# **JavaScript: Ajax and Fetch**

### **Creating and working with API requests**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Sasha] Ajax is the backbone of interactive high-performance web apps. It allows you to incorporate new content into a web app without reloading the page. But using Ajax effectively requires understanding different JavaScript syntaxes, as well as working with data and manipulating the contents of the browser window. Building code using both XHR and Fetch can help you get a handle on how Ajax is implemented across different code bases. Examining Ajax requests and responses in the browser and working with data structures to modify webpage content can give you practice putting Ajax to practical use. In my LinkedIn Learning course, I explore modern coding practices, as well as those found in some legacy code, and I also show you how to secure your API keys in a proxy server. I'm Sasha Vodnik, and I've been writing JavaScript since the browser wars of the '90s. If you want to get in a firmer footing with making API requests and incorporating the results into your apps, I invite you to join me for this course on Ajax and Fetch.

### **Setting up your environment**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] To work along with me on the files for this course, you need two applications: a web browser and a code editor. You undoubtedly already have a web browser installed on your machine, and any major modern browser, Chrome, Firefox, or Microsoft Edge, is fine for this course. I'll be using Chrome in these videos, which is a popular choice among web developers because of the extensive and powerful developer tools it has built in. I've customized Chrome with the JSON Formatter extension by Callum Locke, which is helpful when examining JSON data in the browser window. A number of great code editors are available, both free and paid apps. Any editor that lets you edit and save plain text is fine for this course, so if you have a code editor you like, such as Sublime Text or Atom, it's fine to use it. I use Visual Studio Code in these videos, which is a version of Microsoft's Visual Studio created specifically for web development. Visual Studio Code is free and has Windows, Mac, and Linux releases. The code is available on GitHub, and users can submit issues there as well. I've turned on word wrap in my editor. If you want to do the same, just click View, and then Toggle Word Wrap, this ensures that long lines of code don't run off the screen. I've also installed a few extensions. Bracker Pair Colorizer 2 by CoenraadS color codes each nested pair of parentheses, brackets, and braces to make nesting levels easier to distinguish. Note that this is a new, improved version of a previous extension called simply Bracket Pair Colorizer by the same creator, so be sure to grab version two for the most up-to-date code. Indenticator by SirTori highlights your current level of indent, which is also helpful in identifying your location in nested code. Finally, Live Server by Ritwick Dey is an HTTP server you can launch with a single click that automatically opens the current document in your default browser. Testing Ajax code during development benefits from an HTTP server, and this one is really easy to install and use. If you want to learn more about anything I use or talk about in this course, I encourage you to explore the library for a deeper dive on that topic. Now let's get started.

### **What is Ajax?**

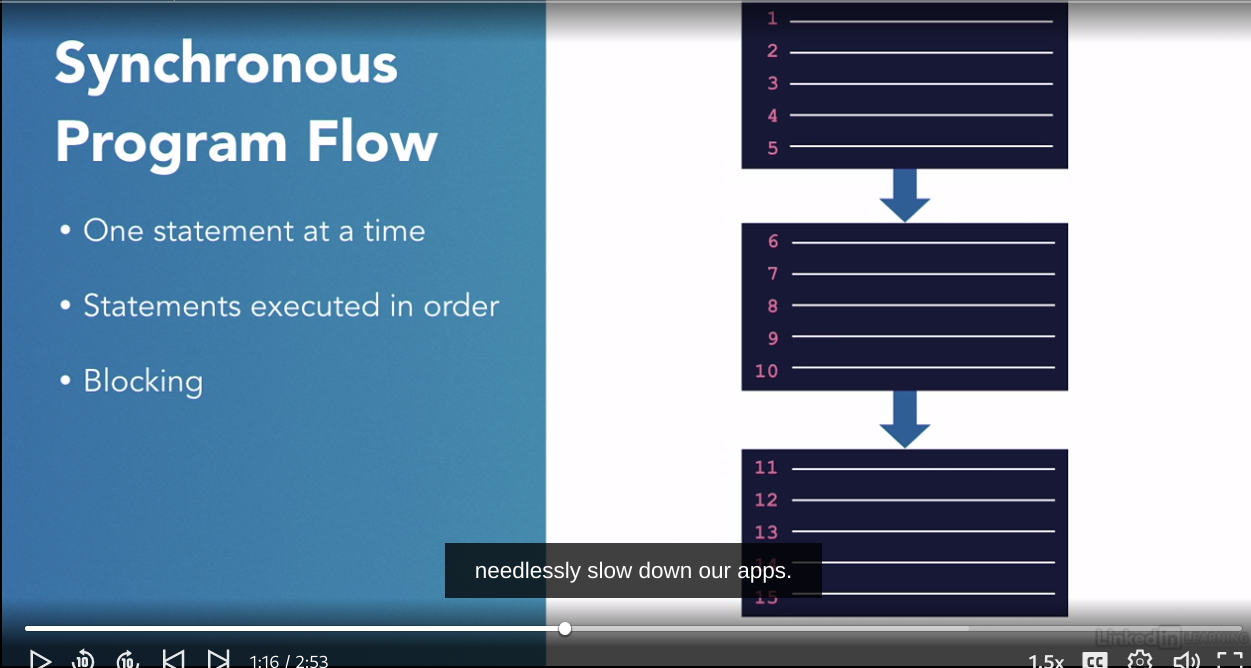
Selecting transcript lines in this section will navigate to timestamp in the video

