More API Calls.

Mari Good

CS 233JS

Topics

- Event Registration App
  - .env file
  - JS Modules
  - Regular expressions
  - More practice with AJAX, fetch and promises
  - Chart.js api
  - Google maps api
  - Building production (vs development) code
  - Lab 6
    - Event Registration app – ONLY ONE app!

Introducing the Event Application

- The last lab involves an application that could be used to allow people to register for any kind of event. The application has 3 pages
  - The home page allows the user to register for the event
  - The status page allows the user to view charts that summarize data related to folks who have already registered
  - The about page shows a (google) map of the location of the event

Demo On My Machine

- Let's start by running the app on my machine and looking at its functionality.
  - The app uses an npm module called json-server that allows us to both submit the registration information and get data about registrants for our charts. json-server is frequently used to "mock" a web api during the development process. Some of you may want to use it for your project.
    - In order to test the application I have to start json-server. In package.json, notice that there is a script called apiStart. It starts json-server on port 3000 and uses the db.json file in the data folder as its "database".
    - From a terminal window using the main application folder, npm run apiServer.
  - The app uses a webpack development environment that's very similar to the one we created for the MemeCreator.
    - Notice webpack.config.js.
    - Notice package.json script watch.
    - In a terminal window using the main application folder, npm run watch.

Setting up Your Machine - The Service

- Now let’s try it on your machine
  - Clone the github repository. Look at the structure of the file system.
  - Install all of your node modules
    - Open a terminal window on the main application folder.
    - npm install
  - Start the “service”
    - npm run apiServer to start the server

Test the Service

- Now let's try it on your machine
  - Test the service on your machine
    - Open a browser
    - Enter http://localhost:3000/participants in the address bar
      - You should get json back. That's all of the contents of the participants array in the db.json file
    - Enter http://localhost:3000/participants/1 in the address bar.
      - You should get back the data for the participant that has an id of 1
  - Just in case you want more information about working with json-server: https://medium.com/codingthesmartway-com-blog/create-a-rest-api-with-json-server-36da8680136d

Setting up Your Machine - The App

- Now let's try it on your machine
  - Open another terminal in the main folder for the app
    - npm run webpack to create the dist folder
      - You should see that the html files have been copied, the js files have been copied and have corresponding map files and that the assets folder has been copied too.
      - You should also notice that there are lots more js files. I'll talk more about this when we talk about creating a production build. In the meantime, a screenshot of my dist folder is on the next slide.
    - npm run watch to start the application

My Dist Folder

Test the Starting Files

- Now let's try it on your machine
  - Home page looks ok but doesn't do anything. We'll start writing code for this page. We'll add
    - Form validation using regular expressions
    - An ajax call that uses POST as the method to submit the registration form
  - Status page doesn't do anything. We'll add
    - An ajax call that gets the data from service
    - API Code that uses charts.js to add pie charts to the page.
    - Code that changes bootstrap styles to create tabs the page. Each tab will display a different chart.
  - About page works. We'll look at the code AND change the map location etc.

One More Setup Thing

- Now let’s try it on your machine – the app
  - Create a .env file that stores configuration information for the app.
    - Why shouldn’t configuration information like the api key for google maps OR the url for the service that processes the registration information go in our source code?
    - JS developers generally put this kind of information in a .env file (that’s gitignored) and create an .env.example file (that’s not ignored) so that other developers know what technical information the app needs.

Create a .env file

- Now let's try it on your machine
  - Edit the .env.example file and save it as .env in your editor

    NODE_ENV=dev

    SERVER_URL=http://localhost:3000/participants

    GMAP_KEY=YOUR_API_KEY_GOES_HERE
    - You'll need to get a google maps api key. Or you can use mine. BUT WHATEVER YOU DO DON'T post my key to a public repository.
    - AIzaSyBYZeL9diaVDvdIfkLD4UuK-5XCM7DJ2n8

The Home Page – html file first

- Let's start writing code for the home page. Look at
  - index.html – Notice
    - The bootstrap nav bar. This same nav bar is duplicated on all 3 pages. Hmmm … shouldn't there be a better way to do this?
    - The form tag action and method attributes. See pages 92 – 94.
      - If this was a "traditional" web app, the data entered by the user in the form would be submitted to the server for processing by an "endpoint" with /registrataion appended to the end of the url for the home page.
      - The data would be submitted via post (as part of the http header of the request sent to the server) as opposed to get (as part of the url).
      - We're going to submit the data by making an ajax post request in our client side code.

The Home Page – html form validation

- Let's start writing code for the home page. Look at
  - index.html – Notice
    - Each input tag uses SOME html validation attributes. What attributes do you see? How do each of those attributes work? What happens if you leave the fields empty? What fields are required? What other attribute might we add to enforce that validation without code? How would you select a default value in the combo box? A radio button?

