Front End Web Development: Get Started

by Joe Eames

Start Course

Bookmarked Add to Channel Download Course

Course Introduction

Course Introduction

Hello, I'm Joe Eames, and welcome to Pluralsight's course: "Front End Web Development: Get Started. " The very first thing I want to do is discuss front end web development, what it is and what it isn't. Front end web development involves the browser and everything we send to it. This includes: HTML, CSS, JavaScript, Images and anything else sent to the browser to make our web application. What it doesn't include is any frameworks rendering on the server. It only includes the server so far as to specificy how those assets are transmitted, so long as it's something that matters to the browser. So, for example, making sure that the data transmitted is compressed is a concern of front end developers, but how to make the server compress that data is not a front end concern. That being said, rare is the job where the developer only has to deal with front end concerns and all server side matters are dealt with by other developers. There are jobs like this, I myself had such a job, but in general, the guy doing the front end is also working on the server side or what is sometimes referred to as the back end. So this course will only be dealing with the front end. That means that we won't discuss any server side items, we will only be discussing them from the viewpoint of the front end. Front end web development is an incredibly complex subject, just by the sheer number of different technologies involved when putting together a webpage. The list of technologies you may hear referenced can become extremely daunting. Here

is just a small selection of common web technologies that are frequently used when delivering a typical webpage. Let's take a moment to soak that all in. All of these different technologies can make it feel like becoming competent in front end web development is something you have to dedicate years towards, but that is definitely not the case. Although mastering these technologies might require lots of time, mastering them is not a requirement for becoming a competent front end developer. Getting a basic understanding of these concepts is very accomplishable in a reasonable amount of time, and a little practice will go a long way to making you comfortable with all of these and more.

Course Website & Errata

When starting out this course, the very first thing you should do is check out my course website. That URL is here. This is where I put any RATA or other updates to the course so that viewers can see what's coming and get access to any public files that aren't easy to get elsewhere. Web technology is moving at a frightening pace, because of that and the significant time it takes to author and publish videos, there can be a pretty big delay between when an issue is discovered or an update is planned and when I can get the videos published. So, please, take a quick minute to check this site to see if there's any pertinent information you need when viewing this course.

Course Outline

Let's look at this course in its entirety. We'll start with this introductory module. I'll tell you more about this module in a minute. Our second module is all about learning and solving problems as a front end web developer. We'll talk about tools to use to solve problems, good reference websites and ways to stay up to date. Our next module will be on HTML and CSS. We'll approach these from a standpoint of front end web development, so we'll pay particular attention to areas that affect development and look at some of the core HTML file APIs that you should be familiar with. After that, we'll learn the basics of JavaScript specifically what makes it different from other programming languages you might be familiar with. Our next module will be on HTTP and interacting with the server. Then we'll talk about browsers and some of the differences between them. How to use the developer tools in them and how to make sure that your apps run across all browsers that you care about. The next module will be about JavaScript and the browser. The browser's the execution environment for JavaScript when building web applications. So understanding some of the particular nuances and challenges that put the browser as a platform is an important topic. After that, we'll take about some of the basic libraries and tools that any

front end web developer should be familiar with. And lastly, we'll discuss performance and some techniques that will help us keep our web apps nimble.

Module Outline

Now that we've seen the course agenda, let's look at the agenda for this module specifically. We already looked at the course agenda. Now we're looking at the agenda for this module, and next we'll look at the prerequisites you should meet in order to get the most out of this course. And then, we'll talk about who should watch this course and what background you might have. And finally, we'll talk about the purpose or goals for this course.

Course Audience

This course is fairly rudimentary, especially compared to a lot of the courses that Pluralsight offers, and especially many of the courses that I myself have authored. Because of this, the prerequisites to this course are fairly basic. First, you should have a basic understanding of JavaScript. If you only have experience with other languages then you will definitely struggle with some of what we go over, but you will still benefit from much of this course. Second, you should have a good understanding of the Web and you should be comfortable with browsers and how to navigate using them. You should also know and be comfortable with common terminology used on the Web. Third, you should know the fundamentals of HTML. Fourth, you should be familiar with, and comfortable reading, CSS. And the final prerequisite is Node. This prerequisite is not nearly so important. I only mention this one because there are some tools that I will use in only a couple of places that are built on Node. If you don't care about these particular tools then you can safely ignore Node, but after seeing them, if you want to try them out, then check out Pluralsights course on Node to learn how to get it installed and running. Even though this course is fairly introductory, don't assume that means that if you have been doing web development that you won't have anything to gain from this course. I will be taking a look at some fairly advanced topics and although we won't go into too much depth about them, we will see how the fit into the overall picture of front end development. So even if you have been doing web development for many years, you are likely to find things to learn in this course simply because of the vast number of topics we will be covering. We will be covering a lot of topics from a very high level. So, if you ever find yourself getting lost, click 'Search' on Pluralsight. com for that subject. We'll probably show you other courses that you can refer to in order to get a better understanding of the topic being discussed.

Prerequisites

I'd like to take a moment and talk about who this course is for. The short answer is that this course is for me. Several years ago, I was a server side web developer who decided to get more serious about front end development and really learn JavaScript and improve my client side skills. At that time, front end web development was not nearly as prolific as it is today. So, it was much harder to piece together all the different skills that I needed. So, I spent a lot of time searching and reading, and much of that time was far less effective than I would have liked. What I needed was this course. I knew a lot about writing web applications, but not enough about doing it on the client. There was tons of stuff I didn't know about that I needed to learn to be competent on the front end. This course solves that exact problem. This course is for anyone who has done some web development and understands the basics of HTML, CSS and JavaScript, but is ready to get serious on the front end, or someone who has started getting serious but wants to fill in the gaps they haven't already filled in. That means that we won't cover the basics. Instead, we're going to cover these possible gaps in your knowledge to take your front end skills to the next level. So, as a result, this course is not going to teach you HTML, or CSS or JavaScript. This course assumes you know those basics. Instead, it's going to fill in the holes in those technologies that you likely have if you have limited experience with front end web development, and help you cross the bridge to doing truly cool things on the client. If you're not familiar with any of these technologies, you should check out some of the many great courses that Pluralsight has on HTML, CSS and JavaScript. If you're watching this course, you are likely from one of the following categories: a server-side web developer who wants to do more on the client, or a new programmer who has only done some smaller projects on the client but wants to continue to grow your skills, or possibly a non-web developer who has been doing web development on the side. Of course, you might not fit into any of these backgrounds but can still benefit from watching this course. Regardless of which category you fit in, if you want to improve your front end web development skills then this course is for you.

Course Goals

The primary goals of this course are to first, lay down the foundation of front end web development and to fill in any holes in your knowledge of the landscape to help you on your way to becoming well rounded in front end web development, and thereby, enable you to move on from here into more specific areas without feeling lost. We will endeavor to give you the widest possible basis of understanding, so that you can from there, move on to as many further topics as possible. But you should take note that we will cover a lot of topics in a relatively short amount of

time. So, we are not trying to help you gain a mastery of any specific area, but instead to give you a general level of understanding so that you can leverage that fundamental understanding and jump into more advanced topics. And that will conclude our introduction into this course. So let's move on to the next module and get started learning.

Learning and Solving Problems

Introduction

Hello, I'm Joe Eames. In this module, we're going to look at different ways to learn about front-end technologies, try out solutions, and solve problems that you may have. We'll start this module by learning about the Mozilla Developer Network, a website that should be a go-to resource for any web developer. Then, we'll look at the browser console and how to use it to try out things. After that, we'll look at four different online coding utilities, JSFiddle, JSBin, CodePen and Plunker. These utilities will enable you to try out more complex problems than a console will but will also keep you from having to worry about dealing with files in your hard drive and eliminate the time it takes to set up a new project in your favorite code editor. Finally, we'll talk about staying up to date with front-end web technologies.

Mozilla Developer Network

When looking up information about different aspects of front-end development, not all search results are created equal. For some of the more basic front-end technologies, there is a specific website that generally has the most accurate and useful information, and that is the Mozilla Developer Network or MDN. The MDN is a type of wiki that anyone can contribute to. It has quickly become the best resource on the web for reference information about HTML, HTML5, CSS, CSS3, JavaScript, the DOM and related technologies. It also usually has a high ranking when searching for something specific about one of these technologies. Let's say for example, that I wanted to find out more about the h1 tag. If I just search for h1, we can see that the MDN is actually the third result. Unfortunately, the first result is from a less effective website, w3schools. This is definitely a website that you want to avoid. It really doesn't cover anything that the MDN doesn't cover, but it occasionally has incorrect information and examples. So you can quickly fix

that by prefixing any search with MDN. And now you can see that this gives me MDN as the first result. Let's look at another example search. Let's say that I'm creating an input box, and I want to use one of the new HTML5 types but I can't remember how many types there are, so in Google, I type in MDN input and type. I'll select this first result, and if I scroll down, I can see the complete list of types that are available on the input element. So whenever you're doing any searches like this, check the MDN first, then look at other resources. As a reference for these technologies, the MDN just can't be beat.

The Browser Console

Perhaps no tool is a better friend of the front-end developer than the browser console. Let's take a look at it. In Chrome, on a Windows machine, to open up the console, use the key stroke combination Control-Shift-I. On a Mac, that key stroke is Command-Option-I. The console is a full JavaScript environment so you can run commands within it like creating variables, and just typing in the name of a variable will display that variable to you. Of course, the standard way to interact with the console inside of one of your programs is to use the console. log statement. Now you can do more complex things inside the console like defining a function, and I can call up function directly from the command line, and we can put in more complex functions as well. This is one of the ways that we can get our console to run multiple statements of the same line is by putting things inside of a function and then executing that function. Now, we can always fit everything onto one line, but if we need more room within the Chrome console, we can type in Shift-Enter, and that will take us to a new line where this will all execute together when I hit Return. You can also use the console to explore objects. Let's say I had an object like this with these three properties. Now if I knew that there was an obj object, but I couldn't remember what it looked like, I can inspect it from the console by just typing in the name, and it will print out the object for me. But if the object is more complex, such as this object here, then when I write it out to the console, I actually get a short summary of the first few properties of the object and then I get this explorer that I could click down here and it will show me all the properties of the object. And if these properties have their own properties, I could drill down within them as well. Now, some objects are rendered differently depending on the nature of the object. For example, the document object, appears to be just a regular JavaScript object. But if we drill down inside of it, we can see we're actually getting the HTML for the document, and as I drill down, I get more HTML. But the document object also has properties and methods that we can call. If I'm interested in those properties and methods, and not the HTML, then instead of just writing out the document, I can try console. log(document). Well that's going to give me the same result as

before. console. log is going to show in its default representation. So there's another command that we can use instead of console. log, and that is, console. dir. This renders the object as a JavaScript object, so if I click this open, you can see I'm now looking at properties and methods that belong to the document object. And when my console gets a little bit too cluttered up, I can always come up here and click this to clear the console. Now let's take a quick look at the Firefox console to compare how it works versus the Chrome console. In order to open up the console in Firefox, use the same key stroke Control-Shift-I, and you can see that this console already has a lot of statements written out to it. If I click Clear, then that will clear it up. And one thing that's different, is the place where we type is down here at the bottom instead of at the top. Let's look at the document object inside of Firefox. When we just log it out, we don't get the same ability to browse through the HTML inside of Firefox as you do inside of Chrome, but if we do a console. dir, then what we get is this little window inside of the console that shows the properties and methods on the document object. And we can scroll within that and view those, and we can drill down inside of them as well. Other than that, the consoles work fairly similar. There's some filtering options here and you can turn on and off certain kinds of logging. Whereas if we go back to Chrome, there's a filtering option right here which is very similar to the Firefox filter, but you can see that there are some minor differences. So there's an introduction to the consoles in Chrome and Firefox. The consoles' a great way to try out code instead of being forced to create a JavaScript file just to try something out. We'll be using this tool frequently in this course.

JSFiddle

The console can be a great way to test out some code but it can be really limited so sometimes it's just not enough. Sometimes you want to try something out that's a little more complex than the console can handle but you still don't want to create a whole bunch of new files, create an entire project and track it on your hard drive. Or maybe you want to easily share your idea with other people. There's an entire class of tools that have been developed in order to enable this. We'll look at three of them in this module. The first of those is JSFiddle. JSFiddle allows you to try something out in the browser without having to worry about tracking files and it will even save and version your ideas for you. Let's take a look at an example so you can see what I mean. Let's say I want to play around with how buttons handle clicks, so I'm going to specify a little bit of HTML. I'm just going to create a button. One of the nice things about JSFiddle is that it supports some convenient key strokes when authoring HTML. You can just type in the name of a class and hit Tab which is called Zen Coding, and it will turn the word that you typed into an element. I'm going to give this button a text Click Me, and I'm going to give it a non-click event and put in

some JavaScript. And I'm going to call an alert that says clicked. And I'm not going to put any JavaScript in but I'm going to add a little bit of CSS. I'm going to add a rule to make this button Pluralsight orange. And now that I've got that already I can just click Run. We can see that our button shows up down here and if I click it, then our alert shows up. Now let's talk a little bit about some of the options that JSFiddle has. The first option is you can choose a library here to use to manipulate your HTML. So it gives you all the standard options. It starts out with jQuery and Mootools, Prototype and YUI. Then it gets down into some other libraries such as Raphael and Right and Three, Enyo, Knockout, Angular and Amber and Underscore. So you can choose the library that you want to use and then you can tell it when you want that library loaded. You can choose onLoad, onDomready or just to put it in the head or the body. You can also set some options about your Fiddle. You can name it and give it a description. You can tell it you want Normalized CSS. We'll explain that in the next module. You can tell it what you want the body tag to be like. Say for example you need your body tag to have a specific class, you can type in body class="joe" and now that's the body tag that your sample document will have. You can also specify your DTD, of course it defaults to HTML5, which is what you should mostly be using. The Framework script attribute is something you probably won't have to worry about too much so let's skip that. On the External Resources, you can add in additional libraries that you want used inside of your Fiddle. So for example, if you want to use a couple of jQuery plugins, so long as those are published somewhere on the Internet, you can type in their URL and add them here, and they'll be added to this Fiddle. You can also select your Language so, for example, instead of JavaScript you can author in CoffeeScript. And instead of CSS, you can author in SCSS which will be explained in the next module. You can also set some specifications for Echo apis for Ajax calls, and then of course there's some Legal, Credits and Links down here. So now that I've got this Fiddle looking the way that I want, I want save it. Now you can see that I've logged in. Creating a new account with JSFiddle is free. Once I've done that, if I click Save, you can see that the URL has changed and now this is the Fiddle. If I just copy this URL, I can bookmark this and come back to it later on. But there's also some other nice things that JSFiddle will do. If I click here and go to my Dashboard, I can see a list of all the Fiddles that I've created. Here's the Fiddle that we just created so let's go back to that. Let's say i need to make a change to this Fiddle. For example, instead of using alert, I want to do a console. log. If I click Update, you can see that my URL has changed. It's actually versioned this Fiddle so it's storing both versions for me to use, and I can go back a version if I want. If I've decided that this version is really the valid version and the previous versions don't matter, then I can click Set as base and this becomes the new base for this Fiddle. And if I work on a Fiddle for awhile and I have it where I want it, but I want to try an entirely different idea but not version it, then I can click Fork and it'll create an entirely new Fiddle based

off of this one. Now let's say that I've authored some JavaScript and for one reason or another, my indentation got off. And I want to fix the indentation. I can just click this TidyUp button and it will go through and fix the indentation in my code. And of course, one of the really nice features is JSHint. Let's say I've got a small problem in my JavaScript, the JavaScript is still working but I want to run JSHint against it, I can click this and it'll show up here and tell me that I'm missing a semicolon. JSFiddle also has a Collaboration feature so that you can work on a Fiddle with somebody else. And lastly, it's got the Share feature, so you can go down here and grab a link to share this Fiddle or a fullscreen or a way to embed it on a page, and of course, a link to share it on Twitter or Facebook. So that's JSFiddle in a nutshell. In the next section, we'll look at JSBin.

JSBin

JSBin and JSFiddle have a lot of feature parity. Most of what you can do in JSFiddle you can do in JSBin, and vice versa. There are a few features that are different and of course, the layout is a little bit different as well. One of the nice things about JSBin, is that you can turn on and off the different pieces, depending on what you need. So if you don't need CSS, you just need to manipulate HTML, you can click that and the CSS goes away. If you do need CSS, you can turn it back on. Also you can turn on JavaScript and get a panel for that as well. Another really nice feature of JSBin is the fact that the preview is live. So as you modify your HTML and CSS, the preview will instantly update. So let's go in, and do the same thing we did in the JSFiddle demo. Let's add a button, and just like JSFiddle, you can use the Zen Coding by just tabbing after you type in a word to turn it into an element. And you can see that as I type that in, the button automatically appears without having to hit any kind of a run button up at the top. Unfortunately our button's in a funny place, so let's add a BR tag, move it down, make it a little bit more visible and let's add in our CSS. And you can see that it changed as we typed in the CSS, and that gives us some really nice features if we want to play around with, say the background color. We can try different shades of gray and it'll instantly update and we can keep that up until we find one that we like. And since it's auto-updating we don't have to click Run, we can just click the button and we see our alert. Another really nice feature of JSBin is that you get a console window. So I can do the same things I can do with the console like declare variables, and inspect them, but unfortunately this console does not operate exactly like the actual console. Because if I go over here and change this to say console. log instead of alert, and then I click my button, you can see that it's not printing anything out here but if I open up the console, we can see that the message clicked has actually appeared inside the console. Another really neat feature of JSBin is that it has this feature called Export as gist, which will create a new GitHub gist from this JSBin. So if I log

into JSBin using my GitHub account, then if I click this button, it will create a new Github gist and I can click Open and it will show me that just here. JSBin will do the same thing that JSFiddle does by letting you save, by creating a milestone and you can also do the same thing as Fork by just clicking Clone or using one of the keyboard shortcuts for those two functions. You can also see a list of all your bins here and you can delete the bin from here. Now if you want to add an external library, you just click this and you can see a list of all the libraries that they have available for you to add. Now this is a little bit different from JSFiddle because JSFiddle actually lets you put in your own URL to add a resource. JSBin doesn't have that feature. But what you can always do is come right into your HTML and add a script tag and from the script tag add in the source attribute and specify the URL that's available on the Internet and that will be saved as part of your bin. And of course, just like JSFiddle, when you're ready and you want to share it with other people, you can click Share and it'll give you appropriate URLs in order to share this with other people. And that concludes our look at JSBin.

Codepen

The third tool we'll look at is CodePen. CodePen has essentially the same features of the other products we've looked at. But it does have a few small features that distinguish it from the others. So let's start by putting in some demo code. We'll put in our button that we've put in the previous demos, and we'll also style it up with some CSS. And you can see, as I paste in code, the result automatically updates itself. Now I'm going to go ahead and save this pen. And that will give me some new options. If I make any changes, clicking Save will update the pen that I'm using, so let's change this alert to a console. log, and I'll click Save, and you'll notice that the URL didn't change. One of the features that CodePen doesn't have is keeping track of the different versions of your code. You can Fork CodePen which will give you a new version that you can take in a different direction but that's not the same as version tracking. You can click Info here in order to give your pen a title and a description and a set of tags, and of course you can click Share to share this with other people either using the URL or here's a unique feature, you can actually send an SMS text with the full page URL, and you can save a pen as a GitHub gist, or you can export it as a zip. Now one of the things that makes CodePen unique is all the different options that you have for each of these different panes. These options are available by clicking the gear icon next to each of the panes. So let's start with the HTML options. You can see that we've actually got a lot of different languages we can choose to author our HTML. Either plain HTML or you can use Haml, Slim, Markdown or Jade, we can add classes to our HTML tag or put stuff into the head, and we can analyze our HTML using the HTML Inspector provided by the W3C. For CSS, we've got plain CSS,

then we've also got two different versions of Sass, LESS and Stylus available. You'll learn more about those in the next module. We can also Normalize it or Reset it or do Neither. These are things that we'll learn in the next module as well. You can use Prefix free CSS. This is a nice little feature that prevents you from having to use browser prefixes. And you can also link to an external CSS or another pen, and you can analyze your CSS via CSS Lint. And finally for JavaScript, you can author your JavaScript in plain JavaScript or CoffeeScript or LiveScript. You can use the latest version of several different libraries. You can see here that the list of libraries isn't nearly as comprehensive as it is in some of the other products, and you're limited to only the latest version. Of course, you can always link to an external JS file if you wish. And there's a nice check box to add in Modernizer. And then like the other two panes, you can analyze your JavaScript using JSHint. And if you don't want your JavaScript to automatically run every time you change it, you can turn the Auto Run feature off. So that will conclude our look at CodePen.

