method. This method will take the artist as a parameter, and use functional syntax to convert the existing songs list into a new one with an updated vote value for the relevant entry.
- Pass the addVote method to the Song component using its' recordVote property
- In the Song Component, call the parent addVote method, passing through the artist name.

## APPROACH 2

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

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

# ACTIVITY - LOOPING

In the SongList component

- Place the loop using functional syntax directly into the JSX

# CONDITIONAL RENDERING

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

# ACTIVITY - CONDITIONAL RENDERING

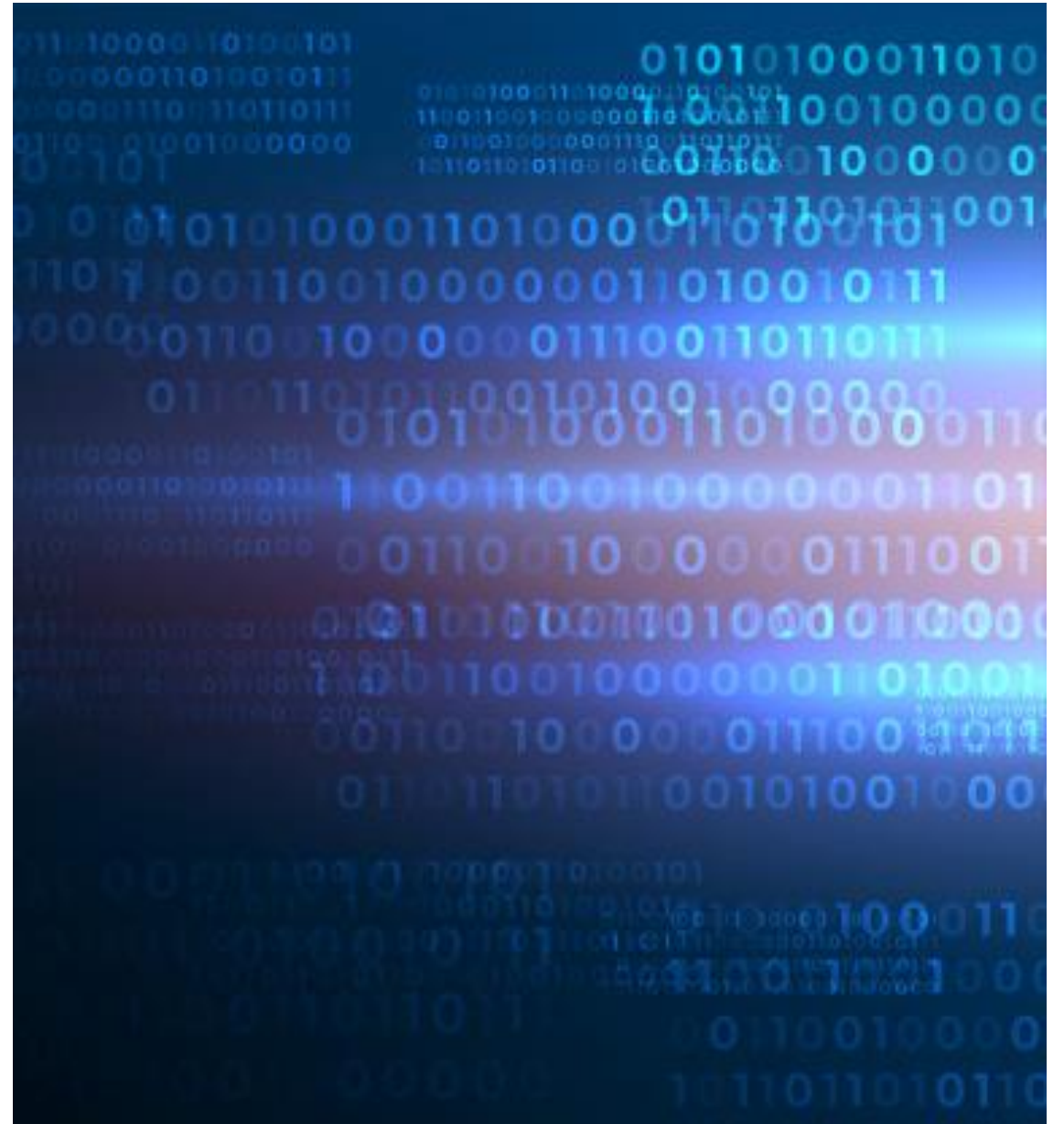
In the SongList component

- Create a boolean variable called `showAll`, with an initial value of `true`
- Create a button which when clicked will toggle the value of `showAll`.
- If `showAll` is `true`, show all the songs. If it is `false` show only those songs that have 2 or more ratings.
- Make the text of the button change based on the value of `showAll`, using the ternary operator
- Display the relevant songs list within the component using the logical `&&` operator



# DEBUGGING

02\_04

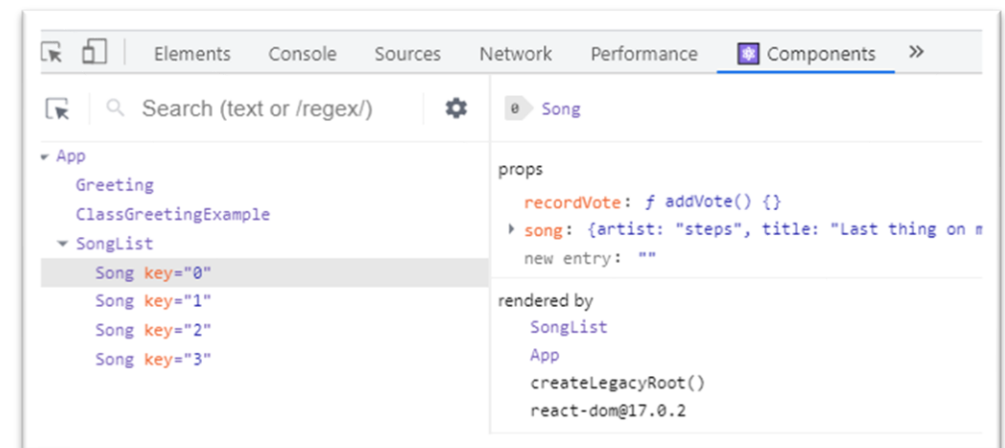
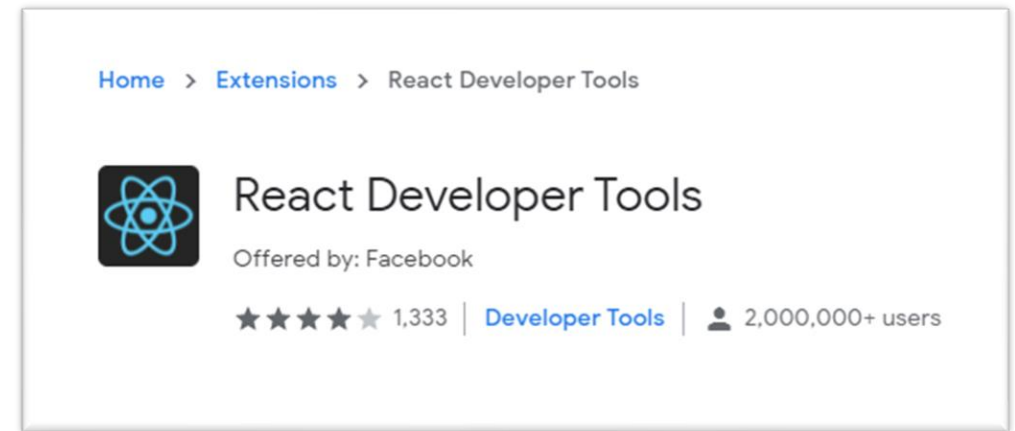


# AGENDA

- 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 already
- Open the developer tools window and find the components tab
- Navigate through the components and select one of the songs
- Review its' properties
- See how voting for a song changes its properties

# USING BREAKPOINTS

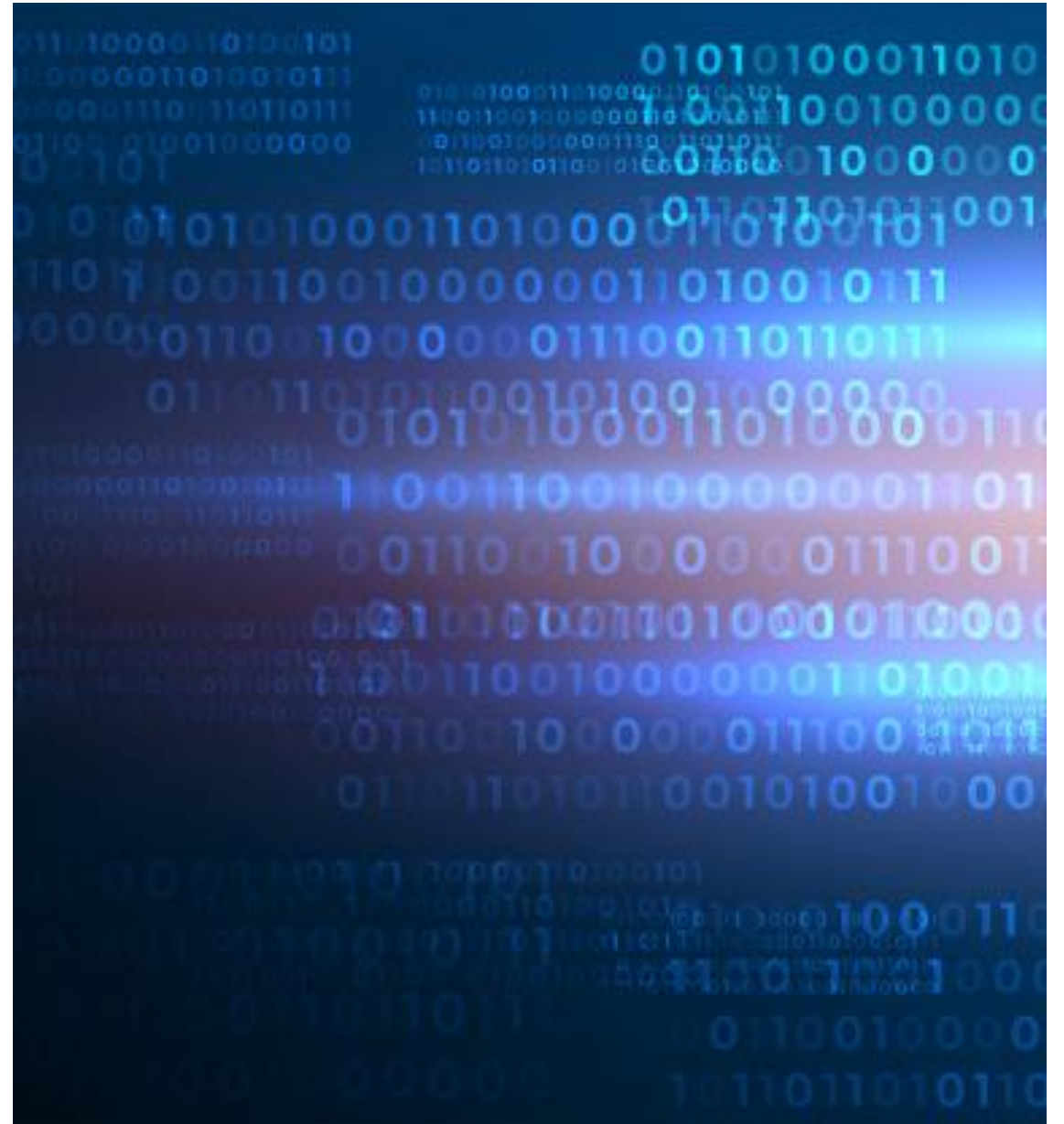
- To place a breakpoint in the code use the debugger keyword;  
`debugger;`
- This will make the browser go into debugging mode, but only when the react dev tools are visible.
- In debugging mode you can:
  - Hover over variables to see their values
  - Step through the code
  - Add extra breakpoints

