Lesson Weekend

Intermediate JavaScript (/intermediate-javascript) / Test-Driven Development and Environments with JavaScript (/intermediate-javascript/test-driven-development-and-environments-with-javascript) / Modern JavaScript Development

Text

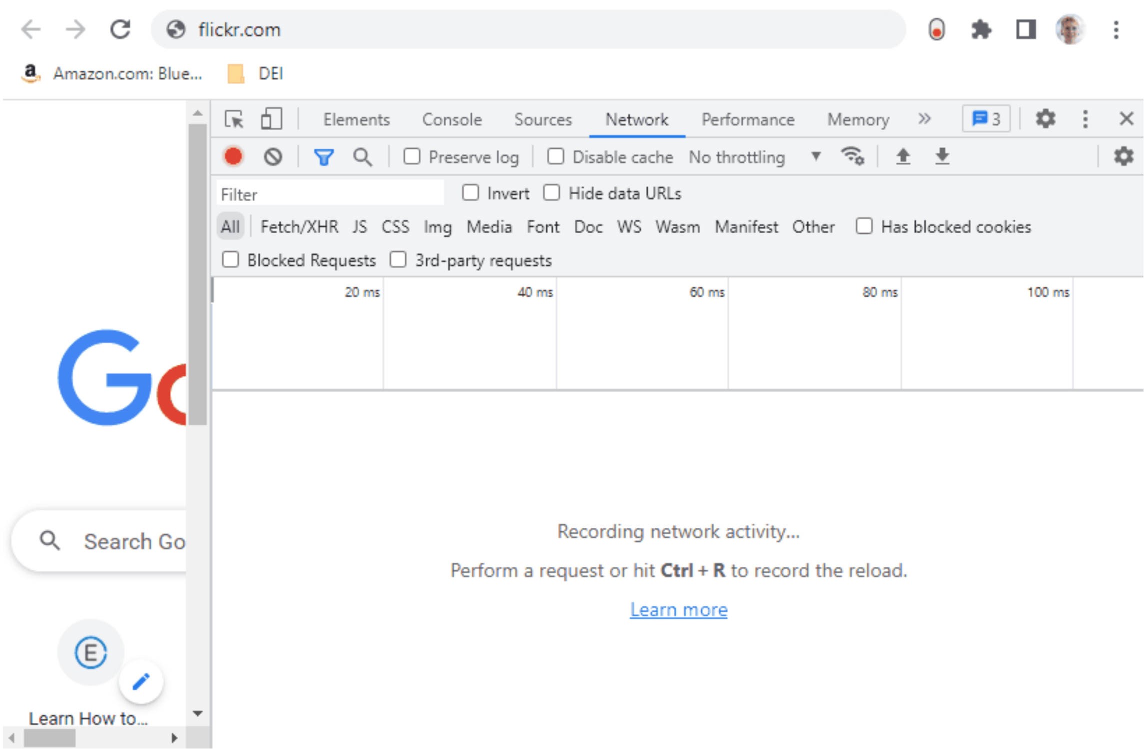
Imagine we're part of a development team working on a huge application. The application is live and thousands of people visit our site everyday. It has hundreds of pages, JavaScript files, and dependencies (like Bootstrap, or any other library we can incorporate into our project). After all, it's a huge site and many people use it!

Now imagine our site uses only the HTML, CSS and JavaScript techniques we've covered so far. Our `index.html` page has one hundred script and link tags in it, each representing a JavaScript file or dependency we need. This would be very messy and hard to work with. A single small change to a file could break everything.

There'd also be another huge problem: our site would be incredibly slow. Each file we load has its own overhead. When multiplied over many files, that overhead quickly adds up and leads to lost customers and lost revenue. Our customers expect the site to load as quickly as possible and it should be reliable!

We can actually get a sense of how many resources a site is loading via the *Network* tab in Chrome Dev Tools. This tab shows all the resources that are being uploaded and downloaded when we visit a page. We can open Chrome Dev Tools, click on the *Network* tab, and then visit any site we like to see this exchange of resources.