Plunker

The last tool we'll look at is Plunker. Plunker is quite a bit different from the other tools that we've looked at so far because Plunker doesn't give you just one JavaScript file, one HTML file and one CSS file to manipulate. Instead, it allows you to create any number of files. Over here on the left, you can see that there are four files in the sample template Plunk. index. html, readme markdown, a script file and then a style file. But if I wanted, I can add new files by clicking the New file button and then name the file whatever I want. Plunker also allows you to create new plunks from a template. If you click this drop-down here, you can see we can choose a new jQuery template, jQuery UI, AngularJS, and then choose our version. Bootstrap, Backbone, YUI, or KendoUI. So let's create a new plunk using jQueryUI. And now if I want to run my plunk, I click Run, and it pulls up my preview over here in this panel. So if I want to have more script files, you can see all I have to do is refer to them here inside of my HTML document. So if I add another script file, I go back to my index. html, copy that line, put it in and script2, and in this script file, let's call it alert. And then it auto-runs and says hi. Now if I don't want it to auto-run, I can click Stop and then just modify my files, and then I can turn the Run back on, and it's running again. Now let's take this alert out since that can be annoying. If I save my Plunker, then I get some more options. At this point I can Freeze, and keep it at the same version. I can Fork it into a new plunk. If I click this, when I fork, I can fork it to a public plunk instead of a private plunk. You can see the Privacy is set down here. I can delete the plunk up here, I can create a new one, and of course, I can turn off the preview. Over here, I can beautify my code. I can save it to my list of templates. I could save it to my favorites. I can download it as a zip file and I can open the embedded view. This is a pretty cool

feature. When you click this, then you get the preview where it takes up the entire window rather than a netted view. Another nice thing about Plunker is that it'll tell you how many versions you've got and give you a nice interface for looking through the versions. Over here in this right panel, over here, you've got a bunch of options for changing what you want to see over here. So there's the Live Preview which we're seeing now. You can find an external template so you can search packages. You can also look at your own templates and popular public templates. You can see the info about your plunk, you can look at code linting information about your plunk, and then you can go into the Collaboration options so that you can work live with somebody else. And of course, get into Editor Options. So as you can see, Plunker's got a lot of really cool features, and of course, it really sets itself apart from the others because you can have multiple files inside of your plunk. Plunker sharing options are really quite straightforward. You simply give out the URL to anybody else that you want to be able to see your plunk. The other users will be able to edit your plunk but they won't be able to save those changes. If they want to save changes, they'll have to fork it off themselves. A private plunk like the one I've got here is only visible to those who I give the URL out to. If it's a public plunk, then that's discoverable by other users. So for any quick trial projects where you may need more than one file of a given type, then Plunker is the tool to use. One of the big reasons to learn to use these tools is being able to get other people's help when trying to solve and diagnose a problem that you're having with your code. Trying to explain a problem in a form or comment just isn't sufficient. You really need to be able to create a live version of what's breaking about your code if you want other people to help you out. So being able to adequately work with one of these online coding sites is really critical in order to obtain help from other people. Each of these tools that we've looked at allows other people to look at what you're doing, help diagnose what's going wrong, fork it, and show you how to fix your code. So no matter which tool you choose, become familiar with at least one of these and learn to use it when asking for help on any semi-complex problem.

Keeping up to Date

One of the important pieces of improving your front-end skills is staying up to date with developments and changes in this particular segment of the industry. Even among computer technologies, front-end development is one of the more rapidly changing ones. Developments and innovations on the front-end are frequent as the industry seeks ever more effective ways to quickly build maintainable, performant, and usable web front-ends. There are many ways to keep yourself up to date and in this section, I will cover a few recommendations. The first thing I'll do is recommend a couple of people to follow over social networks like Twitter, Google Plus and their

blogs. And that is, Paul Irish and Addy Osmani. These two have a long track record of producing some of the most frequent and easily consumable media on front-end developments, mostly because it's their job. They both work on the Chrome team at Google, and their jobs are primarily about improving the web for developers and acting as advocates. So it's their responsibility to stay aware of developments in the industry and you can save yourself a lot of time by just following them and seeing what kinds of things they're talking about. There are so many other people who are great to follow, it would be impossible to list them all, but I have pick these two simply because of how long they have been at the forefront of our industry and the large quantity of consumable information they produce. The next recommendation is a couple of podcasts. Some of the criteria I use to recommend a podcast is that it is focused on web front-end and that it has been around awhile and regularly produces content. Sadly, at this time there are only two podcasts that fit this bill. The first is the Shop Talk Show which is focused on front-end design, development and user experience. They publish roughly weekly and every episode has a new guest talking about a particular topic related to front-end development. The second one is the JavaScript Jabber podcast. This is a little self-serving since I'm one of the panelists of the podcast but it really is a fantastic weekly podcast focused on JavaScript and all related technologies and has some of the coolest guests. And of course, the panelists are pretty okay as well. Lastly, I'll talk about a few newsletters that you should subscribe to. This is a pretty easy area to cover. The best newsletters on front-end technologies are four weekly newsletters conveniently named JavaScript Weekly, HTML5 Weekly, CSS Weekly and Node Weekly. Now you may think it odd that I listed Node Weekly. There's so much of Node that's about front-end developer tooling, that even if you don't care about it on the backend, it's still a technology that you should keep up to date with. Of course, there are many other ways to keep up to date with developments on the front-end and I highly recommend you search out other people to follow, podcasts to listen to, and newsletters to read, as well as anything else that helps you keep yourself up to date.

Summary

In this module, we looked at several different ways to learn and solve problems with front-end web technologies. We looked at the MDN and several different online utilities and we also looked at some recommendations on keeping up to date. All of these tools and recommendations will be a big help to becoming a more competent front-end developer. But the one thing that is truly more important than any specific technique or tool is simply a strong desire and excitement about learning. This is a field where those who aren't learning rapidly fall behind. So do yourself a favor and invest some time in yourself. It'll be one of the best investments you can make.

HTML and CSS for Front End Developers

Introduction

Hello, I'm Joe Eames. In this module, we're going to be looking at HMTL and CSS, but we are going to do that from a Front End Developer's viewpoint. These core end technologies need to be well understood, and the assumption in this course, is that you already understand the basics of them, but we will dive into some concepts that Front End Developers should know about but can easily be overlooked. We'll start by looking at what it means to write good HTML that will help us follow convention, and keep our HTML readable. Then, we'll look at some of the HTML5 APIs that we should at least be aware of. It would be best to be comfortable with the basics of these APIs, so we'll actually look at some simple demos of each of them. The HTML5 technologies that we will look at are woefully under used. A good Front End Developer should have these APIs in their tool belt, and be able to easily implement them. We'll start with audio and video, which can allow users to access rich content in your apps. Then we'll look at local storage because way too many people are storing data in cookies that shouldn't be there. Local storage is the answer for that. Then, we'll look at the canvas, which allows us to draw in a web application and can be used to produce effects that are unfeasible with other means. After that, we'll look at the offline API to see how to make our applications work when a user is disconnected, which matters a great deal with apps that need to be accessed on mobile devices. And then finally, the history API, which is a core piece to single page applications, and then a Geolocation API, which is really under used. Any time your users need to look up something by zip or location, you should start by defaulting them to their current location, which the Geolocation API lets you do. This will wrap up the HTML5 APIs that we'll look at, then we'll move into CSS. CSS is one of those technologies that has had some serious advancements in the past few years. But too many developers are using it as if it were still 2005. We'll start by looking at where we should place our CSS, which improves performance and maintenance. Next, we'll look at how CSS resets and normalization work, and which of these we should use in different situations. If you're not using normalization, you should have a good reason. After that, we'll look at CSS Preprocessors, and see examples of each of the currently popular ones. CSS Preprocessors turn CSS from a scooter into a bullet bike. If you're not using a Preprocessor, you're missing out on a serious amount of features. Then, we'll look at the fundamentals of responsive design. The time has long passed that we can ignore mobile devices when building application. A site without responsive design is a great way to lose

a lot of users and customers. Next will be CSS grid systems, which are the foundation of most responsive design, followed by a short introduction to Bootstrap, the most popular CSS framework. And finally, we'll look at icon fonts and how they work, and how they can make your life significantly easier.

HTML Structure

You will hear the terms valid and well formed floating around. In this course, we won't go into the specifics of what those concepts mean, although if you're familiar with XML, then you may already have some idea of what they mean. Instead, we're just going to cover what is generally recognized as acceptable HTML. When it comes to HTML, unlike XML, having valid and well formed HTML is not necessarily important, because browsers have a long history of being highly forgiving when it comes to parsing and displaying HTML. In fact, you have to have a seriously broken HTML document before a browser won't display it. Most problems, will, at worst, cause your page to display a little strange. Because of this, the rules about HTML aren't very strict. There are standards, however, and the World Wide Web Consortium has a page that you can use to see how standards compliant your HTML is. But browsers really are quite forgiving. If you want to play around with a W3C validator to check the mark up validity of some of the HTML that you have written, this is the URL. Nonetheless, there are definitely ways to make your HTML easier to deal with as a programmer. Both for reasons that will make a difference as you modify HTML programmatically, and also some style conventions that will make your HTML more, well, conventional. The first guideline is that your HTML elements and attribute names should all be completely lowercase. Like this. Don't do this. Second, attribute values should be enclosed in double quotation marks. Don't enclose them in single quotes. And don't leave off the quotes all together. Lastly, think of other developers when you are formatting your HTML. Your source HTML should be formatted nicely. So use new lines for new blocks. Like this. And use indentation to show how elements nest inside each other. Like this. If you want to dig more into some recommended formatting and style guidelines for HTML, Google has published a great document with a lot more recommendations on style guidelines than I have shown. You can access it at this URL.

HTML5 Video, Audio, and Local Storage

In this section, we're going to look at some of the HTML5 APIs. We're going to start with the audio API. So we'll start with the skeleton for a page. Within this, I'm going to start by adding an

audio element. This is the basis of the audio API. Within the audio element, I can give it a source. Notice how the source element doesn't have a closing tag. This is one of those tags that you don't need to close in the typical manner. Within this tag, I've given it a source attribute and a type attribute. This tells the web page where to get the sound file, and what kind of sound file it is. Now one other thing about the HTML5 audio and video APIs, is that not all browsers support all kinds of audio. So in order to make sure that your audio or video is playable in every browser, you're going to need to include multiple sources. That's easy enough by just adding another source element. Here, I've indicated that this audio element should use either an MP3, or an OGG file for its source. The browser will try the first source, and if it doesn't know how to process it, then it will move on to the next source until it finds one that it supports. Another thing that we can do inside of this audio element is add in some text content, that will display if the browser does not support the audio element. This is due to the default nature of how browsers handle HTML. If they encounter an element that they don't know about but it has text content, the browser will still display the text content. Now this is all well and good, but if we were to render the page at this point, there's nothing that we could do with this audio element. So let's start by just making the sound play as soon as the page loads. I'll do that by creating a script element, and I'll create an audio variable and add an event listener for when the page loads, and I'll have that event call my init function. And inside of my init function, I'll grab ahold of that audio element. Here I'm getting all the elements that have the audio tag name, of which this page only has one. That will return to me an array, so I'll grab the very first element from that array and put it into the audio variable. Now that I've got ahold of the audio element, I can go ahead and tell it to play. Now let's add this MP3 file to our project. Now if we were to render this page in a browser, it would immediately begin playing this MP3 file. Unfortunately, we have no way to stop it, so let's give ourselves the ability to stop it by hand, by adding a click event listener to our page. And we'll have that call a function named playpause. So now let's create that playpause function. And we'll just check and see whether or not the audio is paused. And if it is, we'll tell it to play. Otherwise, we'll tell it to pause. Now WebStorm is telling me that I've got a typo up here in this playpause name. So I'm going to fix that by renaming it, and using CamelCase to separate out those two words. And now that I've got this ready, let's run this in the browser. (clarinet sounds) And we can see if I click, then the audio pauses. And if I click again... (clarinet sounds) Then the audio starts back up. Let's look at one more feature that the audio API provides for us. If I go here to the audio element, and I add the attribute controls, and this attribute does not need any value, just the attribute itself, and now I'll come down here and remove these two event listeners. And now refresh this page. You can see that we've gotten a little control bar that we can use to control the audio. If I click play, it'll play, and I can pause it, I can adjust the volume from right here, or I can

mute from here, and that's a nice little feature of the audio API. Now it's important to note that the controls themselves are dependent based on the browser. So for example, the controls inside of FireFox actually look quite a bit different. So if you need a consistent experience with your controls, it's best not to rely on the built in ones. Now let's take a glance at the video API and how that differs from the audio API. So let's create a new page. And we'll just start off with this same content. And instead of here, we'll change the audio to video. And let's change the source files. We'll make this one. ogg as well, and now we can change this. And we'll change our title. And then let's change our code. And let's put in our load event listener but not our click event listener since we have the controls in place... Now let's go view this in the browser. And that's the little video of my daughter getting thrown off of an inner tube. And just like with the audio API, the controls look different in other browsers. So that's the basics of the video and audio API. Now let's take a look at local storage. Close these other windows down, and we'll just put in a skeleton document again. And within our script tag, we're going to write some code to demonstrate local storage. Now local storage is an API that lets you store data on the browser separate from cookies. There are two types of local storage. The storage space available to local storage varies based on the browser, and then you've got session storage which is only valid for the current session. So let's start out by looking at local storage. And to interact with local storage, we call window. localStorage. And it has two key functions, set item, where we give it a name, and a value, and now that we set that we can access it through a similar function called getItem. Now let's go run that in the browser. If we open up our console, we can see that it's printed out the value one. So we were able to store it and retrieve values using local storage. Now here, I just stored a primitive which is of limited value. If you can store more complex things like objects and arrays of data, session storage works similar to local storage, except instead of calling window. session storage, we just call it global session storage object. But the functions to set and retrieve values are the same. And if we go refresh our page, we'll see that it's accessing both values now. Now this may not seem like much, but because local storage can store complex pieces of data, you can actually do some really useful things with it.

HTML5 Canvas

In this section, we're going to look at some more of the HTML5 APIs. We're going to start with a canvas API. Canvas is a two dimensional drawing surface. We're going to demonstrate the HTML5 canvas by doing a little bit of drawing. This is going to be extremely simplistic stuff, but you can do all kinds of very complicated things. There are even games built on canvas. When we're working with a canvas we always start out with a canvas element. I'm going to give it an ID so I

can access it easily, and I'm going to specify its width and height right on the element. Now for fall back reasons, you can always put something inside of your canvas element, that will only display if the canvas development isn't supported by the browser, such as a text message, or even an image tag. And again, this would only display if the canvas element is not supported by your browser. Now we have our canvas element on the page, let's draw something on it. Let's start with an event listener that will execute some code once the page is loaded. And we'll call a draw function. Within this function, I'm going to start out by giving a handle to this canvas element. Once I have the canvas element, I need its context in order to do any drawing. And here, I need to pass in the string 2D, unfortunately, there's not an option for 3D. There are some other experimental options besides this string, but this is the string you will use for typical use of the canvas. Now that I've got the context, I'm going to do something very simple by just drawing a rectangle, so I'm going to call context. fillRect, and that will create a rectangle, and fill it in with a solid color. The first parameter it expects is X position, so I'll give it 100, the second parameter will be the Y position, and finally, the width, and the height. Now that I've got this, let's go ahead and view it in the browser. So here's the results of drawing to our canvas. You can see that it's drawn a black square. So let's just show a couple more things. First, I want to change the color of that square from black to blue, so I'm going to call context. fillstyle, and here, this takes in a string. This is a color specification, and I can do all kinds of things, such as a red, green, blue setting, or I can just do named colors. So I'll just use the string blue, and now let's go back and see how this looks in the browser. Okay, we can see now, that our rectangle is blue. Once I set the fill style, from that point on, all fill drawing such as the fill rect call, will be in that color. So for example, if we draw another rectangle, and just change the position a little bit and go back to the browser, you can see that it's in blue too. If I want to change the color of that rectangle, I'm going to have to change the fill style again in between these two calls. Now what we've shown will be extremely basic, but as we've said before, you can do extremely complex things with canvas. You can draw lines, you can draw curves, you can draw images onto the canvas, and there are many third party libraries built on top of canvas to allow you to do some pretty amazing things.

HTML5 Offline

Now let's move on to the offline API. The offline API in HTML5, allows you to take your applications offline. This is useful for applications that may be used when the user isn't connected to the internet, such as on a laptop on a mobile device, or on some other semi-connected environment. So let's start by creating page to demonstrate this. We'll start by pasting in a page skeleton. Now the key to making a page work while offline is using the application cache. We do

this by specifying a manifest on the HTML element. And at this point, we specify the name of a file. This is an application cache file, so I'm going to name it app. appcache. An important note, your server must serve up this file as my type text cache manifest. Now we need to create that file. This file is a simple text file that has the following format, we start off by telling it that it's a cache manifest. And then we give it a list of files that the browser will cache for use while offline. So we'll start by specifying the page that we just created, and let's specify a second page that we want to be available while offline, and I'll go back to my offline HTML page, and inside of here, I'm going to create a link that points to that second page. And we'll give this page a little header. Now let's create that second page. Go back and just copy the contents of this page, and paste into here, but we can take the manifest out of the HTML declaration. And let's change our header, and let's change the link to go back to page one, and let's go view this page in the browser. So you can see that our page is rendering. You'll also notice up in our address bar, that I'm hitting some kind of a web server. This is actually a feature of the latest version of WebStorm. It's running a lightweight web server, that will serve up my pages through HTTP. Now if I go back to my WebStorm and close it down, it will take that server offline. So let's close down our WebStorm. And now the server is offline. But, if I were to browse to page two, you can see that I'm still able to get there, and I can go back as well. And if I refresh the page, it's still rendering the page without any problems. Now if I try to go to a page that I know exists but isn't available in offline mode, such as the canvas page we made just a minute ago, you can see it eventually times out because it can't find that file. So those other pages were only available because of the offline API.

HTML5 History

And now let's move on and talk about browser history. The browser history API, allows us to navigate backwards and forwards through the browser history, and also to manipulate that history by adding new items to the history, or replacing existing items within the history. So let's start by creating a new page. And we'll start it with a skeleton page again, and typically with the way the page gets new entries inside of its history, is by browsing to new pages. But another way that we can do that is by navigating to an internal anchor tag on the same page. We can do that by adding an anchor tag that starts off with a pound sign. Now let's view this in the browser. So if I click on this link, we can see that we're now at a different URL, and if I click back and forward, I go back and forth within the same page at two different entries. Now let's see about manipulating that programatically. Let's create some h1s that will help us demonstrate this. I'm going to give this h1 an onclick handler, and I'm going to call a function called go back. And I'll create another h1 that does the same thing except go forward. And now let's add those two functions to our page.

The way that we tell the browser to go back within the history is to call window. history. go, and we pass in a positive or a negative number. Negative numbers go backwards, and the number tells us how far back to go. So negative one will go back one entry. To go forward, we do the same thing with a positive one. And now let's see how this works in our page. Let's click the go backwards link. I'm going to go back to the previous state, and if we click the go forwards header, and then we go back to that new state URL. Now there's another thing I want to demonstrate. We can programatically add another state to our history. So let's create a new h1. And this time, we'll call a function called addHistory. And we'll change the text to just Click Me. We'll create that function, and within here, we call window. history. pushstate. This pushes a new state onto the window history. This function takes in three parameters. The first is an object that is associated with this state. For our purposes, we'll just pass an undefined. Then it takes a name, we're going to pass an empty string, and the third parameter is the URL. So in this case, we'll call our new state, newState2. And notice how I put the string in there. And let's go back to our page. We'll refresh, and I'm going to click that Click Me header, and see how the URL is changed. It has actually given us an entirely new URL. We can still go backwards and forwards, and do all the other things that we can normally do within our history. In fact, we can browse backwards and forwards using the links up here at the top as well. The states we've added are no different to the browser history, than if we'd been clicking anchor tags to navigate from page to page. Now this may not seem very useful right up front, but using this API allows NVC frameworks like Angular and Ember to do client side routing.

HTML5 Geolocation

Now let's look at one more HTML5 API. We're going to look at the Geolocation API. The Geolocation API is relatively simple. It just allows you to get the location of the client's computer. This can be done through many different means. If you're on a mobile device, it might use your GPS or sales signal in order to determine your location. If you're on a hard-wired computer, it'll probably use your IP address in order to get your location. Ultimately, what the Geolocation API will give you is a latitude and longitude. So let's start by creating a new page. And we'll put in some basic HTML, and I'm going to add a script tag, and inside of here, I'm going to call navigator. Geolocation. getCurrentPosition, and this takes in a function, which receives a location parameter. And I'm just going to take that location and print it out to the console. So I'm going to log out the location. coordinates. latitude, and then a comma, and the longitude. Now let's view this in the browser. Now if we open up our console, and look, we can see that it's printed out our latitude and longitude. Now the first time you try to do this on a browser, it's going to give a little

pop up that will ask the user to allow access to the Geolocation API. This of course, is for security reasons, so malicious code can't actually track where you physically are. And now that we've got a latitude and longitude, we can do all kinds of things like using a Google Maps API, in order to pin our position on the map, or using some third party service to find a nearby restaurant. Now we've taken a look at a few HTML5 APIs, but this is certainly not all the APIs that are available within HTML5. There's a lot of other useful APIs, such as the file API, or the socket API that allows for web socket communication. HTML5 is a very big technology that includes a lot of different features. You can find out more about the APIs that we've looked at, by going online and looking at some of the many resources that are available about HTML5.

CSS Placement

There are three places where we can specify CSS. The first is inline on the element itself. We do this by going onto the element, and adding a style attribute. And inside of that style attribute, specifying our style. So let's say we want to put a border around this h1 element. So I give it a simple border specification, and now I specify the style on the page, inline, that will put a black border around this h1 tag. Now this will certainly get the job done, but there's a lot of problems with specifying styles like this. For one thing, if we had a lot of styles that needed to apply to this h1, we would end up with a lot of text here in the h1 tag, and it could easily scroll off the screen. Another problem with it is duplication. Let's say that we wanted this black border around every h1 tag that we've got. So if we put in a second h1 tag, now we've got duplicated style information. This duplication will one, leave to bloat in the size of our page, and two, it's hard to maintain. Let's assume we wanted to change the color of our borders from black to blue. Now we've gotta go change it in both places, so that brings us to the second way to specify styles, and that's to use a style tag on the document itself. Typically, that's put up in the head. Now we can't just start specifying our styles directly here. We've got to tell the page which element a particular style is supposed to apply to. So we start by using a selector. Let's use the basic selector, which is the name of a tag. So I'll type in h1. And that means that all these styles are going to apply to all h1 tags. Then I use a curly brace, and everything inside the curly brace is going to apply to all h1 tags on the page. Now let's say that I want my background color for all of my h1 tags to be gray, and now all h1s will have a background color of gray. There is another benefit to doing it this way, and that is that I can apply this style to multiple elements. Let's say I wanted all my h1s and divs to have a gray background. I can just go in here and add div to the selector. Now all h1s and divs are going to have a gray background. Now there's still a problem with specifying styles this way. Most sites have multiple pages, that means I'm going to have to duplicate the style information across

every page in my site. And that means, if I do any maintenance on the styles for my site, I've got to go into every page and update the style information. So let's talk about the third way to specify styles. And that is, to use a link to a CSS document. We do that with a link tag. We give it a rel attribute of stylesheet, and the href is the URL to the style sheet itself. In this case, I'm going to name my style sheet css. css, and this means it's in the same directory. And now let's create that CSS document. And let's assume that I wanted to make the foreground color of all of my divs and spans reds. Now within a CSS document, I simply specify a style the same way we did inside of the style tag. I give a selector, and then I list the styles that apply to those selectors. Let's say I want to make the text red for all of my divs and spans. So the three ways to specify style information for a page, this is by far, the most common, and best way to do it. Rarely should you use inline styles or the style tag in your HTML page. Instead, a separate CSS document, is almost always the way to go. Now let's talk for a minute, about CSS3. CSS3 is the name of the latest features that have been added to the CSS specification. There are a lot of different pieces to CSS3, but we're only going to talk about a couple of the really important pieces. The first of those is transitions and animations. Transitions and animations really open up what CSS can do. Let's quickly take a look at how our page looks in the browser. Here's our page in the browser. We've got our two h1 tags with gray backgrounds and a blue border, our div with a gray background and red text, and our span with just red text.

