Forms Extended Notes

Forms Outline

My first thought on this day is that the trie stuff might be overdoing it. I haven't had much experience with them though, and they look really interesting (reminds me of my Comp Sci days). I'm happy to go with it though, and this is more just my initial feeling here. It will depend a lot on the students and where they are at with things so far. It's a great schedule, but packed, and I wonder if they might be a little fried by this time.

With forms I would build out a very basic web form in HTML5. Along with this would be a discussion about HTML, and a little about browsers, parsing, and web communication. I'd keep this relatively brief.

I feel like with forms the emphasis should start with trying to push them to get over the fiddly nature of things at the outset. They have some 'annoying' elements, and various oddities that make them a bit of a hassle (hence dubious tools like Form.IO). I would overcome this by part explanation, and part throwing the task back on them to make a few forms, and to get a sense of the building involved.

Then we would need to chat about where that data is going. This leads to a chat about servers and more on web communication. I think it's worth a brief discussion of the various ways of communicating between browser and server (url, query strings, forms, fetch).

Here I would make a very simple web server using express. The absolute basics. In here I would have them send data from the forms, and then console.log out the various incoming data elements.

Pathway

- HTML5 forms
    - different form types
    - different methods
- HTML5 validations
- Connecting this to the back end, and responses
- Exercise around front to back
- Summarise all the methods of sending data from front to back
- Dealing with data, pre DB
- Make a small app without a DB

Forms

We have so far learned one ways that the client (browser) can send some useful information to the back end, which consists of two elements send through the URL: a. The path: the extra bit after the domain, and we have seen that we use this in the server as routes. `http://my.site.com/this-bit-here/` b. Query strings: the bit after the path, and we can usse this to send some extra data even with a `GET` request. `http://my.site.com/path?first-key=first-value&second-key=second-value`

Everyone has used forms (and often been frustrated by them I imagine). This is out *second* way to send some data to the server from the browser (with the URL being one way with a few internal methods).

One caveat that I will put here with forms, is that they are fiddly. They have some very idiosyncratic behaviours, largely due to the history of forms over time, and they can be somewhat annoying to deal with on occasion. What I would suggest is that you see this as a challenge to be overcome quickly, as they are a big feature of web programming.

As might be becoming clear, the whole game here is sending something from the server, having the user see this and react to it, and in doing so they send some information via a request to the server, and we react and send a response back to the browser, and repeat. Each request and response is a discrete interaction that we have discretion to use on the backend to present user with the next state.

Forms are a way to group a few pieces of data in the one hit.

The key new piece of information is the type of request we make from a form (usually), which is a `POST` request. The main difference between a GET request and a POST request, is that you can attach data to the body of the post request. This means that the format for the request is the same, but you have the option of a body with data after the mandatory blank like to separate the metadata.

Before we talk about request types in more detail, let's get things going, and make a form, submit it to the back end, and then see what we find out.

```
<form action="/my-handling-form-page" method="post">
    <ul>
        <li>
            <label for="name">Name:</label>
            <input type="text" id="name" name="user_name" value="by
default this element is filled with this text">
        </li>
        <li>
            <label for="mail">E-mail:</label>
            <input type="email" id="mail" name="user_email">
        </li>
        <li>
            <label for="msg">Message:</label>
            <textarea id="msg" name="user_message"></textarea>
        </li>
    </ul>
    <div class="preference">
        <label for="cheese">Do you like cheese?</label>
        <input type="checkbox" name="cheese" id="cheese">
    </div>
    <div class="button">
        <button type="submit">Send your message</button>
    </div>
</form>
```

As you can see, there is a bit going on here. Let's break it down:

We have used some `li`s in a `ul`, and some `div`s. We can use what we like in the form, and we will be able to style them as we choose too. You will see shortly that they come with their own styling by default.

The big new things are `label` and `input`.

Input

The `input`, as you'd expect, is the driver here. This is where the user takes charge and has the ability to choose their response. These create the boxes on the screen, or the checkbox, or whathaveyou.

The most important element in the `input` is the `type`. This specifies what type of input we are expecting - here we have instances of `text`, `email`, `checkbox`, and the special instance of `submit`. It's merely referring to what type of thing the user will be presented with.

The next attribute of an `input` to discuss is the `name`. This is the thing that will be used as the *key* for that piece of data. Let's leave that there for now, and talk more about that when we hit the backend.