# ACTIVITY - BREAKPOINTS

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

# GETTING READY FOR REST

03\_01A



# AGENDA

- What is a webservice?
- What is REST?
- Rest principles
- Fetch
- Promises
- useEffect
- Axios
- Error handling



# HTTP VERBS

- Before we start learning about webservices, there are 2 useful terms to understand, Verbs and Status codes.
- When you send a request to a server over HTTP, such as a request to view a 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 link to a web 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 values on 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 status code.
- You might have come across the idea of a 404 error page – 404 is a status code.
- The most common status codes are:

200 (OK)	301 (URL has changed)	400 (bad request)
401 (unauthorised)	(403) forbidden	404 (not found)
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 standard format (such as XML or JSON) over an internet protocol
- Clients can send an HttpRequest to a URL (the “endpoint”) , which will be 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 and optionally some data.

# SOAP

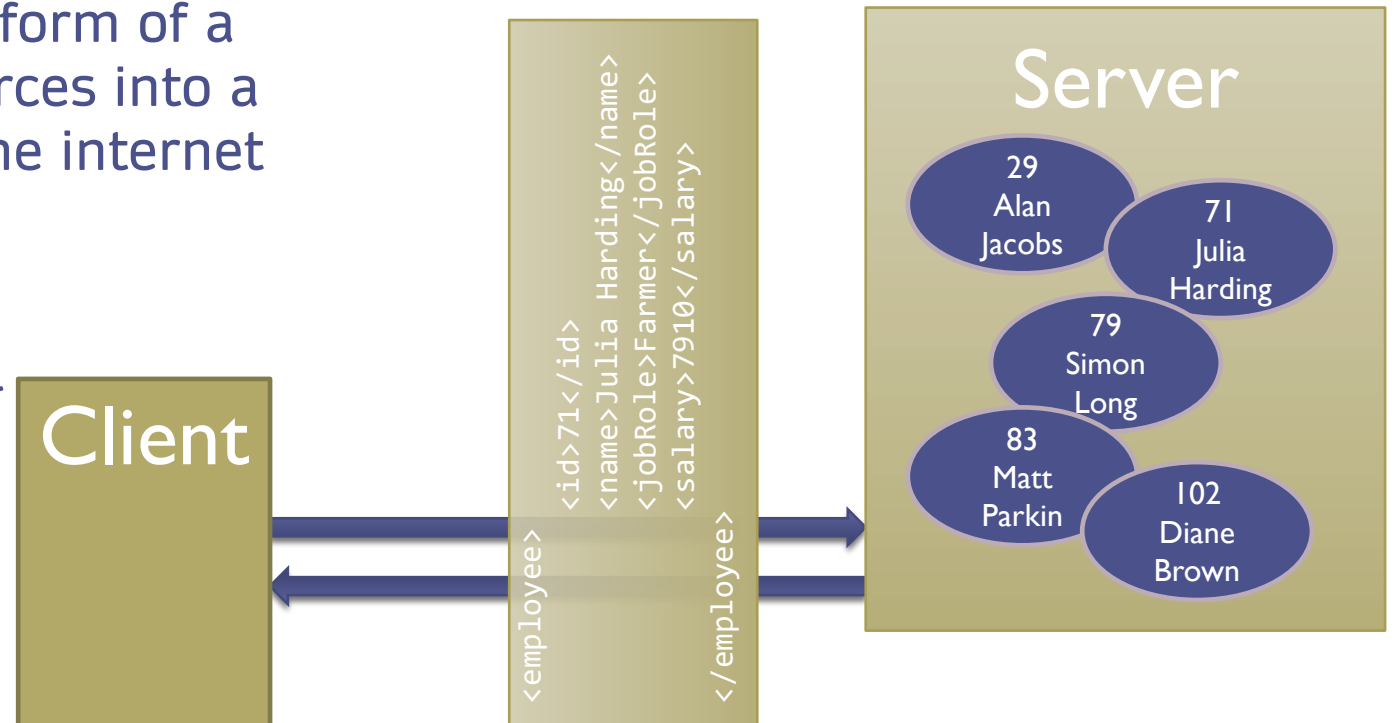
- SOAP was the first generally accepted webservice standard, originally released in 1998. It uses XML as the format for data. Many systems still use SOAP.
- One of the principles of SOAP is that for users of the webservice to find out what the different endpoints are, and how to use them, they first visit a known URL where they are provided with a list of options – this is in a format known as WSDL (Web Service Definition Language).
- SOAP requests are always sent as POSTs and because they use XML for the 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 published in his doctoral thesis in 2000. The thesis contained a number of best practices that were followed in designing the World Wide Web.
- Roy's thesis doesn't set out to design what a good web service should look like – he is trying to define what a good architecture of any distributed system should be.
- It has however become the most successful network architecture of all time!
- Rest is based on 4 key principles, but to understand them we first need to 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 entities.
- 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 I : UNIQUE IDENTIFIERS

- The first key feature of a REST based architecture is that we're going to have 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 might have a 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 we expose the entities in our system to the outside world, each entity has its own URL.

## PRINCIPLE 2 : RESOURCES ARE MANIPULATED THROUGH REPRESENTATIONS

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



## PRINCIPLE 3 : MESSAGES SHOULD BE SELF-DESCRIPTIVE

- The third key feature of a REST based architecture is that messages should include information about how to process its data.
- This simply means that there should be some meta-data attached to the message (an HTTP header) that tells us the format that the message is in, such as JSON or XML.
- We sometimes describe this as “content negotiation” - the agreement 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 message should include links to other URLs, to tell the client what else it might want to do.
- For example if a request to `http://myserver.com/webservice/employee` returns a list of all the employees, it should also include information about the URLs for each employee, or the URL to be used to add a new employee.
- 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 detail can be found at: <https://mailchimp.com/developer/marketing/api/>
- Mailchimp is an email marketing platform. A user of Mailchimp will have 1 or mailing lists, each containing subscribers, and can send email messages to their lists (called campaigns). The key resources are mailing list, subscriber, and campaign.
- This is a good example as they have mostly followed the REST rules and the URLs are logical.

# DESIGNING A REST API

- Use verbs appropriately. GET to retrieve resources, POST to add resources, PUSH to update resources, DELETE to delete resources
- Use logical URIs...
  - GET /api/employee should retrieve a list of all employee resources.
  - GET /api/employee/7 should retrieve the employee resource with ID 7
  - GET /api/employee/7/contract should retrieve the list of contracts for employee 7
  - GET /api/employee?name=Smith should retrieve all employees with a name of Smith
  - 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 state of an auction

The screenshot displays an eBay auction for an Apple iPhone 7 32GB Silver (Unlocked) - in Great Condition - UK Seller. The current bid is £132.00 with 34 bids. The time left is 7m 35s (08 Jun, 2020 13:37:50 BST). A REST client overlay is shown, displaying the current state of the auction as a JSON object:

```
{"CleanAmount": "132.00", "Amount": 132.0, "MoneyStandard": "\u0026#163;132.00", "CurrencyCode": "GBP" ....}
```

The REST client also shows the request URL: `https://www.ebay.co.uk/lit/v1/item?pbv=1&item=174309344364&si=SkWBvDGBsDNP...` and the response headers:

- content-encoding: gzip
- content-length: 475
- content-type: application/javascript; charset=UTF-8

# ACCESSING REST AS A CLIENT

- To test a rest server we need to have a client which can make the requests
- 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 browser
- Most developers use the command line tool CURL.

