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 (or, alternatively, TypeScript). 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 (node -v and npm -v). 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 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 (questions below with my answers). Feel free to change the name, but leave everything else as defaults. You can try one with JSX and then create another with TypeScript, same with Tailwind CSS – ultimately your stack is up to you! It'll take a while to create because it's downloading a lot of stuff.

(my defaults)
What is your project named? my-app
Would you like to use TypeScript? No
Would you like to use ESLint? No
Would you like to use Tailwind CSS? No
Would you like your code inside a `src/` directory? No
Would you like to use App Router? (recommended) Yes
Would you like to use Turbopack for `next dev`? No
Would you like to customize the import alias (`@/*` by default)? No
*What import alias would you like configured? @/**

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 and repetition 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 every page in the website. You could define the footer as in the example above, and the navbar as below (although it is ugly – use vibe coding to generate a better one with css):

```
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 (or follow the instructions below to add the page to your existing React app). In the first instance, it only needs to be front end (it's all that tutorial covers). There is a very rough solution file for this ToDo page on Moodle (rename it "todo" and add it to the pages folder in your React app to make it do something).

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 headless 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 <2-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. If you use it, also notice what parameters it expects from your url (i.e. check the CORS).

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 will be somewhere in pages).
In todo.jsx (the file with your todo app – or whatever you've called it), 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 server (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 yet but it could be used as a way to remove items from the todo list (and delete those items from the server). **You might choose to completely redesign this code so that ToDo items are stored one at a time to both the database and to the screen (the betterTodo.jsx file shows how to do this).**

Remember: You are only one person. The database is only serving one person and keeping persistence between browser refreshes, but it could serve more than one person. Pair-up with someone and share a database (remember, it's just the URI that needs to be shared and a suitable hole poked in the firewall where the database is hosted – use the Ubuntu machines). Alternatively, have your web app open in two browser windows. What kind of issues arise with the ToDo list? What could be done to fix them?

ToDo list updating (deleting "Done" items)

So far, we can retrieve the Todo list from the database through our server, and add items to it (GET and POST). We can also DELETE the whole list from the database. We've got a button on the front end that can mark individual items as "Done", but it doesn't actually do anything on the back end. We need to add findOneAndDelete to our back end, and also edit the front end functionality so that the DONE button calls this api.

On the back end, the delete functionality could be modified to include

```
            var n = req.query.todoNumber
             var s = req.query.todoText
             console.log("deleting only one item "+ n +" "+ s)
             const query = {todoNumber: n, todoText: s };
             await dbo.collection("todo").findOneAndDelete(query, function(err, result) {
                if (err) throw err;
             });
```

(that is, the "delete" API call that you already have could be modified to take arguments just like the "add" API call, and these arguments used to delete from the database)

The front end needs:

```
var requestURI = "http://127.0.0.1:8000/api/todolist?todoNumber=" + toDoItem.id +
"&todoText=" + toDoItem.name
             console.log(requestURI)
             axios.delete(requestURI)
```

Added when the Done button is clicked (or you might decide to rename the button "Delete", or implement it in some other way).

Lab Functionality Extension

This is an optional part of the lab to give you a way of expanding your React development methodology.

We've already used API calls for POST, GET and DELETE. Is there a way we can PUT or PATCH? That is can we update our ToDo list items after they have been created? This might involve adding a third parameter to our database/api calls. You could call it the "Done" parameter and it might appear as a check box on the front end.

Moving Away from a "Headless Server"

Up until now, you have had a separate back end (Node and Express) which acts as a "headless" server. This is good practice, because that back end server acts as an interface to the code and could be used with any front end. The back end can run on a separate server. **Think of some reasons why this is good practice. Think of some reasons why it is annoying.**

You could design an app that would use your own API bundled in with Next. This is actually quite easy to do, but you need to follow the correct folder structure. This is what we do here.

First, create a folder called "api" within the "pages" folder in your Next (React) app. Then, create a .js file (call it todolist.js so that it will match with our existing API convention). We will use a RESTful API, so we need different http methods. Stub them in like this:

```
export default function handler(req, res) {
  if (req.method === 'POST') {
    // Process a POST request
  } else {
    // Handle any other HTTP method
  }
}
```

You will need the database stuff (just the same as from the original backend):

