A first React app
React's official documentation is quite good. See here:
https://react.dev/learn

React is developed by Meta and was originally focussed on developing reusable components for **single-page applications** (i.e. front-end only, but you could add your own back-end API). Multi-page apps could be handled with React's router. As time has gone on, though, there is a drive to make React a more full-stack framework. That's being done by combining React components with existing back-end frameworks. When you use frameworks like these, there is usually an application (from npm) that lets you set up all the boiler plate code before you begin to edit it.

So many "languages"
You should already be able to recognise HTML, CSS and JavaScript. But React uses something called JSX. JSX is a JavaScript syntax extension that allows you to describe visual components in an HTML-like way, embedded within your JavaScript. Files written in JSX have the extension .js or .jsx.

TypeScript is a superset of JavaScript. It "transpiles" (translates/compiles) to JavaScript. Files written in TypeScript have the extension .tx or .tsx. TypeScript will "consume" JavaScript (i.e. if you wrote JavaScript, it would pass through the transpiler unchanged).

What do I need installed?
- node
- npm
- npx
- (nvm)

You'll need node and npm installed to use these things. Make sure your versions of these things are not ancient. Go with the most up-to-date version if you can (or the most recent stable version). You will also need npx. You can use npm to install npx. Ironically, npm and npx are actually both node packages. The scripts to run them (.ps1 or PowerShell scripts on Windows) simply call node and specify some JavaScript to run.

Remember, if you're having trouble with node and npm ("node package manager") installations and versions, then look at installing **nvm** ("node version manager"). DigitalOcean have a good resource on how to update Ubuntu's node version using nvm.

Creating your first Next.js App.
(Skip this section and move straight to React if you like)
We are going to use Next.js as a back-end framework, but, in the first instance, only to host our front-end. This very first example is just to show you how the default Next.js app looks. Make sure you have your terminal in the folder where you want to create your app. If you want to, you can set up a GitHub first ready to take the files.
We will create our app using:

```
npx create-next-app@latest
```

This will walk you through creating the most basic boiler-plate code for your Next.js project. Feel free to change the name, but leave everything else as defaults.

Now let’s see what it created. Browse to the folder where all the new files have been put and take a look. Lots of boilerplate and a node_modules folder.
Now start up the dev version of the server with

```
npm run dev
```

running in the terminal where package.json is. It might take a while to run. And use the browser to look at the page. This is your chance to show a little curiosity. See a thing, change a thing. We’ve only just started so you’ve nothing to lose. See if you can:

- Read the ReadMe file (it tells you the port to use)
- Change the favicon (the picture that appears on the browser tab)
- Change the words that appear on the browser tab
- Figure out if you need to restart the app when you change stuff (or just refresh the browser).
- Find the name of the function that represents the home page

Creating your first React (with Next.js) App

As above, there’s already an example to set up the boiler plate:

```
npx create-next-app --example with-react-bootstrap with-react-bootstrap-app
```

(yes, the spaces are correct: see create-next-app - npm for understanding). And then, to start the server (remember to be in the right folder – the folder will be called *with-react-bootstrap-app*):

```
npm run dev
```

This time, see if you can figure out:

- What you need to do to change the port of the server (Hint: editing the dev script in package.json might help).
- What you need to do to add a page to the front end (i.e. make localhost:3000/about work and show something you’ve written) (Hint: it might be just as easy as adding a file to pages: https://nextjs.org/docs/pages/building-your-application/routing/pages-and-layouts)
- Work out how to make a page consisting of components (Hint: you will need to add a folder called “components”

If you want a really, really basic React component or page, it looks like this:

```
export default function Footer() {
  return (
        <p>
         Footer
        </p>
);
}
```

If you are templating React components or pages, this is a minimal example. The default function name will be the same as the filename (but capitalised). In that example, a file called footer.jsx contains the above. There is no styling information included in that component, it simply returns a <p> formatted word, but it will let you see your component appear on the web page and it will let you see if you've made something work in terms of page routing or components.