CSS Resets and Normalizing

In this section, we're going to talk about resetting and normalizing CSS. When it comes to CSS, most people don't think about the fact that each browser automatically chooses all kinds of styles such as font, margins, padding, and other styles for use so that any unstyled content you put onto a page has some default styles. This seems like a good idea, but as soon as you want to provide a consistent experience across multiple browsers the difference is between the default styles in the different browsers can really mess things up. It can look great in FireFox, but when you go to Internet Explorer the display of the page can be completely broken just because of the differences in margins. That is where CSS resetting and normalizing comes into play. These two techniques are two different angles at getting the same result, which is a consistent experience across browsers. The idea of a CSS reset was pioneered by a guy named Eric Meyer. He created the first widespread CSS reset. Let's compare the two techniques. Normalizing is a process of seeing what's different about the default styles in the different browsers and making adjustments so that they all produce the same look. CSS resetting is based on the premise of removing just about all default styles and giving you a blank slate. Let's look at an example of both of these.

First, you can see that I have added a few files to my project. There's two CSS files I want you to pay particular attention to. The first of these is the HTML5 doctor reset style sheet. You can get this file at this URL. I have also downloaded the extremely popular normalize. css from its GitHub repo which you can find here. Obviously the HTML5 doctor file is a CSS resetter, where as the normalize. css is a normalizer. I have taken a page with a decent selection of HTML elements which you can see here, and I have created three different versions of it. The first one is just using the default styles from the browser. The second one, normalize. html links to the normalize. css style sheet. And the third one, reset. html links to that HTML5 doctor reset style sheet. Let's look at each of these in the browser. Here's our page that just uses the default styles for the browser. Now let's look at our normalized page. You can see that there's been quite a few changes. The font has changed to a Sans Serif font, and the margins have been taken away so that everything is flush against the left side. You can also see that the font is a little bit bigger. Now let's look at our page that is using the reset style sheet. Here you can see that the font size between the h1 tag, the h2 tag, and the other normal text tags have no difference in size. The only real difference, is that the two header tags are bold. You can also see that the margins have been completely removed, so that is the difference between a normalized style sheet, and a reset style sheet. In practical terms, you will usually want to just use a CSS normalizer, and every decent CSS framework out there will do this for you, such as the ever popular Twitter Bootstrap. Or, if you aren't going to use a CSS framework, then just grab normalize. css from its GitHub repo that listed earlier. You may have particular needs for your website, for which a reset makes more sense, but because it wipes out so many styles you'll need to redo a lot of the work that your browsers have already done for you, so in most cases, a normalizer makes a lot more sense than a reset.

CSS Preprocessors

CSS Preprocessors were built out of frustration over limitations with CSS. As designers and developers began doing more and more with CSS, and pushing it to its limits both in functionality, and in size and maintenance it quickly became obvious that there was some significant problems with CSS that could be overcome without needing to change anything in the browser, but instead, changing how CSS was authored. Thus, Preprocessors were born. One of the drawbacks of Preprocessors, is that they add an extra step to your development process. Essentially, a compile step is added to your CSS authoring. You author your CSS in a different syntax, then have to compile that to CSS. This step is typically done at build time, so that requesting a CSS file is just a request for a static asset. This compile step can be automatic or manual, but in any case, it

must be done. There are many options for doing this automatically through file watchers, which can ease any development workflow issues but most projects still end up with some kind of build process to make sure that all the style sheets have been compiled before deploying to a non-development environment. The main features the Preprocessors add to CSS are, Variables, Math, Mixins, Nested Rules, and to a smaller extent, Logic, such as if statements and looping. We'll take a look at examples of the first four items, but since logic varies so much in its support from one Preprocessor to another, we'll leave that to you to discover on your own. Here I've got a simple style sheet that has four rules inside of it. The first two rules are using the same color, which is this orange color that you see here in the margin. The bottom two rules are dividing this color, which is this gray that you see here. Now the style sheet for this size, maintaining these colors is going to be very simple. But let's imagine a big project, where our style sheet is large, hundreds, or even thousands of lines, and went across many different CSS files. In that case, maintaining these colors can actually become quite a problem. What if we wanted to change these colors, or what if, whenever we wanted to use a color, we couldn't remember, exactly, the code, and hunting around to find the code was a real big pain in the butt. In that case, what would be nice would be to be able to define these colors as variables, that way we could name the colors, and also change them whenever we needed. So if something like this was available... Then we could change our styles to look like this. Now, all of a sudden, our style sheet is a lot more maintainable, and it's a lot easier to use these colors in new areas. Now of course, this isn't possible with CSS, but using Preprocessors, this is extremely easy using variables. Now let's look at an example of using math. Let's say I have the following styles to find. Here, I've got a content class defined with a 780 pixel width, and a secondary class defined with a 220 pixel width. Really, what's going on is I have a thousand pixel width design, and I want my content to pick up 780 pixels, and I want my secondary to take up the rest of it. What would be nice is if I could define a secondary in relationship to the content, and just tell it to take up the rest of the space. That way, if I ever change the width of the content, or the total width of my design, then the secondary could change in response. So what I'd like to do is this... And then, down here in the secondary, do a little bit of math. Again, with CSS, this kind of math isn't possible, but with Preprocessors, this is entirely possible. Now let's look at why we might use a mixin. Let's say we have a box class, and we want to define, in our box class, a border radius. To do this in CSS, you actually have to use quite a few vendor prefixes. That's a lot of typing everywhere I want to use a border radius. What we would like to do, is be able to define a mixin, that could package up all these styles together. That might look something like this, and also have it take in some kind of a parameter. Then, inside of our box class, we could simply use this mixin. And we wouldn't have to specify all those styles separately. Again, this is not possible with CSS, but with a Preprocessor, this is extremely

easy. Now let's talk about our fourth feature, nested rules. Let's say we have the following styles. Here I've defined a content class whose color is that gray, and I've said that any span inside of that content class needs to have a three pixel padding, and any h3s inside of that content class, you'd have a top margin of two pixels. Well, these three styles logically belong together, so what we would like to do, is be able to nest the span and the h3 style inside of the content style, with something like this. And now that logically groups those styles together and makes them a lot easier to maintain in relationship with each other. There are three main CSS Preprocessors in use today. The first one, Less, just builds on the existing syntax in CSS. So, for Less, every CSS style sheet can simply have its extension changed to. less, and then any Less enhancements can be immediately used. Less' supportive logic is quite a bit limited compared to the other Preprocessors. Less is built on JavaScript, so its service and component runs on node. The second one is Sass. Sass actually has two different syntaxes. The original Sass syntax is a bit different from CSS. It essentially removes the curly braces and semicolons from CSS, and relies on light space. Because of this, you can't simply take a CSS spreadsheet and change the extension and treat it as Sass. So for existing projects with large CSS style sheets, porting into Sass is problematic. Because of this, an alternative syntax was developed, which has become the de facto syntax for Sass, and that is called SCSS. This syntax is identical to CSS still using curly braces and semicolons, but just adds on some enhancements. Sass also has another product built on top of it called Compass. Compass is essentially SCSS with a whole bunch of pre-built mixins and utility functions, plus some other extras, like having the ability to easily create image sprites. Sass runs on Ruby, so you'll have to have Ruby installed to run it. The final Preprocessor we'll discuss is Stylus. Stylus is a relative newcomer that has some interesting features. It is backwards compatible with CSS syntax, so just like Less and SCSS, you can simply change your CSS file extension, and it is now a Stylus file. But it has a unique feature in that the syntax is a lot more versatile. You can remove curly braces semicolons, and even the colons in CSS, and it will still work. So you can have a syntax that appeals to a lot of people because of a lack of ceremony, but still use existing CSS style sheets. And you can mix the syntax. One rule can look just like CSS, and the next rule can have no punctuation at all. Just like Less, Stylus is built on node. So here is my first style sheet. This is using the Less syntax. You can see I've got a couple of colors defined up here using variables and a couple of widths. I'm using those colors down here. Here's that primary gray and primary orange. The widths, I'm using down here to do some calculations. The content box is set to that width, and the secondary is set to the total width minus the content width. I've also got the border radius mixin. It takes in the radius parameter, and defines all these styles based on that, that is used right here, and finally, you see I've got a nested style. The content box has a couple of styles defined, and then the header class right here has its own styles

defined, but it's only headers that are inside of a content box class. Now let's take a look at the CSS that this produces. Using a great feature, WebStorm, I actually got a file watcher attached, that will turn this Less into CSS for me. Here's the style sheet that it produces. You can see the content box has the background color, all the border radiuses, and the width. The content box header has that color specified, and the secondary has the appropriate width. It's been calculated based on some math that the Preprocessor did. Now let's look at our second example, Sass. As you recall, Sass has an entirely different syntax from CSS. It doesn't use curly braces, and it doesn't use semicolons. The definition of the variables is a little bit different. You use a dollar sign to prefix your variables. In order to define a mixin, you use this app, mixin, and then to use the mixin, you use this @include. Other than that, it's pretty similar to the Less file. Let's take a look at the CSS that this produces. Now this CSS is pretty much identical to the last CSS, other than formatting of where the closing curly brace goes. Now let's look at our SCSS example. Here, again, variables are defined pretty similar to how they were in Sass. Other than we have semicolons at the end of those lines, we got the curly braces back, but other than that, this looks pretty similar to the Sass syntax, and again, if we look at the output CSS, we can see that this is entirely identical to the output CSS from the Sass file. And finally, we look at our Stylus example. Stylus' syntax is by far, the most unique. I've taken advantage of all the enhanced syntax, and removed as much of the ceremony from the CSS as I can. There's no colons, no semicolons, and no curly braces. Defining a variable uses the equal sign. Defining a mixin just uses parentheses, and you can see there's another feature where all arguments can be passed into the style. This is reminiscent of JavaScript. Now if I needed to use multiple arguments, I could actually define them up here... And then use them here. But in this case, just using the arguments will work just fine. Looking down here, you can see that I've actually included a couple of colons. That's because these are completely optional. If I have them or don't have them, it still produces the CSS correctly. Calling that mixin simply involved specifying the border radius, a space, and then the radius. Now let's go look at the CSS that the Stylus file produces. These CSS is completely identical to the CSS the that the Less file produced. You can see, as I tab back and forth between them, they are completely identical. So there's some quick examples, of the different Preprocessors, and how their syntax looks, and what the differences are. If you want more information, Pluralsight has a full course on Less and Sass, and my own course on developing with the mean stack uses Stylus for style sheets, so you can get a taste of that, there. Of course, in the future, there may be more courses on Pluralsight for these technologies.

Responsive Design

In this section, we're going to take a look at responsive design. One of responsibilities for developing for the front end, means your site needs to be available to a target set of users. But what if some of those users will access your site on a device other than a PC? For example, a phone, or tablet. You need to understand the purpose and principles of responsive design, and when to apply its techniques. This section will give you that knowledge. Responsive design entails several different aspects of a site, and includes, but is not limited to the layout of the site. As the device gets smaller, you will usually have to change your layout, since less can be visible on the screen at the same time. The size of the text used on the site. Again, as the screen gets smaller, you will need bigger fonts to make your site readable. The methods of input. On a phone, or tablet, there's no mouse, so things like hovering, and double clicking don't work anymore. Site navigation, as you have less screen real estate, your navigation scheme will have to change. Plus, many sites use a hover based navigation, which won't work on a phone or tablet. And finally, images. You may want to use a different image when the screen is small, or when optimized to a retina display if the user has a retina display. Let's take a quick look at a couple of responsive sites, and see how they adapt to the size of the user's screen. We'll start with the ng-conf 2014 site, designed and built by a couple of friends of mine, Brian Sweeting, and Merit Christianson. So let's start by unmaximizing it, and then I'm going to take the side and slowly begin to drag it in, so that we see what happens as the site gets thinner, and thinner. First, notice, that as the site gets thinner, our text doesn't start to sit on top of the image. The image pops down underneath this text. Let's continue making it small. And at a certain point, the entire site switches, so that the layout is entirely different. You can see that the tickets button now occupies the entire width of the screen. So does this book your hotel room button. If we go back to a wider screen, the tickets is only a small button right here. Now let's scroll down and look at a little bit more of the content. Right here, we have the list of the speakers. Notice as I make it thinner, at a certain point it switches so that the images of each speaker are using a small version, instead of the larger version we were seeing when the site was wider. Let's go back. And you can see, here we're using the big images, and at this size, we use the small images. So there's a couple of things this site does to the size of the user screen. And let's look at another site. We're going to look at the Twitter Bootstrap site. Watch what happens to this text right here as I make the site wider and smaller. See how it's gotten bigger? And if I go smaller again, then the text gets a little bit smaller. Now let's scroll down, and look right here. Here you can see three columns of information. Watch what happens as I make the size of the screen smaller. All of a sudden, it switches so that instead of three columns, each of these pieces of information takes up the full width. If I go back, we go back to three columns, and if we go back thin again, each of those takes up just one column, but that column is the full width of the screen. Also notice how the images are not a fixed size but

instead respond, and take up as much space as they can. Now let's go to our code, and see a demo of how to do some of these techniques. Here, I've created a page called responsive. html. I've also created a style sheet for that page called responsive. css. I've got a little bit of content on this page. I've got one header, then I've got a containing div, and inside of that div, there are two divs. One for the content, and one for the sidebar. This is a fairly typical layout for a site. If we were designing a fixed width site, we might go in and specify that the class content has a width of say, 600 pixels, and class sidebar has a width of 300 pixels. That may work fine for most computer monitors, but on a phone or a tablet, that layout is going to be less effective. Let's start off by making our content and sidebar, a little bit easier to see. I'm going to go to the style sheet, and add in a couple of rules. First, look at the content, a background color, and a height. And then I'll do the same thing to the sidebar. Now, at this point, the content and the sidebar are not going to render side by side like we intend. So the typical way to make that work would be to specify a width in each of those classes, and then also just specify a float. We're going to do that, but we're not going to do that inside of the two rules which we've already created. Instead, we're going to create a new rule, which will only apply only if this screen is 800 pixels or wider. So we'll use a media query for that. Inside here, we'll specify a style for the sidebar, and tell it that its width should be 33 percent, and it should float to the left. And we'll specify that the content has a reciprocal layout. With these styles, if the screen is 800 pixels or wider, we'll see two columns. But if it's less, then we'll only see one. Now, let's also adjust the font size of our h1. Let's go back up here above our media query, and we'll specify a default size of 22 pixels. This will apply when the screen is small. Let's add a new media query for 600 pixels or greater. And inside of here, let's set the font size for the h1 to 28 pixels. Now it's important to know that if the screen size is 600 pixels or greater, then both rules will appear inside the style sheet. We'll get a directive that says, make the font size 22 pixels for an h1 tag, and then, later on, another one that says make the font size 28 pixels for an h1 tag. Because this rule comes after this rule, it will override it, and the font size will be 28 pixels. That's not a problem for CSS, but if that bothers you, what you can do is wrap this style inside of a media query that's only for screens of less than 600 pixels. Now since we've already got our media query for 800 pixels specified down here, let's go ahead and add one more style, and change the size of the h1 tag, to an even greater size if the screen is over 800 pixels. And let's view our site in the browser. As you can see, we're getting the larger font in the h1. Our content occupies the left two thirds of the screen, and the sidebar occupies the right two thirds of the screen. Now let's start sizing down the screen and see how it changes. Once we get under 800 pixels, then the content in the sidebar no longer sits side by side, but instead stack on top of one another. Let's continue to size it down and watch the h1 tag. And you can see that it sizes down again after you get to 600 pixels. So these are the more fundamental techniques that will

be used in order to create responsive sites. Of course, there's a lot more that you can do in order to make a site responsive and view well on smaller devices, but with these basic techniques you can at least get started. And of course, there's always more information to be had by viewing the courses on responsive design available at Pluralsight. com.

CSS Grid Systems

In the last section, we looked at the basics for responsive design. One of the things we did, was create a design that had two columns, one for the content, and one for the sidebar. But we made sure that our design was responsive so that, at a certain width, the columns went away, and we were left with a single column layout. Well, this is such a common thing, that there is an entire class of CSS frameworks that helps you to solve this problem. They are called Grid Systems, and there are a lot of them. If you simply Google CSS Grid System, you will see what I mean. Grid Systems are a response to solving a known problem, and that is managing column based layouts and supporting responsive design. They are also far more flexible than what we did in our last demo. Instead of just supporting a two column layout, they let you divide your site or a portion of your site into a number of virtual columns, and then create elements that span a certain number of those columns. Typically, they support 12 virtual columns. 12 is extremely flexible, because you can go half and half or into thirds, or quarters, or sixths, or even 12ths. So in our demo, the main content would occupy the first eight columns, and the sidebar would occupy the last four, which would give us a layout with two thirds, and one third column size respectively. But, any variation would work. You can tell an element how many of the virtual columns it occupies by assigning a class to the element. Let's look at an example. I have arbitrarily chosen one of the Grid Systems to show you in this demo. It is called responsive. gs, but I don't particularly prefer this one over any of the others. One of the features it has, is that it supports 12, 16, or 24 columns, in case your desired layout doesn't quite work with 12. You can see here the HTML for an example page. We've got a containing div, and inside of that div, we've got a header element and a main element. It's the content of this main element that I want to look at in detail. There are three articles inside of this main element. You can see that on each of them they have two classes. The COL class, and a span underscore four class. This is how I specify to the Grid System that these should be columns, and they each should span four of the 12 columns. So in other words, each of these article elements should take up exactly one third of the available space. Let's view this in the browser, and see what it looks like. Here's our page in a browser. You can see that header section up at the top, the two sections for content over here, and then finally, the sidebar. Let's go ahead and make our page smaller, and see how the design switches to a single column layout. We've now reached

the cut-off point, and it switched to a single column layout. So at this point we have the header, our first content, our second content, and then the sidebar section. So going back to our HTML, you can see how using a grid system greatly simplifies the amount of work that we have to do in order to make our layout not only versatile, but also responsive.