The Home Page – js code

- Let's start writing code for the home page. Look at
  - home.js – Notice
    - This is an ES6 style class that encapsulates the functionality of the page.
    - Most of this is similar to things that you've already done and I'll give you time to write this code after we talk about a couple of things.

ES6 Modules

- Let's start writing code for the home page. Look at
  - home.js – Notice
    - There are 2 import statements at the top of the file that refer to the services folder in this application. I "borrowed" a couple of "utilities" that are used for validation.
    - Each of the utilities uses ES6 module (import and export) syntax to make a function available in other js files.

Declaration in services file

```
export default function validateRegistrationForm(formValues) { … }
```

Usage in home.js file

```
import validateRegistrationForm from './services/formValidation/validateRegistrationForm';
```

Regular Expressions

- The actual validation is done using regular expressions.
  - Regular expressions allow you to match strings to complex patterns using a declarative (rather than a procedural) approach.
  - In the starting files, I've given you an html file demo_regexp.htm that you can use to experiment with regular expression syntax.
  - By the time we finish experimenting together, you should be able to
    - Recognize a regular expression when you see one
    - Interpret simple to moderately complex regular expressions
    - Write a simple regular expression

Regular Expressions- Syntax

- Using a regular expression in js code is relatively simple
  - Declare a variable that contains the regular expression. Delimit the expression with / and /

```
const pattern = /^a+/
```

  - Call the method test of your regular expression object passing the string you want to test as a parameter.

```
pattern.test('alpha') // returns true
pattern.text('beta')  // returns false
```

validateRegistrationForm.js

- You're ready to write the validation code in validateRegistrationForm.js
  - The function that the page uses calls a number of simple validation functions. I've given you all of the code for that function.
  - You need to write each of the individual functions at the bottom of the file.
    - You might test the expressions in the test page I've provided.
    - You might test your functions using codepen or jsfiddle.

The Home Page – Everything But AJAX Call

- Now let's work on home.js
  - Part 1 is done (the validation module)
  - Part 2 - create the constructor
  - Part 3 – write onFormSubmit
  - Part 4 – write 4 other UI related functions
  - Instantiate a Home object at the bottom of the file in the usual way.
  - TEST. The ajax call that submits the form won't work but the validation should work. We'll do the ajax call next.

Using Fetch

- The general syntax for making an ajax call using fetch looks like this

```
fetch(theURLOfTheService)
    .then(response => response.json())
    .then(data => {
        //data is a js object that was returned by the api and
        //converted to json
        //do something with it in the body of this arrow function
    })
    .catch(error => {
        // do something with the error
    });
```

fetch returns a Promise and promises are "thenable" . The arg to then is a function (in this case an arrow function) that will be executed when the promise is fulfilled.

Asynchronous activities can be chained or executed one after the other.

Promises are also "catchable"

Using Fetch to Make a Post Request

- The general form for making a post request

```
const options = {
      method: 'POST',
      mode: 'cors',
      headers: { 'Content-Type': 'application/json' },
      body: // data you're sending to the server as a json string goes here
    };
fetch(theURLOfTheService, options)
      .then(response => {
         // there might or might not be data returned from a post
         // but an app should do something to let the user know that everything is ok
      })
      .catch(error => {
                // do something with the error
      });
```

The second parameter is optional in a get request. In a post request it is not. It will often look exactly like this.

Don't forget to let the user know about an error

The Home Page – the AJAX Call

- Now let's finish home.js
  - Parts 1 – 5 are done.
  - Part 6 – write the submitForm function. It makes the ajax call using fetch.
  - TEST. Use the developer tools to watch the ajax request go out and come back. If you look at the details for the request you should see that the method is POST as well as the detailed data that is submitted. You should also check the details of the response. Let me show you this on my machine as a reminder of what we did during lab 5.

The Status Page – html file first

- Let’s start writing code for the status page. Look at
  - status.html – Notice
    - The tab area. Tabs (for each chart) are created by styling a ul and muliple li elements with bootstrap.
    - The chart area. Each chart is drawn on a canvas. The Chart constructor takes a canvas at it’s first parameter. You’ll hide or show the appropriate canvas when the user clicks on one of the tabs.

The Status Page – js Code

- Let’s start writing code for the status page. Look at
  - status.js – Notice
    - This is an ES6 style class that encapsulates the functionality of the page.
    - Most of this is similar to things that you’ve already done.

The Status Page – Charting Functionality

- Now let's work on status.js
  - Part 1 – create the constructor
  - Part 2 –
    - Write loadData. It makes the ajax call and draws the experience chart.
    - Write addEventListeners. It adds the event handlers for the tabs.
  - Instantiate a Status object at the bottom of the file in the usual way.
  - TEST. Use the developer tools to watch the ajax call go out and come back. Only the experience chart will work at this point.
  - Part 3 – write the other chart drawing function
  - TEST. You should be able to click on the other tabs and see other charts.