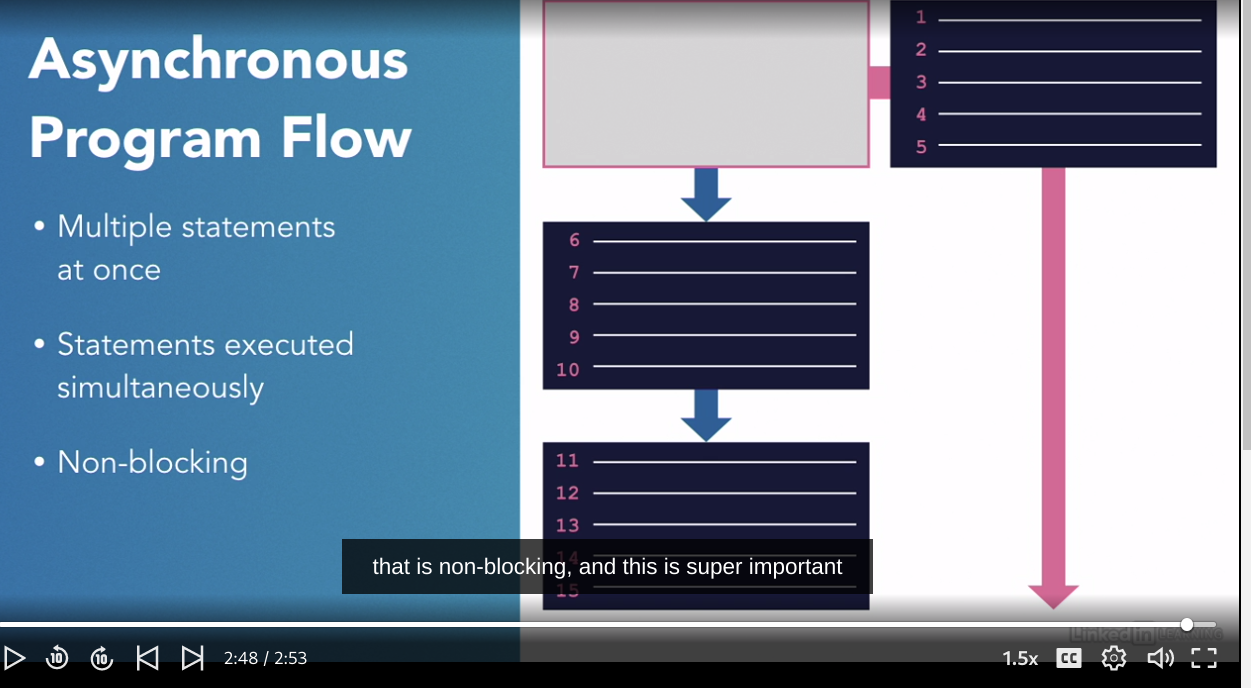
- [Instructor] The dynamic nature of web apps and web pages is an integral feature of the modern web, but this was not always the case. In its early days, the web was composed of static pages. Each page displayed only the content coded into its HTML document. Changing the information displayed meant requesting and loading a different HTML document. But new documents take time to load and to render. What if we could, instead, simply request content and update the display of the current page? In fact, this is exactly what AJAX lets us do. A page that uses AJAX, which is most everything on the modern web, requests data in the background, and then integrates it into the document already displayed in the browser window. Originally, AJAX was an acronym, based on the fact that the process was asynchronous, was scripted from JavaScript, and originally used XML as the data format for the new content. Today, AJAX describes a more generic process. Although JavaScript is still a core building block of AJAX, it's much more common to use JavaScript object notation, or JSON for data exchange. AJAX is at work anytime you interact with a webpage, and the page changes to display data that was not present when it first loaded. For instance, in may search engines, when I start typing a search into the search box, the app sends what I've typed so far to the server, receives a set of likely search phrases based on that, and offers them to me in a context menu. As I continue to type, these options change, always based on what I've typed, but without loading a new HTML document. Another place we often see AJAX is in feeds that use infinite scrolling. This feature enables the feed to load just enough articles that a user can continue scrolling. So I can go on this site and grab the scroll bar, and when I scroll down to the bottom of the feed, more articles load so I can continue scrolling. This enables users to continue scrolling through a feed for as long as they want to, but ensures that the page does not request a lot of unnecessary data. Instead, the data is requested in chunks, and only in response to the user's intent to continue scrolling.