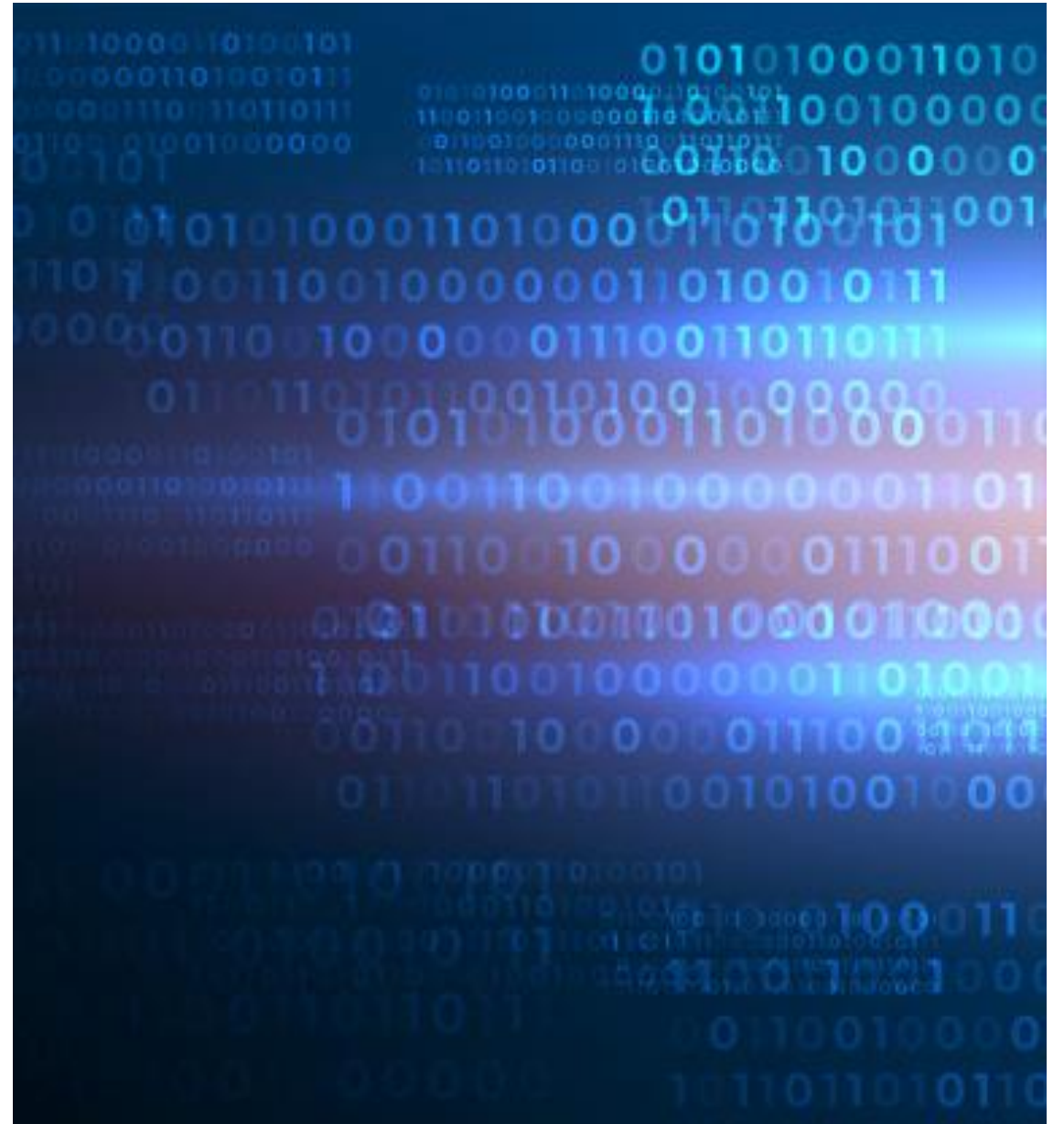
# ACTIVITY – ACCESS A REST API

- Run the application paymentgateway-standalone (use the batch file from the repository).
- This project provides a sample rest api and a web page which will generate the curl commands.
- Open the client.html and try out the commands
- Try running the CURL commands from the command line
- Try creating a .rest file in IntelliJ and running the commands there.

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

# MAKING REST CALLS

03\_01B





# AGENDA

- Fetch
- Promises
- useEffect
- Axios
- Error handling

# 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 2<sup>nd</sup> parameter can be removed.
- This function returns a Promise – it will return immediately but the data won't be there at that time.
- We can use the .then() method of the promise, to execute code when the data arrives.
- The promise will return a Response object. We need to extract its' data by calling the .json() method... and this again returns a promise.

## ACTIVITY - FETCH EXAMPLE

- Ensure the Payments back-end application is running.
- Create an additional method in the DataFunctions class called `getAllPaymentsRestVersion`.
- Call this method from the transactions component, but for now just as a method call
- In the `getAllPaymentsRestVersion` method, use `fetch` to issue a get to the payments get all transactions rest end point. (note you will need to provide headers
- Print out to the console the response object and inspect it – notice you can't see the data.

# ACTIVITY - FETCH EXAMPLE

```
export const getAllPaymentsRestVersion = () => {  
  fetch("http://localhost:8080/api/cctransaction", {  
    method: "GET",  
    headers: new Headers({'Accept': 'application/json'})  
  })  
    .then(response => console.log(response));  
}
```

## ACTIVITY - CONVERT TO JSON

- Take the response object that we received, and call its `json()` method. Return this from the fetch function – this is therefore returning a promise.
- When the `jsonPromise` completes, print out the data received to the console.

# ACTIVITY - FETCH EXAMPLE

```
export const getAllPaymentsRestVersion = () => {  
  const jsonPromise = fetch("http://localhost:8080/api/cctransaction", {  
    method: "GET",  
    headers: new Headers({'Accept': 'application/json'})  
  })  
  .then(response => {  
    return response.json();  
  });  
  
  jsonPromise.then( data => {  
    console.log(data);  
  })  
}
```

# USING FETCH

- Because getting data is a 2-step process, which can take some time, the user experience we want to create is
  - Load the component, with a placeholder for the data, so that the user sees 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 promises, 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 state of the data (whether it is loading or not)
  - Use this stateful variable to determine what gets shown on screen (a spinner or the data)
  - Run the fetch and json methods, and when both of these are complete, change the loading variable's value
- It's not quite this straightforward...



# ACTIVITY - ATTEMPT TO IMPLEMENT

- Make the `getAllPaymentsRestVersion` method return the response promise
  - In the transactions component:
    - Set up a stateful variable called `loading`, with an initial value of `true`
    - Change the return method of the component so that the table is only displayed if the stateful `loading` variable is `false`. Show a friendly message if it's `true`
    - Change the `payments` array to be a stateful object with an initial value of an empty array
    - Call the `getAllPayments` method.
    - When the promise completes call the `json` method.
    - When this completes, set the values of `payments` and of the `loading` variable.
- (Warning this won't work!)

# ACTIVITY - ATTEMPT TO IMPLEMENT

```
const Transactions = () => {  
  
  const [payments, setPayments] = useState([]);  
  const [loading, setLoading] = useState(true);  
  
  getAllPaymentsRestVersion()  
    .then(response => {  
      if (response.ok) {  
        response.json().then(  
          jsonData => {  
            setPayments(jsonData);  
            setLoading(false);  
          }  
        );  
      } else {  
        console.log("something went wrong");  
      }  
    })  
};
```

# 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` method.
  - When the `setPayments()` line is called, this changes state...
  - Which makes the component re-render, so the `getAllPayments` method is called again
  - And we're in a loop!
- There is a react hook called `useEffect` which can be used to create code that runs once only, the first time that a component is mounted.

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

- The second parameter of the `useEffect` hook can either be an empty array – this means run once only, or it can contain a list of stateful variables. If any of the variables in this list changes, the code block will re-run. If you don't include the 2<sup>nd</sup> parameter then this code will run on every re-render of the component.

## ACTIVITY - USE EFFECT

- Move the code that will load the data and set the value of payments + loading into a separate method
- Create a useEffect hook that will call this method.
- Ensure you have a console.log line at the start of the transactions component and in the fetching data method, so that you can see how many times each runs.
- This should fix the issue, with the exception that the data table is empty initially... that should be easy to fix!

# AXIOS

- Some Rest developers choose to use an alternative to the built in fetch – a 3<sup>rd</sup> party library called axios.
- With axios, the retrieval and conversion to JSON are done as a single promise, so the code becomes simpler.
- The syntax for the method call is slightly different – it only takes a single argument:

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

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

- The object returned from the promise contains a .data() method that contains the json data

## ACTIVITY - USE EFFECT

- Stop the application from running
- Install axios by running the command  
`npm install axios`
- Restart the application
- Convert the fetch method to an axios method... you'll need to import axios

# ERROR HANDLING PROMISES

- Promises might not always work. The server might not respond, or the data might not be valid JSON.
- The error handling concept is the same whether we use Fetch or Axios, but 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()` method, and optionally a `.finally()` method.

