Assignment 8

Contents

This assignment is a tutorial in which we will write JavaScript code to make the browser send HTTP AJAX requests and receive the server's response. We'll use a third-party library called Axios to help us formulate the HTTP requests, send them, wait for them to finish, and process the response. Finally, we'll run the code we write directly in the Google Chrome browser and view the requests we make through Chrome's DevTools window.

1. Getting Started

1. Open Visual Studio Code and take a look at the a08 assignment directory.
2. You should see one HTML file (`a08/index.html`), one linked CSS file (`a08/style.css`), and one linked JavaScript file (`a08/script.js`). There should also be another JavaScript file `a08/submission.js` containing empty functions. Your job for this assignment is to fill in the functions in this file.
3. Open a terminal in the a08 assignment directory and initialize a new npm project using `npm init`.
4. Use `npm install` to install the package `axios`. Refer to a02 or the official Axios documentation if you need to review installing packages with npm.
5. Link the downloaded Axios library to the index.html file. To do this, find the Axios `TODO` comment in `a08/index.html` and add a `<script>` tag with `src` attribute pointing to the correct Axios `.js` file, located somewhere in the `a08/node_modules/axios` folder.

   Hint: If you're having trouble finding the correct file to link, refer to a02 and/or a04 to reread the process that we used to link to Bulma and jQuery. The Axios code is organized very similarly to jQuery; there is a special `dist` folder containing the relevant "distribution" Axios code.
6. Run `browser-sync -sw` to start the Browsersync local development server. This should automatically open a new browser tab showing the contents of `a08/index.html`.

   Note: If you get a `command not found` error, it likely means you have not installed Browsersync as a global npm package. Go back to Section 5.4 in Assignment a02 to see what it means to install packages globally, and follow the instructions to install Browsersync as a global npm package.

2. Using Axios

2.1 A simple request

All the functionality provided by the Axios library comes packaged in a single object named `axios`. In particular, the `axios` object is a function. When you call it, the library automatically initiates an HTTP call for you according to the configuration you specify as arguments to the function. Here's a simple example of what this might look like in code:

```
const result = await axios({
  method: 'get',
  url: 'https://comp426fa19.cs.unc.edu/a08/heroes',
});
```

This tells Axios to make a GET request to `https://comp426fa19.cs.unc.edu/a08/heroes`, and store the response in the `result` variable.

But wait! You may have noticed the `await` keyword in front of the function call to `axios()`. *This means that `axios()` is actually an async function*! That's right---a call to `axios()` actually returns a *promise* that resolves once the response is received from the server. This is fantastic news, because it means we can take advantage of the simple async/await syntax with Axios and can avoid using callbacks! It also makes total sense: HTTP requests are perfect candidates for promises and async/await because they are I/O-bound operations that take time to complete.

2.2. The HTTP response

Once the `axios()` promise resolves, the data that was received in the HTTP response will be automatically stored in the `result` variable. Here's an example of what `result` may look like, taken directly from the Axios documentation.

```
console.log(result);
{
```

```
  data: {},         // `data` is the response that was provided by the server

  status: 200,      // `status` is the HTTP status code from the server response

  statusText: 'OK', // `statusText` is the HTTP status message from the server response

  headers: {},      // `headers` the headers that the server responded with
                    // All header names are lower cased

  config: {},       // `config` is the config that was provided to `axios` for the request

  request: {}       // `request` is the request that generated this response
                    // It is the last ClientRequest instance in node.js (in
                    // redirects) and an XMLHttpRequest instance the browser
}
```

Some of these descriptions may be familiar to you from KMP's lecture about the HTTP protocol. In fact, `status`, `statusText`, `data`, and `headers` directly correspond to the parts of an HTTP response as mandated by the HTTP protocol:

- `status` - a status code sent by the server which is a 3-digit number indicating whether the request was successful
- `statusText` - a human-readable description sent by the server describing whether the request was successful
- `headers` - The HTTP response headers sent by the server
- `data` - The optional body of the HTTP response, containing JSON, HTML, XML, text, or any other data that the server sends back

The `config` and `request` parts of the Axios response are less important, as they just provide information about the HTTP request that was originally made in order to produce the given response.

2.3. More HTTP request options

The HTTP request that we made in Section 2.1 was quite simple: it only specified the HTTP verb (`post`) and the URL (`https://comp426fa19.cs.unc.edu/a08/heroes`) that we were requesting. However, Axios actually provides a huge list of configuration options to help you formulate your HTTP request just right. The official documentation provides a complete list of all the Axios request options. There are a lot of settings in that list, and it can look a bit overwhelming. Keep in mind that settings are optional, so you only really need to add settings that you specifically want. Settings can be specified by adding them to the `axios()` function's argument.