Bootstrap

Bootstrap, according to its site, is the most popular front-end framework for developing responsive, mobile first projects on the web. It was developed by a couple front-end developers at Twitter. It is often called Twitter Bootstrap, although just Bootstrap is the official and most common name for it. So what exactly is Bootstrap? Well, what Bootstrap does is provide for you a set of styles that will give you a really good starting place for a nice looking, responsive site. With a little bit of added color, you can create a really sharp looking site, although many people just opt to use the built in colors, which admittedly are a little bland. But for developers who aren't very artistic, and don't have access to a designer, the result is greatly enhanced over what they would produce themselves and Bootstrap is super easy to use, and has a great documentation. Due to the ubiquity of Bootstrap, it is important to understand and know the basics of it. Even if your main work project doesn't use it, knowing Bootstrap will allow you to quickly put together a decent looking site for either a quick work project or a side project, or a personal project. Let's build a really simple page using a selection of Bootstrap styles, to show how easy it is to put something basic together with it. The first thing I did was download Bootstrap using the Download Bootstrap button right here on the GetBootstrap. com page. From that download, I extracted the Bootstrap. min. css file, then put it into my project, then I created this bootstrap. html file, and set a link to that style sheet. The very first thing I'm going to do is add a class to my body tag. I've added the container class, which will center the content on the page. Now the first thing I want to do is add a navbar to my site. I'll start that process by adding the navbar class to a div. Within that, I'll create a navbar header, and instead of this navbar header, I'll create a link that will go to the homepage of my site. And now I'll create the rest of my header by adding the other menu items. I'll do that by creating a ul with a couple of special nav classes. Within this ul, I'll add three list items. Each of these will be wrapped inside of a link. Now let's view our page in the browser and see how it looks so far. Here's the site we've created so far, and you can already see that the brand and the menus are spaced appropriately, have a nice font to them, and already have a default color. Now the next thing we want on our site is a breadcrumb, just like the menus, it's very easy to do with Bootstrap. I'll add an ordered list with the class of breadcrumb, and inside of this, I'll create two list items. One inside of an anchor tag, that will go back to the home

page, and then the other list item will have the class of active. And that will be where we currently are. Now let's see what this renders. And there's the breadcrumb that Bootstrap produced for us. Next I want to create a large area on the page that will grab the eye. This is called a jumbotron. We create that by adding the jumbotron class to a div, and inside of that, I'll create an h1, with some plain text inside of it. Let's see how that looks. There's our jumbotron. You can see that the h1 inside of the jumbotron is extremely large, and it really draws the eye to it. Now I want to create some content on the site, so I'll add a new div with the class of content. Inside of this, I'll create another div with the class of row. I could have multiple rows on my site, but for this site, we'll just do one, and inside of this row, I want to create two columns. I want to use the same layout that we did in the responsive design section, with one column taking up two thirds of the available space, and a sidebar column that will take up the remaining third. Bootstrap has a grid system in it just like what we looked at in the last section. In order to create columns with Bootstrap, we use classes that are in the format of col, a dash, and then a side specifier. I'm going to use the small columns, which are sm, there's also extra small, medium, and large. The difference between these is the width of which the layout turns from multiple column, to single column. And then, finally, the number of virtual columns this column should take up. Bootstrap uses a 12 column system, so in order to take up two thirds of the content, I'll use eight. Within this column, I'm going to create a section of text. I want to put that in an area that's visibly contained, so I'm going to create another div that uses the well class. And inside of that, I'll create a paragraph, and I'm just going to paste in the lorem ipsum text, so that we have some sample text to look at. Now let's see how our site is looking. Okay, you can see our text is now inside of a section that only takes up two thirds of the width of the available content. Let's go back to our code, and add in something else inside of this first column. Let's say I want to show some data inside of a table. With Bootstrap, it's very easy to create a nice styled table. After my well class, I'm going to add a table. And I'm going to give it the class of table, and I'm also going to give it the class of table-bordered. This will put borders around the cells and around the table itself. I'm also going to give it one more class, table-striped. This will zebra stripe each of the rows inside the table. Within this, I'll create a table head. And within that, I'll create some th tags for my data. And then I'll create my tbody, and put the data inside of it. Now that I've got some sample data in, let's see how this looks in the browser. So there you can see that creating a nice, formatted table is very easy to do. Next, let's add a sidebar that will allow us to collect some data from our users. I'll go outside of the column that I created, and create a new column that will take up the rest of the space. Again, within that I'll put a well, and within this, I'll create a form. Inside of this form, I'm going to create a form group for the email address. I'll give my input the class of form control, which will style it appropriately. And last, I'll add a submit button. Now let's go see how that form looks. So you can

see with just a few styles we've created a nice looking form to collect email addresses. Now let's go add one more thing. I want to add a footer to my site. So I'll go down here, after my content section, and I'll create a footer tag. And I'll give it a class of text-center, that will center any text inside of this footer tag. Now let's go see how this footer looks in our browser. So there, we put together a very basic, yet decent looking page, with very little work done on our part, in order to get some nice spacing between different sections of our page, and a decent layout, and look and feel. This is just a little tiny bit of the power that's available in Bootstrap. There is so much more to it, so for more information, check out some of the great courses on Bootstrap that Pluralsight offers. I especially recommend the course on Bootstrap by Shawn Wildermuth.

Icon Fonts

Icon fonts are a way to put icons and pictures on your site, by taking advantage of the fact that fonts are vector graphics, meaning that they can be manipulated in ways that typical images can't. Therefore, using a font for images has a lot of benefits. For example, you can scale them up to a large size without increasing the size of the data that is sent over the wire. And you can do that on the fly. You can also change the color with just some CSS, and again, you can do that on the fly, and you can apply text effects like drop shadows, but there are some limitations. For example, they can only be one color. Therefore, they are particularly suited for use with icons, but not so much for other types of images you may use on a site. There are a lot of different icon fonts available. Some of them are free, and some cost money. Bootstrap even ships with an icon font. In this demo, I'm going to use an icon font called Font Awesome. I chose this one because it is quite popular, and once you get the hang of it, other icon fonts will be straightforward to use as well. We're going to start with the same page that we built in the last section, but we're going to enhance it with some icons. The first thing I've done is gone to the Font Awesome website, and downloaded their distribution package. I extracted that package into a directory called Font Awesome and put it inside of my project. Then, I created this new icon font HTML, which is exactly identical to the Bootstrap HTML file that we created in the last section, except for one change. I added this style sheet reference, to the Font Awesome style sheet. And that was all I needed to do to enable Font Awesome in this page. Let's look at our page in the browser and see what changes we might want to make. Let's start by looking at our menus. Let's assume that these are actually drop down menus, even though they're not. I may want to convey that information to the user with a little down arrow icon, next to each menu. So let's go back to our HTML, and I'll go down into the menu, and for this first menu item, inside of the anchor tag, I'm going to add a new icon. I'm going to do that by adding an i tag. Now the i tag was originally

meant for italics, but it's been hijacked by a lot of icon fonts, because it's very short, and it's easy to look at it and think of i as icon, instead of italic. So I'm going to add the i tag, with the class of fa for Font Awesome, and then another class which tells me which icon I want to use. In this case I'm going to use the caret down icon, which is fa-caret-down. This is the same format for all of the icons in Font Awesome. And now that I've expanded that out into actual HTML, let's go back to our page, and see how it renders. And now you can see right here next the menu run, the little down caret. And that's exactly how easy it is to add icons to a page. Let's add a few more icons and see some other features that Font Awesome offers for us. For my next menu, I'm just going to add a different kind of down arrow. And for my third menu, I'll add yet another one. Okay, let's view that in the browser. And you can see we've got several different options for a down arrow. Now let's go back, and let's add a home icon to our breadcrumb for the home link. This icon, I'm going to add before the text. And back to the browser, and you can see that home icon. Also notice that the color of the icons matches the color of the text that it's with. Now let's go back, I'd like to add an icon to the jumbotron. So down inside of the h1, I'm going to add an icon. I'm going to use the HTML5 icon. And now we can see that that icon's appearing, but there's a couple of things that I don't like about it. I'd like it to be bigger, and I also want it colored. So let's go back to our HTML, and I'm going to add on the class fa-2x, which will make it twice as big. I'm also going to use a Bootstrap style called text primary, which will color the font the same as the primary text color. Now you can see that our font is bigger and it's colored the light blue that is the primary color for the color scheme that comes default with Bootstrap. Now let's go back to our code, and let's go down to our table. Let's manipulate the table a little bit and add another column. We'll add a registered column, but rather than using the words yes or no, let's display an icon that will indicate whether or not that person is registered. We'll make the first user registered, the second one unregistered, and the third one registered. I'll use the check square icon to indicate that they're registered, and a square icon to indicate that they're not. And now let's go see those changes in the browser. Okay, you can see that the first user has a check, the second user has an empty box, and the third user has a check again. Clearly indicating, to whoever is viewing this page, that this person is, indeed registered. This one's not, and this one is. Now let's do one more thing. Something that's very common is to have icons on your buttons. So let's add an icon to our submit button that indicates that it's going to save the data. So going back into here, let's go down to that button. And I want to change the text of the button to be save, and right before the text, I'm going to add my icon, and I'm just going to use the save icon. And we can see in the browser, that we now have a little save icon next to our save text inside that button. Now let's show another feature of Font Awesome. Let's assume that I don't like that save icon button, I want to use a different button, the double arrow, but, I want it pointing up. If I

looked at the fonts for Font Awesome, I would find that there's two double arrow icons, one pointing to the left, and one pointing to the right. Neither of them points up. But I can manipulate one of those for my purposes. So let's go back to the code, and instead of fa-save, I'm going to use the fa-backward icon. But I'm going to rotate it 90 degrees. And now if we go and look at that in the browser, you'll see that there's a double up arrow. If I take off the rotate 90, and refresh, you can see it points to the left. But if I put the rotate back on, it now points up the direction that we want. There's corresponding rotates for 180 and 270. Now let's assume that I do want the up arrows rotated, but instead of rotating to point directly up, I actually want them to point up and right at a 45 degree angle. Well, there's not a built in style for that, but because this is just text, I can manipulate it using CSS. So I'm going to add a class called rotate 135, and then go up into the head of my document, then I'm going to add a style section. Again, you would typically do this inside of your CSS style sheet, I'm just going to do it inside the page for convenience. And I'm going to add a class called rotate 135. And I'm going to add the webkit transform, and tell it to rotate 135 degrees. Now let's go back to the browser, and refresh, and you can see that the icon is now rotated exactly the direction that I want. Now I'm going to show you one more thing we can do. Rather than having those two upward arrows, I'd like to go back to the disk icon. But I'd like the disk icon to be on a dark background. Well, the way that we can achieve that, is by stacking two icons on top of each other. So let's go back down to our submit button, and I'm going to add a span tag to wrap around the icon. Now on the span itself, I'm going to add the class fa-stack. That tells Font Awesome that I'm going to stack two icons inside of the span. I'm going to add a new icon. And this one's going to be that background color, so I'm going to use the box that we used for the unchecked user, but I need this new icon to be a little bit bigger. Since it's stacked, I'm going to add the fa-stack-2x class. And now this icon I'm going to change back to the fa-save, and then I'll add the fa-stack-1x, because I want this to be the normal size. But remember that save icon had a dark foreground and a light background. Now that we've got this dark background using a square, this disk is not going to show up at all. So I'm going to add the inverse class. And now let's go view that result in the browser. And there you can see that the disk icon is now a white color on top of the dark background. And now you've seen some of the convenient and useful things you can do with icon fonts.

Summary

In this module, we looked at quite a few HTML and CSS technologies that front-end developers should be familiar with. We looked at some guidelines on HTML and several HTML5 APIs. After that, we dug into some CSS concepts, such as where to put our CSS, how resets and

normalization work, and how Preprocessors work. Then, we took a look at responsive design, and the role that Grid Systems play in that, and lastly, we looked at the extremely popular Bootstrap, and also, icon fonts, which are commonly put together to quickly turn out decent looking websites, without an over-abundance of design time, even though, with more designer work, we can produce even better looking applications. These technologies plus JavaScript, are the core foundations of front-end web development. So be sure to keep yourself up to date with developments in HTML and CSS, to keep your front-end development as sufficient as possible. For more in depth look at these technologies, Pluralsight offers many courses on HTML HTML5, CSS, and CSS3.

JavaScript

Introduction

Hello, I'm Joe Eames. In this module we're going to look at Java Script as a language. We're going to start out this module with this introduction to JavaScript. Then we're gonna look at objects in JavaScript and a couple of key points about them. After that, we'll talk about the asynchronous nature of JavaScript. Which is a very important thing to understand, especially when coming to JavaScript from another language. After that we'll talk about hoisting, one of the unique features of JavaScript and the places where not understanding it can get you into trouble. We'll also talk about scope and closures. One of the things that most developers have the most difficult time with. And after that we'll look at context and this. Which JavaScript handles differently than most other languages. So understanding this can be very important and keep you from making a lot of mistakes. If you want to be a confident front-end developer you need to understand JavaScript. You can't simply treat it like a light version of C Sharp, or Java, or C++. There are some really important points about JavaScript that you need to understand, and we will talk about them in this module. JavaScript was built in 10 days by Brandon Eich at Netscape. It's important to understand, because JavaScript has some pretty big holes. One of the things that Brandon Eich really got right, was making JavaScript extremely versatile. In fact, one of the best things about JavaScript is that it's so versatile that it can plug it's own holes. Because of that, I can't strongly enough recommend that you watch the Pluralsight course, "JavaScript The Good Parts" by Douglas Crockford. This course will go into depth about all the great things about JavaScript, and all the less effective parts of JavaScript that you should avoid. When Brandon Eich designed

JavaScript, he designed it to script a few little things here and there. Writing entire applications in the browser with it, means adding on a lot of adaptations. But the name JavaScript, is also not doing it any favors. It is truly a full fledged language and has far outgrown it's original design. Thinking of it as lightweight scripting language doesn't do you any favors. Strangely enough JavaScript has less issues when used for large applications on the server than it does on a browser. The reason for that, is that a lot of the difficulties and problems with JavaScript have a lot more to do with running code in the browser than with JavaScript itself. The DOM is a difficult API to work with. And shipping code over the wire is a difficult proposition regardless of the language that you're using. Starting with a little code, and then growing and growing that code as you execute and continually ship more code over the wire, causes all kinds of issues. But one of the best things about JavaScript is that so many people are using it, that it is receiving constant attention and innovation. And is growing into an amazing language. So in this module we'll look at a few key points of JavaScript, that a front-end developer should really understand.

Objects

There are two things about JavaScript objects that you need to understand. The first is that everything in JavaScript besides primitives, is an object. That means that both arrays and functions are objects. They're just special kinds of objects. Let's look at a couple of quick examples to see what I mean. I'm gonna use CodePen to show this demo. Then I'm gonna switch it so that the code windows are on the top. Then I'm gonna turn off HTML and CSS. And I'm gonna open up the console. And I'm gonna keep the result window closed. Because we're just going to be seeing output in the console. Size this down a little bit. And give us some more room here. And then I'm gonna do one more thing. I'm gonna go in my JavaScript options, I'm gonna turn off auto run. And that gives me a little run button up here so now the JavaScript will only run when I click this button. So let's start by creating a simple object. We know that after we create an object we can add properties to it. And then if we view that object through the console, we'll see representation of that object and the properties it contains. Well functions are objects as well. So if I create a function. Now if I just log out that function to the console. I now it's a little mind bending to log out a function that internally calls console dot log. But just go with me here. All it does is print out the text of the function. But if we use the console dot D-I-R. Now we can see representation of this function where we can explore and drill down into it. And we can see it actually has properties. Arguments, color, length, name. This length is the counter parameters that it has. So that means that we could add properties to a function. And those properties will stick with that function. So if you open up, you see that we now got a new property called "my

prop. " Now the same things goes for arrays. Let's clear this out. And I'll create a new array. And we'll set a property on that array and log it out. And if we open up in Explorer we'll see that it has this my prop property. Now this property is distinct from an element of the array. If I go in and give this array a single element which is the value three. Run this again. We can see that this array only has one item in it. The length property is just one. The item at index zero is a three, but the my prop property also has a three. Now the other thing you should understand about objects in JavaScript, is the two different ways you can access properties on an object. Let's go back and create just a simple object. And let's give this object a property called name. And now if I want to log out that name to the console, I would just type in console dot log, O-B-J dot name. And if I run that it will log out Joe. But there's another way to access properties. And that's using the bracket syntax. If I come over here and type in a bracket and then I put in this property name in a string, and run this, I'll get the same result. Because this with a dot syntax is entirely equivalent to this with a bracket syntax. So what that means is, I can add or access properties using a loop. And now when I view this object you'll see it's got five properties named prop zero through prop four. I can also write a function that will read or write from any property on an object. And now if I go up here and call console dot log, read prop, my object. And the property in string form, which would be the string name. And run this. You can see it'll write out the name property. So if I give this another property. I can call it with that other property name. And access that property as well. So there's a lot that you can do with metaprogramming about an object by reading and writing properties using the bracket syntax. One of the nice things about the bracket syntax is that you can use it to create properties whose names wouldn't be legal using the dot syntax. So for example if I wanted to add a property to my object named "Full name. " I can create that property even with the space. And you can see there's a property now, named full space name. Another place where this can be important is if you see an object with a bracket syntax. You might initially think that it's an array. But in reality the code is just accessing the property using the bracket syntax. So if you assume that that object is an array at that point, the object won't behave correctly because it's not an array. So learning not to be fooled whenever you see the bracket notation for property access is another good reason to understand this principle.

The Asynchronous Nature of JavaScript

We're going to be learning about the asynchronous nature of JavaScript. Which really means that we are talking about the event loop. It's important to understand how JavaScript's event loop works, since that's what enables JavaScript to be asynchronous. Now before we look at how JavaScript's event loop works, let's take a look at how asynchronous language works since it's

fairly straight forward to conceptualize and it will be familiar to those who work with most of their languages. First, as you can see we have some event handler code. Say from when a user clicks a button. And then inside that code we call an IO operation. Let's say we make an AJAX call over the internet. And then with asynchronous language we sit and wait for that operation to complete. That means that no other code can run, no matter how long the operation takes. And any other code that needs to run will also have to wait for that IO operation to complete. And the result processing of that IO operation to complete, before the other code can run. Now let's compare with JavaScript. Here our event loop has the same click event handler. And then in that click event handler, we call the IO operation and we register a call back. That new call back is then set aside waiting for another event to occur to trigger it. In this case, the event would be when the AJAX call finishes. Then our original click event handler finishes executing and is removed from the event loops que. At this point the event loop is empty and any other code, say the user clicking another button, can be executed because no other code is actually executing right now. At some future point, the AJAX call finishes and that causes the callback to be put on the event loop. It can't execute yet, because we're still executing the other code. So our callback will wait patiently for it's turn. Then when that other code finishes, our callback can move to the front of the que and will get executed. So of the two methods, the synchronous method is simpler to grasp conceptually and easier to understand the code when we look at it, because we don't see call backs. Instead we just see function calls. But we are unable to run any code while IO operation is executing unless we do something like writing multi-threaded code. So if we don't do that, then our UI is probably frozen. So that forces us to go to multi-threaded code to make our programs more responsive and efficient. And multi-threaded code is significantly more complex than asynchronous code. This is the essence of JavaScript asynchronous nature. This means that we're never waiting for any IO operations. So code can keep executing, which means our user interface's remain responsive even during long IO operations, another code can run. This is one of the reasons that Nova was developed. Because this method is just as advantageous on the server as it is on the client. I've created here a pen that's got a fake AJAX function in it that will simulate an AJAX call. Inside of that fake AJAX function I just set a time out and tell it to wait for one second, and then call whatever callback was passed in and pass in to that callback the string AJAX data. To stimulate the AJAX call passing the data that returns back into the callback function. And you can see that that callback function is a parameter of the fake AJAX function. So now let's go ahead and call our fake AJAX function. We'll do that by first logging out to the console that we're starting. Then we'll call our AJAX function. And here we need to pass in our callback function. So I'll define a new function, that takes in a parameter which is the AJAX data. And inside of this function, I'm just going to log out to the console that we've returned from the

AJAX call. And also log out the data that we got. I'll close up that call to my AJAX function. And then I'll also log out that I'm finished. Now let's run this code and see what happens. All right, so you can see that the start and the end appeared immediately and then it took another second before the AJAX returned AJAX data line was logged out to the console. This doesn't correspond at all to how the code looks in the code window. According to the code window we see out start then AJAX returned and then end. In asynchronous language, that would be the order that we see the log statements written out to the console. But since JavaScript is asynchronous, this function became a callback that was set aside and then later on placed on the event loop. Once this one second time out occurred. So it's very important to get used to seeing in line functions, meaning that something's going to occur later on. And not necessarily occur right now. Now if our callback function needs to be a little bit more complex than this, then we can write a separate function to handle the callback and pass that function into our fake AJAX call. So let's do that. We'll write another function called process data. That takes in the same data parameter. And we'll log out to the console. Now we'll go up to our fake AJAX call. And instead of putting in an inline function we'll just pass in the process data function. Now it's important to note that we don't put on the parenthesis to actually invoke this function. We're just passing the function in. This function we'll again invoke later on, right here when the callback is actually invoked. This asynchronous nature is core to JavaScript. And learning to see these callbacks as code that we'll execute at some later point is key to being able to work effectively with JavaScript.

Hoisting

Hoisting is one of those features of JavaScript that is fairly straightforward but can cause you consternation if you handle it incorrectly. Let's look at a quick demo. If I try to access a variable that doesn't exist. Like say logging it out. When I run it I'm going to get an error. So you can see that it's erroring out because A is not defined. Now let's go to the line after and declare A by just typing in var A. Now if I run it again, you'll see that the output is undefined. I'm no longer getting an error. So now JavaScript thinks that line one, that A has been defined. Even though I'm not defining it til line two. The reason is that with JavaScript, any variable declarations are hoisted up to the top as if they look like this. Instead of what the code actually says. Now if I go back, and instead of just declaring A, I actually assign a value. If I run this code, I'm still going to get undefined. The reason being is that JavaScript will essentially treat the code as if it looked like this. It'll hoist up the declaration of A, but it won't move the assignment of A that will continue to happen down here on line three. So at line two, when we log out the value of A we're going to get an undefined. Now that's fairly straightforward but there's one specific scenario, that can cause

you problems. There's two ways to declare a function. The first one is called a function declaration. And that's when you create a function like this. The second is a function expression which looks like this. There's a very distinct difference when creating a function using these two methods. And one of the primary differences is how the function is hoisted. In this case the function isn't hoisted at all. The variable is. So this way acts as if we wrote the code like this. Where we have a separate statement for the declaration of the function two variable, and then another statement for the assignment of the function two variable. Where as when JavaScript hoisted function one it'll actually hoist the entire function and not just the variable name. So you can see that in action. Let's change that back. And move this code here. And let's log out our two functions. We'll clear our console and run. And you can see that function one, because it was hoisted, we have the entire definition of it as being a function hoisted up to the top above the first line of executable code. Where as function two, only the variable declaration was hoisted. So it knows that function two is a variable, but it doesn't know what it contains so it prints it out as containing undefined. Not understanding this can cause you some problems. Let's say we have the following scenario. Here I've got a fake implementation of a factorial function, and I'm calling the factorial function, here in this line of code, even though it's physically in the file before the declaration of the factorial function. It'll operate just fine because the factorial function will get hoisted. So let's give it a fake implementation. And we'll run it. And we'll see that it prints out factorial. Now clear this, and let's change this from a function declaration to a function expression. At this point when I go to run the code, I'll get an error. Because at line one the factorial variable is not a function. It's simply the primitive value of undefined. So always be aware of the fact that whenever you use the function expression, you need to assign it before you actually call that function.