The `value` attribute is a way to set the data content of that input. You can see in our one that this prefills that box with the string that we used to set value. This can change if the user chooses to override the value that we give by default. Often we can use this attribute to give the user some indication about what should be entered into this field.

The `id` is just the `id`, for CSS and JS purposes. And for the `label`, as we will see..

Label

The `label` is a bit interesting. It seems a little old school, and you might wonder why have something specific to attach to the input - why not just use a `div`? Same effect, no? Well.. no. In defining a `label` and specifying which `input` it is attached to, you are connecting them programmatically, and this has the following effects: increased click area to focus the `input`; better for screenreaders; and better for assistive technology. The connection is made through the `id` attribute.

You can read about more of the details here:

https://developer.mozilla.org/en-US/docs/Web/HTML/Element/label

You can see a few quirks here too. I want to discuss these, but not get caught up in them. They are just part of what you will need to get used to when dealing with forms, and sometimes may not have any real rhyme or reason. They are just things to get used to.

The first is that an `<input>` tag is an empty element, meaning that it doesn't take a `</>` closing tag. This is different from `<textarea>`. This has an impact on a specific feature of forms: the way you define the default value. To define the default value of an `<input>` element you have to use the value attribute, but the default for `textarea` is provided between the tags.

The `<button>` element also accepts a type attribute — this accepts one of three values: submit, reset, or button.

- A click on a submit button (the default value) sends the form's data to the web page defined by the action attribute of the `<form>` element.
- A click on a reset button resets all the form widgets to their default value immediately. From a UX point of view, this is considered bad practice, so you should avoid using this type of button unless you really have a good reason to include one.

- A click on a button button does... nothing! That sounds silly, but it's amazingly useful for building custom buttons — you can define their chosen functionality with JavaScript.

Form

All of the above is wrapped in the `<form>` tag, which has two elements:

- The first is the action, that tells the browser which path to take, and will be matched by us on the backend in our API.
- The second is the method, which here is `POST`. This is as opposed to the requests that we have been making so far, which have all been `GET` requests. We will talk a lot more about this shortly.

Some of this content is taken from here:

https://developer.mozilla.org/en-US/docs/Learn/Forms/Your_first_form

But please don't try to learn everything about forms from a document. There are too many little quirks, and it's mostly not a good use of time. Better to make a few forms and look up the docs when something isn't clear, or you need to check a detail here or there. As devs we will always have little things here or there that we need to double check, or that you haven't remembered. That's not to say don't commit these things to memory! Just that it's not a requirement of being a dev, and it's absolutely fine and expected to google many things during you day. Some things will become second nature, but if your job is interesting, expect Google and Stack Overflow to be your friends.

This is getting to be too much preamble. Let's see the results of our efforts on the backend. But how?

We know how to make an HTML file, and we know how to see it using our Live Server from VSCode. We have since made a server that can respond to requests itself, so let's get in there, and make a route temporarily to serve our form.

```
const express = require('express')
const app = express()
const port = 3000

app.get('/our-form', (req, res) => {
  res.send(`
        <form action="/my-handling-form-page" method="post">
            <ul>
                <li>
                    <label for="name">Name:</label>
                    <input type="text" id="name" name="user_name"
value="by default this element is filled with this text">
                </li>
                <li>
                    <label for="mail">E-mail:</label>
                    <input type="email" id="mail" name="user_email">
                </li>
                <li>
                    <label for="msg">Message:</label>
                    <textarea id="msg" name="user_message"></textarea>
                </li>
```

```
            </ul>
            <div class="preference">
                <label for="cheese">Do you like cheese?</label>
                <input type="checkbox" name="cheese" id="cheese">
            </div>
            <div class="button">
                <button type="submit">Send your message</button>
            </div>
        </form>
        `
    )
})

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`)
})
```

Let's remember a few important things about our app while we do this.

Firstly, the paths that we use as the first arguments to our `.get()` functions are arbitrary, meaning that we define them to suit our needs. I'm calling the route to serve the form `/our-form` just to be descriptive.

The second thing is that our code is getting rather unwieldy. We will be fixing this, but for now we are going to have to cope while we get our heads around what is going on. We also see a way to make a multiple line string here in JS (Node) with the ´ character.

Now we need to hit that route, so lets go to the localhost at port 3000, and to the route we just made: `http://localhost:3000/our-form`.