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

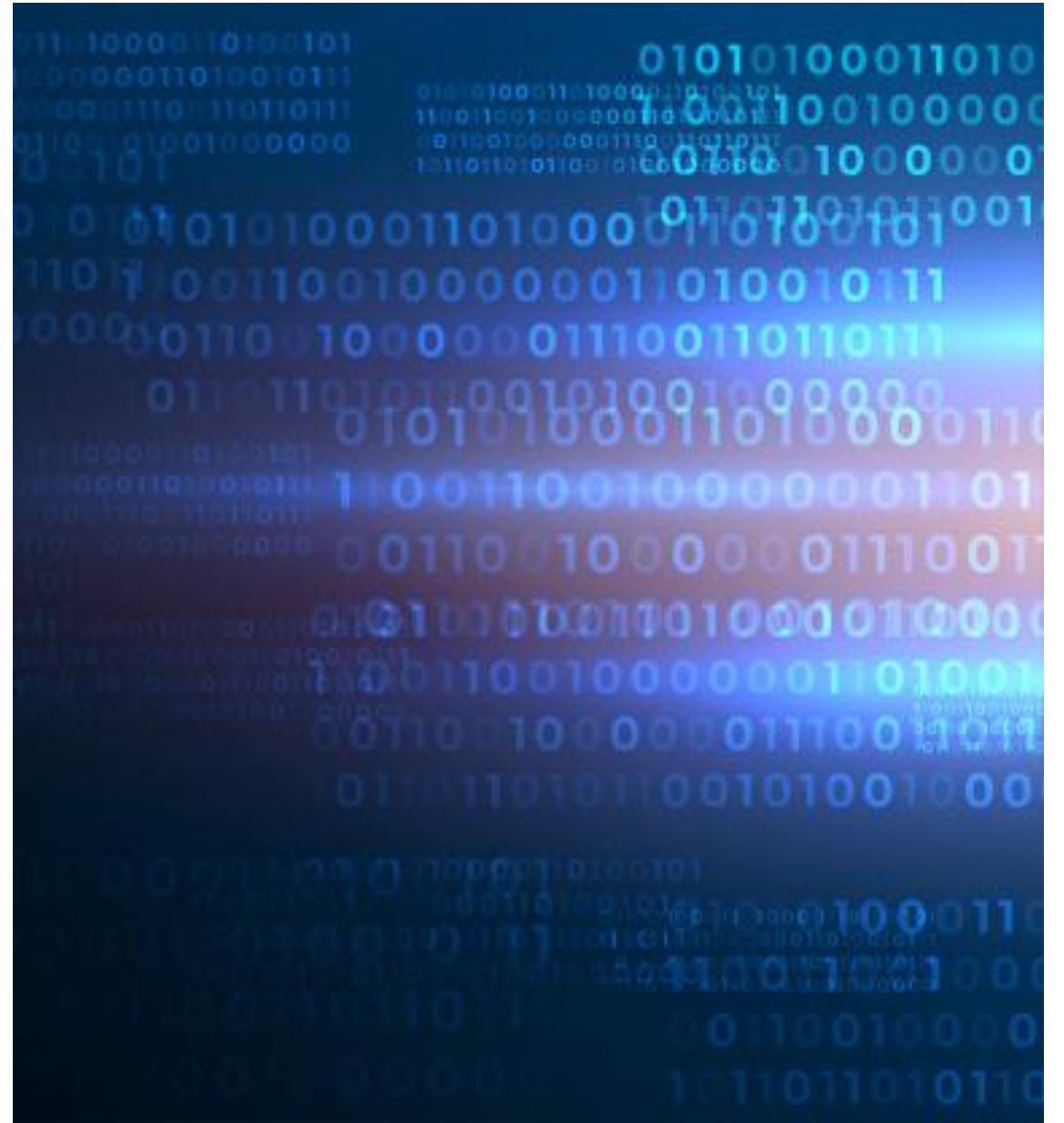
# ACTIVITY - TIDY UP

- At the moment, we are getting all the data in one go – this isn't sensible – we shouldn't load 1000's of records unnecessarily.
- A better work flow would be:
  - When the application loads, load a list of countries and populate the dropdown
  - When a country is selected, then load the relevant transactions for the selected country
- This will need an extra feature in the back end to retrieve the list of countries



# FORMS

03\_02



# AGENDA

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

# HTML VS DOM

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

# STATE TO THE RESCUE

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

# ACTIVITY - SEARCH BOX

In the Search component:

- Create a stateful variable called `searchTerm`, with an initial value of `""`
- Bind the `searchTerm` to the value of the input.
- Create a method to be executed when the input's value changes, to update the search term (use the event's `.target.value` property to find out the data in the input)
- Call the method from the `onChange` event of the input
- For now, just to test, add a method to the `onClick` event of the form to print out the search term to the console.

# USING FORMS

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

```
const handleSubmit = (event) => {  
  event.preventDefault();  
  ...  
}
```

# ACTIVITY - USING FORMS

- Change the search component so that it is now constructed as a form.

# FORM VALIDATION

- We can validate the form in the submit handler method that we write – or even in the onChange handler.
- You can use stateful variables to display meaningful errors to the user.

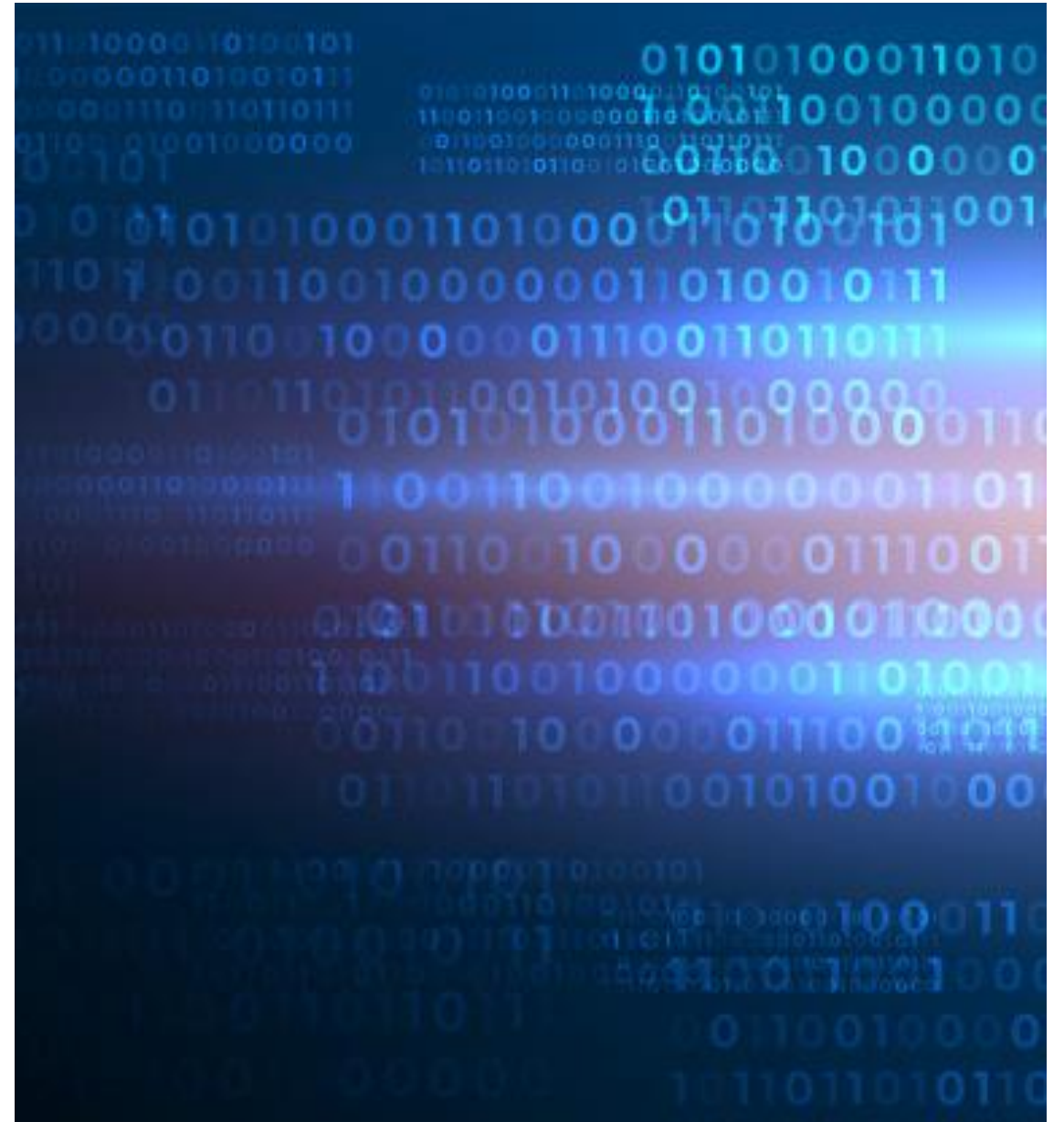


# ACTIVITY - USING FORMS

- Update the form so that:
  - As the entered value changes, if the length when trimmed is 0 characters, the input should have a red box around it
  - The search button should only be enabled if the form has been completed and is valid

# COMMUNICATING BETWEEN COMPONENTS II

03\_03



# AGENDA

- Passing State
- The Stage triangle
- Application-wide state

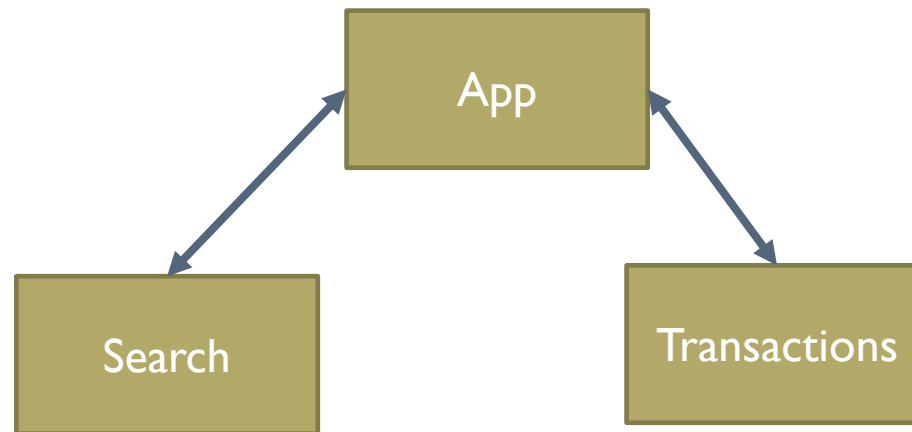
# PASSING STATE

Earlier we learned that

- a parent component can pass state to a child component using its properties
- A child can only pass state to a parent if the parent gives it access to a function that will accept the value. This function is passed from the parent to the child using its properties.

# STATE TRIANGLES

- In our example, the Search component has a searchTerm. This needs to be passed to the transactions component.
- These two components have no direct relationship – they are both children of the App component





## PASSING STATE

- State can only be passed through the direct relationship of a parent to a child.
- If there is a variable that needs to be accessed in multiple components, it needs to be held by a common parent component – this need not be a direct parent-child relationship – the state can be passed up or down multiple layers

# ACTIVITY - PASSING STATE

The searchTerm that is determined in the Search component needs to be accessed by the transactions component. To achieve this:

- We'll first create an extra interim component to be the parent of both of these. This can be the FindTransactionPage component, and later we'll add a NewTransactionPage component – these are the two "pages" of our application
- Create a stateful variable at the FindTransactionPage level
- Pass the variable as a property to the Transactions component, and the setter method as a property to the search component.

## ACTIVITY - PASSING STATE

- In the search component, use the parent component's setter method to pass the search term up the chain
- In the transactions function use the search term to retrieve the transactions from the server - if a search term is entered, we'll not use the countries drop down.
- Add a "clear" button to the search form
- At this point it might be sensible to add the orderId to the table too.