Scope and Closures

Scope is another concept in JavaScript that could be fairly straightforward most of the time, but occasionally get you into trouble. The number one thing to remember about Scope is that JavaScript works off of function Scope. So in the sample function here, the full name variable which I describe right here is Scope to the function. When the function finishes executing, this variable will go out of scope and can be garbage collected. It wouldn't matter if I declared this variable inside of the If statement. It would still get hoisted up to the top of the function right here. As long as you never get confused and think that scoping happens to do with blocks like with most languages. Then you'll be able to handle most scoping issues. Now what if we want to do some work that involves some variables, but we don't wan those variables to pollute the global

scope. Let's say that we have the following setup code. We're calling a function to get a utilities object, then we're getting our settings out of a local storage. And if they come back undefined, presumably because they haven't been set into local storage. Then we'll get default settings. Then we want to take those two objects, wrap em up inside of another object, and put that onto the window object in a variable called app. But what we don't want to do is have these variables, utitilies, and settings, polluting the global scope. We want to put those variables inside their own scope. Since scoping is based on functions one thing we could obviously do is wrap a function around all of this then call it. So that's the way we can make sure these variables stay inside their own scope. But even though this method works, it's a little inelegant. Creating a function and naming it just to call it down here, seems like a lot of extra work. Well there's a way that we can do this that is a little bit more elegant that has become very popular in JavaScript. And that's using an IIFE. Or Immediately Invoked Function Expression. We do that by taking our function declaration and turning it into a function expression. We can turn it into a function expression by wrapping it inside parenthesis. At this point we don't need to name the function so we can remove the name. And then down at the bottom, instead of invoking it like this, we can simply put the invokation parenthesis right here. Immediately after the closing curly brace. And then we close the whole thing instead of a closing semi colon. And now what will happen is JavaScript will create the function here and then immediately invoke it when it sees these two parenthesis right here. At that point window dot app will receive it's value. But these two variables will go out of scope. So basically we've created a temporary private scope. Now there is a feature of JavaScript that could effect scope quite a bit. And that's closures. Let's look at a different example. I've got a fake AJAX function here. All this function does is wait for one second and then call a callback that's given and return the string approved. We'll assume that this fake AJAX function handles requests to determine whether or not a user is approved for some action. Now somewhere else in my code, let's assume I've got some kind of a user. Now I'm might have gotten this user from some kind of an action that the user selected, that made this user the active user. Now that this is the active user, we want to call that AJAX function and find out if they're approved. And then we want to take the result of that call and print it out to the console. So in that code we would call our AJAX function and we would pass in some kind of a callback. And that callback takes one parameter only and that is the status. Then I want to take that status and print it out to the console. But I also want to print out the user's name. So I'll put in user dot name. And a string that indicates their status. Now something very important is going on here. This function here that's becoming a callback is closing over the user variable. You can see that right here. It's accessing the user variable, but it's not passed in as a parameter and it's not created locally. It's actually created in a parent scope. Remember when we were talking about the asynchronous nature of

JavaScript. And we talked about how when a callback is registered it gets saved off to a temporary location, waiting for an event to trigger it to cause it to be put back on to the event loop. Well it's not just the callback that's saved, but it's the entire context for the callback which includes any variables that it closes over. So in this case the user variable is being closed over. Let's make this a little bit more exclusive by wrapping all of this in a function. And then down here we'll call that function. And at this line of code right here, user is no longer defined. User's only defined while that function is executing. It's defined at the very top of the function and then when the function ends, that variable goes out of scope. Except that it can't be garbage collected because this function here is being saved off and it closes over the user variable. Let's adjust our function just a little bit so we can see that it's being closed over. Instead of calling this AJAX function, I'm gonna return out a new function. All I'm going to do in this function is log out the user's name. Now down here, I'll capture that function as the return value. And again the user variable shouldn't be in scope. But if we call that function, and we run our code. We'll see that the user variable is still alive and well when that function is executed. Because it's still knows that the user's name is Joe Eames. So long as this function that's returned never goes out of scope, then the user variable will not go out of scope. Closure's are very useful in a lot of situations. For example, the situation we showed earlier. Where we have to pass some kind of a callback to some other function. But that function will only execute our callback passing in one parameter. But the function that we actually want to execute requires more than one parameter. And that case we can close over the additional pieces of data and pass in a function that accesses those other pieces of data, but only receives the parameters that are given when it's evoked. Learning to see closures and understand closed over variables, is key to being able to work effectively with JavaScript.

Context and This

Understanding context or what the this key word means at any given moment is very important in JavaScript. In JavaScript, functions are not bound to the objects that they belong to. As we saw earlier, functions are merely an object. So in JavaScript, there's really no such thing as a method which is bound to an instance of an object. They're simply functions that can be invoked in the context of an object. Let's look at an example of this. Let's say I have an object called Joe. And that represents me, so I'll give it a first and last name. And I'll give it a method called full name. Now if I log out that full name, then we'll get the expected behavior, in that it will print out Joe Eames. But this function right here is not bound to this particular object. I could take this function and apply it to other objects. And it would behave is if it were a member of that object, and not

this one. So let's see how that might work. I'll create another object called Dan. And you'll notice that this object has no full name method. But if I take the full name function from Joe and attach it to Dan. And then I invoke it. The function will now execute as if it belongs to this object and not to this one. And you can see it's printed out Dan Wahlin. So if I attach that function to an object that didn't have a first name or a last name, we would get an error. But the important part to understand is, when a function is invoked, whenever it's invoked with a dot syntax, then whatever object is before the dot becomes the this object inside of the function. So when this function asks for this dot first name, it's looking inside the Dan object, not inside the Joe object. Even though it was originally attached to the Joe object. Now you can also execute functions without the dot operator. For example if I just set a variable to it. Now at this point I can call the full name function without using the dot operator. Now the question is, what will this be. If I run this, we can see that it printed out undefined and undefined because full name and last name were not defined for whatever this was when this got executed. So watch what happens when I change this code just a little bit. I'll commit this out. And run this again. And now it's John Papa. So when we execute a function without using the dot operator, then the this object for that function will be the same thing as the this object will be on the line before the function is called. Or in other words, whatever this is inside the code that invokes the function. It's a very important point to understand. So let's look at a way that we can use a closure to build the fact that context is not fixed to functions. I'm gonna close my console and open the HTML editor. And then I'm gonna add in jQuery. I pasted in some HTML that might be from part of some kind of a quiz or questionnaire site. I've got three sample answers here that are represented by the question marks. And I've got two buttons up above to mark all the answers yes or all of them no. To implement this with jQuery is relatively straightforward. So I'll start out with a typical jQuery function. This function will only get executed once the document's ready. And inside of that I'm gonna grab all the items with the class of all which includes these two buttons. And I'm gonna register a click event handler for them. And whenever one of these buttons is clicked, I want to grab every item that has an answer class, which would be these three divs here. And set the text of that item to be the same text as the button that was clicked. So I start out by just grabbing the answer items. And I go through each one of them. And notice this function is executed for each item. So inside of this function the this keyword is going to refer to the actual item or the div. So I'm gonna wrap that div in jQuery, and then set it's text property to some value. The problem is, what I need is the text property from this outer all class. So inside of here, this would be the button. But inside of this function this, is the div. So I need to hold a reference to this when I'm inside of this interior function. I can do that by using a closure. And create a new variable called that. And set it equal to this. And now I can set the test to the item to the text of the all button.

Where I just called a text function in order to get the text of the all button. And now I close all this up. Let's go ahead and run it. And if I come down here and click yes, all these turn to yes. And if I click no, they all turn to no. So there's a great example of using a closure to hold onto the correct context, and understanding what the context is here, and understanding what the context is here. So being able to identify context and make effective use of closures is an important skill when working with JavaScript.

Summary

In this module we looked at several key points of JavaScript. We talked about it's history, and pro's, and con's. We talked about objects in JavaScript and how everything is an object, and the different ways to access properties of objects. We also talked about the asynchronous nature of JavaScript and how it differs from synchronous languages. We looked at hoisting and how that effect's the code that we might write. We also looked at Scope and closures, and discussed how thinking in closures is an important part of understanding JavaScript. And lastly we talked about context for functions. And how to use closures to deal with the inherent nature of functions that aren't bound instances of an object. JavaScript is a powerful and complex language and learning to treat it as a real language and spending the time to understand it will greatly benefit you in the long run.

HTTP and Interacting with the Server

Introduction

Hello, I'm Joe Eames. In this module we're going to take a look at HTTP and how we interact with the server. Front-end programming almost never relies solely on the client. Communication with the server, in one form or another is almost always a part of building a front-end solution. Our applications receive their initial data and display from the server, and operate based on that. At some point we will need to send data back to the server, either for persistence or as part of a request for more data or display information. This module will look at some important aspects of that communication. We will cover four areas that every front-end developer should have a reasonable understanding of regardless of the nature of their application and which third party

libraries they use. First is the basics of the HTTP protocol itself. In this section we'll discuss some important concepts to understand about how HTTP works. After that we'll look at the XHR object which allows us to make AJAX calls and is a critical piece of the functionality that we have come to expect from modern web applications. Next we'll look at JSON. JSON is the de facto communication format for front-end development. Other formats can certainly be used, and occasionally have to be used, such as XML. But in general, JSON is the preferred format. We'll look at why that is and cover some commonly misunderstood points about it. And lastly, we'll look at the page request life cycle, to make sure that we understand how a browser processes a page. As this is important in many aspects of maintenance and performance in a web application.

The Basics of HTTP

HTTP requests follow a request response cycle. First the browser creates a request and sends it to the server. The server then processes the request, creates a response to that request, and then sends that back to the browser who then processes the response. These requests and responses are just plain text. Not some binary format. If you look at a raw request, all of it would just be plain text. Requests are made up of two pieces. One of which is optional. First is the request headers, and then the request may have a body depending on the nature of the request. Responses work the same way. Both pieces of information are sent together when a request is made that has both headers and a body. The nature of the request varies by the method used. Each request has one of a small set of methods. This is a legacy from when the web was first developed. Since it was simply a way to store documents. Mainly academic papers. So the different request methods were developed to work with those documents. We have gradually adapted their meanings for ever more varying purposes. HTTP requests methods are commonly called verbs, so you will hear that a lot. The first type is the most basic request which is a GET request. This is a request for the server to send us a document or resource. And anytime you type in a URL in the browser or click on a link, that is the method of the request that you make. The GET request itself does not have a body, but the response to a GET request typically has a body. Next is the PUT method. This method is used to create a new document or resource. It is common to use this verb to replace an entire document with a new version. Although some implementations call for this to only be to create a resource and not modify one. With this method, the request obviously has a body. Since that will be what we are asking the server to create. The response can also have a body depending on what the server does. The next request method is POST. This verb is used when you want to add some kind of addition to a document or resource in some way. For example, to append something to an existing document or to post a message to a form. This verb is

commonly used to modify an entire entity as well. This request will always have a body. Lastly is the DELETE method. This method requests the server to delete a document or resource. The request have no body with this method and neither does the response. With modern implementations, these methods have been mapped to the standard CRUD operations for manipulating any kind of resource. Not just documents, but users, or blocked posts, or financial transactions, or anything else your website may deal with. This concept is called REST, and isn't used everywhere, but is getting more and more popular. Let's go look at a couple basic requests and see some of this information. First we'll start out with the request of Pluralsight dot com. And open up the console, and go to the networking tab. And then I'm gonna refresh the page. And you can see that it made a bunch of different requests. Let's filter this so that we only see the document requests. Which is the going to be the page that we actually requested. And you could see that it was a get request. The return status was 200. Meaning that it was successful. The request type was text HTML. We see the size and the amount of time it took. Let's click on the request, and let's look at the header's. You can see that the first header we see is the request URL which is the URL that we requested. Then we have the method. And then we have the status code. And then we have this section called request headers which has a whole bunch of other information. For example, what kind of encoding we can accept, what kind of language, cash information, cookies, which we'll look at. And then there's also response headers. More cash control, some information about the content, the date, etcetera. Now let's look at the body of the response. If I just click response, we'll actually see just the HTML document. There's nothing special about this. It's just plain text in the HTML document format. Now let's go see a different kind of request. Let's go read codepen, and let's type in some kind of HTML. And maybe a little bit of JavaScript. And then let's open up the console. We'll go to the network tab again. And let's save the pen that we've created. Now you can see right here there's a POST request. Let's click on that and look at it. We'll start with the headers. The request URL was to codepen io slash boomerang. It was a POST. The response status was 200. The request headers have a bunch of information about what it can accept and cash and content. And then we see the data that was actually sent or the body of the request. Again, this is just plain text. And lastly we see the response headers. If I look at the response, we'll see that all we got back was some information that indicates that it was successful. So there's a couple of examples of request headers. Another piece of information that's sent back and forth with a request in addition to the headers in the body are cookies. Cookies are just additional pieces of information that the server asks the browser to store and send back to the server with every request that's made. Let's take a look at the cookies that were involved in this request. Here you can see that there are several cookies that were involved with this request. All these cookies were sent in the request and then the

server sent back this cookie and asked the browser to store this cookie as well. We can see the cookies programmatically by using a little bit of JavaScript. If we type in document dot cookie. It will show us exactly how the cookies are stored. They're just a big long string, with semi colons separating out each piece of the cookie. And each piece of the cookie is a name and a value separated out by an equal sign. So here we have the screen with equal 1680. If we go back to our network tab, we can see the screen width cookie right here. Often times websites will use cookies to store information that's purely for the client side. But there are better ways to store client's that information. Such as local storage which we saw in a previous module. Cookies really are about information that the server needs to see on every request. Such as authentication or configuration information.

XHR

The XMLHTTP object or XHR as it is commonly called, is an object built into each browser that allows JavaScript to programmatically make what is commonly known as AJAX calls. Which are sometimes just called XHR calls. Let's first talk about what an AJAX or XHR call really is. An AJAX call or AJAX request is just an HTTP request made by JavaScript, which doesn't involve navigating whatever page the browser is on. So here you see your browser. And you're looking at a page and everything seems normal. But behind the scenes the JavaScript you write, running in the browser, can be making requests, receiving responses from the server, and processing those responses. The responses to those requests are simply passed back to your JavaScript through a call back. And it's up to you to decide what to do with that data. Where as when you have the browser make a request, such as when clicking on the link to a new page. Then when the response comes back the browser will display the new page returned and update the URL automatically. With AJAX that doesn't happen. These AJAX requests can use any of the verbs we talked about earlier. Now although the XHR object is built into the browser, and that is all you need to make an AJAX request yourself. The interface for it is a bit unweildy. So I highly recommend using some kind of third party library when doing these requests. We'll use jQuery, and show a quick example of making a GET request to Pluralsight dot com. Here I am at Pluralsight dot com. I'm gonna open up the console. And I'm gonna check to see if jQuery is used by the Pluralsight site by just looking for the dollar sign object. If I print it out, I can see that it's some kind of a function. Not many different libraries can actually use the dollar sign object. So a better way to check for it is to actually ask for the jQuery object. Which is there. Now these two different variables actually point to the same object. So I can also check what version of jQuery is loaded by doing dollar sign dot F-N dot jQuery. And we can see that one eight one is the version

of jQuery that's running on Pluralsight dot com at this time. So let's utilize jQuery to make a GET request on the Pluralsight dot com default page. I'll do that by calling dollar sign dot GET. Which will issue an AJAX request using the GET verb. And I'll give it the URL. And then I'll give it a call back that will process the response when it returns. A callback takes in one parameter which is the data returned. And I'll just log out that data to the console. Now you can see that what's acutally been written out to the console is a whole bunch of HTML. That's because issuing a GET request to the base URL on Pluralsight dot com returns the default HTML page. Now I could also use jQuery to issue a POST, or a PUT, or a DELETE. But that's not really gonna produce anything useful. With your own server, you could define an API that will handle POST requests, PUT requests, DELETE requests, and GET requests. And do all kinds of things with those requests. Whatever you do with those requests is really up to you. So that's how you make simple XHR requests using the built in tools in the browser and jQuery.

JSON

JSON is quickly becoming one of the most popular data interchange formats. But it's really good to learn what JSON is and what it isn't. For example, what we're looking at right here is not JSON. This right here is the declaration of a JavaScript object literal. JSON, as I said before is a data interchange format, which means it really only exists as a string. Where this is JavaScript. This is not a string, this is code. There's some other differences between what you're seeing here and JSON. For example, here we're declaring a variable. In JSON we don't declare variables, we simply define data. Another difference is that in JSON the keys must be strings so they have to be inside of quotation marks. With JavaScript you don't need quotations marks around your keys. Another difference between JavaScript and JSON is that in JavaScript you can define properties that are functions. In JSON, functions are not a legal property. One of the reasons that JSON is becoming so popular is because of how succinct and expressive it is compared to the alternatives. Let's look at an example of JSON and XML. Here I have a sample JSON document. This describes a Pluralsight course. Now let's take a look at the corresponding data in XML. As you can see, XML requires a lot more ceremony in order to describe the data. Where JSON is a lot simpler. That make JSON easier to read and it also makes it smaller for transmitting data. Which is obviously a big advantage when transmitting data over the internet. Now many products and technologies that handle data interchange, such as Server-side frameworks and Client-side communication libraries will handle the JSON conversion for you. But if you need to convert some JavaScript objects to and from JSON there's a utility built into the browser that you can use. Let's show how to do this with our CodePen example. Now that I've got this person object I want to convert it

into JSON. So I'm gonna take this JavaScript object and convert it to JSON and log it out to the console. I'll do that by calling console dot log, and then I'll use the JSON utility that's built into the browser, and call it stringify method. This will convert a JavaScript object to JSON. I pass on the JavaScript object. And now let's set this to auto run off. Open up the console. Clear our console, and run it. And you can see that it writes out the object. And it's removed the illegal function property and formatted everything else correctly. And now let's look at how to take a JSON string and convert it back to a JavaScript object. I'll close the browser down a little bit. Move this to give us some more room. And instead of logging out the JSON version, we're gonna capture that into a variable. And then I call the JSON dot parts method to turn a JSON string back into a JavaScript object. And let's log that out to the console. And here you can see that it's logged out an actual object with the name, age, and state. But of course, the give key note function was removed when it went to JSON so it can't restore that property when it turns it back into a JavaScript object. And there's the basics of working with JSON.

Page Request Lifecycles

When you request a webpage in your browser, either by clicking a link to the new page or typing a URL in the browser. The server sends down the HTML to your page in the body of the response. And that is it. That's all that is sent and nothing else. At that point the request is now finished. Now that the browser has received the HTML it begins processing it. And part of that processing is to look four special kinds of directions in the HTML. These are CSS links, Script tags of the source attribute, Image tags, and Font face directives in CSS rules. Whenever the browser encounters one of these tags, a new GET request is created and sent off. And the response includes the asset requested. All this happens at speeds that are ideally very quick to us. The browser knows that it can make multiple requests at a time for these other assets. So it can speed things up by requesting several images at a time. There are limits as to how many simultaneous requests a browser will make before it will pause and wait for some request to come back or time out before making a new request for assets. Gathering these assets is the first part of what we refer to as page loading. The other parts of the page loading process include rendering the HTML, styling it with the CSS rules that have been received. Whether those are in HTML or CSS style sheets. Displaying text in the font specified, displaying images in their correct place as they are received, and executing scripts that have been downloaded. All of these pieces work together to cause the page to render the browser. Let's look at an example of a typical page request. Here is Pluralsight dot com. I've opened up the network tab, and we're going to look and see the all requests that are made in order to render Pluralsight dot com. You can see that the very first

request was made to training. That's the URL that you get redirected to when you type in Pluralsight dot com. Looking at this request, we can see that we sent off the headers, and the response was the HTML that was sent back. Looking at the timeline, you can see that while this request was being made no other requests were made at that same point and time. All these other requests were made after this response was received. That's because the browser didn't know what other requests to make until the HTML had come back for that page. And then it could look through it and find other assets that need to be requested. Looking at these further requests you can see that there are lots of them. Here in the type column we can see what type each of these requests were. We've got JavaScript, CSS, JavaScript again, and image, some CSS, more JavaScript, and another image. All of these requests were made together. Then the browser paused. It had reached it's limit for simultaneous requests and was waiting for some to back. Then it made an additional one here, a one here, and then a bunch more here. These are all images. And then as we continue to scroll down we'll see even more requests. We got more image and text. And then here we have a font request. You can see that the method used for all of these requests is the GET method. No Posts were made. You can also see the status of each of these requests in the status field. 200 means that the request was successful. And you can see the size and the time that each request took. Looking down at the bottom, you can see that a total of 61 requests were made. With a total of 1. 1 megabyte transferred. 2. 24 seconds was the total amount of time it took the page to render. So there's the breakdown of the page request life cycle, for a typical page.

Summary

In this module we looked at several important aspects of the HTTP protocol. We talked about the basics of HTTP, how headers, cookies, and request bodies work to transmit data, how the different HTTP verbs work, and the basic request response cycle. We then talked about the XHR object and how it allows us to communicate with the server behind the scenes. After that we looked at JSON, the default communication format that we use when communicating with a server. We can certainly use other communication formats, but JSON is the easiest and most efficient format when doing front-end development. Lastly, we looked at the page request life cycle and looked at a typical page, and the requests associated with it. HTTP is an important concept to understand as a front-end developer. As it effects all the communication you do with the server. So making sure that you don't have any important holes in your knowledge about HTTP is important. If you want to learn more about this important topic, go watch Scott Allen's course on HTTP.

The Browser

Introduction

Hi, I'm Joe Eames. In this module, we will be looking at the browser which is the runtime environment of our applications. As the the track is to the race car driver, so is the browser to the front end developer. The browser is where our code executes and it is the rendering engine which determines how your HTML and CSS are displayed. Understanding this environment and how it enables or limits your development, maintenance and the features of your application can be critical. In this module, we will start by looking at browser capabilities and have different browsers limit what we can do and discuss this as not as a limitation to be considered but also a constantly moving target. Then we'll look at browser development tools and how to best take advantage of the browsers we are using in our development. Finally, we will look at multiple browser testing and how to leverage different tools to give us increased confidence in the applications we build.