Here is a non-exhaustive list of the most commonly used Axios request settings:

```
const result = await axios({
  // `url` is the server URL that will be used for the request
  url: 'https://comp426fa19.cs.unc.edu/a08/heroes',

  // `method` is the request method to be used when making the request
  method: 'get', // default

  // `headers` are custom headers to be sent
  headers: {'X-Requested-With': 'XMLHttpRequest'},

  // `params` are the URL parameters to be sent with the request
  // Must be a plain object or a URLSearchParams object
  params: {
    ID: 12345
  },

  // `data` is the data to be sent as the request body
  // Only applicable for request methods 'PUT', 'POST', and 'PATCH'
  // Must be of one of the following types:
  //   - string, plain object, ArrayBuffer, ArrayBufferView, URLSearchParams
  //   - Browser only: FormData, File, Blob
  //   - Node only: Stream, Buffer
  data: {
    firstName: 'Fred'
  },

  // `timeout` specifies the number of milliseconds before the request times out.
  // If the request takes longer than `timeout`, the request will be aborted.
  timeout: 1000, // default is `0` (no timeout)

  // `withCredentials` indicates whether or not cross-site Access-Control requests
  // should be made using credentials
  withCredentials: false, // default

  // `auth` indicates that HTTP Basic auth should be used, and supplies credentials.
  // This will set an `Authorization` header, overwriting any existing
  // `Authorization` custom headers you have set using `headers`.
  // Please note that only HTTP Basic auth is configurable through this parameter.
  // For Bearer tokens and such, use `Authorization` custom headers instead.
  auth: {
    username: 'janedoe',
    password: 's00pers3cret'
  },

  // `responseType` indicates the type of data that the server will respond with
  // options are: 'arraybuffer', 'document', 'json', 'text', 'stream'
  //   browser only: 'blob'
  responseType: 'json', // default

  // `onUploadProgress` allows handling of progress events for uploads
  onUploadProgress: function (progressEvent) {
    // Do whatever you want with the native progress event
```

```
  },

  // `onDownloadProgress` allows handling of progress events for downloads
  onDownloadProgress: function (progressEvent) {
    // Do whatever you want with the native progress event
  }
});
```

Most likely, the only request settings you're required to use for COMP 426 are `method`, `url`, `headers`, `params`, `data`, and `withCredentials`.

- `method` - The HTTP verb to use when sending the request (`'get'`, `'post'`, `'patch'`, `'put'`, or `'delete'`).
- `url` - The URL of the server that should receive the request.
- `headers` - An object of HTTP request headers that should be sent with the request.
- `params` - An object of key-value pairs that should be sent to the server as part of the URL using GET parameters (see notes from lecture).
- `data` - Also an object specifying data that should be sent to the server; however, information specified in `data` will be passed to the server in the **body** of the HTTP request, not in the URL. This is traditionally used for `post`, `put`, or `patch` requests, but it can be used in any situation where you want to pass data to the server in a way that is a little more hidden than having it show up directly in the URL.
- `withCredentials` - A boolean value that is beyond the scope of the course, but which you'll need to set to `true` when making HTTP requests in the next assignment (a09).

2.4. HTTP request with data

Finally, here's a simple example of a POST request to the server where data is sent in the body of the HTTP request.