We can see here that we are getting our form on the front end. This is a reminder that HTML is just a string, and the parser in the browser does the work to put the elements on the page for us. It's a bit ugly on the code side, but we are going to fix this soon.

I'm going to fill in the form and attempt to send it, and let's see what happens. I am going to put `first`, `second@email.com`, and `third` into the text spots, and check the checkbox. Now submit..

It complains at you! This is because HTML5 has the ability to validate the input, and because we asked for an email, it is checking that we are trying to submit a value that has the format of an email. Validation is going to become very very important with the database we will set up soon, and this is a first taste. So we will change that to `second@email.com`. Submit..

Ok. Now we are getting somewhere. But where? You should see what amounts to a error mesage: `Cannot POST /my-handling-form-page`. There's a little bit going on here, but we are going to fix it first, and then we will go into the explanation.

It's telling us that we need a `POST` route in our backend app, and it's reminding us of what that route needs to match, and that's `/my-handling-form-page`. So we need to get into the backend and sort it out.

```
const express = require('express')
const app = express()
const port = 3000

app.get('/our-form', (req, res) => {
  res.send(`
        <form action="/my-handling-form-page" method="post">
            ...
        </form>
        `
    )
})

app.post('/my-handling-form-page', (req, res) => {
    res.send('Sending something back from the form submition route')
})

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`)
})
```

I've removed some of the form for clarity, and added the `POST` route. Express gives that power to us in the app object where we have a `.post` method that looks really similar to our `.get` method. It again takes two arguments (at least), the first of which is a string, and the second is our callback, where we are given the `res` and `req` objects. (A reminder: these objects are provided to us, and we don't get any say in what they are - but we could name them anything: `cat`, `dog`, whatever. But because these are the JS versions of the HTTP request and response objects, we find the more useful names `req` and `res`.)

HTTP Methods (REST), and CRUD

Let's get past the issues around GET and POST here and now. We have talked about our form sending a POST request. We have talked about our API and its `.post` method to accept this `POST` request, and now we need a bit of theory.

This involves two concepts that are two-sides of the same coin.

REST or RESTful API design (Representational State Transfer) is paradigm of API design that uses the HTTP methods: GET, POST, PUT/PATCH, and DELETE. It's a way of using the existing HTTP protocols to create an API that conforms to a particular way of interacting. That is a somewhat a confusing starting definition, and like everything, it's better to see it in action. It's also a somewhat slippery definition, but we can cover the basics and then talk more about this down the track.

It is tied to CRUD, which are the operations that you would run on a DB. The first is create, to make a new entry in the DB. The second is READ, which is to access an entry in the DB. The last two are updating and deleting. CRUD.

The way they are tied to being RESTful, as it's termed, is that you use a GET request to read from the database, you use a POST request to add an item to the database, a PUT/PATCH for an update, and a DELETE request to delete an entry.

We are yet to create a DB, so this relationship is just a wishlist for us at the moment, and we will be talking a lot more about this when we have a DB. But we have made a GET request, and we are currently making a POST request via this form, and so we need to start thinking in these terms now, even if it is a little confusing.

As you can see, it's a convention, a paradigm, and so it's something to be aimed at, but it is not strictly enforced. But it is strongly recommended that you follow this unless you have a very good reason not to. You will get used to this as we work through things over the coming days.

Hang on though, API?

API is Application Programming Interface, which is a slightly jargony way of saying that it's a digital place that you can get things from. As a result, this broad definition covers pretty much anything on the internet, and you will hear the same concept used in a variety of circumstances. In a more narrow version though, it's what we are building. We are making an API, and someone could some to our API, hit our routes, and get something in return. And when we make this API, we will be following convention and making it in a RESTful manner.

Back to the form..

Now we go back to the form and submit. We see that we are now getting our response sent back to the browser. This is good! But where is the data that we sent? This is where we will start getting a look at the request object that Express is dishing up.

But first we need ti get an overview of Middleware.

Middleware

Express does a lot for us, but there is still a need to customise things somewhat. Let's add a line to our code and discuss it. (You can see that I have stripped out some lines of code here to concentrate on the lines we are discussing.)

```
...
const port = 3000

app.use(express.urlencoded({ extended: true })) // added

app.get('/our-form', (req, res) => {
...
```

Middleware is basically any processing that we want to do after the request is recieved in our API, and before the response is sent back. With everything we are doing on the backend here, we are in broad terms merely performing that action: recieve the request, deal with it in some way - which can be very minimal, or be more involved - and then send back a response, which again can be a short string, or can be a full webpage, or can be GBs of data. Request, then response.