Let's take a look at Flickr's home page:



Recording network activity...
Perform a request or hit **Ctrl** + **R** to record the reload.

Notice the red box that pops up in the gif when we go to www.flickr.com — it's tracking the number of requests for resources that Flickr's homepage makes as it loads. At the end of the gif, the number of requests total to 94! That's a ton of resources! And that's just for Flickr's *splash page*, without even logging in.

Modern JavaScript Development

In the world of modern JavaScript development, performance and reliability are extremely important. It's essential to optimize our applications so they run faster, whether in production or development. A site like Flickr seamlessly downloads many photos, files, and website styles, all while giving users a good experience. It's simply not possible to build efficient and resource-intensive sites like Flickr with just the basic tools we've been using so far.

As our applications grow bigger, we need to consider two very important issues. We need to make our lives easier, not harder, as developers. We also need to make our applications faster, too. However, these two goals often conflict with each other. Here are some examples:

- With very large applications, it's much easier to organize and understand the codebase if we break it down into smaller files. So far, we've only worked with single custom JavaScript files (`scripts.js`), however, it's more common to have multiple JS files, each of which focuses on one slice of functionality in the entire application. As an example using our Address Book application, we might have one JS file for our `AddressBook` business logic, and another JS file containing the `Contact` business logic, and yet another containing the user interface logic. This is better for developers, especially in large applications! However, because each file has additional overhead to load, it takes longer for our machines to process them. Not so great for having fast, efficient applications...
- Clear variable names and easy-to-read syntax help developers write, read, and reason about code. However, our machines don't care whether a variable is named `veryClearlyNamed` or `x` . But there's one significant difference between the two — `x` takes less space in memory. To make our applications load faster, our files should be smaller if possible. But that's hard for developers to read...

For example, take a look at a small chunk of minified Bootstrap that we've downloaded (https://getbootstrap.com/docs/5.2/getting-started/download/):

```
@charset "UTF-8";/*!
* Bootstrap  v5.2.3 (https://getbootstrap.com/)
* Copyright 2011-2022 The Bootstrap Authors
* Copyright 2011-2022 Twitter, Inc.
* Licensed under MIT (https://github.com/twbs/bootstrap/blo
b/main/LICENSE)
*/:root{--bs-blue:#0d6efd;--bs-indigo:#6610f2;--bs-purple:#
6f42c1;--bs-pink:#d63384;--bs-red:#dc3545;--bs-orange:#fd7e
14;--bs-yellow:#ffc107;--bs-green:#198754;--bs-teal:#20c99
7;--bs-cyan:#0dcaf0;--bs-black:#000;--bs-white:#fff;--bs-gr
ay:#6c757d;--bs-gray-dark:#343a40;--bs-gray-100:#f8f9fa;--b
s-gray-200:#e9ecef;--bs-gray-300:#dee2e6;--bs-gray-400:#ced
4da;--bs-gray-500:#adb5bd;--bs-gray-600:#6c757d;--bs-gray-7
00:#495057;--bs-gray-800:#343a40;--bs-gray-900:#212529;--bs
-primary:#0d6efd;--bs-secondary:#6c757d;--bs-success:#19875
4;--bs-info:#0dcaf0;--bs-warning:#ffc107;--bs-danger:#dc354
5;--bs-light:#f8f9fa;--bs-dark:#212529;--bs-primary-rgb:13,
110,253;--bs-secondary-rgb:108,117,125;--bs-success-rgb:25,
135,84;--bs-info-rgb:13,202,240;--bs-warning-rgb:255,193,7;
--bs-danger-rgb:220,53,69;--bs-light-rgb:248,249,250;--bs-d
ark-rgb:33,37,41;--bs-white-rgb:255,255,255;--bs-black-rgb:
0,0,0;--bs-body-color-rgb:33,37,41;--bs-body-bg-rgb:255,25
5,255;--bs-font-sans-serif:system-ui,-apple-system,"Segoe U
I",Roboto,"Helvetica Neue","Noto Sans","Liberation Sans",Ar
ial,sans-serif,"Apple Color Emoji","Segoe UI Emoji","Segoe
UI Symbol","Noto Color Emoji";--bs-font-monospace:SFMono-Re
gular,Menlo,Monaco,Consolas,"Liberation Mono","Courier Ne
w",monospace;--bs-gradient:linear-gradient(180deg, rgba(25
5, 255, 255, 0.15), rgba(255, 255, 255, 0));
```

