Full Stack

Couch to Coder | BRIGHT NETWORK

Learning Objectives

- Understand what is a request
- Be able to make a GET request from the client side
- Be able to consume the returned data on the client side
- Be able to generate HTML elements in the browser from the returned data
- Be able to create an HTML form on the client side
- Be able to send the contents of the form to the server side
- Understand how to store incoming data on the server side

Requests

After the last couple of weeks, we are ready to finally connect the 2 moving parts of our app - the client side to the server side, front end to back end.

The code itself won't be too long, just like how it was with the server side lesson, but we need quite a lot of background explanation. We will not go too much in depth with all the background info, but we will touch base on evreything that's important.

First off, let's cover what are requests. As we discussed before, in order to get anything displayed in the browser, we need to make a request to a server. So far, the only way we learned how to make a request is via the browser's address bar. Most people are going to use this method exclusively as every day users of web applications, and the only type of requests that can be made via this URL bar is a GET request.

> Note: Another common way to make requests is to click on any anchor tags (links) that takes you to websites. By default, every anchor tag that takes you on a different page or an entirely different website is going to be a GET request.

These type of requests also has an unfortunate side effect: When we make them, the browser will reload and refresh, essentially rebuilding and loading in the entire website again from scratch. This is inefficient, slow, and also costs (relatively speaking) a lot of data. I only want to load in the JSON containing the recipes - the CSS and most of the HTML is fine as is.

Creating requests using JavaScript

Since the early 2000s, JavaScript is able to create requests on the fly without the need to reload the entire page. Essentially it creates a request, and we can tell the browser programmatically using JavaScript that we would like to generate HTML elements on the go without reloading the whole page. Apart from the speed and saved data, it's also better user experience - we will not see the flashes of a reloading page when we run our code like this. Getting notifications and chatting in real time would be impossible to execute well without JavaScript!

Let's see how this would look like with our code. I recommend using some pseudocode first, so we know the steps we need to take. Don't worry - we will reuse a lot of code from last week!

Let's start with the easier step: getting back the recipes from our back end.

```
// 1. make a request that loads in the recipes from our own backend
// 2. Capture the response value
// 3. Turn it into JavaScript
// 4. Loop through each recipe
// 5. Create the html components for these recipes
// 6. Populate the text part using innerText
// 7. Append each recipe to the webpage
```

Looks like a lot, but I promise, the code needed to write this is not going to cause any headaches!

We are going to use the codebase from the last 2 session, so no need to create a new folder for session 5!

First off, we almost forgot step 0: starting the back end server. If the server is not running on `localhost:3000`, then there's nowhere to get our recipes from!

Reminder: Right click on session 3's `server` folder, then click on `Open in integrated terminal` then type in the terminal `node index.js`. If in doubt, follow the video!

Let's continuse with the actual step 1: Making our request. At the top of your `app.js` file, add the following:

```
document.addEventListener("DOMContentLoaded", async () => {
  const response = await fetch("http://localhost:3000/recipes");
  console.log(response);
})
```

Before we open up our console in the browser, a quick disclaimer: JavaScript's asynchronity is a big ol' can of worms we are only partially opening today. In short, we do not want JavaScript to halt the running of the page if we are making a request, because that would be terrible user experience! Imagine not being able to chat with your friends on a social media site just because a video is loading in the background!

The solution is to run things in the background, well, asynchronously. There are different syntaxes to make this work, one of the easier ones, without actually covering big frameworks like React or Angular, is seen above: We are basically saying that when the `DOM` (the JavaScript version of the HTML) is loaded in, we want to run code asynchronously (hence the `async` keyword before the brackets).

With the `async` keyword added, we can say that we would like to wait for `fetch()` to make its request until it's finished.

> Note: JavaScript executes faster than the time it takes for the response to come back from the server - even if the server is on our own machine! That's why we have to tell it to wait for the response to come back. If we don't, JavaScript tries to loop through a nonexistent list of recipes!

The code, however, is not working well yet: If you open your browser's console, you can see a weird error similar to this:

`Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at localhost:3000/recipes. (Reason: CORS request not http).`

What does this mean?