```
const { MongoClient } = require("mongodb");
//const uri = "mongodb://test:password@127.0.0.1:27017/mydb";
// Unsecured database
const uri = "mongodb://127.0.0.1:27017";
```

And you will need to copy across all of the API calls from the old backend (use JavaScript files so that they are just the same code, or else you will have to rewrite some parts in TypeScript). Just take the inner parts of the functions and copy them across to the correct part of todolist.js

Once that is complete, you can shut down your separate backend. On the front end (the .jsx files), you will need to change all the API calls to use port 3000 (the old backend ran on port 8000, but React runs on port 3000 by default). And you will need to run:

```
npm install -s mongodb
```

within the top React app folder (at the same level as package.json and where you usually do *run npx dev.*

The advantages to serving both the front end and the api from the same place are that you do not need CORS, so it is more secure. If you were using https, then you would only require one security certificate. The disadvantages? Your server now has to do a lot more and you cannot separate frontend and backend functionality. If you have a large number of users, or a particularly complex backend, this could be problematic.

Using a third party API (towards adding PayPal).

We're now going to add a page to our website that uses icanhazdadjoke this free API. You've just created your own API to let your frontend and backend communicate. It's now time to use a publicly available (and free) API. First, we want to create a page with a button that will take data from the third party API and display the returned data. You can add it to the "pages" folder of your current project. A lot of the necessary code you can lift from the other .jsx files created in this module. My axios request looks like this:

```
let options = {
  headers: {
     'User-Agent': 'Robert Gordon',
     'Accept': 'text/plain'
  }
}
axios.get('https://icanhazdadjoke.com/', options).then((response) => {
  console.log(response.data)
  setDadJoke(response.data)
});
```

The full code is available in a jsx file (dadjoke.jsx) from Moodle.
When you run it, it will work but you will see an error. The browser refuses to set the header "User-Agent" on the front end. This is a safety feature of browsers, but if you move the code to your own backend api call, this will not happen. So, go ahead and create a new file in the api folder (or in your own headless backend) that simply calls the dad jokes API and returns the response to the front end. Mine looks like this:

```
const axios = require('axios');

export default function handler(req, res) {
 if (req.method === 'GET') {
    console.log("getting the dad joke")
    var response = "";
    let options = {
      headers: {
        'User-Agent': 'Robert Gordon',
        'Accept': 'text/plain'
      }
    }

    async function run() {
      try {
        axios.get('https://icanhazdadjoke.com/', options).then((response) => {
          console.log(response.data)
          res.send(response.data)
        });
      } catch (error) {
        res.status(error.response?.status || 500).json({
          error: error.message,
```