Note on the file _app.jsx

If you want to make changes to the layout of every page in your website, _app.jsx is the place to do it. For example, changing _app.jsx to:

```
import "../style/index.css";
import Layout from '../components/layout'

export default function MyApp({ Component, pageProps }) {
  return (
    <Layout>
      <Component {...pageProps} />
    </Layout>
  )
}
```

Will apply a layout component across the whole website (i.e. every page in pages). You can define the "Layout" component as you like, but one example would be to add a Navbar and a Footer like this:

```
import Navbar from './navbar'
import Footer from './footer'

export default function Layout({ children }) {
  return (
    <>
      <Navbar />
      <main>{children}</main>
      <Footer />
    </>
  )
}
```

And assuming you have defined the Navbar and footer components, they will be added to the beginning and end of every page in the website. You could define the footer as in the example above, and the navbar as below:

```
import Link from 'next/link'

export default function Navbar() {
  return (
            <ul>
                <li>
                    <Link href="/">Home</Link>
                </li>
                <li>
                    <Link href="/about">About Us</Link>
                </li>
            </ul>
  );
}
```

You should now have a multi-page React web app built from pages and components and hosted with a Next.js backend. It’s using bootstrap css from within the styles folder in the app, and most of the backend Next.js stuff is effectively hidden. If you don’t have this but want to move on with learning some react, check out the “half_way_and_ugly_solution”.

Actual React in JSX (a Todo app)

We're going to edit index.jsx in the pages directory. There are a number of things to notice first. This file currently defines a page and returns a default function. This is a common way of using React called "Hooks". React *can* have a class and in some tutorials you will see pages which extend the React.Component class. The class defines a "render()" function which is where the JSX website code goes (the "View" if you're thinking about Model/View/Controller). However, these days, React is all about skipping the class definitions and using hooks. The general format for a React "hook" file is:

```
Imports (including state)
import React, { useState } from 'react';
export default function Myfunction(){
        const [variable, setVariable] = useState(…state type…);
        return(
                ….JSX code describing the website component and all related functions/stuff
        );
}
```

That means that all your functions for your component are defined within the exported component function itself. See here: https://react.dev/learn/responding-to-events for an example.

The code from here: https://react.dev/learn/updating-arrays-in-state can be modified to create a basic ToDo list (i.e. change some variable names). Work through the samples there to get a ToDo list that works for you. In the first instance, it only needs to be front end (it's all that tutorial covers). There is a solution file for this ToDo page on Moodle.

Now that it's up and running, you need to notice a couple of things. First what happens (to your todo list) when you refresh the page? What does that mean about where the code is running? If you're really curious, take a look at the "Network" tab in the developer tools and figure out whether the routing on your web page is front-end or back-end.

Do you wanna add a database?

In the interests of having a fully portable 3-tiered system where front-end, back-end and database can all be hosted on different servers, and because it's what you know already, we're going to add a ME_N back-end. That is, Node/Express and Mongo.

Backend

You will have noticed in your testing that your "todo" list vanishes when you refresh the page. There is no persistence because there is no back end. But we can use/modify the back end from last week to provide a database. (Yes, you could also modify the next.js backend if you would like to have a <n-tier system, but we'll build on what we have in the first instance)

You will need to create a separate backend. A .js file (server.js or index.js) that is run on node. You can base it on the backend from previous labs. It will need to be run in its own

terminal (along with a database in *its* own terminal). You can put it in its own GitHub if you like, but at the very least it should go in its own folder.

Some of the code from the "Adding a Database" lab (the "storeQuote" route in server.js) allowed you to connect to a database and use **insertOne()** to insert a document into a collection in a mongo database. In my example, I store both the number and the instruction in the todo list.
The following code (derived from https://www.mongodb.com/docs/drivers/node/current/usage-examples/find/) will allow you to query the database (assuming you have a collection called "todo" and a MongoClient called "client"). It needs to go in an api call in the backend server. It returns an array of the result as its response.

```
async function run() {
   try {
      const dbo = client.db("mydb");
      const query = {};
      const options = {
         sort: { todoNumber: 1  },
         projection: { todoNumber: 1, todoText: 1 },
      };

      const cursor = dbo.collection("todo").find(query, options);
      if ((await cursor.countDocuments) === 0) {
         console.log("No documents found!");
         response = ""
      }
      // prepare the response as an array
      const response = await cursor.toArray();
      res.send(response)
   } finally {
      await client.close();
   }
}
run().catch(console.dir);
```