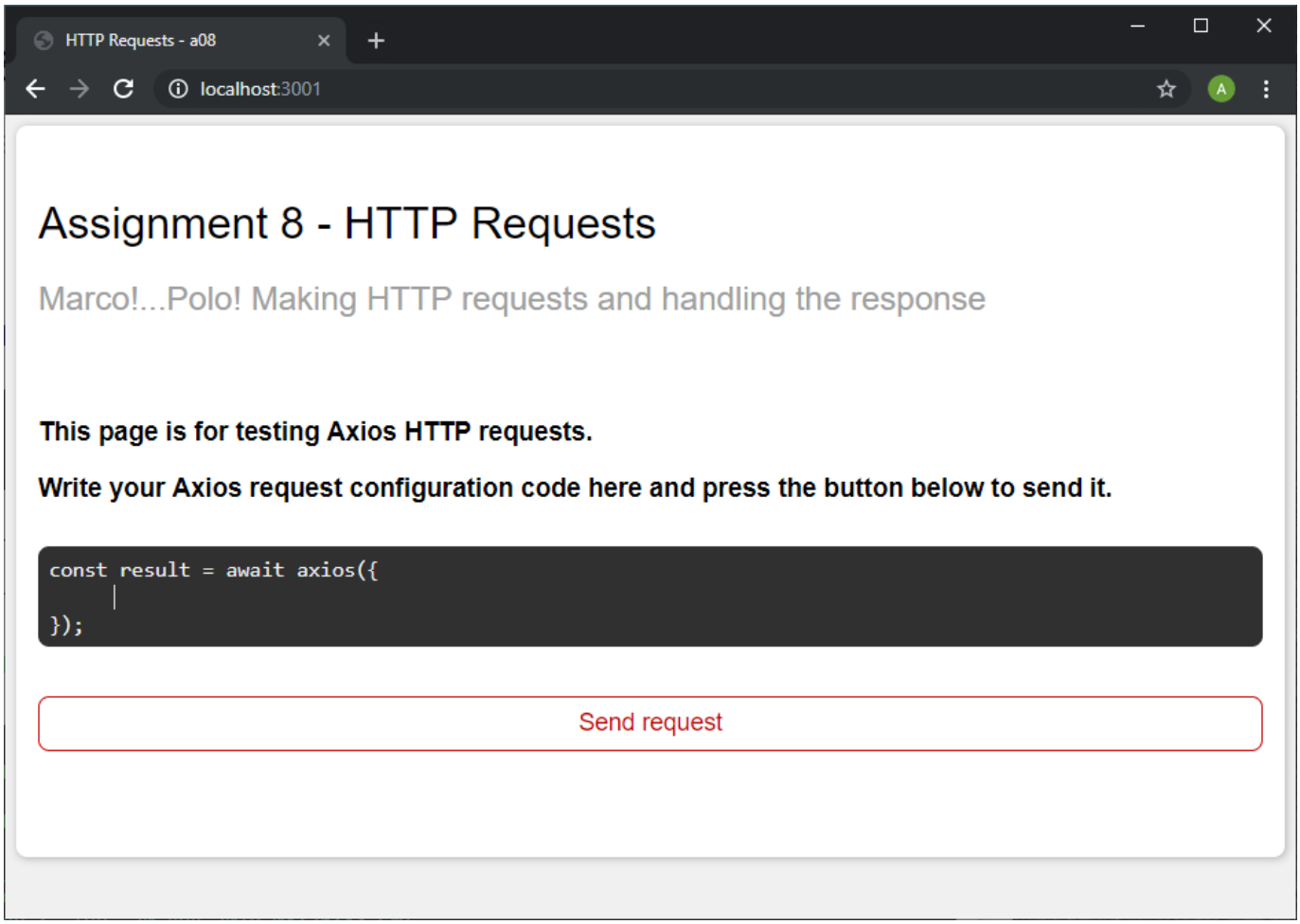
```
const result = await axios({
  method: 'post',
  url: 'https://comp426fa19.cs.unc.edu/a08/people',
  data: {
    first: 'Aaron',  // This is the data that
    last: 'Smith'    // will be sent to the server
  }
});
```

3. Chrome DevTools Inspector

When you run Browsersync in the a08 assignment directory, you should see a basic Axios testing app. For the remainder of this assignment, we will be using this app to write and test Axios requests. Let's try making a basic HTTP request right now.

In addition to playing with the Axios testing app, we're also going to use this part of the assignment to demonstrate a super useful development tool (called DevTools) that is built-in to Google Chrome. With this tool, you'll be able to inspect **all** network traffic that is initiated by a web page, including every HTTP request sent in the background by JavaScript via AJAX.

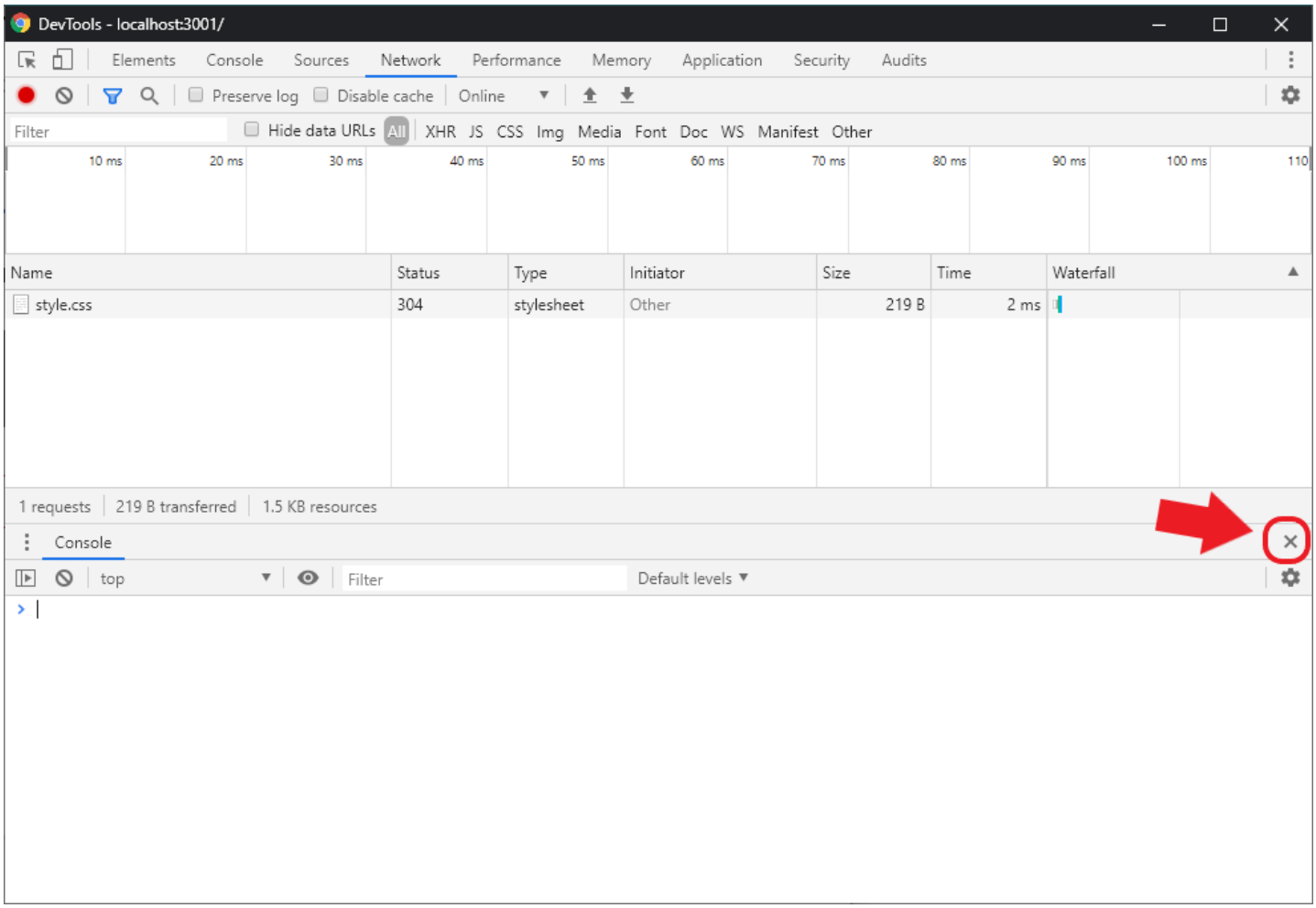
1. If you haven't already, open the Axios testing app in Google Chrome by running Browsersync in the a08 directory.