### **Understand asynchronous program flow**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] A client can experience a lag between submitting a request for data and receiving the response. For this reason, it's important to structure code to minimize the impact of this potential delay on users. We can build program flow in JavaScript in two different ways, synchronously and asynchronously. Using synchronous program flow, code is executed in the order it appears in your JavaScript files. So here, lines one through five of the code are executed first, followed by lines six through 10, and then lines 11 through 15 are executed next, matching the order of the statements in the code as written. In synchronous program flow, the browser parses and executes only one statement at a time. This results in our statements being executed in order. Synchronous program flow exhibits a behavior known as blocking, meaning that later statements are prevented from executing until earlier statements are finished. In many cases, this is what we want. For instance, a value needs to be calculated first before it can be used. But if an earlier statement may take a while to execute, and statements that follow don't rely on that earlier statement having finished, then synchronous program flow can needlessly slow down our apps. This is often the case with an Ajax request, which is why JavaScript treats an Ajax request as an asynchronous process by default. We want the client to make the Ajax request, but then be able to do other things, like respond to user initiated events, while waiting for the Ajax response. Otherwise, a slow response would essentially lock up our apps, which would create a bad experience that frustrates users. So this is an example of asynchronous program flow, also known as asynchronous code, or simply async. Here, lines one through five of the code start execution, but these include some activity that's going to take a while. Now, technically, JavaScript has only a single thread, but does have a mechanism that allows some processes to execute in parallel while other things are going on. So here, the first set of statements is moved into that parallel process, which continues executing even as the main program flow moves to the next set of statements. Now, that main program flow continues synchronous execution, so it keeps moving through the remaining code while that first block of code is executing in parallel. Now, remember that synchronous code is executed only in the main program flow, while we've seen that asynchronous code is instead moved over to execute in parallel. This means that those statements are essentially executed simultaneously. The upshot of this architecture is that asynchronous program flow lets us write code that is non-blocking, and this is super important for things like an Ajax request.