You should consider your server API when you do this. If you were adopting a RESTful API, then you might use POST to add things to a database and GET to retrieve them. And you might create a new "todo" route to use a "todo" collection in "mydb" database. You will be able to run this with a local or remote server. You might also consider adding a DELETE route that would allow you to **deleteMany** (i.e. you can modify the code to use deleteMany in place of find()

To test the server routes you can craft the right url and use curl or Postman or Invoke-WebRequest (with PowerShell) to test the server. An example curl command (to post the todo number and text to the database) would be:
`curl -X POST http://127.0.0.1:8000/api/todolist?todoText="hello"\&todoNumber=1` but it will depend on your operating system, port number, url, api, arguments etc. The modified "Adding a Database" server code (for node) is available on Moodle, and this curl command will work (on a mac) with that. Alternatively, on Windows, the syntax is slightly different:

curl -method POST http://127.0.0.1:8000/api/todolist?todoText="hello&"todoNumber=1

Investigate alternatives to curl.

There is one final thing that you need to modify in the server: CORS. CORS stands for Cross Origin Resource Sharing and it is a security feature. Servers will respond to simple requests automatically, but more complicated requests require CORS. To enable CORS on your server, you can enable it on a specific route and for a specific website. For our example, you can add the following (after the declaration of your "todo" route if that's what you decided to call it (code from https://enable-cors.org/server_expressjs.html):

```
app.use("/api/todolist", function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "http://localhost:3000"); // update to match the
domain you will make the request from
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type,
Accept");
  res.header("Access-Control-Allow-Methods", "GET, POST, OPTIONS, PUT, DELETE");
  next();
});
```

If you forget to add this section of code, when you modify your front end React app, it will give a resource sharing error on the console. This is one place where an error on the browser console indicates an error (or, rather, a security feature) on the server. Note that the above code sets it up for localhost:3000 (which is where your React app will be running unless you've changed the ports). You might consider this as something you could use environment variables for. Note that you can also use * (wildcard) for these if you want to be really liberal with your CORS.
Notice that your CORS origin MUST be correct for the entire URL. It does distinguish between 127.0.0.1 and localhost (so if your browser is on localhost, your server should be on localhost).

There is a possible backend solution on Moodle. The CORS in this solution uses 127.0.0.1 (not localhost) and it uses port 8080. If you use it, also notice what parameters it expects from your url.

Frontend

Now, with the server API set up, you need to call your own backend API from within your new React front end. We're going to base this on this tutorial for adding Axios and React-query imports to the code and this blog for handling Axios requests. We're going to query a "remote API". The trick is that the "remote API" is your server (modified from Lab 3 that you've just prepared). To create the query, we'll use react-query and axios. You will need to install them in your node_modules and add them to package.json. This command should do both:

```
npm install -s react-query axios
```

You'll need to modify _app.js in the React app (pages directory), add the import (with the other imports):

```
import { QueryClient, QueryClientProvider } from "react-query";
```

add the const (with the other const – they go above the function)

```
const queryClient = new QueryClient();
```

and surround the <Layout /> code with <QueryClientProvider client={queryClient}> like this:

```
  <QueryClientProvider client={queryClient}>
    <Layout />
          <Component {…pageProps} />
    </Layout>
  </QueryClientProvider>,
```

That adds the necessary query client to the whole app which contains our custom component (the "todo" list which might be in index.js depending on whether you've put it in your main page or not).
In index.js (the file with your todo app), you'll need the following imports (with the other imports):

```
import {useQuery} from 'react-query'
import axios from 'axios'
```

You can either save the todo items one at a time, or create a button that will save a whole list to the database at once. We're going to do the latter. We want to add buttons that will store and retrieve the items from the database. Look for the JSX code (it's in the render() function) and you'll see it's similar to html. In here, you can add two buttons like this:

```
<button onClick={() => {

}}>Retrieve ToDo List</button>

<button onClick={() => {

}}>Save ToDo List</button>
```

Notice that we've left space for the onClick functions.