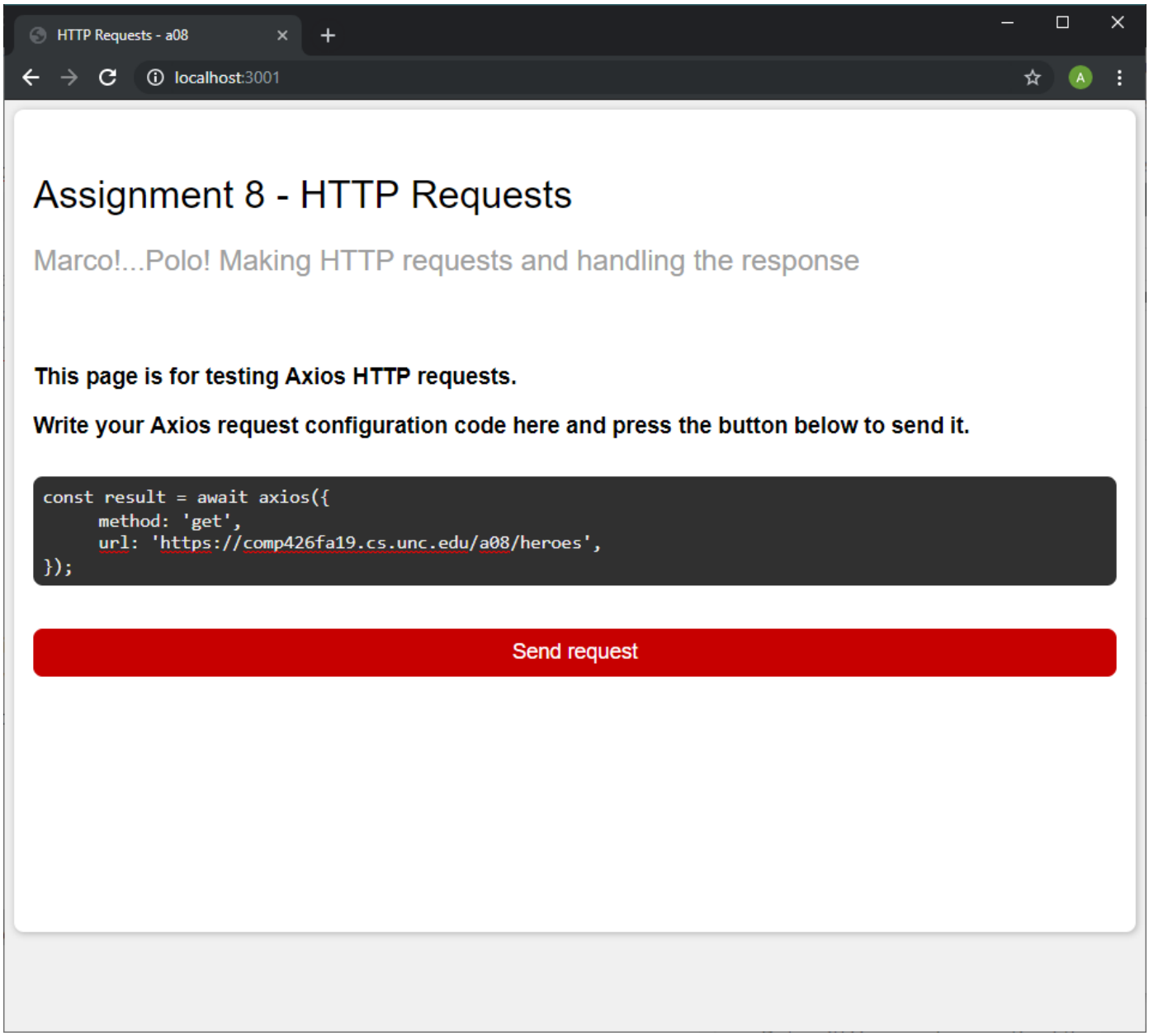
2. From Google Chrome, press the `F12` key on your keyboard. This should automatically open up a new DevTools Chrome window pointing at the "network" tab. As long as this window is open, DevTools will monitor all incoming and outgoing network traffic to and from the current tab/web page.