Browser Capabilities

When discussing browser capabilities, we should start out with the discussion of evergreen versus non-evergreen browsers. An evergreen browser is one which simile upgrades itself as new versions are available, and therefore, is guaranteed to always be up to date, and can continually support newer and newer features. Nowadays, all browsers claim to be evergreen but there is definitely one possible exception to that rule that you should be aware of. Internet Explorer, although advertised as evergreen, is still a part of the operating system. That means that when a specific operating system becomes obsolete, and Microsoft discontinues support for that OS, any users who continue to use that OS will be stuck on some version of IE. So be aware that IE may or may not actually be evergreen. Now even if you have two browsers that are completely up to date, which features of HTML, CSS or JavaScript they support can vary based on what the manufacturer has decided to support. There are many different browsers and many different pieces of HTML, CSS3 and JavaScript that are in various levels of support in the different browsers. Because of all this, knowing which features are supported in which browsers can quickly become unmanageable. Thankfully, there are great tools out there to help us with this. We will look at one of these, which is caniuse. com. Let's go look at it. This is the current version of caniuse. com. You will quickly see that it keeps an extensive list of the various features that are currently in some form of adoption by the various browsers. Let's take a look at a couple of examples of feature support. Let's start by looking at support for the audio element. You can see

here that the current version of the different browsers is supported for everything except for Opera Mini. Let's go back and check out a feature that doesn't quite have so much support. The date/time input type, as you can see, is not very well supported. The current versions of IE, Firefox, Safari, Opera Mini and IE Mobile do not support it. In addition, Chrome and Opera only partially support that feature. It's only fully supported in iOS Safari, Android and Blackberry. So you can see that using this site can quickly give you an idea of which browser support a feature that you might be considering relying upon. Now if you do decide to use a specific feature that isn't supported in every browser that may view your application, not all is lost. There are two techniques that we can use to make our application handle these situations. Those two techniques are fallbacks and polyfills. Each of these two techniques handles this situation completely differently. A fallback is a means of dealing with the fact that the browser doesn't support the feature and therefore, providing some kind of secondary implementation. For example, if the browser doesn't support the canvas element, you can detect that, remove the canvas element, and display an image that approximates all the information you are trying to display. Or if the browser doesn't support the video element, you can replace it with a link to a downloadable video file. A polyfill on the other hand, is a way to get the browser to support a feature that it may not support natively by using some extra JavaScript to implement the feature by hand. A very common example of this is the JSON. stringify method which we saw earlier in this course. Older browsers don't support that method so it was common to include a third-party library that had an implementation of it and use that third-party library if the browser didn't support it, but use the native implementation if it did. These techniques are great for increasing the possible audience for your web application, but like all things, a balance must be sought. Worrying about users who are on IE 5 is probably not something you should much time and money on. There's a finite amount of resources you can put into any application, so be sure to consider how much of those resources you should spend on dealing with things like this. There's another technique called progressive enhancement. That technique heavily emphasizes accessibility and involves planning for the lowest common denominator and browser support. For example, browsers that don't even support JavaScript and making your site work with those restrictions and then putting in code that will detect more capabilities and taking advantage of those that are available. This technique hasn't seen extremely wide adoption due to the fact that most sites rely heavily on JavaScript, and instead, the burden is being shifted to the market to enhance the browsers to enable application developers to worry less and less about archaic browsers without support of even the most basic features. Where you will fall on that scale would depend on your application. But in any case, knowing what browsers you will support, what features those browsers support that you may want to use, and how to utilize fallbacks and

polyfills to deal with any unsupported features is an important part of front-end development. The situation you want to avoid is spending a lot of time planning and building a site that won't work correctly on some browsers that you care about, when a simple look at caniuse. com would have saved you a lot of wasted effort. This is one of the important design and architectural responsibilities of front-end developers.

Chrome Developer Tools

In this section, we're going to be talking about the developer tools that come with the browser. When doing front-end development, you can easily spend more time in the browser than you do in your editor of choice. So knowing the tools that are available and how to leverage them is an important skill that every front-end developer should know. We'll start by looking at the developer tools available in Chrome. The Chrome developer tools can be opened by either hitting Control-Shift-I or F12. On a Mac, that key stroke combination would be Command-Option-I. We've already looked at the console tab so we're going to focus on a few of the other tabs. This isn't going to be an exhaustive analysis of all the tabs available, we're just going to look at the core ones that a front-end developer should know. Pluralsight has a course about the developer tools available in Chrome. If you want to look deeper into the developer tools in Chrome, Pluralsight has a course on that very topic. So first, let's talk about inspecting our elements. You can see here that I'm viewing a page that is very similar to Google. This is a simple test project that I created that has a couple of JavaScript files, a CSS file and an HTML that ties it all together. Let's start by looking at how to inspect elements in a browser. In Chrome, we can activate the Element Inspection Tool using Control-Shift-C. Hitting this key stroke combination will allow you to pick an element and click on it, and that will highlight that in the Element box. Let's first go to the Elements tab and see what's available. You can see that it's showing you an entire HTML document as a tree. If I hover over just the h1, then just the h1 is highlighted. Now, of course, with any reasonably large HTML document, it can be difficult to find the element you care about by digging through this tree. So there's another tool that the developer tool provide and that is the Element Inspector, which is this little magnifying glass right here. If I click this, and then select an element in my page, that will automatically get highlighted in my Elements tab. There's a shortcut key stroke combination for this. On Windows, that's Control-Shift-C, and you can see that by using that key stroke combination I was able to select my input box. You'll also notice that once you select an element, there's a panel over here that shows a bunch of information about that element. I'm on the Styles tab which shows me the styles that apply to this element. There's a lot of things I can do with this tab. I can hover over a particular style and turn it on or off. I can add a

new style to that element. And I can delete styles. The Computed tab will show me the computed styles for that element. So this shows me the layout of the element, how much padding and border and margin it has. It also shows me all the computed CSS styles for that element. So for example, on the h1 tag, I've actually got two competing styles, one that says use a black font, and one that says to use a red. If we look at the Styles tab, we can see that. Here I've got the red style, and you can see that it's been marked out showing that it's being over-ridden, and then I've got the black style. If I go to Computed, you'll simply see that the color is black. If I open this up, it'll show me where any conflicting styles are and which one is taking precedence. In addition to being able to edit my CSS, these tools are very useful in case some of my CSS isn't working out exactly like I want, I can use this to play around with the CSS until I get it exactly where I need it to be. I can turn this off and see how it would display if this rule wasn't there. I can adjust rules, I can change from center to right, or left. And I can add new styles until I have the element displaying exactly the way that I want and then implement those new styles in my source files. In addition to editing the CSS, I can also edit my HTML. If I want to see how this page would look with longer text inside the h1, I can go in here and type in more words, and you'll see that it updates up here. I can edit not only the text but the HTML as well. By right-clicking I can say, Edit as HTML and see what this would look like with an h2, instead of an h1. So that's the Elements tab. Another tab that's going to be very important to be familiar with is the Sources tab. This is the tab that lets you work with your JavaScript. One of the first things that you should learn is the keyboard shortcut, Control-O, which allows you to find the source file and open it up. I'm going to look at my button1 JavaScript file. This allows me to see the JavaScript inside of this file and from here I can set break points, I can also add Watch Expressions. And once the code hits a break point, I can actually step through the code, by using the tools over here. This will let me resume, this will let me step over. I've also got into and out. I can also deactivate all my break points or turn on and off pause on exceptions. The last tab I want to take a glance at is the Network tab. We've already looked at this tab once before but it's useful to see it again. There's a few features of this tab that are useful to understand. The first is the ability to filter. If I only want to see my JavaScript files, I can click Scripts. If I only want to see my HTML documents, I can click on Documents. I can also turn on Preserve log. This will prevent the Network tab from clearing out all the entries whenever I refresh the page or browse to a new page. This can be very useful when diagnosing issues related to redirects. So that is just a brief overview of the developer tools available in Chrome. Learning and becoming familiar with these developer tools is a core skill that you would definitely want to obtain.

Firefox Developer Tools

The Firefox developer tools are fairly similar to the developer tools in Chrome. You can open up using the same key strokes. On Windows, Control-Shift-I or F12 will open up the developer tools. Let's go straight to the Inspector tab which is analogous to the Elements tab in Chrome. This tab allows us to see the structure of our HTML document, and if we select an element, it will highlight it up on the page. Firefox also has the same element selection tool. It lets me pick an element that I want to look at. But one thing that is different is it has this interesting little pop-up that lets you do a few things, such as copying the HTML or deleting the node. Firefox also has the same panel for looking at the CSS rules about an element. And you can do the same thing by adding new rules. And you can enable and disable rules using the check boxes next to the style. You can also delete rules. And do pretty much everything you can do in the Chrome tools. You can go to the Computed tab and see the same information about the computed styles. The Box Model tab is what shows you the size of the element, its margin, border and padding. You can also edit elements the same way. Or to edit the tag, you double-click on the tag itself. So you can see that these two browsers essentially have the same features, with a couple of minor differences. The Debugger tab is the same thing as the Sources tab in Chrome. It shows you your script files off to the left and it allows you to search them over here. You can set break points the same way you do in Chrome and once you hit a break point, you could do the same things by stepping through your code. Finally, let's look at the Network tab. You can see a list of the files were requested just like in Chrome. And down here at the bottom of the tab, you can filter the type of file shown. So as you can see, the feature set supported by both of these browsers' developer tools is essentially the same with a few small differences.

IE Developer Tools

The last set of developer tools we'll look at is the IE developer tools. With IE, you have to use the F12 key to show the developer tools. The other shortcut doesn't work. Just like with Chrome and Firefox, you can select elements to highlight them or there's a tool that you can use in order to select a specific element. In IE, the shortcut for that tool is Control-B or you can just click right here. Just like with Chrome and Firefox, you can go and see adjust the styles that are involved in that element. IE's Computed tab shows you the same things as Chrome and Firefox, and the Layout shows you the size, border, padding and margin of the element. Just like with Chrome and Firefox, you can edit the text of an element and you can edit the element itself by right-clicking and selecting Edit as HTML. The Debugger tab over here is what allows you to look at your JavaScript and set break points and add watches. It works essentially the same as the Chrome

and Firefox debugger. You can hit Control-O to open up a file. You can set a break point, and once you hit that break point, you can step through your code. Lastly, let's look at the Network tab. Again, just like Chrome and Firefox, you can see all the files that are requested. One difference is, you actually have to turn it on and off. By default, it's not on, you'll have to turn it on and then refresh the page. You can clear all the entries here. If we refresh, you can see all the same metadata and timing information you can see in the other browsers and by double-clicking on an entry, you can see the details for that entry or you can go back to the summary. So again, IE support for developers is pretty much the same as the other two browsers, again, just with its own unique look and feel. No matter which browser you use, you should learn the developer tools available in that browser as well as possible, and be familiar with how to use the developer tools in the other browsers as well.

Multiple Browser Testing

In this section, we're going to talk about dealing with differences in browsers using automated testing. So obviously, since browsers have all kinds of differences, then testing our code in multiple browsers is a valuable thing. Doing that in an automated fashion is even better since we reduced or eliminate the possibility of human error when manually testing, and we also reduce the cost such that we can test faster and more frequently than we could with manual testing. When we are testing a web application, there are two basic types of tests we can run, visual tests and non visual tests. Let's talk about the differences between these two types of tests. Visual tests are the process of automating the front-end of a web application, and in some way, testing that the visual portion of the application looks correct. This maybe as simple as testing whether or not a specific element is visible or not through programmatic means, or it might be capturing screenshots for a human to review later, or it can also mean capturing a baseline of screenshots and then, after making changes to the code, taking new screenshots and comparing them to old screenshots in an automated fashion, where any differences are kicked out for human review. There are many, many different kinds of visual testing. Whereas non visual testing is all about testing that the code works correctly. This kind of testing will never involve human review like screenshots, and a test will pass if the output of that test matches an expected output, which may mean calling a small piece of code and seeing if the return value is correct, or running the entire application again in an automated fashion and seeing if the text in a given element matches an expected value. Again, like with visual testing, the methods for doing non visual testing are many and varied. Because of this, the tools used to do visual testing can also be used to do non visual testing. Each type of testing has different kinds of problems it will help you avoid. Let's look at the

benefits of each in turn. Visual testing is all about whether or not the application looks correct in a given state. It will help you do the following things. It could help you manually inspect your application by just capturing screenshots for human review, but do it much faster than a human can do it, so that the process of testing for correctness just involves looking at screenshots instead of actually trying to use the application. It can also help you find meaningful visual differences in browsers. If one browser doesn't support a specific feature, it can cause the application to render completely incorrectly. Lastly, it can help you find visual regressions. If you change your CSS rule thinking that it will affect a specific element but it actually affects more than that and causes some unintended consequences or regressions, then this type of testing can help you find these problems. Non visual testing on the other hand isn't about worrying whether the display portion is right, but whether or not the application code works correctly. This kind of testing will help you find the following things. It will help you know if your functionality is correct. For example, if your interest calculations are working correctly in a banking application. It can also help you determine if a given browser doesn't support something that is important to your JavaScript code. It can also help you find regressions but of the code kind. So if you change a method and that causes a break in a far corner of your application, this kind of test can help you find that. There are many different tools out there to help you do these kinds of testing. We won't list all the options because the list is constantly changing. For visual tests, googling the words multiple browser testing will give you more results than you will know what to do with. For non visual testing, there are many tools and we'll take just a quick look at one of the more popular ones which is Karma. Karma was built by the Angular team at Google, but is not tied in anyway to Angular so it can be used no matter what kind of libraries you are using in your application. Karma has a configuration file that lets you specify what browsers you want to run your tests in. Let's look at a sample run of Karma. Once Karma's configured correctly, all I have to do is type in karma start and it will run all I test and all the browsers I specify. In this case, I have specified Chrome and Firefox. The browsers are launched by Karma and they look like this. There's Firefox, and here's Chrome. It runs extremely quickly and will even watch your files and re-run your tests whenever you change your code. For more information on Karma, in my course I'm testing client side JavaScript, I have an entire module on it that you can view to see how to install and configure it. Between visual and non-visual testing, neither kind of test is inherently superior to the other, but what is important is to leverage automated multiple browser testing whenever possible as this will save you a lot of time and money in the long run.

Summary

In this module, we took a closer look at the browser. We talked about the fact that different browsers support different sets of features, from the latest features in HTML, CSS and JavaScript. We also looked at the developer tools available in browsers, and common functionality you should be comfortable with to help you design and debug your web apps. Lastly, we talked about testing our applications using automated multiple browser testing tools. These concepts are important for front-end developers to know as the browser is the runtime environment for applications, so the importance of being competent with them just can't be understated.

JavaScript in the Browser

Introduction

Hi, I'm Joe Eames. In this module we're going to learn about working with JavaScript in the browser. In this module we'll be looking at some specific items that apply to JavaScript as it runs inside the browser. The browser, as an execution environment, brings its own set of challenges for working with JavaScript. We will be looking at how to handle these challenges, and what aspects of working with the browser are particularly important to understand as we build increasingly complex applications with JavaScript. We'll start by looking at how scripts are loaded in the browser, then we'll talk about the global scope and which object represents the global scope in the browser. After that, we'll talk about working with the DOM, and look at the basic DOM operations that we should be familiar with. Lastly, we'll look at modules in JavaScript, or specifically, the lack of them, and how that affects the code we write and how we can work around that, and what is coming down the pike. JavaScript is our main vehicle for building complex apps, and understanding how it is affected by the browser is a core skill that we will need to develop.

Script Loading

In this section we're going to talk about script loading. In the HTTP module, we talked about how the browser processes the HTML page that it receives, looks for script tags, image tags, etcetera, and sends off requests for those assets. We also talked about how the browser will send off multiple simultaneous requests in order to improve performance. The same thing works with scripts, the browser will send off multiple script requests, and those scripts will come back in whatever order they're received. With images, the browser is free to process those images as

they're received. There's no need to wait for a previous image in order to display a later image. But the same is not true of scripts, scripts must be processed in order. I've built a sample page here, based on a previous demo, but I've made a few changes. I've got five script tags in this file, first is the main script, and then the button script, then the jquery script, and then the addins script, and then the utilities script. The first two and last two scripts are very small scripts, whereas the jquery script in the middle is very large. Each of these is less than one kilobyte, the jquery script is almost 100 kilobytes. But no matter what the size is, and no matter how long it takes for any given script to come back, the browser must process these scripts in the order specified in the page, otherwise you could encounter some major problems. For example, inside of our addins. js script file, we can rely on the fact that jquery is already present and loaded, so we can use jquery. But inside of buttons and main, we can't rely on jquery, because it hasn't been loaded beforehand. If the addins script comes back really quick, before the jquery script comes back, and the browser processes it as soon as it receives it, then at that point jquery won't be available and this script would crash. Instead, even if this comes back first, the browser has to hold onto it, run the jquery script when it's received, and then it can run the addins script. Let's look at a sample of loading this page. Now to demonstrate this, inside of each of these scripts I've added in a console. log statement that will print out the name of the script file. I even did that to the jquery file, even though I also left all the jquery code inside of it as well. So let's look at an example of rendering this page and see how the scripts are loaded. Here's the page and you can see down in the Network tab, the order in which the scripts were received. First the main. js script was received, then buttons, then the addins was received, then jquery, and finally utilities. Notice that in the script order, jquery comes in between buttons and addins, right here, but when the page ran addins was actually received before jquery. Now if we look at console to see what order they were processed in, we'll see that it reflects exactly the same thing as the HTML page, main. js, then buttons, then jquery, then addins, and then utilities. The browser held onto addins until it was able to run jquery, and then it ran the addins script. This is extremely important to understand, to make sure that you request scripts in the proper order.

The Global Object

Let's talk about global variables. We learned in a previous module that in JavaScript variables are scoped according to function, so functions create scope. It's important to realize that that means that files do not create scope. In our demo here we have a whole bunch of script files, buttons, main, addins, etcetera, these different files do not create a separate scope. They're just files, so therefore they all execute inside of the global scope. Let's demo that by creating a variable inside

of one file and accessing it inside the other. Inside of my main. js file, I'm going to create a title variable. And then I'm going to go to my buttons script file, which executes after main, and log out that variable. Now let's go to the browser and run that page. So you can see that the second script file was able to access the variable created inside the first. They operate inside the same scope. As I mentioned, they execute inside of the global scope, so this title variable that it created is global. Now one of the things it's very important to understand when using JavaScript inside of a browser, is that the global scope actually belongs to the window object. We can prove that by going into our console, and typing in window. title and you can see that it's the word search, even if I create a variable with the var keyword, if that variable's created inside of the global scope it goes on the window object. So defining a variable inside of the global scope using var is the same thing as just putting it on window. Executing this line of code is exactly the same thing as we executed in this line of code right here. Remember, in a previous module we checked for the presence of jquery by looking to see if the $ variable was defined. That variable exists on the window object. We can see that using the equality operator. Those two are the same exact object, so it's important to note that whenever defining variables inside of a global scope, those variables are actually added to the window object. The window object is the global object.

The DOM

The DOM is an API, or Application Programming Interface. It allows your JavaScript code to read and manipulate the HTML page that you view. A lot of people who do web development simply rely on third party libraries to manipulate the DOM, but those libraries just simply wrap on top of the DOM, and although they provide a large value, many operations can be done without a third party library. There are certainly situations when bringing in a large third party library, just to do some simple reading or manipulation of the DOM, is overkill. So we're going to look at how to do some of the most basic operations using the DOM. I'll go to the console, and I'm going to start by showing you how to select an element. Let's assume that we wanna look at the DOM node that represents our first button, the search button. There are several different ways that you can select elements, the first question I need to answer is how do I want to select that element? Let's assume that all I know is that it's a button, in which case I'm going to use the getElementsByTagName function, on the document object. This will return an array of elements. I don't want all of the elements, I only want the first button, so I can access that by asking for just the first element. And with that I'm able to select just the first button. Now that I look at that button, I can see that it's got an ID and a class. So let's select it by its ID. Let's also try selecting it by its class. You can see when we select it by ID, it just returned the object, but when we select it

by class name and tag name, it returns an array. That is because ID's are unique, so they can only return one object, whereas tag names and class names are not, so it will return an array. If I just want that object from the array, of course I can again, access the first element. Now let's say I want to add an element to my page. Let's say that when a user clicks Search, we want to add a div that has the results in it. To do that we start by creating a new element. This element will be detached, and not part of the DOM, until we append it into the DOM. We create an element using the createElement function, and we simply give it the tag name, and now that I have that tag, I want to add it to the document. In order to do that I first have to select the element that I want to be the parent of this new object. In this case I want to add it to the body, and that's still going to return an array, so I'll grab the first element, and then I can use the appendChild function, and now I just pass in that detached element that I created. At this point, that div has been added to the DOM. We can see that by going to the elements, and we'll see right here is that div. Unfortunately, there's nothing in it, so let's add a little bit of text to it. Let's go back and set it's inner text property, and now we can see that div, and now that div is visually represented. Let's add another element. When we added this results tag, we used the appendChild function which just appends it to the end of the list of children. I want to add another tag, but I want to put it in between two existing tags. Looking at our elements, we can see that we've got this h1 and then this div with a class of center. I wanna add another h1 tag in between these two. So I'll go back to my console, and I'll create the new h1 using createElement, and I'll set that new tag in our text property. So when I add that new tag, I need to add it to the body, so I'm going to grab a reference to the body element, and looking again at the elements panel, I want to add that tag before this div with the class of center. So I'm going to grab a reference to that div. This time I'll use the getElementsByClassName, and now that I've got references to those two elements, I can use the insertBefore function, called on the parent element, where I pass in the new element, and then I pass in the reference element, which this new element will be inserted before. And there you can see it, the original h1 is here, our new h1 is in between it and the div that is containing the text box and the two buttons. Now let's do one more thing, let's add an event listener. I want to add an event listener to my Search button, so I need to grab a reference to that button. I know that it has the ID of search, and then I'll call the addEventListener function. The first parameter is the type of event I wanna listen to, in this case, that's the click event, and then the second parameter is the callback function. Now that we've registered that listener, if I come in and click the Search button, you can see that we get a log message of clicked search button. So there's how to do basic operations with the DOM in raw JavaScript. It's good to learn to do these basic operations, so you're not relying on a large third party library when you don't need it. This becomes important when building your own library for others to consume. In this case, you

should try not to rely on any third party code, so doing any DOM manipulation using raw JavaScript can make it much easier for others to consume your library. That's not to say that third party libraries don't have their place, they do a lot of other things the code that we wrote doesn't do, for example, they deal with differences in browsers, and support older browsers as well, but they're not necessary 100% of the time, and learning when you can get away with not using a third party library, and just utilize the built-in DOM API, will keep you from bloating your applications unnecessarily.