This is a built in security measure to block basic attempts at trying to run scripts from the same host as where the server is. Imagine how much damage you can cause against a big app if you could just try to force running code on the same host - you could access databases, sensitive data, even the app's source code itself! Since we know it's not a big deal, because we are running our own development code, we can set up our own code to allow this to happen.

First we need to install a new NPM package. Stop the running of the server in the terminal for the back end, and enter the following:

`npm i cors`

Now we can add the following code to our `index.js`:

```
//same as before
const path = require("path");
const cors = require('cors')

// enabling CORS for any unknown origin
app.use(cors());
```

Let's restart the server and check out the browser's console again after reloading!

Now this is certainly... something. The type of object we get back is a response, and on that response, is a JSON object, which we can access by using a method called - `.json()`! Since this one is async too, we need to `await` it!

```
document.addEventListener("DOMContentLoaded", async () => {
  const response = await fetch("http://localhost:3000/recipes");
  const recipes = await response.json()
  console.log(recipes);
})
```

Well look at that! An array - with 3 recipes in it. Great job!

We are done with pseudocode steps 1-3. We need the next one - looping through them. We can do so with the code we learned about previously:

```
document.addEventListener("DOMContentLoaded", async () => {
  const response = await fetch("http://localhost:3000/recipes");
  const recipes = await response.json()

  for(recipe of recipes){
    console.log(recipe)
  }
})
```

Since we can loop through them, we just need to modify the code to carry on. We could move the entirety of last week's JavaScrpt code into the for loop!

```
document.addEventListener("DOMContentLoaded", async () => {
  const response = await fetch("http://localhost:3000/recipes");
  const recipes = await response.json()

  for(recipe of recipes) {
    // Create the container that will hold the recipe
    const recipeContainer = document.createElement("div");

    //create the name tag (h3), add text to it, glue it onto the container
    const nameTag = document.createElement("h3");
    nameTag.innerText = "Spaghetti Carbonara";
    recipeContainer.appendChild(nameTag);


    //same with cuisine and time tags
    const cuisineTag = document.createElement("p");
    cuisineTag.innerText = "Cuisine: Italian";
    recipeContainer.appendChild(cuisineTag);

    const timeTag = document.createElement("p");
    timeTag.innerText = "30 minutes";
    recipeContainer.appendChild(timeTag);

    //create the unordered list element for the ingredients
    const ingredientsListTag = document.createElement("ul");

    //create the list items for the ingredients list
    const ingredientsListItemTag = document.createElement("li");
    ingredientsListItemTag.innerText = "pasta";
    ingredientsListTag.appendChild(ingredientsListItemTag);

    recipeContainer.appendChild(ingredientsListTag);


    recipeContainer.appendChild(document.createElement("br"));
    //create the ordered list element for the steps of the recipe
    const stepsListTag = document.createElement("ol");


    //create the list items for the steps list
    const stepsListItemTag = document.createElement("li");
    stepsListItemTag.innerText = "Cook the pasta";
    stepsListTag.appendChild(stepsListItemTag);

    recipeContainer.appendChild(stepsListTag);

    const recipeList = document.querySelector("#recipe-list");
    recipeList.appendChild(recipeContainer);
  }
})
```

This, of course, just repeats the short carbonara recipe as many times as many recipes got returned from the server - in this case, 3! But what happens if we change the code to not hardcode the values, but get them from the recipes as we loop through them?