The About Page – html file first

- Let's start with the code for the about page. All of the code for this page is already written but look at
  - about.html – Notice
    - The container that should have information about your event. Change this. Maybe your end of year party?
    - The map div tag. The map will be created here.

Google Maps Developers Documentation

- Let's start with the code for the about page. Look at
  - https://developers.google.com/maps/
    - You add a map to a page by instantiating a Map object and passing configuration information about the map in the constructor.
    - You're going to add a script block that references an external file at on a google server. You'll also need to identify yourself using your apikey. AND you'll have to identify the name of the function (your code) you want to call to draw the map.

```
<script src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY&callback=initMap">
```

Google Maps – You'll Need an API Key

- Let's start with the code for the about page.
  - The example on the web adds the script block in the html file. We're going to do the same thing but in js code.
  - You'll need an apikey if you don't have one already.
    - https://developers.google.com/maps/documentation/javascript/
  - Just in case you don't remember, the apikey for google maps is in our .env file. You'll want to replace my apikey with yours once you have it.

The About Page – js Code

- Let’s look at the code that creates the map. Notice
  - initMap is exported
  - The body of the function creates a bunch of map related objects based on configuration information you provide. You’ll want to change that information so that you get a map for your event (graduation party).
  - At the bottom of the file, we add the script element to the page after the page has finished loading.
  - Notice that the callback function is initMap. I made this function available globally by adding the line of code on line 34.

One Last Thing – The Nav Bar

- At this point the version of the application from the text should be finished. I'd like you to do one more thing
  - Develop a strategy for sharing the navigation elements at the top of each page across all 3 pages. Implement your strategy and replace the navigation elements on each page with your implementation.
  - Does anyone have any ideas about how you might share the code across multiple pages? What's the same across all of the pages? Is there anything that's specific to an individual page? How might you handle that?
- When everyone is done, we'll work on creating a production version of the application and deploying the app and its services to citstudent.

Creating a Production Build - .env file

- Some of the things that work in a development environment don’t work so well for production applications.
  - Change the .env file to include production values
    - NODE_ENV
      - production
    - SERVER_URL
      - The “service” is NOT running on citstudent (where your web app will be hosted) and even if it was, it wouldn’t be running on port 3000!
      - The base url for the production service is: http://citweb.lanecc.net:5000/participants
      - You should be able to test the service in your browser using the base url

Creating a Production Build – webpack.config

- Notice all of the comments in the webpack.config.js file. Let me point out the most important things.
  - I use a variable, isProduction to apply config changes differently for dev and production.
  - Create content specific filenames for files to make sure that when content changes in a production environment, anything that is cached is replaced with new content.
  - Remove the source maps file from production js code to protect your intellectual property
  - Use MiniCSSExtractPlugin in production rather than style loader to separate js and css code. Link elements go in the head and js elements go at the bottom of the body so that the page looks ok when loading.
  - DefinePlugin (part of webpack) exposes variables from .env file globally. Both production and dev.
  - Optimization property (part of webpack) finds all of the js and css that is used in multiple bundles and extracts it so it only has to be loaded by the browser once. That speeds up load time. Both production and dev.

Production dist Folder

- Some of the things that work in a development environment don’t work so well for production applications.
  - Notice the content of the dist folder before you start. How big are the js and css files? Are there source map files in the folder?
  - npm run webpack from the command line
  - Compare the content of the dist folder to what you started with. What do you notice?

Production html Files

- Some of the things that work in a development environment don't work so well for production applications.
  - Look at the production html files in the dist folder. Notice
  - Link elements with hashed filenames are added to the head section of the document. That allows your pages to look ok to the user while the js code is loading.
  - CSS that you use on multiple pages such as bootstrap is separated from your page specific css. Multiple link elements are added to the page.
  - Script elements with hashed filenames are added to the body section of the page.
  - JS code that you use on multiple pages such as your navbar is separated from your page specific js. Multiple script elements are added to the page.

Test the Production Build

- Some of the things that work in a development environment don't work so well for production applications.
  - Test your application. You CAN NOT use the webpack development server at this point but you can use the live server VSCode extension.
  - Live server will server your application from localhost:5500.
  - Deploy the dist folder to citstudent.
  - Test your application one last time.
  - Apache on citstudent is hosting your application. json-server and node.js on citweb is hosting the "service".

What's Next

- Your Project
  - At this point you should have several of the deliverables for your project complete. You'll spend the last 2 weeks in class working exclusively on your project.
- Don't forget
  - Reading Quiz 6
  - Participation Score 6
  - Lab 6