Module Systems

JavaScript was originally built to handle small tasks. Its initial design never considered the possibility of building large applications with tens, or even hundreds, of thousands of lines of code, yet that is the world we find ourselves in today. One of the features that most languages have, that JavaScript does not, is the concept of modules. Other languages natively can package up code into modules that house classes or objects, that can then be used by or depended on by other modules, and ultimately assembled into programs. This sort of building block pattern makes it easy to organize code and allows for simple use of third party code, since that code can be encapsulated and won't pollute your global space. This is not true of JavaScript. JavaScript has no module concept. Putting code into separate script files doesn't change the fact that all JavaScript code lives together. The only way to separate out code is to put it into a function, which moves it into private scope, but there is no facility to identify a set of code and group it all up into some kind of a module. With JavaScript in the browser, it gets even more complicated, because code has to be shipped across the wire. If we had the above situation in JavaScript, and each of these of modules was just a separate file, we have to very careful to make sure that the order of script tags in our HTML page matches the dependency order, so Math has to come first, then Graphics and Geometry, in either order, and then finally Main. If we specify the Main file first, then things will not work. This is no problem when we have four files, but if we have 400, then it's an entirely different matter. What we need for large applications is a real module solution. This solution should meet the following requirements. We should be able to expose only a certain subset of variables within a module. In other languages, we call these the public objects or classes. We'll call these exported items. We also need a way to specify what modules a given module depends on, and since we're in the browser, we'll have to be able to get that code across the wire and load it when it arrives. Finally, after the code's been loaded, we'll need to execute it in the proper order based on how the modules depend on each other. We're going to talk about three solutions, two of them only briefly, and the third, we'll take a look at a demo. The first solution is called the

CommonJS module system. This module system really developed for Node, it hasn't gained much popularity in the browser. The second solution is the ECMAScript 6 Module solution. ES6 Modules is still in the defining phases for the next version of JavaScript, so at this point it's not a viable solution. The third solution is called AMD, or Asynchronous Module Definition. There are several third party libraries that implement AMD, and the most popular of these is called RequireJS. Let's take a brief look at how AMD modules work. I've got a small demo here based on the diagram that we looked at earlier. I've got three modules that I create, the math module, graphics and geometry modules, and then a program in main. js that consumes them. So let's start by looking at the math module and see how we create that. We use the define fuction, and then we pass in another function that returns out what the module has to export to the public world. So in between this line where I create the function, and the line where I return it, I could put in any kind of code that I want. Ultimately, I'm going to return something, and that is the public portion of this module. In this case I'm returning an object that has one function, calculateArea, which calculates the area of a circle. Next let's look at the geometry module. This module depends on math, so the first parameter is no longer a function, but instead an array that has a string in it that says math. This indicates that it depends on the math module. This list of dependencies lets me tell my module system that it needs to load up those modules first before it executes this module. After I name the module, then here I define what that module's going to be imported as. So it's imported as a variable named math, and you can see down here, I used that imported module in order to call its calculateArea function. Let's look at the graphics module as well. That one is essentially the same, it depends on math, we bring it in as a variable named math, and then we can utilize it. And just like with math and geometry, we return out an object that is the public export for this module. Now so far we've been defining modules using the define function, let's go over to the main program and we can see that we're no longer defining a module, we're simply calling the require function which says I'm going to require other modules, but I, myself, am not a module. In this case we depend on graphics and geometry, so the system will see these, load up the graphics and geometry modules, when it loads those up, it's going to see inside them that they depend on the math module, so it will load that one up as well. Once that's all loaded, then it can finally execute this module, passing in graphics and geometry as the parameters graphics and geometry, doing whatever it needs to do, and then at that point it can run all of its own code. Now this may seem like a lot of work, but remember what our index page would have to look like if these four files were simply just script tags. We'd have to put them all in the correct order. Again, easy with four, but difficult with 400. Now using require. js let's look and see what our index file looks like. Here you can see we've added in the script file for require itself, and then we've also got the script file for the main script, and then we've got a little bit of configuration, which you don't need to

pay attention to, and that's all that we've got. No other files need to be noted here in the index file. So that means I don't have to worry about making sure that math is before graphics and geometry, and graphics and geometry are before main. I simply specify any code that uses the modules, and then the module system itself will figure out how to bring in those files. And even though we've looked at how to do this with require. js, using ES6 Modules or CommonJS modules, essentially the same basic principles apply, you create modules and declare their dependencies, and ultimately use them, and then the module system itself handles the loading of those dependencies. This system is obviously superior to doing this by hand, but also incurs overhead in the form of additional complexity inside of each of our modules. At a very small program size like this, it's really not worth it, but as your application grows, this kind of functionality can become indispensable.

Summary

In this module we looked at several items that affect our code when running in the browser. We talked about script loading, and making sure that the browser loads our code correctly, then we talked about the global object, which is the window object, and how variables appear on the window object by default. After that we talked about the DOM API and saw how to do basic operations without leaning on a third party library. Finally, we looked at JavaScript module systems and how they help us compensate for the lack of a built-in module system, which makes a huge difference as applications get bigger and bigger. These different items affect how we work with the browser in our JavaScript code, and learning them will help us increase our confidence in building more and more complex front-end applications.

Basic Libraries & Tools

Introduction

Hi, I'm Joe Eames. In this module we're going to be learning about the basic libraries and tools used by front end developers today. There are many different libraries and tools out there, and obviously we aren't going to cover them all, but instead we'll be looking at a few that are so widely used and so important that everyone who does front end development should at least be aware of these tools, including their features and uses if not actually comfortable with using them. We'll start by looking at Bower, a Node utility for installing client side third party libraries into

your application. Then we'll look at jQuery, everyone's favorite library. After that we'll look at Underscore and Lo-Dash, which are utility libraries that bring things to Javascript that it desperately needs. Next we'll look at Modernizr, a way to detect features in the browser. After that we'll look at JSLint and JSHint, which are code quality checkers. Then we'll talk about Grunt and Gulp, which are JavaScript task runners. And lastly, we'll look at a few of the more popular MVC Frameworks and make a small comparison of them. This module will be all about getting familiar with these tools and libraries and not about mastering them. Most of these topics have one or more courses in the Pluralsight library that you can check out for a more thorough examination.

Bower

If you've worked with. NET before, you may be familiar with NuGet. Or if you've worked with node, then you will be familiar with NPM. These tools are package managers. They allow you to easily add third party libraries to your application. For example, if you want to install a real time socket communication engine, then you can use NuGet to install Signal R for. NET, or NPM to install socket IO for node. Installing these libraries is easy with these tools and if you ever need to reinstall, that's a simple matter too. Bower is a package manager very similar to those two products, but specifically for front end web development. Let's take a look at a simple example. Here I've got an app somewhat similar to a previous demo that we've looked at. Let's assume that I need to add jQuery to this application. Now, I could go out onto the internet, browse to the jQuery site, download the JS file, and then include it in my application. But using Bower, there's a much simpler solution. I'm going to simply go to the command line. I'm going to change directory into my js directory. And I'm going to type in Bower install jQuery. Now I can only do this because I've already got Bower installed. Bower is a node tool, so you already have to have Node installed and you can install Bower by typing in NPM install Bower. And it's usually a good idea to just install it globally. So going back, we use Bower to install jQuery by typing Bower install and then giving the name of the library. And you'll notice that this is very similar to how NPM works. Now once I hit enter, Bower will go out to GitHub, get jQuery, and bring it in and install it into my application. If we go back to the application, you can see that it's added this Bower components directory, and inside of this it's added a jQuery directory. Now, it's brought in a lot more than just the jQuery file, but I'm going to ignore all that and just look for the jQuery file, which will be in this dist directory. And there's the unminified file and the minified file. Now that I've got those files, I can simply set a reference to them in my index file, and then I'm good to go. Now another thing that Bower will do similar to NPM is store a list of the third party dependencies that you have so

that you can reinstall them easily. I go back and type in Bower INIT. I'll need to answer a few questions about my application. I'll accept the defaults on most of these. And it will create an initial Bower file for me. Let's go back to our project. You can see this file appear here. And you can see that it lists my dependencies and it's automatically added jQuery in because it sees that it's already been installed. And if I go in and delete jQuery out of my Bower components. I can go back to the command line and just type in Bower install and it will read that Bower file and install any dependencies that I've got for my project. And now that I've got my Bower file created, if I want to add any more dependencies but I want them to be saved in that Bower file as dependencies, when I install them I simply need to add the dash dash save parameter. Let's say I want to install Bootstrap. I just add the dash dash save, and when I install it will add Bootstrap as a dependency to my application. Now this is just a basic introduction to Bower, and it does have more features, but it shows you how easy it is to manage your third party JavaScript libraries with Bower.

jQuery

jQuery is an entirely unique library in the front end development world for two specific reasons. And those are its widespread use and its huge plugin library. There are many neat features that jQuery has, but none of them are unique to jQuery except for the fact that it's so popular and has an amazing selection of plugins. It has a much higher penetration to the market than any other JavaScript library. The exact numbers can be misleading, but the short of it is that it's one of the libraries that you should be familiar with. And when looking for a plugin to do something, the problem will always be that there are to many choices. In fact, the chances are good that most web developers have more experience with jQuery and know it better than they know JavaScript itself. But that is another problem. As a front end developer, jQuery should be one of the libraries you are familiar with, regardless of whether or not you use it in your professional or personal web applications. Not knowing it can hamper your efforts at obtaining a job since it really can be considered a baseline skill. Of course, Pluralsight has courses on jQuery, so here we'll only talk about the important functions that you should be able to do with jQuery and show a simple example of each. There are five main features that jQuery has and therefore five main tasks that you should be comfortable doing with jQuery. Those five features are DOMNode selection, DOMNode modification, animation, event handling, and XHR calls. Let's discuss each item. Here I've got a simple page based on a previous demo. Let's first talk about how to select nodes using jQuery. Selecting nodes with jQuery is extremely simple. You simply use the dollar sign function, and inside of that function pass in a string that is your selection. You can use a tag name such as

div or you can use a class name or you can select based on ID using hashtag or you can use a combination of these. Let's search for just button. And you will see that what's returned is an array with two items in it. The button with the ID of search and the button with the ID of first result. Now if I only want the first button, since this is an array I can actually access the index that I want. But what is returned from that is not a jQuery object, but instead the DOMNode. That means I can't do jQuery operations on it like hide it. That's going to return an error because the DOMNode doesn't have a hide method. But the jQuery object does. So instead of getting the first button this way, we can use another function called EQ. That gives us the item at that index, and then we can call the hide method. You'll see that when I called that, the button disappeared on the page. Let's bring our button back by calling show. And now let's talk about manipulating the DOM. If we want to add more elements to the DOM, there's a lot of methods that jQuery gives us. Let's say we want to add a subheading to our heading. Looking at the HTML, we'll see that our heading is an h1 and after that is some divs. I want to add an h4 after the h1. So going back to the console. I'm going to grab the h1. And since there's only one, I don't have to worry about grabbing the first one. And then I can call the after function and just pass in a string of HTML. And that adds the h4 to the DOM. In addition to adding Nodes, I can also manipulate the existing Nodes. I can change the CSS of the h1 to have a red color. So now you've seen how to select and modify the DOM using jQuery. If you'll remember back to our section about selecting Nodes and modifying Nodes using raw JavaScript, you can see that doing it with jQuery is quite a bit easier. Another advantage that jQuery has over raw JavaScript is that it works in a wider selection of browsers. Now let's talk about using animations with jQuery. Let's say I want my buttons to fade out. I can simply select them and call fade out and they'll fade out. And if I want them to fade back in, I can call fade in. And if I want them to fade out and then fade in, I can simply chain those calls. jQuery has a lot more animations than just those two, plus there's a lot of plugins that give you even further animations. But you can see just from this example that animating with jQuery is quite simple. Now let's talk about handling events with jQuery. This is another area where jQuery shines. If I want to put a click event handler on my first button, all I have to do is grab that button, and call the click method and pass in an event handler. And now when I click on my search button, you can see that the event is handled and the alert is raised. In addition to the click event, jQuery supports a lot more events. The last thing that jQuery does really well is HXR or Ajax calls. If I want to issue a get request to the server, it's very simple with jQuery. All I have to do is call the dollar sign dot get function, which will issue a get request. If I want to issue a post, I can change this get to post. Or a put, I changed this to put. We'll just do a get request for now. This get request is going to fail because I don't have a server to respond, but this will show you how it works. The first parameter I give it is the URL. And the second parameter is the success callback

function. You can see that we've got a 404 error because this URL doesn't exist. But that's how easy it is to make a get request with jQuery. Making the get request using raw JavaScript requires quite a few more steps. This is just a brief overview of what jQuery can do. There's really a lot more to it, so if you're not very familiar with jQuery, it would definitely benefit you to spend some time and watch some of Pluralsight's courses on jQuery.

Underscore & Lo-Dash

Underscore and its competitor, Lo-Dash, are utility libraries that bring a lot of functionality to JavaScript that you might think should have been built into the language to begin with. Underscore was built first and then the creator of Lo-Dash created Lo-Dash as a drop in replacement for Underscore, but with better performance and some additional functionality. If you want additional information on the difference between these two libraries, I recommend you check out the Lo-Dash site, which is just lo-dash. com. For our purposes, I will treat these two libraries the same and simply refer to them both as Underscore. Underscore is an extremely useful library. It has utility functions that fall into the following five categories. Utility functions for collections. This includes arrays and objects, where it treats objects as a collection of key value pairs. Array specific utility functions. Utility functions that operate on functions. Utility functions for objects. And finally, some generic utility functions. We'll look at just a couple examples from each area, and although these are some of the most useful functions for each area, there are many more functions, about 15 to 20, in each area. So there's much more to explore with Underscore than we will look at in this section. Fortunately, Pluralsight offers a complete course on Underscore if you want to take a deeper look at it. We're going to start out by looking at a few functions that operate on collections. Collections can be either an array or an object. We'll start by looking at one of the most useful functions, the each function. The each function allows you to execute a function on each element of a collection. So let's say for example we have an array of items and we want to log out each of those items to the console. We can use each, then we pass in our collection, and then we pass in a function that operates on each of those items. Passing in a parameter which will be the item itself. And then we write a function which operates on the item. It's important to note that each will not change the original collection. If you need to change a collection somehow, that's what Map is for. It will create a new function out of an existing function by manipulating each item in some way. Let's say that given an array of integers, we want to produce a new array that has the integers squared. From the function we pass in, we simply return out the new value. Another highly useful function is the filter function, which will create a new array based on the original array, but with certain elements filtered out based on the

function that you pass in. So let's say we have an object that is a list of names of people with their ages. We want a new array that just has the ages of the people who are over 30. There's also a set of functions that operate on arrays. One of the most useful of these is the index of function, which will give you the index that will give an item. The second parameter is the item whose index you want to look up. And you can see that Underscore tells us that three is the item at index two. Another category of utilities that Underscore offers is functions that operate on other functions. One of the most useful of these is the bind function. If you'll remember, in JavaScript functions do not have a fixed context. That means that the this inside of a function was based on how the function itself was invoked. But we can change that by using the bind function. So let's create an object, and we'll create a function that will operate on an object with a first name and a last name property. Now if we just call that function, we're going to get an error. That's because invoking the function like this when it executes this will not have a last name or a first name because this will be the window object. So let's create a new function that is based on the last first function that is bound to the Joe object. I'm going to pass in the function first and then the object I want to bind it to. Now if I just execute Joe last first, you'll see that the last name and first name are based on the Joe object and not the window object. Now we can even prove this is true by creating a new object and giving it a method of that Joe last first function. And then if we call Bob dot last first, you can see that it's still bound to the Joe object. If instead I create Bob with just a last first function, then if I call it, it's going to be bound to the Bob object because that's how I invoked the function. So the bind function is really useful for creating functions that are bound to a specific object as their context. Another useful utility function is the once function. That will create a new function based on another function that will only allow it to be invoked once. Let's say that we've got a function that alerts the user using an alert. Now this function is executed whenever there's a problem with the system, that's fine so long as it's once. But if somehow our system continues to stay down and we execute this function every minute, it can get pretty annoying to the user. So we want to create a new function that will only call an alert once and then on every subsequent invocation we'll simply do nothing. That's what the once function is for. Now if I execute this function, we'll see that it's created the alert. But if we execute it again, nothing happens. The once alert user function will only call alert user once and after that will not call it again no matter how many times we invoke it. The next category of functions is functions that operate on objects. Let's create an object that has a list of states and capitals. Now let's say that I only wanted the keys from this object. Underscore gives us a nice function called keys that we can call an object and will return an array of only the keys from that object. It's got a corresponding function called values that will return an array of only the values. Underscore also has a set of functions called the is functions that will help you easily identify what type of object

an item is. So if you want to test to see whether a given item is a function, you can call Underscore dot is function and pass in some kind of an object and it will return true if that object is a function. And if that object is not a function, then it will return false. There's also an is object function that will return true if that object is an object. There's also an is array function that will return true if the object is an array. There's also functions to tell you if an item is a string, a Boolean, a number, and many different other types as well. The last function I want to look at is one of the utility functions. The escape function allows you to take a string and prepare it to be used as HTML. So let's say I have a string that's help ampersand fitness. This string right here wouldn't work as HTML because the ampersand is a special character in HTML. But using Underscore's escape function, it will turn that string into an HTML string. So as you can see from just these few examples, Underscore has a lot of useful utilities. And there are tons more that we haven't even looked at. I highly recommend that you get comfortable with Underscore and add it to your utility belt as a front end developer.

Modernizr

Modernizr is a browser feature detection library which enables you to easily and quickly detect in code whether or not a specific CSS3 or HTML5 feature is supported by the current browser. This will let you easily and quickly determine if a feature that you rely on is supported, and if not, you can then use some kind of fallback feature or take some other appropriate action based on your application needs. You can install a generic version of Modernizr using Bower, but it's often better to use Modernizr's build tool. You can see that here on the Modernizr site. So for our purposes, we'll assume what we need is to detect whether or not local storage is enabled So I'll go over here and check the local storage button. I'm also going to make sure that modernizr. load is selected. I'll show you what that does here in a minute. Now that I've got the features selected that I want, I'll come down and select generate. And now I can click download. Over here in my project, I've already downloaded the Modernizr file that I created, and incorporated it into this HTML page. I'm going to write some code that'll use Modernizr to detect whether or not local storage is enabled in this browser and then take action based on whether or not it is supported. Checking for support with Modernizr is extremely simple. You just call Modernizr and then a property that corresponds to the feature you're trying to detect support for. This is a simple Boolean value. Based on that, I can take action. So I can use local storage to set a value. And if the feature's not supported, then I'll fall back to another implementation. In this case, I'm going to use cookies. Now let's view that page in the browser. If we open up our console, we can see that we use local storage because if I call window dot local storage, got get item, then we'll get the value

that was stored when the page loaded. Now it's great that we can use this feature, but having to put this check everywhere that we want to store something locally might get to be a little bit cumbersome. So instead, what we can do is tell Modernizr to load one of two different libraries based on whether or not one or more features are supported. We do that using that Modernizr load feature that I mentioned earlier when we created the Modernizr file. The load function takes as a parameter an object that has some keys that tell it what to do based on given situations. So in our case, we're going to test whether or not local storage is supported. If it is, then we're going to use our local storage engine. If it's not, then we use our cookie storage engine. Let's take a look at those two files so you can see how they're implemented. The local storage engine creates an object that has two functions, set item and get item. Very similar to how local storage itself works. The cookie storage engine implements the same two functions, set item and get item, but the implementations of the functions are different. In this case, all I'm doing is logging out to the console which engine I'm using. So either using the cookie storage engine or the local storage engine. In either case, the object created will be named storage and put into the browser, but only one of these two files will actually get loaded based on whether or not local storage is supported. This will save the user the bandwidth of having to download unnecessary code. So if local storage is supported, then I use the yep key to indicate what file I want to load. And I use the nope key to indicate what file to load if local storage isn't supported. And finally, I can use the complete key to execute some code after the appropriate script has been loaded. Now that we've got that implemented, let's go re-run our page. And you can see that it's printed out the string using local storage because local storage is supported in this version of Chrome. So there's a basic introduction to using Modernizr in order to use feature detection and change what code you're running based on whether or not specific features are supported by the current browser.