Change all the hardcoded values to take values out from the individual recipe varible:

```
document.addEventListener("DOMContentLoaded", async () => {
  const response = await fetch("http://localhost:3000/recipes");
  const recipes = await response.json()

  for(recipe of recipes) {
    // Create the container that will hold the recipe
    const recipeContainer = document.createElement("div");

    //create the name tag (h3), add text to it, glue it onto the container
    const nameTag = document.createElement("h3");
    nameTag.innerText = recipe.name;
    recipeContainer.appendChild(nameTag);


    //same with cuisine and time tags
    const cuisineTag = document.createElement("p");
    cuisineTag.innerText = recipe.cuisine;
    recipeContainer.appendChild(cuisineTag);

    const timeTag = document.createElement("p");
    timeTag.innerText = recipe.time;
    recipeContainer.appendChild(timeTag);

    //create the unordered list element for the ingredients
    const ingredientsListTag = document.createElement("ul");

    //create the list items for the ingredients list
    const ingredientsListItemTag = document.createElement("li");
    ingredientsListItemTag.innerText = recipe.ingredients;
    ingredientsListTag.appendChild(ingredientsListItemTag);

    recipeContainer.appendChild(ingredientsListTag);


    recipeContainer.appendChild(document.createElement("br"));

    //create the ordered list element for the steps of the recipe
    const stepsListTag = document.createElement("ol");


    //create the list items for the steps list
    const stepsListItemTag = document.createElement("li");
    stepsListItemTag.innerText = recipe.steps;
    stepsListTag.appendChild(stepsListItemTag);

    recipeContainer.appendChild(stepsListTag);

    const recipeList = document.querySelector("#recipe-list");
    recipeList.appendChild(recipeContainer);
  }
})
```

Better! But we are not quite there yet. As you can see, both the steps, and the ingredients list look a bit silly. Let's do something wild - nested loops!

Sometimes, when our data structures demands it, we encounter a situation where an array contains another array. In order to loop through those, we have to loop through them as a nested array - the `ingredients` and `steps` properties of the outer array's items can be looped through as well. This is necessary, because we want each step to be listed out in a list item. Change the section where we deal with the ingredients like so:

```

//create the unordered list element for the ingredients
const ingredientsListTag = document.createElement("ul");

//create the list items for the ingredients list
for(ingredient of recipe.ingredients){ //remember, this property is an array in
itself!
  const ingredientsListItemTag = document.createElement("li");
  ingredientsListItemTag.innerText = ingredient;
  ingredientsListTag.appendChild(ingredientsListItemTag);
}

recipeContainer.appendChild(ingredientsListTag);
```

And do the same with the steps!

```

    //create the ordered list element for the steps of the recipe
    const stepsListTag = document.createElement("ol");
    //create the list items for the steps list
    for(step of recipe.steps){
      const stepsListItemTag = document.createElement("li");
      stepsListItemTag.innerText = step;
      stepsListTag.appendChild(stepsListItemTag);
    }
    recipeContainer.appendChild(stepsListTag);
```

Well done!

Adding a new recipe

Adding new recipes requires quite a lot of thinking - we will look at a relatively simple solution as a first step.

We need a form for sending data to our backend - we can send a post request however we like, but for the user to be able to interact with it, a form is the best.

To keep it simple, let's copy the following HTML form into our `index.html`, at the top of the `main` tag, so that our form is always at the top:

```html
<main>
  <h2>My Favourite Recipes</h2>
  <h3>Add a new recipe:</h3>
  <form action="">
    <label for="name">Name: </label>
    <input type="text" name="name" id="name">
    <br>
    <label for="cuisine">Cuisine: </label>
    <input type="text" name="cuisine" id="cuisine">
    <br>
    <label for="time">Time: </label>
    <input type="text" name="time" id="time">
    <br>
    <label for="ingredients">Ingredients (each on a new line): </label>
    <textarea name="ingredients" id="ingredients"></textarea>
    <br>
    <label for="steps">Steps (each on a new line): </label>
    <textarea name="steps" id="steps"></textarea>
    <br>
    <input type="submit" value="Save Recipe">
  </form>
  <ul id="recipe-list">

  </ul>
</main>
```

This will act as our form to add new items. Let's quickly cover its building blocks:

- The `form` tag encloses HTML that will handle input fields for our new recipe.

- `label` tags are not mandatory, could use `p` tags or something else, but a label makes it much better for SEOs and screen readers to work with our app
- `input` fields can have many types: text, number, date, checkboxes, but for our recipe elements, text is the most appropriate
- both `ingredients` and `steps` will be `textarea`. We want to keep it that way so we can split up the steps and ingredients easier. There are many advanced techniques for it, we will try to keep it simple - every time the user hits enter to add a new line for a new ingredient, we will split up their text on these new lines, so we can create arrays from them easily
- finally, a `submit` button to, well, submit the form. When it's pressed, a special event will trigger, and we can add code to execute when it is pressed.