Good luck reading and understanding that! The truth is that the above code isn't for humans — but rather to make the files smaller so our machines can load them more quickly. It has automatically been **minified** (made smaller) to keep the file size as small as possible.

As we can see, a good developer experience and efficient applications can be at cross purposes. However, we can address the issues above by separating our code into **development** and **production** environments. Code in a development environment is convenient for developing — easier to read and work with, but slower. Code in a production environment is easier for machines to work with and much more efficient.

Because we aren't building polished, production-ready applications yet, we'll be focused on working in a development environment. Even so, we will still utilize tools to make our code more efficient and our lives easier as developers.

Here are some of the tools we'll learn to use in the following lessons:

- **npm** (**Node Package Manager**) will allow us to easily install and manage JavaScript packages in our projects. A package is also called a **dependency**, and it's any prepackaged code that we can download and use in our project. A good example of a dependency that we've used so far is Bootstrap, a style library that also uses JavaScript for interactivity.
- **webpack** is a module bundler that will **concatenate** and **minify** our code. A **module** is a file — JavaScript, CSS, an image, or another asset. That means we can write code that's easy for developers to read, and webpack will handle automatically concatenating all our files into just one file and then minify the code (like the Bootstrap example above). We can use webpack to add lots of other useful functionality to our projects as well.
- **Jest** will allow us to write and run automated tests to ensure our business logic is working correctly. We will install the Jest package using npm, and it will be saved as a dependency in our project.
- We'll also install additional packages (dependencies) to improve our development experience:

- `html-webpack-plugin` will allow us to easily generate HTML files based on templates.
- `webpack-dev-server` will automatically reload our code in the browser as we make changes to it. Great for development!
- `eslint` is a code analysis tool that we can set up to automatically review our code and notify us when any code is poorly-written or has errors. This helps us quickly identify and fix bugs.

We'll do all of these things and more in this section. In the process, we'll learn how to use new JavaScript features that were released with ES6 (released in 2015 as ECMAScript 2015, the latest stable release of JavaScript that is implemented in all modern browsers).

One Last Thing...

The next two sections of class are going to look and feel very different from the work we've been doing. Specifically, these sections are a big step up in terms of difficulty. Up until this point, we've been taking things slowly and focusing on learning the fundamentals. From this point on, we're going to be writing our code the same way it's done in a professional setting. We're about to jump into the deep end.

Creating a development environment can feel very abstract and it's often difficult to debug when things go awry. This is a common feeling among developers both new and experienced. You may feel frustrated while going through these examples but keep pressing forward. While struggling through this material may feel frustrating at times, there is no other way to get comfortable with setting up and problem-solving environments than to just jump in. Do your best, accept partial understanding, and know that your efforts will benefit you not just at Epicodus but also in the programming industry.

Right now, we are building the toolset we need to succeed as developers. That includes problem-solving, being patient with ourselves, managing frustration, and learning how to be comfortable even when we feel like we are in over our heads.

Previous (/intermediate-javascript/test-driven-development-and-environments-with-javascript/test-driven-development-and-environments-with-javascript-objectives)
Next (/intermediate-javascript/test-driven-development-and-environments-with-javascript/basic-project-structure)

Lesson 2 of 49
Last updated January 19, 2023

disable dark mode

Epicodus (http://www.epicodus.com)

© 2023 Epicodus (http://www.epicodus.com/), Inc.