Servers

In order to communicate with other applications or services, we need to create a server. These are standalone files, separate from our frontend, that we will run and deploy.

It's extremely common for companies to use another language to host their backend, rather than continue to use JavaScript. This has some advantages:

- Many large companies have had these backend infrastructures for a long time, even before modern JavaScript frameworks were created.
- Java, C#, and Go are usually more efficient than JavaScript for processing speed.
- These backend languages typically also have built-in security features like type-checking which make them more reliable for systems like banking and healthcare.

However, we're going to continue to use JavaScript for our backend, since it would be confusing to try to learn JavaScript and something like C# at the same time. Just keep in mind that you may have to learn a bit of another language for the backend if you work at a large company.

But first, what is Express?

For JavaScript backends, we're going to use a library called `Express` to create our server.

- Express is one of several modules that allow for easy construction of HTTP servers.
- They can be run on multiple different platforms including locally on your own machine.
- Incredibly robust and allows you to serve HTML files to be rendered client-side, or render files server-side.
- Also allows for interaction with databases.

So how do we get started?

- There are a ton of options for setting up the server, but the bare minimum is something like this:

```
// Imports the Express library and sets it to a variable called 'express'
import express from 'express';

// Creates a new instance of Express and stores it in our 'app' variable.
const app = express();

// Sets up a variable that holds what port our server is on (not required, but can be useful)
const port = 3000;

// When a GET request is made to the root of the server, it executes a given function
// The function takes two arguments: req (request) and res (response)
app.get("/", (req, res) => res.send('Hello World!'));

// Starts the server and makes it listen on the port.
// You can also give it a function to run when the server starts, but not
```

```
required.
app.listen(port, () => console.log(`Example app listening on port ${port}!`));
```

- So that's great if you want a server to just say `Hello World!`. If you want it to do more, we'll have to start looking at more options.

Basic App Configuration

- You can set up `app.use()` to declare specific things you want to use in the whole app. This can be for routes (see below), middleware (see below), specific modules etc.
- If you're using the express generator you'll probably something stuff like:

  ```
  app.use(cookieParser());
  ```

- The above is telling the entirety of the application to use a library called 'cookieParser' when the server is running.

So what if you want all routes for `/users` to go through an authorization process of some sorts ahead of time, or some other functionality?

Middleware:

- Sometimes you want to make custom functionality. Let's say you want to take every request to the servers `/users` routes and check to see if a token/api key matched
- You COULD declare that function at each call inside the callback. But instead you could declare it as a standalone function:

  ```
  function checkAuth(req, res, next) {
    if (req.whateverTokenKeyYouWant) {
      next();
    } else {
      res.send("Invalid token");
    }
  }
  ```

- In this case, `next()` tells the server that this middleware is done and to move on to the next one. If you don't send a response in a route or exit out of a middleware with `next()`, the server will NEVER conclude the call (well, it will time out eventually).
- In your routes you can now simply call `router.get('/route', checkAuth, (req,res,next)=>{})` and it will automatically call that middleware.
- Alternatively you could use the `router.all` or `app.all` and attach middleware to ALL routes or all http methods to a specific route (`router.all('/users', middleware, callback)` would attach to any `GET`, `POST`, `PUT`, `PATCH`, or `DELETE` to `/users`)

Starting the server

To start the server, the easiest way is to run `node SERVER_FILE_NAME` in the terminal. Since it is a node app, it will start the server. That method, however, requires you to restart the server every time something changes in the files. You can also utilize the `nodemon` package if you'd like. Install it with `npm i nodemon` and then setup a script in your `package.json` to start up with `nodemon SERVER_FILE_NAME` instead of `node` that way as you save and update files, it will automatically restart for you!