Middleware is the term for the things we do in between the request and response, and here we have two examples.

Let's talk about the JS, so we don't lose sight of that.

We can see again that `app` is an object because of the `.` that follows it. We can see that `.use` is a method or function that is attached to that `app` object. In both cases we can see that it takes at least one argument. In the first instance we can see that its argument is `express.urlencoded({ extended: true })`. What is this? Well the snippet of code here is a function *call*. We know that every function returns something when called, but we don't know what that is in this instance. We could look in the docs, and we might do at some point, but that's getting a little deep in the weeds.

We can see that the function being called is a method on the express object, and that it is taking one argument (in this instance), which is a simple object. The object looks to be configuring something in the function, but that's speculative. When `express.urlencoded({ extended: true })` returns, that value is then passed to the `app.use()` function call.

What we know is that this function is being used in Express to parse the incoming request, and as a result make the parts of it available to us in JS. The URL encoding refers to the way that the data in the body will be presented, and how then to parse that as a result.

Exercise 1

Given that we know a bit about URLs, and if I tell you that the data in the body is going to resemble a query string when sent by the browser, please attempt to write up what you think this POST request will look like leaving the browser.

POST Request answer

The Req Object, and the Body

Let's put a `console.log` in the callback function for our route that receives the form submission, and then have a look at the object in the server logs.

```
console.log(req)
```

We will see that it's absolutely enormous. There are a few bits and pieces in there I could explain, but there is also many many things that I can't even start to imagine what they are doing. It's worth a quick scroll just to have a look about. But let's look at the data we need:

```
console.log(req.body)
```

We can see the result in the server logs:

```
{ user_name: 'first',
  user_email: 'second@email.com',
  user_message: 'third',
  cheese: 'on' }
```

It seems to be a JS object, so let's see if we can dig further:

```
console.log(req.body.user_email)
```

And we get back `'second@email.com'`.

The data that was sent by the browser as a string we now are able to manipulate using the JS that we are now familiar with. Now that we have the power, let's react to the incoming request by sending back the email that was sent through.

```
app.post('/my-handling-form-page', (req, res) => {
    res.send(req.body.user_email)
})
```

We can see that there is no issue here, because HTML is just a string, and we are sending a string back.

Change the email, and send it again.

We are now getting a bit closer to a working app, even if it is a simple little thing at the moment. We could do something like this and send back some HTML around our data:

```
app.post('/my-handling-form-page', (req, res) => {
    res.send(`
        <h2> ${req.body.user_email} </h2>
    `)
})
```

We are using string interpolation here, and we can send back a full webpage if we wanted. We could marshall our webpages on the backend, and call up the webpages and send them back when the requests come in.

A Rudimentary DB

```
const express = require('express')
const app = express()
const port = 3000

app.use(express.urlencoded({ extended: true }));

const data = []

app.get('/', (req, res) => {

    console.log("HIT THE ROOT ROUTE")
```

```
    console.log('DATA')
    console.log(data)

    const htmlData = data.map(email => {
        return `<li> ${email} </li>`
    })
    console.log(htmlData)

    const htmlString = htmlData.join(' ')
    console.log(htmlString)

    res.send(`
        <h2> Emails </h2>
        <h3> Number of emails: ${data.length}
        <ul>
            ${htmlString}
        </ul>
    `)
})

app.get('/simple-form', (req, res) => {
    res.send(`
          <form action="/my-handling-form-page" method="post">
              <div>
                  <label for="email"> Name: </label>
                  <input type="text" id="email" name="user_email"
value="">
              </div>
              <div class="button">
                  <button type="submit"> Send your email </button>
              </div>
          </form>
        `
    )
})

app.post('/my-handling-form-page', (req, res) => {
    console.log("IN THE POST")
    console.log(req.body.user_email)
    data.push(req.body.user_email)
    res.send(`
        <h2> ${req.body.user_email} </h2>
    `)
})

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`)
})
```

Let's break down this code, with our very rudimentary database in the form of a JS array.

However, although a perfectly acceptible want to make an app, what I we would ordinarily do is separate out the front end code from the back end. ???? .....