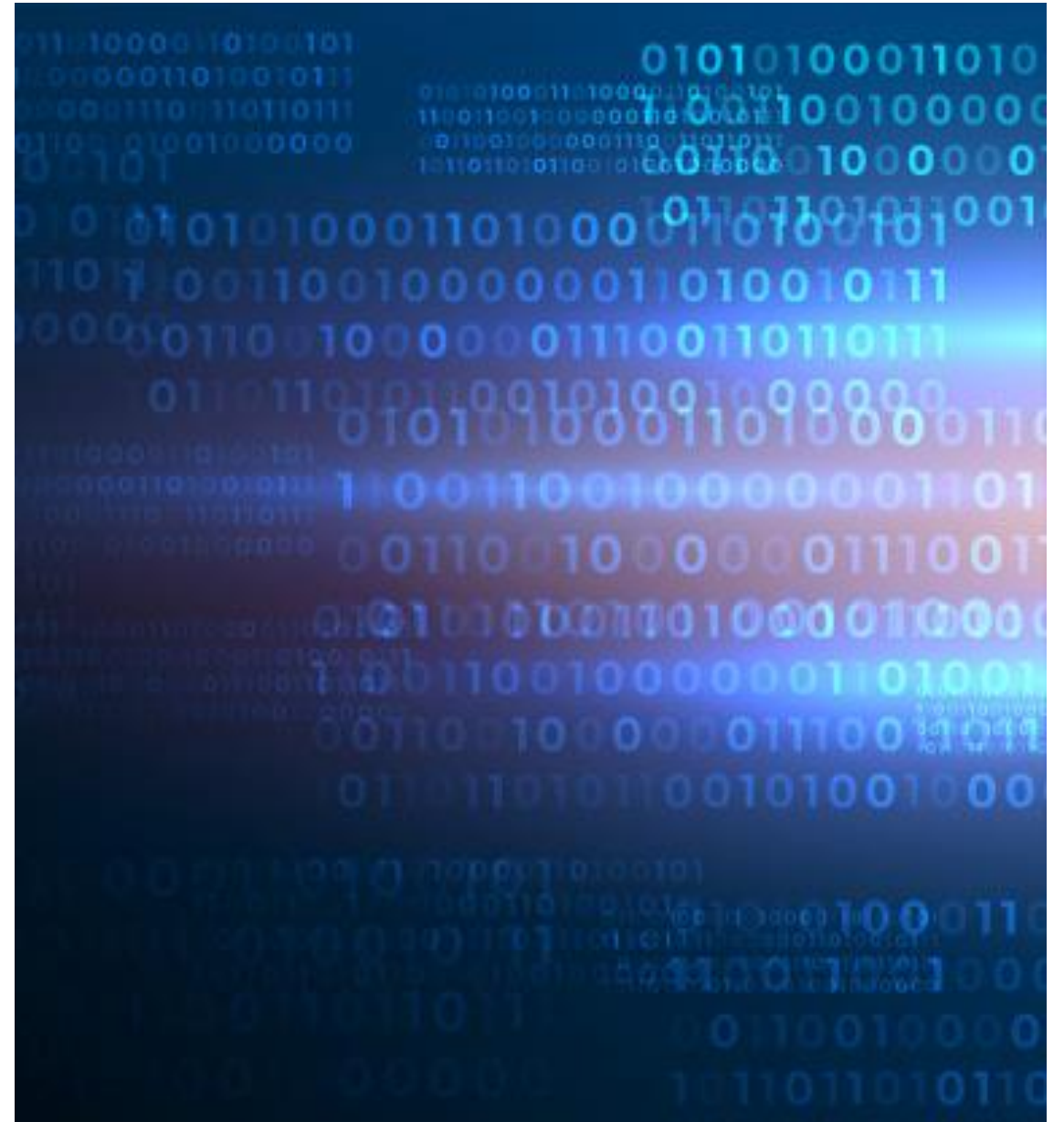


## APPLICATION WIDE STATE

- There is an alternative approach to what we have just created – we can set up state which is globally accessible throughout the application – this feature is called Context.
- However larger applications use a 3<sup>rd</sup> party library to do this called Redux, and we'll learn about that later.

# REDUCER FUNCTIONS

03\_04



# AGENDA

- useReducer
- Reducer functions
- The dispatch function

## USE REDUCER

- When you have a lot of state variables in a single component, things can get a bit messy... it's possible to combine some of them together into a single 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, we'll 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 coming soon) we'll just put it below the current page.

- Create a new component called AddTransaction and wrap it in an AddTransactionPage component.
- Within this component return the required form – a sample form is 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 function is to adjust the stateful variable – we will be giving the function, for example, the data `{country : "FR"}`, and we want the reducer function to change just that field of the stateful variable.

```
const reducer = (state, data) => {  
  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" – that's the method we'll use when we want to change a value.
  - There's an extra parameter – the reducer function we just created.
- You may get a warning that useReducer expects a 3<sup>rd</sup> parameter – this is optional and you can ignore that warning

# THE DISPATCH FUNCTION

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

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



# ACTIVITY - USING A REDUCER

- Use a reducer to store the state of the form as it is filled in.

Note that

- To pre-populate the date with today's date you can use:

```
new Date().toISOString().slice(0, 10)
```

- Every input element can call the same method for its onChange event, which will be something like this:

```
const handleChange = (event) => {  
  dispatch( {field : event.target.id, value : event.target.value});  
}
```

(if you had a checkbox, then the value would be event.target.checked)

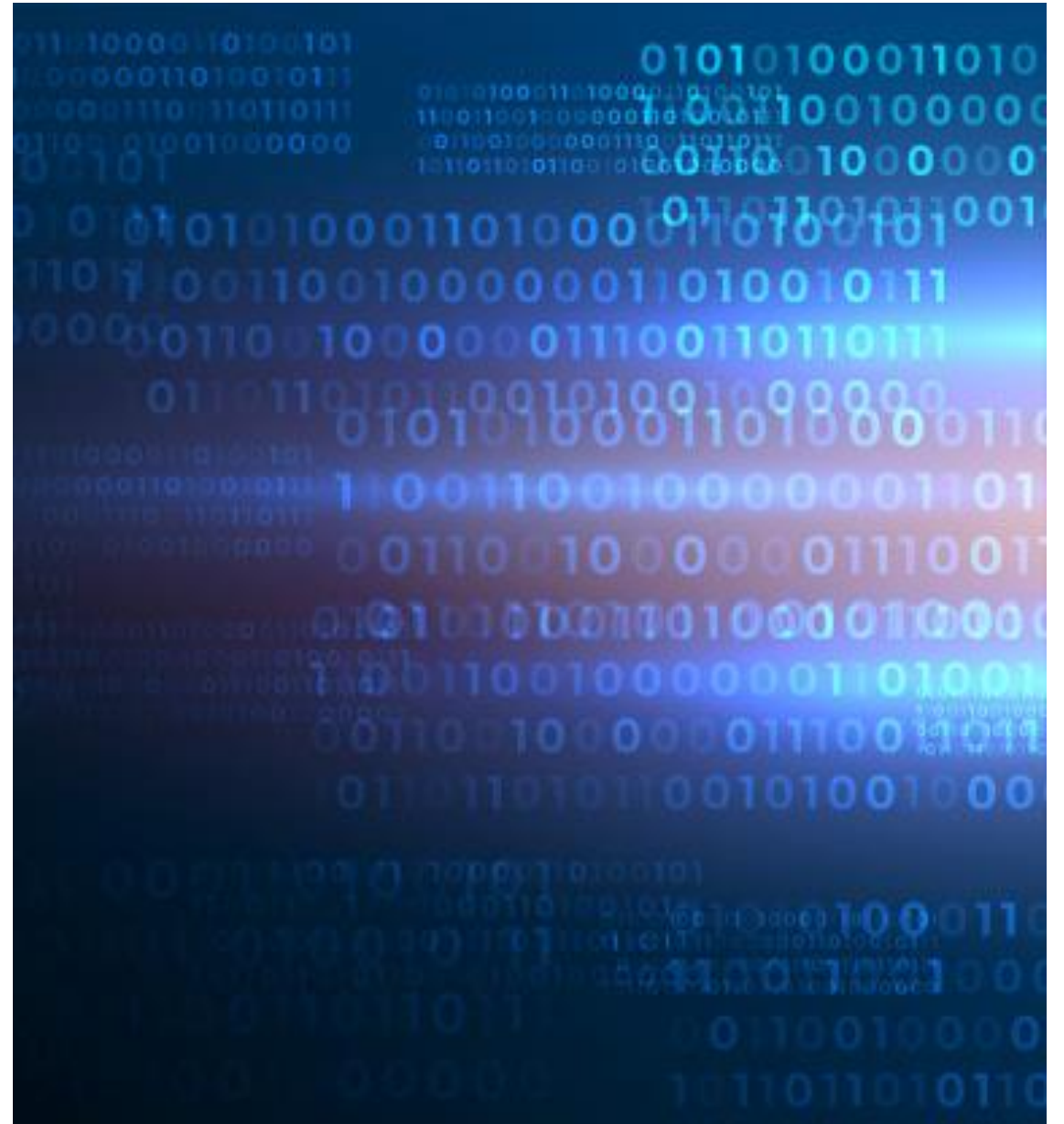
## ACTIVITY - COMPLETING THE FORM

- Now complete the form – when the data is entered, send it to the server.
- Check the status of the response, and display an appropriate message to the user.

(We are not doing form validation in this exercise, but you should do that in a live system!)

# DEPLOYMENT

03\_05



# AGENDA

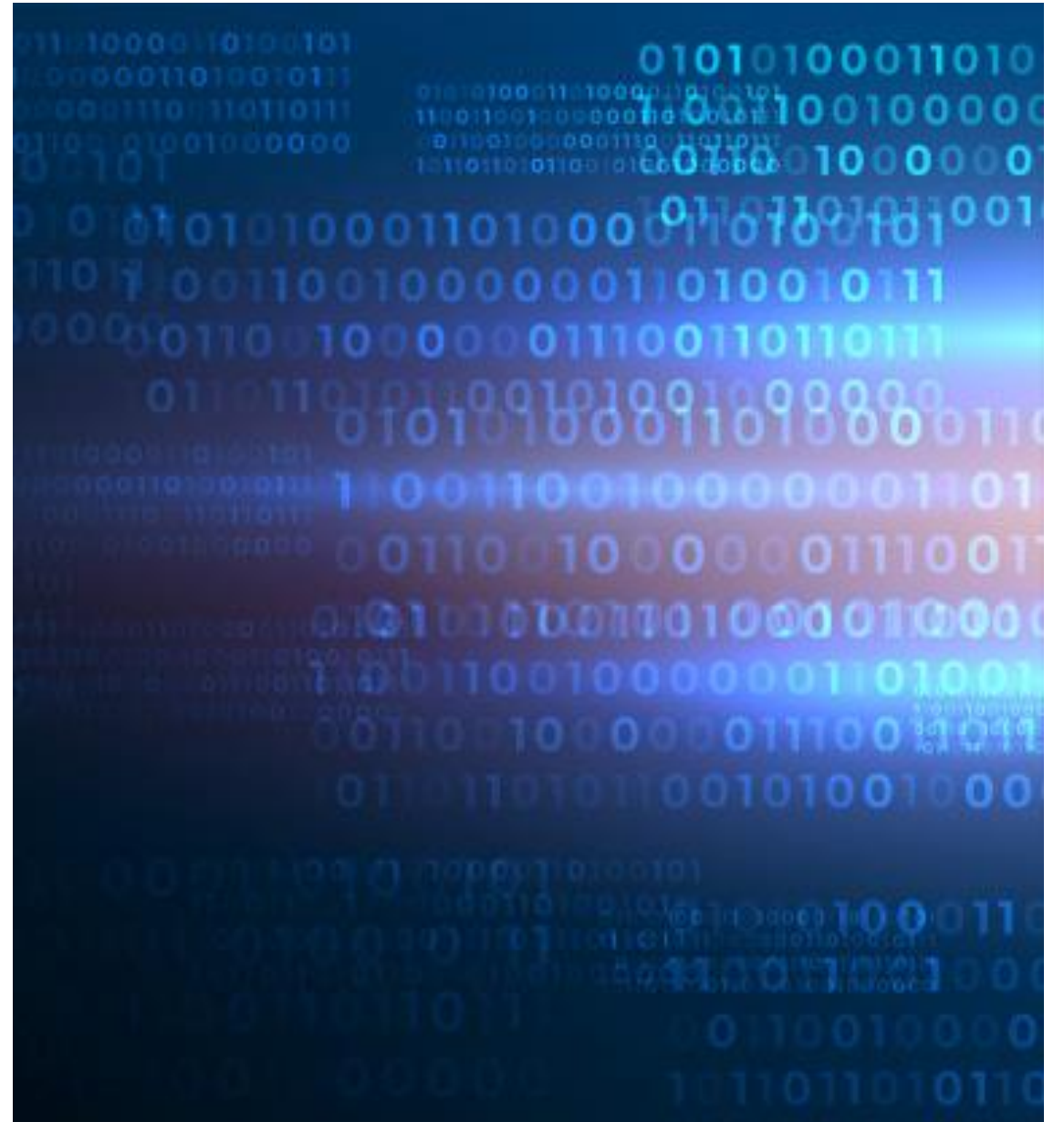
- Building the application

# BUILDING THE APPLICATION

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

# ROUTING

04\_01



# AGENDA

- 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 – this means 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 progress through the application
- We will however want to give the impression of navigating through pages to 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 default React 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 shown within each Route
  - We will define Links to allow the user to navigate to the different URLs
  - We will need to ensure that as the parent components in each route are loaded, they check the URL for any parameters, so that the data is loaded 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 single Router. 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 this route is active
- **Switch** This acts as a wrapper for each of the Routes. The first matching route within the switch for any given URL will be used.
- **Link** This is the replacement for the anchor tag – we'll use it to navigate to URLs which will be processed on the client, without a visit to the server

# ACTIVITY - IMPLEMENTING ROUTING

We'll now split our 2 pages in the Payments UI into separate URLs:

- Ensure the application isn't running and run the command `npm install react-router dom`
- Set the outer tags of the App component to `<BrowserRouter>`
- Split the 2 pages into separate routes, wrapped with the switch component – note that the order is important as the first matching route will be used
- Run the application and check that visiting the URLs shows the correct page