```
                details: error.response?.data
            });
        }
    }
    run().catch(console.dir);
} else {
  // Handle any other HTTP method
  console.log("Unsupported HTTP method")
  res.send("Not supported")
}
}
```

You can then just replace the call to the 3rd party API with a call to your own server. This means that all calls to the 3rd party API go via your server ("backend").

Read this: javascript - How can I ensure my API is only called by my client? - Information Security Stack Exchange and consider the security implications of having the API call on the front end or on the back end. This API is free, but if you utilised at alternative API (e.g. one of the LLMs), you make have your own token and pay for accesses using this token.

Extension: The next task is using PayPal's API. This API needs to be called on the backend - you authenticate the web app with PayPal in order to pay the money into the businesses' paypal accounts. If you want to test your skills, consider moving the API calls in this exercise from the front to the backend.

Actually using PayPal's API

The last task of this lengthy lab is to add some sort of "check out" system to your React front end. We use PayPal's developer tools: Get Started with PayPal REST APIs and Paypal's developer Dashboard. Go ahead and sign up if you're comfortable doing it.

We will be following this How to Integrate PayPal Standard Checkout tutorial with some differences:

- we will integrate the checkout into our React app (i.e. any backend api files will be in the pages/api directory of our React app). If you like, you could also integrate it into the headless server that we used earlier in this lab.
- We will use axios for our http calls
- We will store our tokens and keys directly in the code (it would be better practice to store them in a .env file, though)

Our checkout will be very basic – initially it is just a button that pays a pre-determined amount (set on the backend). Eventually, we will just need some sort of way to calculate a price followed by the necessary PayPal call. Feel free to ask a large language model to generate a React .jsx file that will display some boxes and calculate a price. My stub pay.jsx file is on Moodle (8_pay.jsx) but we don't use it initially.

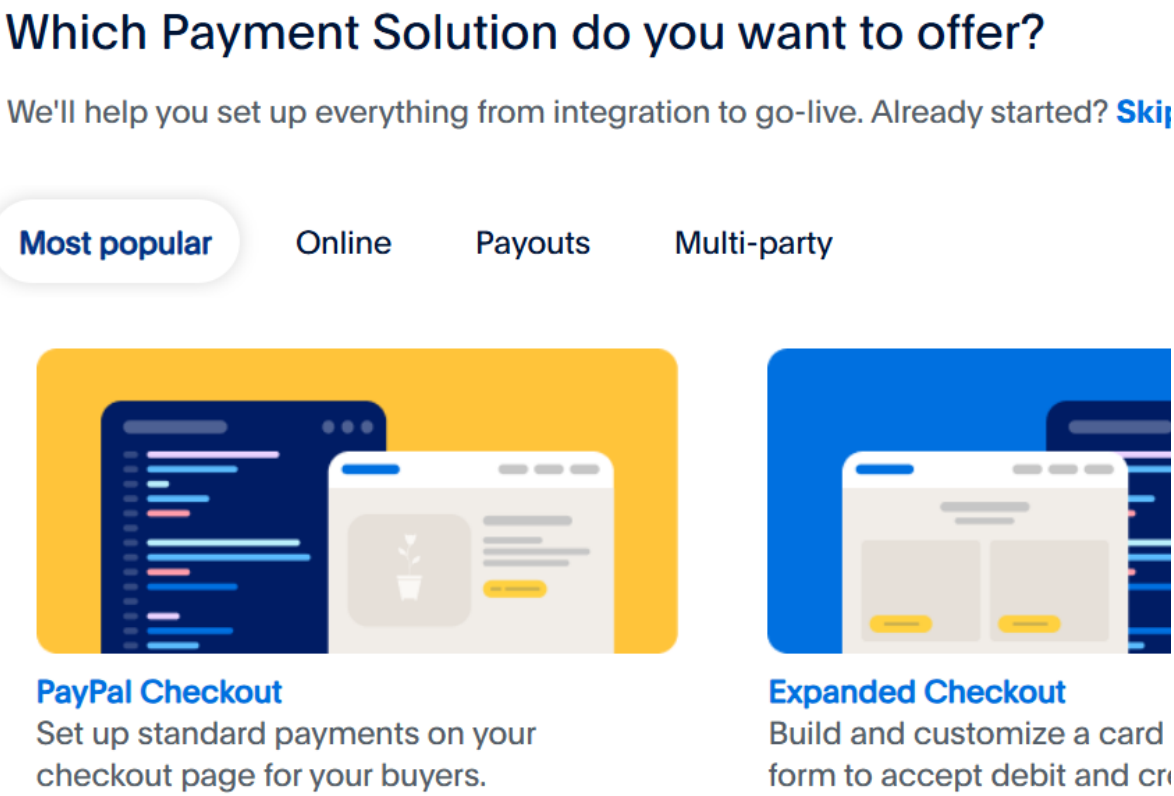
The Paypal order workflow goes like this:

- Load the buttons

- Start the order on the front-end
- Create the order on the back-end (including an oauth token so that the correct paypal account is contacted)
- loads the Paypal pop-up on the front-end (handled automatically if you create the order correctly)
- The user fills in the paypal form and “completes” the order on the front end
- Passed to the back-end which handles order completion and communicates back with the front-end.

Getting a Paypal developer account.

If you want to, you can create a new PayPal account to do this. That way you can be sure it does not have access to your actual money. This lets you try out multiple things with the API but you would need a proper account if it were to go *live* and take actual money. Sign up and log in to the developer Dashboard. Go to “Home”



And select PayPal Checkout (the most popular).
You will be able to create your Client_ID and your Client_Secret on this Dashboard (or view ones that you have already created. While you’re looking at the developer dashboard, see if you can find where you will see any Paypal interactions (the logs or debug logs).

Paypal Resources

Paypal’s developer site gives a variety of resources. For this tutorial, we will be loosely following: this video: How to Integrate PayPal Standard Checkout and the associated GitHub: PayPal-Standard-Checkout-Tutorial/script.js at main · paypaldev/PayPal-Standard-Checkout-Tutorial. And this tutorial PayPal Community Blog | How to add PayPal checkout payments to your React app and the code from this GitHub PayPal-React-FullStack-

Standard-Checkout-Sample/package.json at main · paypaldev/PayPal-React-FullStack-Standard-Checkout-Sample. The tutorials are written using Node, and we'll be adding our backend .js API file and front end .jsx on to our Next.js web app (for convenience – you could create a new one if you like)

To use the Paypal API, you need to install the correct package (using the -s option to save it to packages.json). We're using React, so we will use the react paypal package:

```
npm install -s @paypal/react-paypal-js
```

The Paypal front end implements some buttons with over-ridable callback functions.

Paypal frontend

First, we will put together the most basic shop. It is literally a single .jsx file with a header and some paypal buttons. The code looks like this:

```
import React, { useState } from "react";
import { PayPalScriptProvider, PayPalButtons } from "@paypal/react-paypal-js";

// Renders errors or successfull transactions on the screen.
function Message({ content }) {
  return <p>{content}</p>;
}

function paypalskeleton() {
  const initialOptions = {
    "client-id": "YOUR_CLIENT_ID",
    "data-sdk-integration-source": "integrationbuilder_sc",
  };

  const [message, setMessage] = useState("");

  return (
    <div className="App">
      <h1> Donate £1 </h1>
      <PayPalScriptProvider options={initialOptions}>
        <PayPalButtons
          style={{
            shape: "rect",
            //color:'blue' change the default color of the buttons
            layout: "vertical", //default value. Can be changed to horizontal
          }}
        />
      </PayPalScriptProvider>
      <Message content={message} />
```

```
    </div>
  );
}

export default paypalskeleton;
```

If you spin it up as a .jsx file and add it to your React app, you will not see any Paypal buttons. There is something in that code called YOUR_CLIENT_ID. Replace it with your own Client_Id from the React developer dashboard (you get a client id and a client_secret on the Dashboard, but the secret is only for the back end).

The first over-ride we require for the buttons is “createOrder” and it’s defined below:

```
createOrder={async () => {
          console.log("creating an order")
          try {
            const response = await fetch("/api/pay", {
              method: "POST",
              headers: {
                "Content-Type": "application/json",
              },
              // use the "body" param to optionally pass additional order information
              // like product ids and quantities
              body: JSON.stringify({
                cart: [
                  {
                    id: "YOUR_PRODUCT_ID",
                    quantity: "YOUR_PRODUCT_QUANTITY",
                  },
                ],
              }),
            });

            const orderData = await response.json();
            if (orderData.id) {
              return orderData.id;
            } else {
              const errorDetail = orderData?.details?.[0];
              const errorMessage = errorDetail
                ? `${errorDetail.issue} ${errorDetail.description} (${orderData.debug_id})`
                : JSON.stringify(orderData);
              throw new Error(errorMessage);
            }
          } catch (error) {
            console.error(error);
            setMessage(`Could not initiate PayPal Checkout...${error}`);
          }
```

```
        }}
```

Notice that it makes a call to /api/pay. That means that you need to implement the backend part before it will actually work.

Paypal backend

Following on from our front-end development, we need to create a file called *pay.js* to live in the api folder within the pages folder on our React app. We're going to deviate very slightly from the software architecture that the paypal tutorials use and we're going to put API calls for both capturing and completing the order in the same pay.js file. It is not necessarily best practice but it helps to minimise some repetition within the code.

Pay.js should start with some declarations:

```
const axios = require('axios');
const environment = "sandbox"
const client_id = "my_client_id_from_paypals_developer_dashboard"
const client_secret = "my_client_secret_from_paypals_developer_dashboard"
const endpoint_url = 'https://api-m.sandbox.paypal.com'
```

The client_id should be the same as the one you used in your front end, and the client_secret should be from your dashboard (it's a massive long string).

Paypal uses Oauth2 for secure logins, so you need an oauth token. The function to do this lives just below the declarations in pay.js and looks like this:

```
const get_access_token = async() =>{
    const auth = Buffer.from(client_id + ":" + client_secret,).toString("base64");
    const data = 'grant_type=client_credentials'
    let options = {
        headers: {
            'Authorization': `Basic ${auth}` // from paypal tutorial
        }
    }
    return (axios.post(endpoint_url+ '/v1/oauth2/token', data, options)
            .then(response => {
                console.log("Our paypal endpoint responded")
                console.log(response.data)
                return response.data.access_token;
            })
            .catch(err => {
                console.error(err);
                return err;
            })
    );
}
```

The first part of the exported api function for pay.js looks like this:

```
/// The basic call to PayPal: Everything is $1.00
export default async function handler(req, res) {
```

```
    if (req.method === 'POST') {
        console.log("POST /api/pay/ means we're creating the order")
        const access_token = await get_access_token();
        const order_data = {
            intent: "CAPTURE",
            purchase_units: [{
                amount: {
                    currency_code: "USD",
                    value: "1.00", ///These are fixed server-side. We'll talk about getting the dangers
of getting the price from the client side, too.
                }
            }]
        };
        const options = {
            headers: {
                'Content-Type': 'application/json',
                'Authorization': `Bearer ${access_token}`
            }
        };

        return (
            axios.post(
                `${endpoint_url}/v2/checkout/orders`,
                order_data,
                options
            )
            .then(response => {
                console.log("Axios got back a reponse")
                console.log(response)
                res.send(response.data);
            })
            .catch(err => {
                console.log("Axios request got back an error")
                console.log(err)
                console.error(err);

                res.status(500).send(err?.response?.data || { error: 'Order creation failed' });
            })
        );
    }
//// We can put other API route types in here.

 else {
    // Handle any other HTTP method
    console.log("Unsupported HTTP method")
    res.status(404).send("Not supported")
    }
```

```
}
```

Take a moment and work through that code. It gets an oauth access token, then it declares an order (including price), and some options for the headers for that Paypal API call. Then it uses axios to make the call to PayPal and handles the result (i.e. passes it back to the front-end). Add some console logs and connect it up to the front end so that you can see what it is doing. Try to run the front-end and back-end together so that you can see the Paypal pop up appear, and take a look at the Paypal developer dashboard so you can see what is created.

Now we can create the order, it is time to complete the order. Have a look on your Paypal developer dashboard (Testing tools) and **find the sandbox email and password for your business application** and your "personal" email account. You are going to use the personal account to pay on our website. When the paypal popup opens, use the personal account details to log in to Paypal.

Paypal – completing the transaction

In your web app, you should now be able to click the PayPal button, which then opens up a pop-up which you then fill with your sandbox dashboard (personal email) details. Now you should be able to "complete purchase" and have your app respond to that. To do this, we need to over-ride the button's onApprove function on the front-end. The over-ride lives at the same level as the onCreate function and looks like this:

```
onApprove={async (data, actions) => {
          console.log("onApprove")
          try {
            const response = await fetch(
              `/api/pay?order_id=${data.orderID}&intent=capture`, /// This is rough
              {
                method: "Put",
                headers: {
                  "Content-Type": "application/json",
                },
              },
            );

            const orderData = await response.json();
            // Three cases to handle:
            //   (1) Recoverable INSTRUMENT_DECLINED -> call actions.restart()
            //   (2) Other non-recoverable errors -> Show a failure message
            //   (3) Successful transaction -> Show confirmation or thank you message

            const errorDetail = orderData?.details?.[0];

            if (errorDetail?.issue === "INSTRUMENT_DECLINED") {
              // (1) Recoverable INSTRUMENT_DECLINED -> call actions.restart()
              // recoverable state, per
https://developer.paypal.com/docs/checkout/standard/customize/handle-funding-failures/
              return actions.restart();
            } else if (errorDetail) {
```