### **Get an API key**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this course we'll be improving the usability of the sign up form for Explore California, a fictitious visitor's bureau. When a user enters a U.S. address, city and state, I want the form to automatically look up the corresponding nine digit zip code and fill that in for us. To do this, we'll make use of a web service. A wide variety of web services make data available. These include all sorts of subject areas from weather to Star Wars character references. Some are free while others require a payment. Todd Motto is a developer who maintains this pretty extensive list of free APIs. So scrolling down there's an index that gives us a list of subject areas. And we've got Geocoding which in general is how the data that we're looking for is described. So we have possible web services here from all over the world giving us specific data on specific parts of the world. And we are looking for U.S. data and we are specifically looking for zip code data. So I've looked through this list and I do see one that's interesting which is Zippopotam. And this is clearly a zip code API. And so we'll give this a go. So following that, it's zippopotam.us. So Zippopotamus and it's actually right here on the front page it's got kind of overview of what you can ask it and what it can give you. So it's an API with JSON response format. And we can send it a zip code and it's going to give us back some information about where that zip code is. Or we can send it a state and a city in the U.S. and get back that zip code. Now the thing is for our uses, we want to be able to send a street address, city and state and get a nine digit zip code back. And Zippopotamus doesn't actually do that for us. And so I've looked at other sources already in the Geocoding section on Todd Motto's list and unfortunately none of them are going to do what we want. So at this point I'm going to go web search and I'm just going to search on zip code API and there's a few options. So at this point, it's a question of just checking out the options and seeing what might work. So we've got this one, SmartyStreets that's a zip code API. I'm going to check that one out. And so scrolling through this documentation, I looked through this earlier and what I discovered is that we actually have a precision down to nine digits. So we can actually get nine digit zip codes out of this service. Now SmartyStreets doesn't appear to be free. But if we look over here on pricing, I'm going to United States. And we've got 250 lookups per month for free. So that should be enough to get us through building out this course. This is obviously a service where if we're actually implementing this in an app that has any user base at all, we're going to need to move up to a paid plan. But to be able to build out this course, we should be able to do that on the free plan. And so from this pricing page, if I go to Products in the menu and select APIs, I can see their selection of APIs. Now U.S. zip code API might seem like the one we want. It's actually U.S. street address API that is going to give us back the nine digit zip codes. And so they even have this demo here where we can click this arrow for Try a simple data set. And I'll just click this one in Belmont, Massachusets. And it's filling in basic information, street, city and state. And it shows me the request that it generated as well as the response. And so we look here and we have a nine digit zip code. And it's broken down into a five digit zip code and a four digit plus four. So that's exactly the data we need. So this API can do what we need. So great. Now looking in this demo, we can see that one of the fields is this auth-id. And this is super common with web services. And it's generically referred to as an API key. This is a string of characters associated with your account that you provide as part of any request you make to the API. For this sample page, they've created a general public auth-id that probably can only be used with this page. So in order to create my own requests, I'm going to have to sign up for my own auth-id on this website and you'll need to do the same. So scrolling back up, we have a Sign Up button at the top. I'm going to click that. And just going back over to the free section, click the Start Now. Okay, and so to sign up, I need to give them my first name, I need a phone number, we need an email address and then a password and then I'm going to create an account. So my account's all set. I'm going to click Start Using Your Account. And so scrolling down then we have this API Key section so I'm going to click that. And we have a couple different sections here. We have Website Keys for client-side code. We have Secret Keys for server-side code. So we're going to be working primarily with client-side code. And so I'm going to click this plus next to Create new key and the host name we use here is going to be dependent on our testing environment. For now, I'm going to use 127.0.0.1 which is the IP address used for local testing. But if we need to adjust that later on we can do that. I'm going to go ahead and click Okay. And now I have a website key which is this string of numbers here. And I have a host that is associated with that. Note that this key, by the time you see this video this key will have been deactivated. So you'll need to make sure you get and use your own. So I find using a password manager to be a great way to stay on top of web credentials. And if you are using that system, that is a great place to stick your API key. For now I'm just going to copy it. I'm going to open Notes 'cause I'm on a Mac. And I'm going to create a new note which is SmartyStreets API key and paste that in. And now any time I need to grab my API key if I need to use it in files, I can simply come to this document and copy and paste it in.

### **Research and create an Ajax request URL**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Once you've identified the API you want to use, the next step is to build a request. Modern web services are based on an architecture called REST, which uses HTTP requests and responses to exchange information. So we need to write a URL that requests the information we want. APIs often provide access to multiple types of data and you need to examine the documentation to find how to request the specific data you want. When an API gives different URL paths, these are known as endpoints and you need to pick the correct one for the data you want. The SmartyStreets U.S. street address API returns nine-digit zip codes. Even though there's another API called the zip code API, that actually requires a zip code to start. So, again, the documentation is super valuable when you're first picking an API and endpoint and then figuring out how to use it. Now here on the overview page, there's a demo form that creates requests and shows responses. There are a lot of fields here that aren't all that obvious in their names, but I can click Try a sample data set and then I'm going to pick the Belmont, Massachusetts address and the data is auto-populated. Now, up here to the right, is a sample cURL request. This is a command line utility that enables a developer to send an HTTP request. And the useful thing here is that it shows us how our form values are translated into key value pairs in the query portion of the URL. So we have the endpoint first, which is the main portion of the URL and then the auth ID and auth token that are unique to us. In this case, these are the sample auth ID and auth token that are specific to the sample website. And then we have candidates, which seems likely to indicate the number of results we want back. And then we see things like street, city, and state, along with key value pairs with empty values for two of the fields, zip code and street two. And we will be providing values for these. So we can actually take those out of our search with this interface just by clicking the minus next to them. So I'm going to click the minus next to zip code, the minus next to street two, and those are removed from our sample URL. So we have a basic URL structure, which gives results that include those nine-digit zip code sections. So, I want to copy just the URL portion, so I'm going to click in here first and then I'm going to select the URL, but not the quotes around it. I'm going to copy that to the clipboard. And then switching over to my files, I'm going to open up the \_scripts folder and the app.js file. And below that use strict statement, I want to create a new const because my app will not be changing this value and I'm going to call that URL. I'm going to give it quotes and I'm going to paste in that URL that I just copied and I'm going to finish it up with a semicolon. And then with that value still copied, and save my file, go back to the browser and in a new tab, I'm just going to paste in that URL. And here I get the return JSON. Now, by default, that's not easy to read in a browser window. It looks something like that, but I have the JSON Formatter browser extension, which lets me see the parsed view and, again, I can see that we've got the zip code, the plus four code, so all the data I want, and so I now have selected my endpoint. I have a sample URL that returns the data I want and so this is enough to get me started actually building out my app.