```
<BrowserRouter>
  <Switch>
    <Route path="/add">
      <AddTransactionPage/>
    </Route>
    <Route path="/">
      <FindTransactionPage/>
    </Route>
  </Switch>
</BrowserRouter>
```

# ACTIVITY - TIDY UP

Now that we have this structure it makes sense to:

- Move the PageHeader out of the page components and place it in App, above the switch– the URLs will only change the components within the switch. Note that it must be within the router as we'll be putting links in here.
- Let's make a home page for our application, and make this the default.
- Split the 2 pages into separate routes, wrapped with the switch component – note that the order is important as the first matching route will be used
- Run the application and check that visiting the URLs shows the correct page

# LINKS

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

```
<Link to="newURL">
```

## ACTIVITY - LINKS

- Add links to the PageHeader and Menu components to set up the navigation.
- Links generate a regular HTML anchor tag – so you can apply styling to links by creating css rules for the anchor tag.

## 404 PAGES

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

- Change the home page route to be an exact route (so that it matches only against "/" not against every URL)
- Create an extra route for "\*" and display the page not found component in this route.

## ACTIVITY - 404

- Create a `PageNotFound` component
- Ensure the home page route is set to be an exact route
- Add in the `*` route to the page not found component.



# 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 useHistory hook
- In case the user refreshes the page, we need to read in the URL when the page loads and check if there's a parameter there...
- We can read in parameter values from a URL using the useParams hook.

# ACTIVITY - USING PARAMETERS

In FindTransactionPage:

- When the searchTerm is changed, use the "useHistory" hook to navigate to the new URL

In App.js:

- Update the route associated with this page to accept a url with a parameter (this is entered as /url/:parameterName

In Transactions:

- Whenever this component renders, we'll need to load in the orderId parameter, compare it to the search term and if the two are not the same, then set the searchTerm to the orderId parameter. However the search term is part of props, so we can't change that directly – we'll need a new local state variable, which we'll call selectedOrder...

# QUERY STRINGS

- As the users use the application we will also want to consider embedding 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, and then (2) load the data specified in the query string.
- We move to the correct URL using the useHistory hook
- We can read in parameter values the query string using URLSearchParams()

# ACTIVITY - USING QUERY PARAMS

In Transactions:

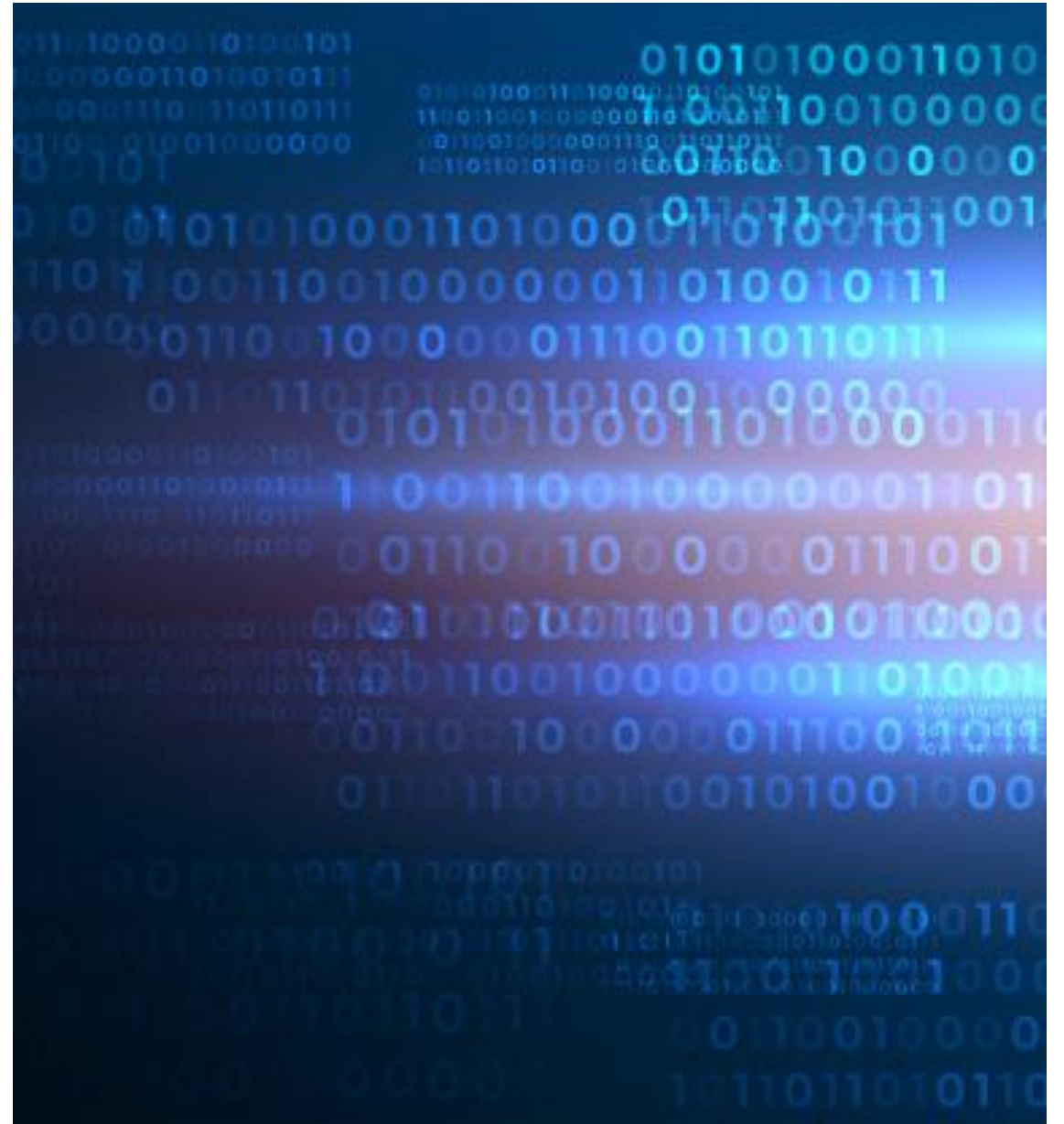
- In the `changeCountry` function, use `useHistory` to set the new URL. Note that this will not affect the route matching.
- When the component loads, check if there is a query parameter, and if so use it to set the `selectedCountry`.

# SURVIVING A REFRESH

- One of the challenges when working with a single page framework like react is how do we survive if the user clicks on the refresh icon on their browser.
- By careful planning of URLs (which we now have) we can survive a refresh!
- Note also that the user can click the back button on the browser and that will work too.
- Unfortunately if there is data that the browser needs to remember that can't be put into the URL, e.g. a user's authentication data, then this is a bit more complicated, and outside the scope of this course!

# TESTING

04\_02



# AGENDA

- Testing libraries
- Testing methods
- Reading the screen
- Assertions
- Working with promises
- Mocking

# TESTING LIBRARIES


- To create automated test for React applications we need 2 libraries:
  - Jest = framework for writing tests (like Junit)
  - React testing library = simulates a browser and lets us inspect the DOM
- Both of these are available by default
- We can run the tests with `npm test`. Once running, tests are re-run automatically as we make changes
- This will scan for tests in any file called `xxx.test.js`
- It is convention that we create a test file per component, but you must have at least 1 test in the file, so don't create it unless you are planning to write a test!



# TESTING METHODS


- A test class is simply a collection of calls to the test function. The syntax for this function is:

```
test('description of the test', () => {  
  render(<Component />);  
  ...  
});
```



- You can optionally group tests into test suites with the describe function

```
describe('description of the test', () => {  
  ...  
});
```



# READING THE SCREEN

- To view what is on the screen we use the screen object. This has methods like `getByText`, `getbyTitle`, etc:
- The methods return an `HTMLElement`
- Each method appears in 3 versions:
  - Get methods will throw an error if the element isn't found
  - Query functions won't throw an error (but an assertion might then fail)
  - Find functions return a promise
- The functions will look for an exact match, unless you specify `{exact:false}`

## READING THE SCREEN

- The `findAll` / `queryAll` methods will return an array of objects. This array may be empty even for the `getAll` method.
- If you want to search for a type of HTML element, you can use the `getByRole` / `queryByRole` methods
- Not every HTML Element type has a role, so you'll need to check whether there is one, and if so what the role is here:

<https://www.w3.org/TR/html-aria/#docconformance>

# ASSERTIONS

- The equivalent to a Junit assertion is the expect method.
- We simply call expect, pass in the HTMLElement, and then use one of its methods like:
  - toBeInTheDocument
  - toHaveValue
  - toHaveAttribute
  - toHaveClass etc
- We can prefix these with .not to test the opposite

## ACTIVITY - WRITING A TEST

- Remove App.test.js as there are no tests we can write for the App component at the moment
- Create a test for the Search component.
- The test should check that when the component is first rendered, the input does not have the searchBoxError class applied to it;

## CHILD COMPONENTS ARE RENDERED TOO

- When you render a component, its children will render also. This can be helpful as you may need a parent component to be present to run a test.
- You can also create dummy components in the JSX for the render command – for example if a component needs a `BrowserRouter` object to allow it to render, we can create one as part of the render command.

# ACTIVITY - CHILD COMPONENTS

- Create a test for the Menu component.
- The test should check that the word "Find" is on the screen, and when clicked would link to "/find".
- When you run the test, you'll see an error – we can't render the menu component as it has to be within a BrowserRouter.
- Change the render to wrap the menu component inside a BrowserRouter.
- To get the test to pass, you'll need to make it a non exact matching test
- When checking the link, check the "href" attribute (the link is rendered as an anchor tag)

## USER ACTIONS + STATEFUL VARIABLES

- We can simulate users typing in values or clicking on buttons as part of our tests by using the `userEvent` object from the react testing library.
- You can only test what appears on the screen. If you have a stateful variable, and it doesn't appear on the screen, you can't test for it.



# ACTIVITY - USER ACTIONS

- Create a 2<sup>nd</sup> test for the Search component
- Make the second test check that whatever you type into the search box will be stored in the stateful variable... this is bound to the input so we can check the value of this object
- Create a 3<sup>rd</sup> test that checks that if the value of the input element is just spaces, this will result in the input element having the class `searchBoxError` applied
- Because we have two tests that are checking the css classes of the input, put them into a test suite.

## USING PROMISES

- We can use the `find` / `findAll` methods to check for items which may appear on the screen after the component has initially rendered.
- For example with the transactions component, we expect the combo box of countries to be displayed, but this will only happen after the rest call has been made and processed.
- We will probably want to mock this (we don't want to make a rest call in a test) which we'll do also, but first let's use a promise...

# ASYNC AWAIT

- When we use promises in tests, we must use the Javascript construct of `async...await`.
- When we do this, we define our method as an `async` method. Then we place the word `await` before the method that returns a promise. This will make the code pause until the method returns, at which point we'll then have a resolved promise to work with.

# ACTIVITY - PROMISES

- Create a test file for the Transactions component
- Ensure the home page route is set to be an exact route
- Test that the countrySelector appears on screen within 5 seconds.
- The test will initially crash with the message relating to `useParams()`. This is because `useParams` must be within a router... so render the component within a `BrowserRouter`

# MOCKING

- At times we will want to override functions within a component with a dummy version. For example to avoid a real rest request taking place during a test.
- To create a mock we can use the following syntax – this must be placed BEFORE the test methods

```
jest.mock('module_name_with_path', () => {  
  return {  
    functionName: () => Promise.resolve(return_value)  
  };  
});
```

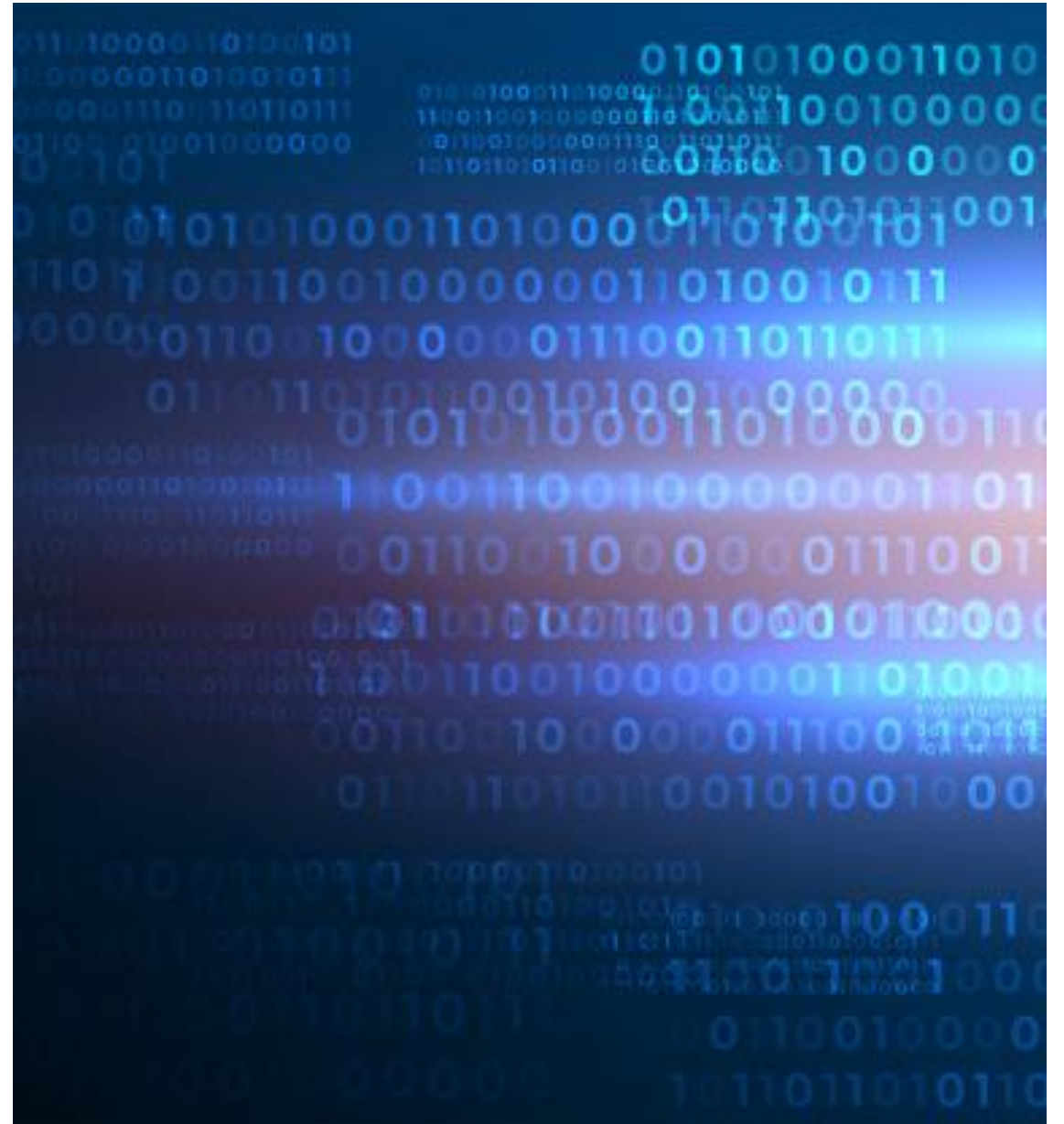
# ACTIVITY - USING MOCKS

In the Transactions test:

- Create a mock to override the `getCountries` method of the `DataFunctions` module. Return 3 countries (the value doesn't matter)
- To prove that the mock is working add another test to check that the number of option items on the page is 4 (the 3 we have allocated + the `--select--` option)

# BASIC REDUX

04\_03



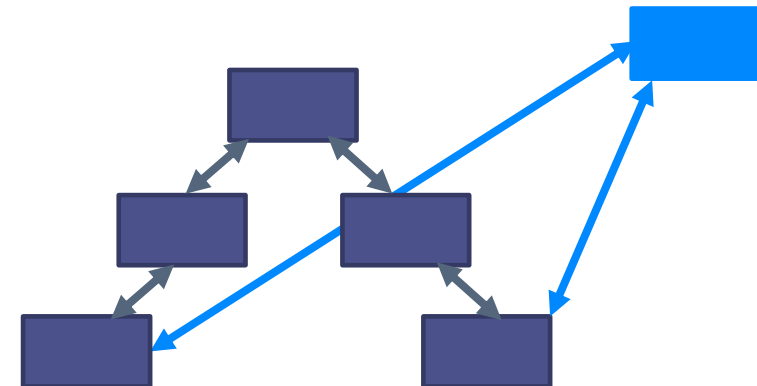
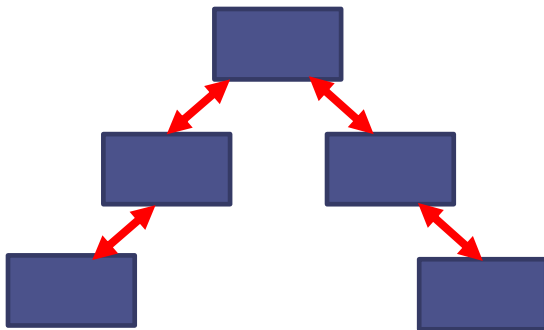
# AGENDA

- What is Redux
- Adding Redux into React applications
- Concept & Principles
- Creating the Store
- Creating a Reducer function
- Getting data from the store
- Dispatching changes to the store



# WHAT IS REDUX?

- A state management system for cross-component / app-wide state
- A separate project to React that can work integrate with it
- This is not (necessarily) an alternative to useState or useReducer
- But it is an alternative to useContext
- In React components can only communicate up and down the tree, and state is defined within a component.
- But all components can access the same Redux store



# ADDING REDUX INTO REACT APPLICATIONS

- Redux is not included in React – you need to add it
- We need both Redux and React-Redux (which bridges the two frameworks)
- We can install it into a React project with