   You can use the `F12` trick on any website---it's not only limited to this silly Axios testing app. Try it on Facebook, Instagram, or YouTube to see in real time the network traffic that goes on behind the scenes!

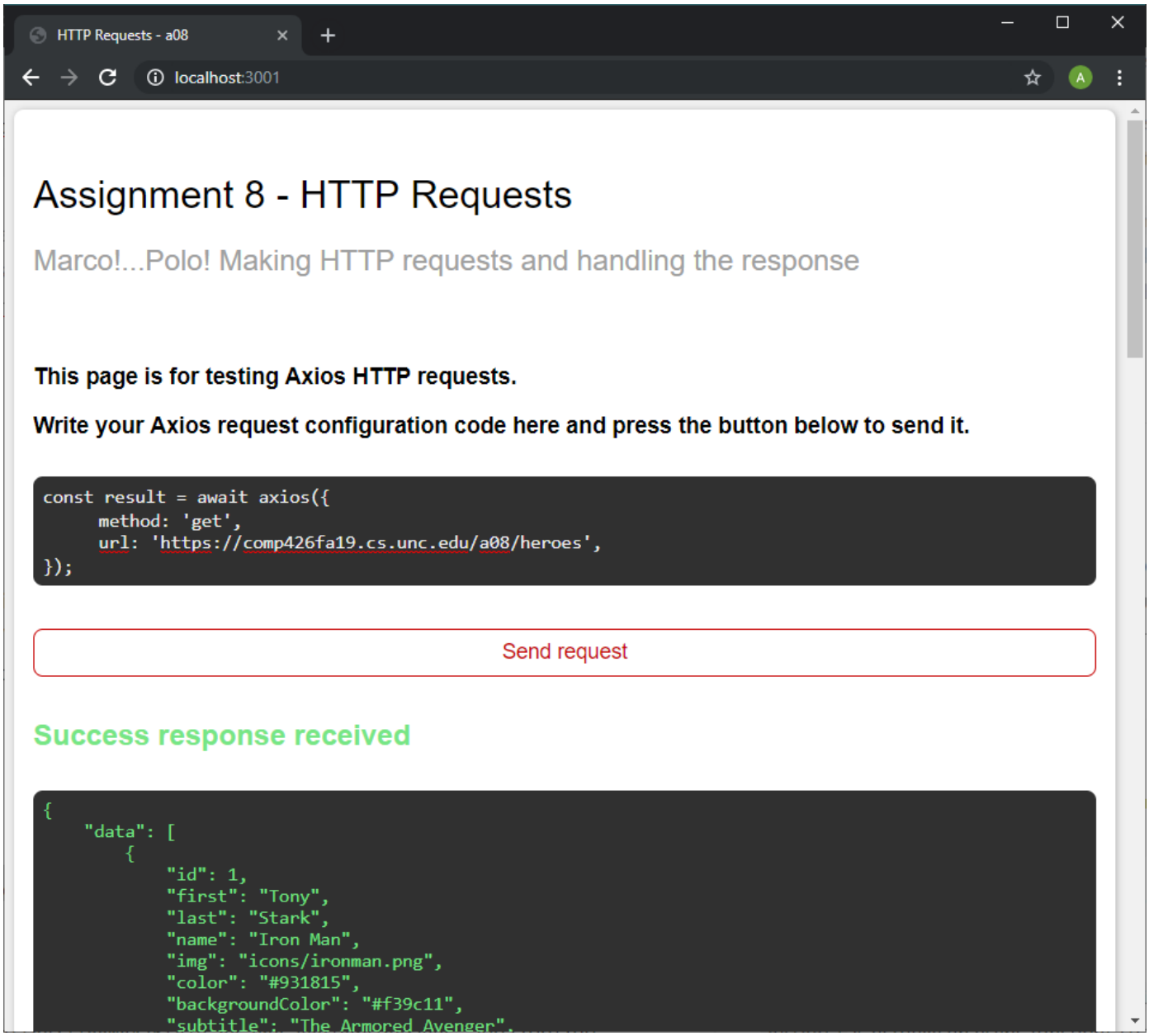
3. If the console drawer is open, click the X button to close it. This gives you more room to see the captured network traffic.