JSLint & JSHint

JSLint and JSHint are code quality tools that check your JavaScript for things that may cause problems. Because JavaScript is such a versatile language and because it was implemented by browsers before it was standardized, it allows all kinds of things that may lead to bugs in code. JSLint and JSHint are tools to help you see potential problems and avoid them before they actually become problems. Sadly, many of the issues that can lead to problems aren't necessarily issues if used carefully. So it's easy to overlook them because your code still works correctly. JSLint and JSHint will provide you a way to force yourself to meet certain stylistic guidelines with your JavaScript that will help you maintain a consistent style in your code and avoid some potential pitfalls as your code grows and changes. This can also help teams keep their code more

consistent to make the code more easily maintained by other members of the team. These tools are not for everyone, but you should definitely be aware of them because many development teams will standardize on them and you may be asked to conform, or you may want to implement them to help other developers or even yourself keep code uniform and as free from potential pitfalls as possible. We are going to use WebStorm's built in JSLint and JSHint integration to demonstrate how they work. But you can use other tools to run JSLint and JSHint against your code to look for quality violations. Here I've got a simple file that creates a couple event listeners on a couple of buttons. This code works 100% correctly in all modern browsers, but a sharp eye will notice that WebStorm is complaining about a couple of things. WebStorm has its own built in code quality suggestion tool, but we're going to ignore that for now and turn on jsHint. JSLint was the first of the two products and was built by Douglas Crockford. Over time, many developers disagreed with the strict rules in JSLint, considering them to be too Draconian. And because of that, a competing product, JSHint, was born, which is a bit more relaxed and offers more customization. We turn it on by going to the settings dialog. And inside of here we select JSHint underneath code quality tools under JavaScript. And then I'm going to check enable. I'm just going to leave the default options set. Now that I've turned on JSHint, you can see that I'm getting some actual errors. We can hover over each of these erros to see what they are. This one is missing a semicolon, so let's fix that by adding in semicolons. And we have the same thing down here. This one is saying that we're missing the use strict statement. We can turn that on and then I get different error, that I should use the function form of use strict. I could use that by wrapping my code in an immediately invoked function expression, or iffy, which is done like this. And now all of our errors have disappeared and we're passing our JSHint check. Now this process is often called Linting, regardless of whether or not you use JSLint or JSHint. As you can see, I've had to do quite a bit to my code in order to get it to pass JSHint. I can go in and turn off some of these options to make it a little bit easier to pass the JSHint check. But instead of doing that, let's actually do the opposite and switch over to JSLint and see what kind of errors we get. I'll turn off JSHint and turn on JSLint. And now you can see we're getting a whole new slew of errors. Let's see what those are. This error here says that we're expecting exactly one space between the word function and the opening parentheses. So we need to go in here and add a space. We're going to have to do that to these two functions right here. And let's see what we've got. Document was used before it's defined. And that's the same problem with window. In order to avoid these errors, we've actually got to define our global variables. So I have to go up here to the top and add in a comment and list any global variables that I want JSLint to ignore. I've told JSLint to ignore the document and window global variables, and now it's passing the check. So you can see that JSLint was quite a bit more strict in order to get the code to pass. So let's go

back and turn off JSLint and turn JSHint back on and disable a couple of features to make it easier to get our code to pass JSHint. I'm going to do a couple of things. I'm going to turn off the warn when not in strict mode because I don't care if my code's in strict mode. And that should be good. And now I can go in and take this code outside of the function for use strict. And I can move it back. And now you can see that JSHint is still passing and I didn't have to wrap my code inside of a use strict function. JSLint and JSHint are great tools to use to make sure you don't do things like forget semicolons where they're needed or structure your code in other ways that can lead to problems. As you have seen, JSLint is quite a bit more strict than JSHint. And both of these have a lot to do with style. A lot of developers disagree with a lot of the rules in JSHint and JSLint. Thankfully with those tools you can customize them. JSHint to quite a greater degree than JSLint. But if you do use them and follow your guidelines, these tools can help you avoid potential pitfalls in your code.

Grunt

Grunt is a JavaScript task runner. It is similar to Windows Powershell or Batch Files or Rake or MSBuild or any other task runner. It is often used for a build process, but it is not nearly so limited. You can use it to automate just about any task. It runs on Node and has a command line tool, but like jQuery it has tons of plugins authored by the Grunt community, so you can probably find a plugin for just about any task you want to automate. For our purposes, we'll imagine that we're not using WebStorm, but we want to Lint our JavaScript code with JSHint. Well, Grunt is a perfect tool for that. You can easily set up a Grunt task to Lint your code whenever you want. In fact, there's even plugins that will let you watch your files and run tasks like Linting whenever any of your development files change. This will keep you from having to manually launch the task whenever you change a source file. This will be a very brief introduction on Grunt, and we will cover very little. So if you want more information, you can check out Pluralsight's other courses on Grunt by searching for Grunt. Let's start by looking at how to install Grunt and then we'll look at how to get it to Lint our files for us. As I said, Grunt is a Node utility. So we install Grunt by using NPM. Grunt actually has two pieces. There's the command line interface, which is Grunt CLI. And you should install that globally. So using NPM install Grunt CLI with a dash G command will install it globally for you. I won't install it on my machine since I've already got it installed. After you've installed that, you'll next need to install Grunt for your project. I've created a demo project and I wish to install Grunt inside of it. So I'll do NPM install Grunt. And that will install Grunt for me. In addition to installing Grunt, I want to install any plugins that I need to use. In this case, I only need to plugin for JSHint. The name of that plugin is Grunt dash catrib dash JSHint. Now that

I've got Grunt installed, I need to create my Grunt file. The Grunt file is the configuration that tells Grunt how to run. I've already created a sample Grunt file that will run JSHint for me. Let's take a quick look at a couple of pieces of this Grunt file. With a Grunt file, you have to do two things. You have to set up your configuration, which is done here in the init config section. And you also need to load in your plugins, which is done here with this load NPM task call. The important part is here underneath the init config. As you can tell, this is just JavaScript. The init config method takes in an object that has a set of keys that corresponds to each task you want to run. I've created a JSHint task, and inside of that I specified which files I want JSHInt to Lint for me. I've given them the name JS files, but I could actually name this anything I wanted. It really doesn't matter. And then the value of this property needs to be an array of strings. Each string is a file path for files that I want JSHint to run against. In this case I specified that I want all the JS files anywhere inside of my JS sub directory. So as you can see here in my project, that includes these four files here. Now that I've got my Grunt file configured, I can go ahead and run Grunt by typing in Grunt JSHint and hitting enter. And you can see that I'm getting two errors. I've got a problem in my addins. js, that there's a missing semicolon on line ten of column two. There's also a missing semicolon in main js on line 16 column 21 and line 18 column two. So I've gotten Grunt to run my JSHint and JSHint is telling me everyplace where my files have Linting problem. So you can see how easy it is to set up and run Grunt in order to automate tasks for you. Grunt is a really great tool and definitely one you should be familiar with as a front end developer.

Gulp

Gulp, like Grunt, is a JavaScript task runner built in Node. Gulp is newer than Grunt, and was built as a reaction to things that developers didn't like about Grunt. The most significant difference is the fact that Gulp uses JavaScript instead of JSON for configuring tasks. Let's look at Gulp briefly by doing the same thing we did in Grunt and get it to Lint our code for us. We start by installing it globally. Once that's done, inside of our project we'll still want to install it locally. And we'll also want to install any tasks that we want to run. In our case, the one task we want to run is the JSHint task. At this point we've got Gulp successfully installed and now we need to configure it. So we'll go WebStorm. And inside of here you can see I've got a blank Gulp file dot js. Just like the Grunt file dot js, this is where we configure our tasks. We start by bringing in Gulp itself. Using the required statement. Then we can bring in our task. At this point we can create and configure our Hint task. We do that with the task function. Passing in a string is the name of the task. And a function. Which is what the task will do. So we'll call the SRC function. This tells Gulp that this is going to be the source of this task. This takes a parameter which is a string that's the path to the

source files. In this case it's going to be the JS folder and star dot js. Then we call the pipe function. Here we're passing in our JSHint task, telling it to Hint those files. And now we have to tell it to write out any errors to the console. We call the pipe function again. Calling the JSHint reporter. And giving it the parameter default, which will be to the console. Now that we've got that task created, we can go back to the command line and call Gulp JSHint. And you can see that that's run JSHint against our files and we're seeing the same errors as before. Three missing semicolons. Gulp, like Grunt, is a very versatile task runner, but its main difference is the fact that it uses JavaScript instead of JSON to configure your tasks, so you can choose based on preference or your local environment. But because of their popularity, it's useful to be familiar with both of them.

MVC Frameworks

As time has gone by, the demand has increased for more responsive and user friendly front ends to our web applications. This has pushed more and more code from the server to the client. The relatively recent emergence of single page applications has shown that significant portions of logic can be moved to the client and the end result is a web application with functionality that just can't be matched by the old model, where most logic resided server side. In order to obtain this kind of application, we can end up with tens of thousands of lines of client side JavaScript code. I've personally worked on a project with over 100, 000 lines of client side JavaScript. This was unthinkable just a few years ago, but with the invention of client side JavaScript MVC Frameworks, we have been able to push far beyond the limits of what anyone thought was possible for the average development team. Today there is a veritable plethora of MVC Frameworks for JavaScript, but in order to keep our discussion focused, we will briefly talk about five of the more popular frameworks and discuss their pros and cons. You shouldn't consider this to be an exhaustive list of the best frameworks out there. Do your own research when choosing an MVC Framework. The first framework we'll discuss is Backbone. Backbone was the first MVC Framework to gain widespread popularity. Much of the development community has said that Backbone is not so much a framework as a library upon which frameworks are built. Backbone actually has many frameworks built upon it. The most popular of these is Marionette. Backbone has many pros, the first of which is size. Backbone by itself is the smallest of the four frameworks and its only prerequisite is the Underscore library, which is also quite small. Another of Backbone's strengths is how unopinionated it is. Because it is so small, it really provides just the basic building blocks of an MVC framework. So you are free to fill in the gaps however you want. It is also one of the most used frameworks for production sites today. On the con side,

Backbone's popularity isn't growing nearly as much as some of the other frameworks. Additionally, its unopinionated nature can be considered a drawback depending on the developer and is a frequent criticism of Backbone. Also, its small size means that it has relatively few features compared to other frameworks. For example, it doesn't handle binding, so you'll have to bring in your own library for binding data to the UI. Typically Handlebars is used, but that limits you to one way binding, which is quite a bit limiting compared to the other frameworks. In any case, it's undeniable that Backbone is extremely popular, and the pros obviously outweigh the cons. The next framework is Knockout. Knockout was another of the early frameworks to gain widespread adoption. Its key pros are the fact that it automatically updates your UI when your data changes, and it also keeps the model up to date when the UI changes using two way binding. This is a feature that is available in most of the other frameworks out there except for Backbone. Knockout also uses the MVVM pattern instead of the more traditional MVC pattern, which can make it feel more familiar to developers who have worked with MVVM frameworks before. Another great feature of Knockout is that it is fairly declarative in nature, and a lot of the coding you do is just decorating your HTML with additional instructions that set up the bindings. And that means you're doing less coding overall. Knockout has a few cons as well. Its main drawback is that it is limited and doesn't natively provide things like Ajax communication and routing, so it needs to be paired with another library to handle these features. Indeed, Knockout is really just a binding library and lacks many of the features that a typical MVC Framework provides. So that is something to keep in mind with Knockout. Another drawback of Knockout is that you can't use simple data structures, but instead you have to put your data into special wrappers and you have to access your data in this manner. The next framework we'll discuss is Angular. Of the many pros Angular has, the fact that it was built by Google is certainly a noteworthy feature. Also, Angular has by far the fastest growing popularity and if making yourself marketable is your primary concern, then Angular is the clear choice currently. More companies are switching to Angular and looking for Angular developers than any other MVC framework at this time. In fact, the Pluralsight. com site, including the video player, is built with Angular. Its dynamic community and the support from Google means that Angular is benefitting from a lot of attention right now. Although Angular is built by a team at Google, it's an open source project so it still benefits from community contributions. Another big advantage of Angular is that it is test focused and was built with testing in mind from the ground up. Everything in an Angular application could be tested and most of it could be tested quite easily. This is definitely something that sets it apart from the other frameworks, where testing is usually an afterthought. Lastly, Angular is particularly expressive. Like Knockout, it's fairly declarative and a lot of the code is done by adding directions to HTML. In fact, you can create your own custom HTML elements that

have their own behavior with Angular, which allows you to build a very component based application. On the drawback side, there are scenarios where performance can be an issue. This is due to the fact that you don't have to wrap your data in a special structure, so as a result Angular might spend more CPU time watching your data for changes. This is typically only a problem with do things like infinite scroll lists, where the page shows thousands of data items. For those pages you'll have to spend a little time tuning your performance. But for most typical pages, performance won't be an issue. The next framework we'll look at is Ember. Ember is an open source project built by a lot of developers but two key contributers are Tom Dale and Yehuda Katz. These two are fairly well known individuals. Ember benefits from most of the same features that the other MVC frameworks have, with a few key features that set it apart. Ember employs a full state machine, which means that you can do some things with deep linking to URLs that you can't do with the other MVC frameworks. Ember is also extremely opinionated, which means that the developer has to make fewer decisions which can save on implementation time and lead to a more consistent code base. But Ember has a few drawbacks. One of these is that it doesn't play well with other libraries and wants exclusive control of the whole page, so just using it on part of a page for part of a site is something that it doesn't do well. Additionally, the fact that it is opinionated can mean that it wants you to do things a certain way, and if you want to do them a different way, then you can end up fighting the framework. Lastly, Ember also requires you to wrap your data, which means that accessing and changing data requires you to call methods instead of just changing properties. Ember is definitely a promising framework and has become quite popular with the RAILS croud. React is the newest player to the MVC field. It was built by a team at Facebook and has made some waves with its extremely original and creative underlying architecture of using a virtual DOM instead of the actual DOM to allow it to only do incremental updates. React's biggest pro, by far, is its speed. There are few client side options that are as fast as React. It's unlikely that you could hand roll a faster solution without significant engineering effort. On the converse side, React is limited. It's really only the V in MVC, so you can't really use it alone for a full application. There's a bigger framework for it called Flux, and there are many open source projects to give you the other pieces in the MVC triad, but those each have their own pros and cons. One of the more novel uses of React has been to use it as the rendering engine on specific pages of an Angular application where performance is particularly critical. Overall, React is interesting and something you should be aware of. There's no clearly superior framework of the ones we discussed. Out of these frameworks and the other frameworks on the market that we didn't mention, the right one has as much to do with what you're doing and what you know as it does with the features of the frameworks that you are looking at. So keep that all in mind when choosing an MVC Framework for an application.

Summary

In this module, we looked at a lot of different libraries and tools. We looked at Bower for an easy way to install third party libraries into our projects. We looked at jQuery, the most popular JavaScript library, and some of the critical functions that front end developers should be familiar with. We also looked at Underscore and Lo-Dash, libraries that front end developers should probably use more than jQuery. We also looked at Modernizr for detecting features in the browser and dealing with missing features. Then we looked at JSHint and JSLint and how to use them to check the quality of our JavaScript code to help us avoid some of the pitfalls of JavaScript. We also looked at Grunt and Gulp, JavaScript task runner, and saw how to set them up to Lint our code. And lastly, we took a look at a few of the more popular MVC Frameworks that are out today and talked about their pros and cons. None of these tools are absolute requirements for doing front end development. But being familiar with each of them will be of benefit when doing any serious amount of front end web development.

Performance

Introduction

Hi, I'm Joe Eames. In this module we're going to be learning about the basics of performance with web front ends. We'll start by looking at where to place our style and script tags in our HTML page and why it matters where they go. We'll also look at the use of content delivery networks and the benefits they have on performance. After that, we'll talk about minifying our code and how and why to do it. And lastly, we'll look at concatenation, what that is and why we should do it.

Script and CSS Placement

When a browser processes an HTML page and encounters CSS links and script tags, it causes the browser to make a request for those assets. Where you place your script tags and CSS links can effect the performance of the page, so it's important to place them properly in the HTML. Let's talk first about CSS. Browsers process an HTML page top to bottom. As the browser processes the page, if it encounters HMTL then it processes and renders the HTML according to the default styles the browser has plus any CSS rules it has already encountered. As it encounters CSS style rules, that will cause the browser to go back and re-render any elements affected by the CSS

rules. Therefore in order to avoid unecessary rendering time re-rendering those elements, it's best to have all CSS processed before any visible HTML. This means you should place your CSS links in the head of your HTML document. That way they will all be processed before the browser gets to the body where the visible HTML is. Conversely, script files work differently. It might seem like the head is also the best place for script files, but that's generally the worst place for them. Whenever a browser encounters some script, it will pause all other rendering and process the script file. That means that a script file in the head will likely cause the page rendering to pause while the script file is processed and ran. So to make your page render quicker to the user, it's almost always a better idea to put your script tags as the last item in the body and then the HTML for the page will be visible and rendered before the browser processes your scripts. So as a general rule, put styles in the head, and put scripts at the end of the body.

Benefits of Content Delivery Networks

A content delivery network or CDN is a third party service where you can host your web assets, such as script files or CSS files or images. This is contrary to the typical method of bringing all your assets together, placing them on your server and then pointing to them from your HTML pages. There are several major advantages to using a CDN. The first is that the contents may already be pre-cached. For example, instead of serving jQuery from your own server, you can just point at a public CDN which hosts jQuery. That way if the user of your site has previously visited another site that points at that same jQuery on that same CDN, then the file will already be cached and they will avoid downloading the same file twice. Another advantage is that CDNs typically have multiple data centers distributed around the world. That means that a request for a file will be routed to the nearest data center. In the age of the internet that may not seem like an advantage, but it can actually have a very noticeable difference for people who are in a different country from your server. The next advantage of a CDN is that they typically have high capacity infrastructures. So that even during high peak times they can handle the load easily and undoubtedly have better scalability than you do. The last advantage of a CDN is the fact that it reduces load and bandwidth on your own servers. All the assets pulled off a public server are bits that aren't getting transmitted over the wire that goes to your server, and the CPU load is offloaded as well. So that means you might be able to spend less money on your own servers and bandwidth. As you can see, the benefits of CDNs are quite significant. Even if you don't put your own custom assets on them, at least using them for third party scripts like jQuery or Underscore is extremely easy to do and will improve performance on your site.

Minification

Minification, which is also sometimes called uglification, is a process of compressing JavaScript for two purposes. First of all, it reduces the size of your JavaScript, which will make your site more performant and scalable and save you on bandwidth. Second, it (mumbling) the names of identifiers in your code, which makes the code harder to understand which will protect your intellectual property from theft. Let's look at an example of how minification works. We have here a sample function called search, which takes in two parameters. Minification involves two processes. First, it takes any local identifiers, like parameters, and renames them to shorter names. This reduces the byte size and also obscures the intent of the code. You can see here the user list has become "a" and users has become "b. " The code still works just fine but now it is smaller and harder to understand. The second thing a minifier will do is remove all unecessary white space. This removes even more space and also makes it even smaller. Although it's a simple process to put white space back in, so this isn't much of protection against theft of intellectual property. Now you may be wondering if all this effort is really worth it. Let's just glance at jQuery to see how well minification reduced the size. Here is the unminified and minified versions of the latest jQuery. As you can see, minification has removed about two-thirds of the size of the file. That's a significant savings. Now you can do the same process for your style sheets as you can for your script files. You'll save less space because the minifier can't reduce the size of the rules, it can only remove the white space. But you can save a little space this way. Now if you want to minify your own files, there are many tools out there to do that, one of which is Grunt. There's a Grunt task called grunt-contrib-uglify that will minify a code for you, and it's pretty simple to set up like JSHint was. But there are lots of other tools out there which will minify your JavaScript for you. The right tool will depend a lot on your environment. In any case, if performance and bandwidth are important to your project, then minifying your code is definitely something you'll want to do.

Concatenation

When the browser processes an HTML page and encounters any style sheets or script tags, it has to send a request back to the server for those files. Every request that the browser has to make to the server for one of those assets means additional time spent waiting for the request to travel across the wire, get processed on the server and sent back. Each of these requests has a certain amount of overhead. So for example, transferring four script files that are each 1K in size will take quite a bit longer than transferring one script file which is 4K in size. Remember that how we separated out our scripts and style rules in a separate file is only for our benefit during development. To the browser, having those assets in separate files or the same file doesn't matter.

The only thing that matters is the order. Styles need to be in the correct order so that the correct styles override previous styles. Scripts have the same requirement. You must have prerequisite script objects created before the objects that depend on them. By having them in the same file or separate files has no bearing. Because of this, concatenation was developed as a means of improving website performance. This process involves taking multiple style sheets or script files and combining them into a single file. This is only done for production since in development this makes it more difficult to debug and diagnose problems. Then your HTML file is changed to request only the combined file instead of all the different original source files. There are a lot different products that will do this for you such as most server side frameworks plus many different utilities such as Grunt. I won't be demonstrating any of these products since each tool works quite a bit differently from any other tool. If you do want to see an example of this, my course on Angular for. Net developers shows examples of doing this with both Grunt and the ASP. Net framework.

Summary

In this module we talked about performance with web front ends. These tips, centered around making your front end more performant, by eliminating any wasted rendering time and rendering the page as soon as possible by placing your script and style tags in the appropriate places in your HTML page. We also talked about using a CDN to offload work to high capacity, third party servers. We also showed how minification can drastically reduce the size of the JavaScript that you transmit over the wire. And finally, we talked about how concatenating your CSS and JavaScripts into a single request can make your site more performant by reducing the number of requests made. All of these techniques will help your web applications perform better, which will make a big difference to your users for a small amount of effort on your part. Taking a little bit of time now to make your applications perform better will benefit you every time a user hits your site.

Course Summary

Throughout this course we have looked at many, many different topics and technologies. It has been a bit of a whirlwind rush to look at so many different aspects of front end development. This is the nature of doing front end work on websites. There are so many pieces and so many new advancements that most developers aren't taking advantage of that it can quickly become overwhelming. So please take some time to try out some of the things we discussed and play

around with them yourself and hopefully implement them in your next projects. The only thing worse than learning something new is not using it. Like the bird who went to flying school and then walks home, don't waste the time that you've invested by doing nothing. I hope you enjoyed this course. If you get a second shoot me a Tweet and say hi. Connect with me on LinkedIn. Leave any constructive criticisms or comments in the course discussion board, and thank you for the privilege of taking this journey with you.

Course author

Joe Eames

Joe began his love of programming on an Apple III in BASIC. Although his preferred language is JavaScript, he has worked professionally with just about every major Microsoft language. He is...

Course info

Level	Beginner
Rating	★★★★½ (1217)
My rating	★★★★★
Duration	3h 47m
Released	24 Apr 2014

Share course