```
npm install redux react-redux
```

## ACTIVITY- INSTALL REDUX

- Run the command `npm install redux react-redux` to add redux into the project.

# REDUX CONCEPT AND PRINCIPLES

- Redux is based on the concept of a store.
- A react store contains the state. All the data is in one (potentially large) Javascript object
- We define which parts of the application can access the store using a Provider component.

```
{
  currentUser : {
    id : 32,
    username: "phil",
    role : "admin"
  },
  countries: ["USA","DEU","FRA","ESP"],
  accessCounter: 271,
}
```

# REDUX CONCEPT AND PRINCIPLES

- We can read from the redux store by using a useSelector hook.
- This hook requires a function that takes the state object, and extracts the part of the object we are interested in

```
{
  currentUser : {
    id : 32,
    username: "phil",
    role : "admin"
  },
  countries: ["USA","DEU","FRA","ESP"],
  accessCounter: 271,
}
```

```
const countries =
  useSelector( state => state.countries);
```

# REDUX CONCEPT AND PRINCIPLES

- The data in a redux store is immutable – it cannot be changed.
- We need to provide a special reducer function which will REPLACE the current state object in the store with a new one.
- The reducer function takes 2 parameters: the current state, and an action
- The action defines how the state is to be changed – it will return the entire new replacement state
- We call the reducer function with a special hook called useDispatch – into here we provide the action

# IMPLEMENTING REDUX STEPS

- Add redux to the project
- Create a store javascript file containing:
  - The initial state when the application starts
  - The reducer function
  - A store created from the reducer function
- Export the store from this file
- Create a Provider component using the store
- Read from the store with useSelector
- Run the reducer function with useDispatch
- \*\*\* take care not to confuse Redux reduce + dispatch with the useReducer() and dispatch function \*\*\*

# ACTIVITY- CREATE A REDUX STORE

Right now we are loading in the countries many times creating lots of round-trips to the server. Let's store this in redux. Then if the component is unmounted / remounted, it can read the countries list from redux.

- Create a file called store.js in a folder called store
- Define the initial state – this will be the state when the application starts, and is used to define the structure of the state object
- Create a reducer function which takes an action containing two attributes – a type of "replace" and a value of the new array of countries. This function should check if the type is replace, and if so return a new state object containing the changed data
- Create a store object using the reducer function and export it as the default export.



## ACTIVITY- PROVIDE THE STORE

- In the App component, use the provider component to define the scope of the redux store

# ACTIVITY- READ AND WRITE FROM THE STORE

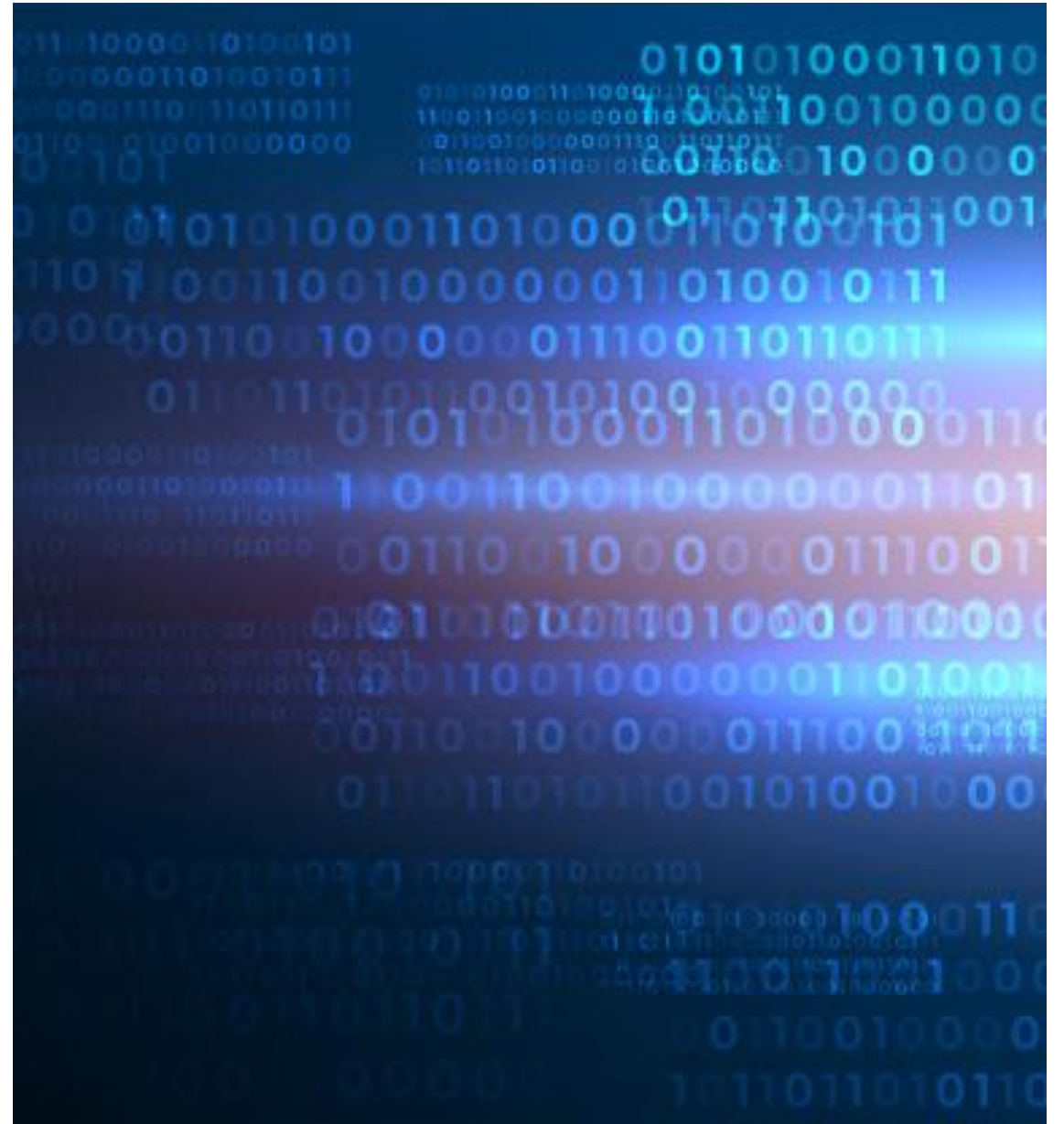
- In the `loadCountries()` method of the transactions component, check first to see if the countries are loaded in the redux store, and if they are use them. If not make the rest call and store the results in the redux store.
- Remember that the hooks must be at the top level of the function – they cannot be within a method.

## ACTIVITY- TAKING THINGS FURTHER

- Update the functionality so that we do get a list of countries from the rest interface if it is at least 1 minute since the last time we got the list.
- Change the AddTransation form so that the country field is now a drop-down. Use the countries from redux if there are some there, otherwise get the countries from react and update redux. Use the same timeout rule.
- Hint... make the country selector a separate component so that you can re-use it!

# CHALLENGES

99\_99



# CHALLENGE I

- Create a new project called payments-ui. This will be the front-end that will connect to the payments database rest application.
- Create a set of components that will come together to form the user 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 transaction”
  - 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 console).

# CHALLENGE I

- Here is an example of what you might build:

**Payments Application**   Find a transaction   New transaction

Order Id:

Id	Date	Country	Currency	Amount
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

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.

# CHALLENGE 2

## Part 1 - looping

- Import the provided DataFunctions.js – this is a simple Javascript module (not a component!) which is designed to simulate the REST connection we'll build later on. I suggest you put it in a folder called data.
- Amend the table you created so that it will be generated by calling the function in the dataFunctions javascript file. You should do this by creating a new component called PaymentTableRow. This component which will take a payment as a parameter, and 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 console).

# CHALLENGE 2

## Part 2 – stat & conditions

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

Hint – the following is an example of how to remove duplicates from an array:

```
array.filter((item, index) => array.indexOf(item) === index);
```

- Based on the value of the drop down filter, change the list of payments so that only payments from the matching country are shown. To do this you will need to:
  - Create a stateful variable which will store the currently selected country
  - When the onChange event runs, use this to update the selected country.



## CHALLENGE 2

Hint - for this, you can find out the selected item in the dropdown if you create the function to be called like this:

```
■    const changeCountry = (e) => {  
        const option = e.target.options.selectedIndex;  
        ...  
    }
```

- 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).

**Payments Application** [Find a transaction](#) [New transaction](#)

Order Id:

Select country:

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

## CHALLENGE 3

- We now have a working application and have implemented the read feature. Now we will implement the ability to add a new transaction. This will be a bit of a challenge!
- At the moment, we don't know how to navigate the virtual pages, so for now we'll place the 2<sup>nd</sup> page after the first page on the same screen.
- Don't worry about form design. It's up to you how much validation you wish to do (none is acceptable!)
- Also take this opportunity to organise the components into a more sensible file structure.

Hint : If you use `<input type=date />` then you need to store the date in the format yyyy-mm-dd. To get today's date in this format use

```
new Date().toISOString().slice(0, 10)
```