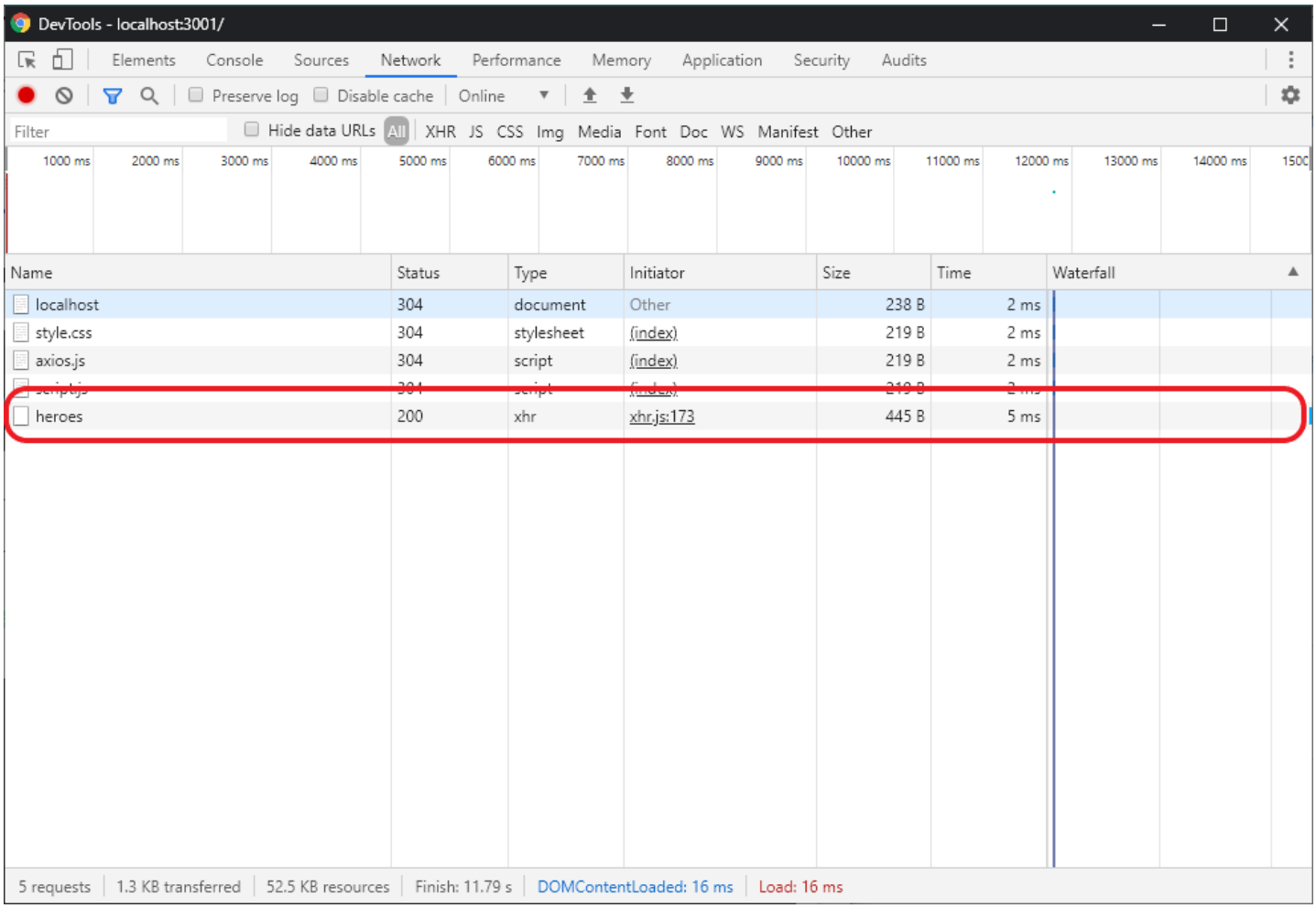
4. Try sending the simple request from Section 2.1 directly in the Axios testing app.