```
            // (2) Other non-recoverable errors -> Show a failure message
            throw new Error(
              `${errorDetail.description} (${orderData.debug_id})`,
            );
          } else {
            // (3) Successful transaction -> Show confirmation or thank you message
            // Or go to another URL:  actions.redirect('thank_you.html');
            const transaction =
              orderData.purchase_units[0].payments.captures[0];
            setMessage(
              `Transaction ${transaction.status}: ${transaction.id}. See console for all available
details`,
            );
            console.log(
              "Capture result",
              orderData,
              JSON.stringify(orderData, null, 2),
            );
          }
        } catch (error) {
          console.error(error);
          setMessage(
            `Sorry, your transaction could not be processed...${error}`,
          );
        }
      }}
```

A few things to notice about this. Again it calls an api on the backend using axios. In this case, it is a PUT call to the same API (/api/pay). We'll write that functionality in a minute. In our case, it is using query parameters in the uri (marked with "This is rough" in the code there because it *is* a rough way of passing information back and forth, but it is the easiest way for us to do it). The last thing this code does is set some text that is visible in the browser to say the order is complete and has been successful. There is also some error handling to inform the user if the backend returns an error.

Our backend now needs to handle the calls to the PayPal API. The code to do this looks like this:

```
if (req.method === 'PUT') {
   console.log("PUT /api/pay/ means we're completing the order") ///YES, the PayPal tutorial
does it differently with two different api routes.
      const access_token = await get_access_token();
      const options = {
         headers: {
            'Content-Type': 'application/json',
            'Authorization': `Bearer ${access_token}`
         }
      };
```

```
    console.log("Here are the parameters")
    console.log(req.query)
    return (
        axios.post(
            `${endpoint_url}/v2/checkout/orders/${req.query.order_id}/${req.query.intent}`,
            {}, // Empty body
            options
        )
        .then(response => {
            console.log(response.data);
            res.send(response.data);
        })
        .catch(err => {
            console.error(err);
            res.status(500).send(err?.response?.data || { error: 'An error occurred' });
        })
    );
}
```

Just as in the function for order creation, this code begins by getting an oauth access token to log in to the PayPal API which it then puts in the headers of subsequent queries. It passes a call to the PayPal API and then it passes PayPal API's response back to the front end. It is fairly minimal, and there are probably a few other things that should be added if this were to be an actual "shop" (like stock-handling, dispatch and databases), but this allows an individual to transfer money from their PayPal account to a businesses' PayPal account.

The last thing to do is to check that the transactions work. The user interface should return a short statement to say that the transaction was successful, and the PayPal sandbox dashboard will show you what emails have been sent out, as well as show a log of the transactions.

The complete files (pay.js and paypaltest.jsx) are available on the Moodle page, but remember, you will need to put your own client_id and client_secret in before the PayPal API calls will work.

PayPal – what do we NOT do?

These are the things that we do differently from the (PayPal) documentation online:

- Next backend in JavaScript rather than TypeScript
- Axios rather than fetch (there are some debates as to which is better, but axios has built in json handling which makes our codebase smaller)
- Query parameters instead of full RESTful API calls to our own backend

These are the things that we'd need to do to make our app fully functional:

- Have a way of calculating the price (this would involve maintaining a "basket" on the front end and converting "items" into "prices" on the back end)
- Have (and maintain) an inventory database (item name, number of items in stock, item price etc.)
- Record sales and delivery details (in a database and then add a user interface on the front so our warehouse staff could do dispatches)

- Have user accounts or guest user checkouts
- Test the credit card functionality using some of PayPal's synthetic cards
- Move from Sandbox to a full business account and test again
- Test your app with other people's email addresses (PayPal might not permit this).
- Replace our query parameters with a better API.

Next Steps

You can now create a React app and maintain both a separate (headless) server and a Next.js API that attaches to the React app. Our overall app is a bit of a "dogsbody" application – it is simply a sandbox of pages where you try out your skills. But you can now think of your own app and try to implement that. Consider the functionality that you will need and what you still do not know how to do.

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. This routing is done by Next.js in our lab. Think of the advantages and disadvantages of this for the user. How will they perceive network speeds? If your website was a shop, how might this impact the user's basket while they shop and when they come to check out?