Now that we have the HTML, let's add some JavaScript in our `DOMContentLoaded` event. First, we need to grab the form like we did with other elements previously, and then add an event when that form is submitted:

```
const newRecipeForm = document.querySelector("form");
newRecipeForm.addEventListener("submit", (event) => {
  event.preventDefault();
  console.log(event.target);
})
```

We need to start with `event.preventDefault()`, because when we are submitting a form, the default behaviour of HTML is to reload the page the form is on. But if we are reloading the page, we are going against what we are trying to achieve - reloading-free interaction with the client. So with this command, we can prevent the default behaviour of the submit event.

`event` is a special parameter in the brackets (we could call it anything though!) - this will grab the event object of submitting the form. This sounds a bit weird, but bear with me - essentially we do not know when the form will be clicked, so we will prepare code for the event. When the form is submitted, code between the curly braces will execute, and we also have a chance to grab the event itself if we need it. The easiest way to access values from the input fields is to use this event, since it has a target - the HTML element that triggered the event, which is the `form`. From the `event.target`, we can see in the browsers console that this has all the input fields accessible - either by index or by their CSS `id` attribute.

Now we only need to create a brand new JavaScript object that will be sent over as JSON via a new request, and set this object's properties to be the ones we added in the form. We can achieve this by targeting the input field's `value` proprety! Add the following to the code:

```
  event.preventDefault();

  const newRecipe = {};
  newRecipe.name = event.target.name.value;
  newRecipe.cuisine = event.target.cuisine.value;
  newRecipe.time = event.target.time.value;
  const ingredients = event.target.ingredients.value.split(/\r?\n/);
  newRecipe.ingredients = ingredients;
  const steps = event.target.steps.value.split(/\r?\n/);
  newRecipe.steps = steps;
```

The `split()` command grabs the string it was called on, and splits it up into an array on a given so called `delimiter`. We can break things up into arrays on a comma, a period, or, in our case, the hidden newline character added when hitting enter in the textarea. We can inspect these values by using the console.

The last step (at least in the client side) is to make the POST request and send the new recipe as a JSON object to the backend. We can send a POST request using fetch like so:

```
fetch('http://localhost:3000/recipes', {
  method: 'POST', // Specify the request method
  headers: {
    'Content-Type': 'application/json' // Set the content type to JSON
  },
  body: JSON.stringify(newRecipe) // Convert the JSON object to a string
})
```

We add a second argument that's an object to the fetch method, which sets the HTTP verb to be POST instead of the default GET. We also tell the request that we will attach JSON to it with the headers property, and finally, we add our recipe as a stringified JSON object.

Final step is to catch this in the server side.

First we have to tell our server side app that JSON is to be expected when we are talking to the endpoints. We can do this by adding the following:

```
// enabling CORS for any unknown origin
app.use(cors());

app.use(express.json()) // NEW
```

Now we can see that we successfully send this request on the back end by adding a new console.log() command to the POST request of the server's `index.js`:

```
app.post("/recipes", (req, res) => {
  console.log(req.body)
  res.send("Recipe added, storing your favourite dishes")
});
```

Don't forget to restart the server!

The last step is us adding this new JSON object to our JSON file.

Adding the following will let us modify the existing JSON file:

```
app.post("/recipes", (req, res) => {

  const newRecipe = req.body;

  fs.readFile(recipesFilePath, 'utf8', (err, data) => {

    const recipes = JSON.parse(data); // Parse the existing JSON array

    recipes.push(newRecipe); // Add the new object to the array
    console.log(recipes)

    // Write the updated array back to the file
    fs.writeFile(recipesFilePath, JSON.stringify(recipes), () => {});
  });
  res.send("Recipe added, storing your favourite dishes")
});
```

- The `fs.readfile()` method opens up the JSON file, and stores the array in the `data` variable
- We use JSON.parse() to make it into a JavaScript array
- We push the new recipe attached to the request into the array
- Finally we write the array back to the file, and we also stringify/JSONify the array. We need the empty callback at the end, which would allow us extra steps like cleaning up the file or changing its name or metadata.