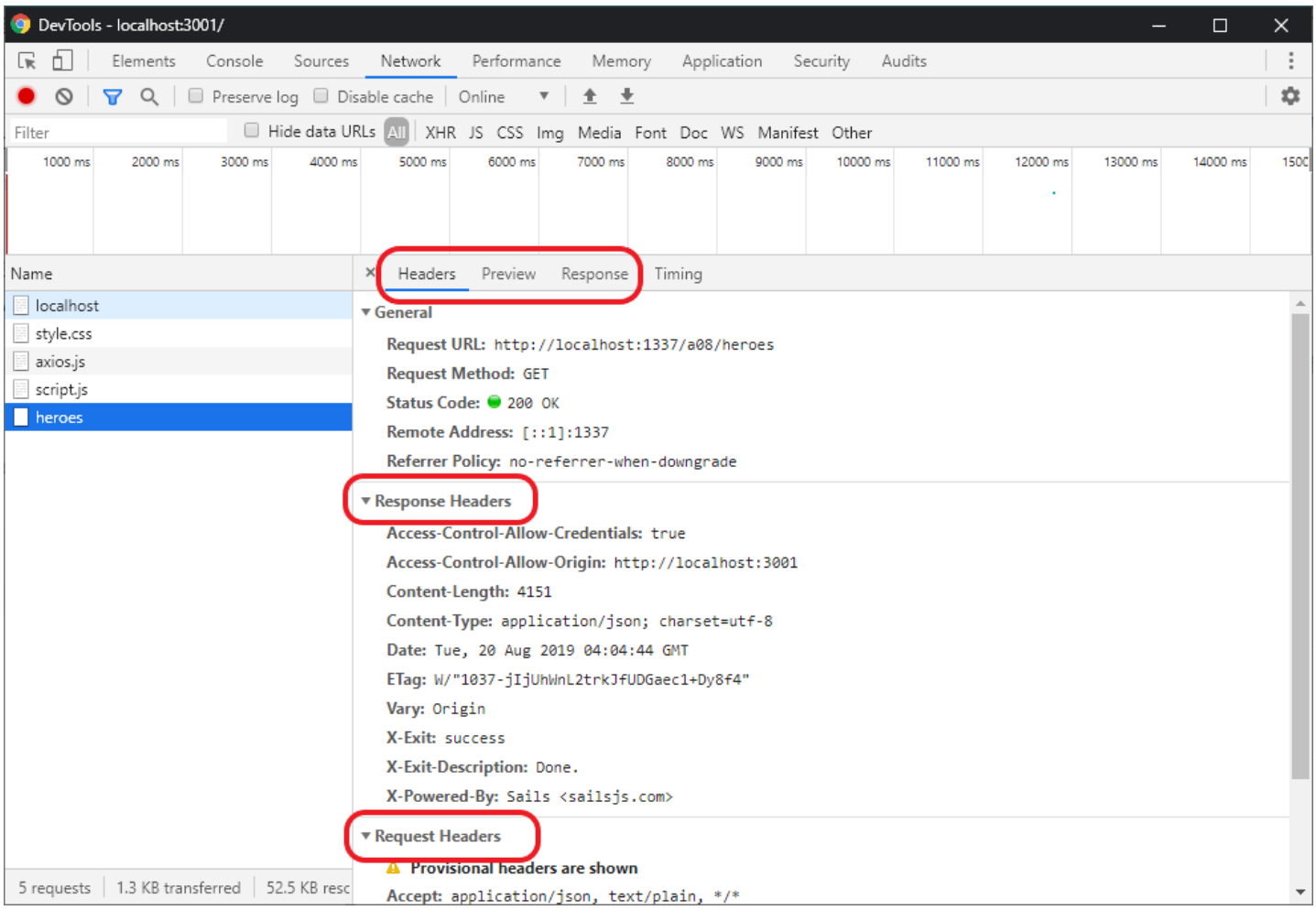
5. If successful, you should see the server respond with the hero data we used in a04 and a05. The text will appear as green-colored JSON in the Axios testing app.



6. However, *you should also see the request show up in the DevTools window*. Click on it to get more information about the request that you made.



7. Take a minute to click around and see all the useful information DevTools gives about the request! It tells you the HTTP request headers, the response headers, the response body, and more!



4. HTTP Error Handling

So far, we've demonstrated how the Axios library can be used to make HTTP requests and access the response. We also showed how to use Google Chrome's built-in DevTools window to capture all incoming and outgoing HTTP requests for a website.

However, so far we've ignored the fact that sometimes HTTP requests fail. Whether due to a loss of internet connectivity, the server going down, or a badly formed request, sometimes an HTTP request is destined for failure. Even if you don't expect your HTTP requests to fail, it's a good idea to write code to handle that odd situation where the request doesn't go through.

When you call the `axios()` function, it returns a `Promise` object---that's why we can use the `await` keyword to simulate synchronous code. When an awaited `Promise` object fails, it throws a JavaScript `Error`. JavaScript errors are exactly like exceptions in Java, which means they can be handled with a try-catch block.