Saving to the Database

We'll start with the "Save" function (unless you want to use mongosh to add some todo items to your list).
The code for the "Save" function could look like this:

```
console.log("Storing items")
toDoList.forEach( element =>
{
  var requestURI = "http://127.0.0.1:8000/api/todolist?todoNumber=" + element.id +
"&todoText=" + element.name
  console.log(requestURI)
  axios.post(requestURI)
})
```

Which, ironically, goes through the toDoList state variable one item at a time and posts them to the serve (with axios) which, in turn, puts them into the database. You could, in theory, modify this code and combine it with the "Add" button so that the application simultaneously adds a todo item to both the state variable *and* the database. The way this code is implemented, it uploads the whole todo list every time. So if you save the same todo list more than once, both copies will be stored in the database. We can get around that using the "delete" functionality provided in server to delete any list before we save a new one. Call it using axios:

```
axios.delete("http://127.0.0.1:8000/api/todolist", { crossdomain: true }).then ((response) => {
   console.log("Storing items")
   //store all the items in here so you know you're not storing items that are going to get
deleted…
}
```

Note: You can (and probably should) remove the console.log() statements from the code.

You can now test the "Save" functionality with the database – make sure your server receives the data from the front-end and that it is written to the database (using mongosh…or you can proceed with the next section and debug it all at once).

Retrieving from the database

The response to onClick for the "Retrieve ToDoList" button should look something like this:

```
console.log("Getting ToDo list")
        axios.get('http://127.0.0.1:8000/api/todolist').then((response) => {
         console.log(response.data)

         // If I'd got the server response to be a perfect match for the react, I wouldn't need
this!
         function untidy_mapping(element) {
                  return{ id: Number(element.todoNumber), name: element.todoText, done:
         false };
         }
         const retrievedToDoList = response.data.map(untidy_mapping);
         setToDoList([
                  ...toDoList,
                  ...retrievedToDoList
     ]);
});
```

The solution code for this is more complicated than it could be if the variable names and response from the server exactly matched the variable names within the React code. My server returns (and expects) *todoNumber* and *todoText* as parameters, but the React state uses *id* and *name* instead.

A few things to notice about this code: it sets the toDoList state variable using the old version (nothing from the front-end is lost) plus the list retrieved from the server. There's a problem with this because the id fields might clash and you might get an error/warning telling you that id keys should be unique. You might consider using the database's _id field instead of the number you set in the front end, or trying to produce a unique key when generating ToDo items. You might also want to consider filtering todo items in some way:

my code has a parameter *done* which hasn't been fully implemented in this lab but it could be used as a way to remove items from the todo list (and eventually delete those items from the server).

Some notes on deprecated software

create-react-app used to be the app to create React apps. It still exists but it is no longer listed on React's official documentation. If you wanted to see how it worked, you can install it (globally) using:

```
npm install -g create-react-app
```

and then you can use:

```
create-react-app <myDirectoryName>
```

To create a react app (in some cases, you will call this script using **npx create-react-app <my-app>**. Note that there are some naming conventions for <myDirectoryName>. You cannot use capital letters. My first React project was called "hello-world". Start the app with:

```
npm start
```

in the appropriate directory and look at the web site it serves up. Play with it and see what you can change about it. See where all the files are (App.js will be important). There's a robots.txt file – do you know what that is for? The old React code required a bundler (WebPack, Babel) to produce "production ready" code, but using this app will still let you experiment with the front end components.
React also had a "react-router" which meant that your whole website was bundled into a single download and then most of the functionality was contained within the browser. Think of the advantages and disadvantages of this for the user. How will they perceive network speeds? If you website was a shop, how might this impact the user's basket while they shop and when they come to check out?