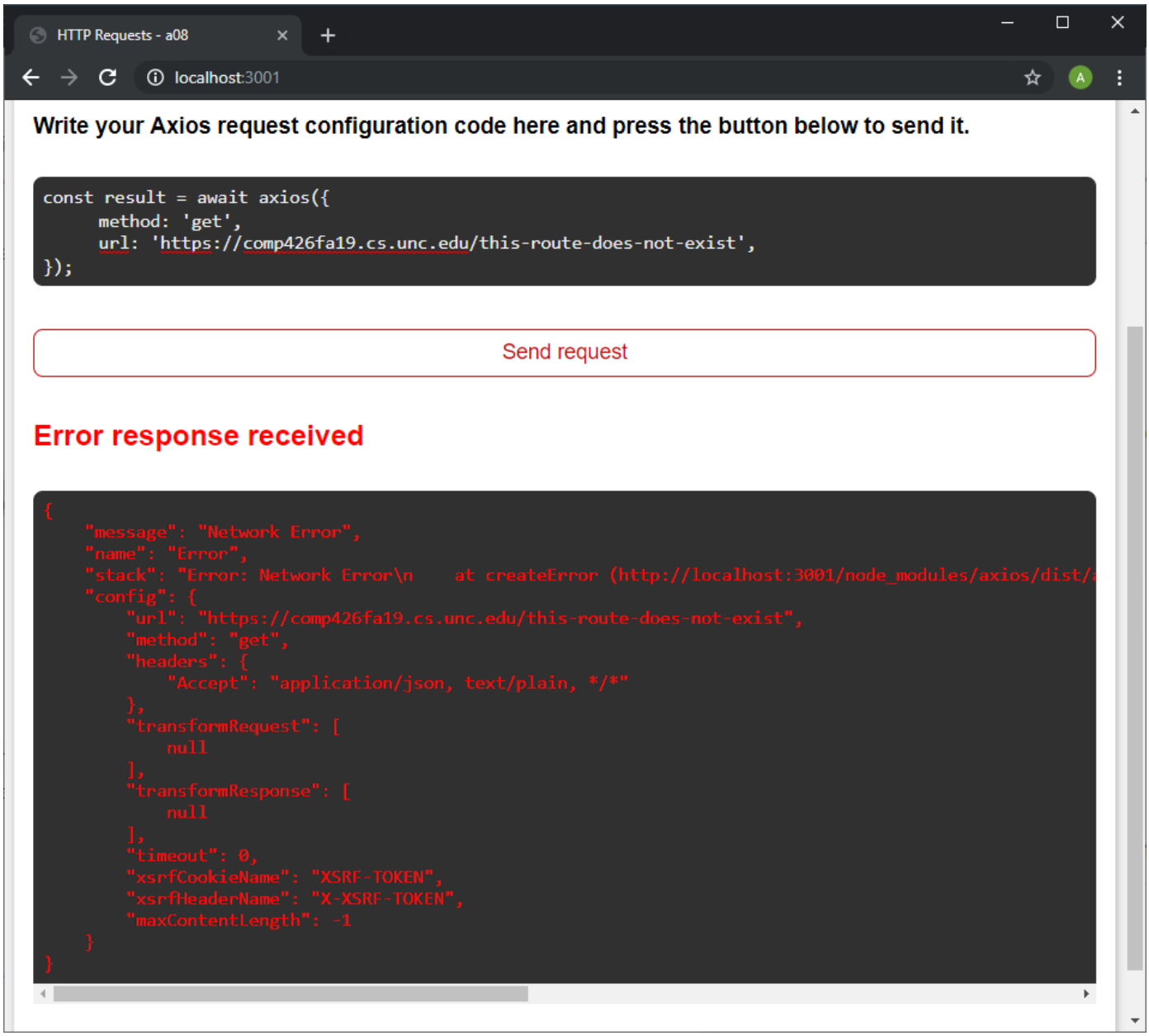
> When an awaited `Promise` object fails, it throws a JavaScript `Error`. JavaScript errors are exactly like exceptions in Java, which means they can be handled with a try-catch block.

Here's an example of an Axios HTTP request that uses a try-catch block to handle any failures that may occur during the request:

```
try {
  const result = await axios({
    method: 'get',
    url: 'https://comp426fa19.cs.unc.edu/this-route-does-not-exist',
  });
} catch (error) {
  console.log(error);
}
```

Notice how the catch block is given an `error` object? That's a lot like an "exception" object you might remember from Java. When an HTTP failure occurs, Axios uses the `error` object to pass valuable information back to you as the programmer about what went wrong during the request.

Try using the Axios testing tool to make a GET request to the URL `https://comp426fa19.cs.unc.edu/this-route-does-not-exist`. Since the URL doesn't exist, the `axios()` promise throws an error. The Axios testing tool is designed to catch this error, and displays the contents of the resulting `error` object as red-colored text in the response window.



5. Submission Requirements

For this assignment, your job is to fill in the functions listed in `/a08/submission.js` according to their specifications. Each of these functions requires you to `await` an `axios()` request, and handle the response. Do not worry about surrounding your code in a try-catch block *unless specifically instructed by the specification to handle HTTP errors*. Some of the functions require you to manage request headers and/or response headers.