### **Create an XHR request**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Ajax requests in JavaScript are based on the XMLHttpRequest object. Now that's a real mouthful. So this object is often referred to as the XHR object for short. You create an XHR object to open a request to a remote URL using an HTTP method such as get. When the XHR object was first created, XML was the standard format for exchanging data on the web. Fortunately, XML is tied to the XHR object in name only and XHR works with any data format including JSON. The XHR object includes quite a few properties and methods but a few particular features are used in most requests. You use the open method to specify an HTTP verb and a URL to hit. While this method attempts to open a connection with a specified endpoint, it doesn't actually send the HTTP request. To do that, you use the send method. Once a request has been sent, the value of the readyState property indicates the current state of the request using a number from zero to four. A state of zero indicates that the request has not yet been initialized. One indicates that the request has been set up. Two that it's been sent. Three that it's in process and four that it's complete. The XHR object also specifies a readystatechange event. XHR code commonly listens for this event to fire and then uses a conditional to check if the state is four indicating that the request is complete. In this case, code to work with the response is called. In the starting code for the Explore California form, I'm going to build a function to create and send an XHR request in the app.js file. So below my URL constant, I'm going to create a const called createRequest, that's going to be a function that takes a single parameter of URL, and that will allow us to pass a URL to the function so that we can customize a request and use the same function for multiple different web services. Now within the function, I want to create an XHR object. And to do that, I use the new keyword with XMLHttpRequest as the constructor and then I'm going to pass that the URL. Now if I just create this object, I don't have any way to work with it in my subsequent code. So I always want to create a variable, I'm going to use const, I'm going to call it httpRequest and now I can use that httpRequest variable name to reference and work with my XHR object. So I create a variable called httpRequest that gives me a reference to this XHR object and allows me to use it later on in my code. And then I want to add an event listener for the readystatechange event. So httpRequest so a reference to my XHR object .addEventListener and the event we're going to listen for is readystatechange and I'm going to use an in line anonymous function so I'm just going to use an arrow function for that and so we will pass in URL as a parameter and we're going to check if httpRequest.readyState triple equals four. And if that's the case, for now we're just going to console.log httpRequest.responseText. The responseText property gives us the text content returned from the remote service which could be our requested data or could be an error message. So this code will let us see in the console whether our request has succeeded. To finish the createRequest function, I need to open a request and I want to use the GET HTTP method and I'll use that variable I created above. So httpRequest.open and we're going to pass in GET and the URL. And remember, just opening it isn't enough. I need to follow that with httpRequest.send and that is going to actually send the request. And then finally outside of that function I need to call the function so we'll call createRequest and pass it in URL and I'll save that. So now we have code that actually creates an XHR object, sends a request and logs what it gets back and we have that function setup in such a way that we can pass in any URL and examine what we get back.

### **Test an XHR request**

Selecting transcript lines in this section will navigate to timestamp in the video

- You can test AJAX code in a browser just like you would any other front-end code. However, it's generally best not to simply open your HTML file in a browser and see what happens. This is because by default a browser opens a local file using the file protocol which browsers treat differently then pages open using HTTP requests including everything you open from the web. Fortunately, it's easy to run a simple HTTP server on your development computer, which lets you more closely match the way your AJAX code will work after you deploy it. In addition, some API's require requests to be linked to a specific URL. So, using a HTTP server is a must in a situation like that. I'm using Visual Studio Code editor with a Live server extension. This extension takes care of all the configuration for me. Similar extensions are available for Sublime Text and for Atom. So, whatever environment you're using, I recommend installing and using an extension like this. For Live Server, I need to start by opening the HTML document in my editor. Then I click the go live button which starts up a server and then automatically opens my page in my default browser using HTTP and the IP address 127.0.0.1 which is a reserved address that maps to my local computer. It's possible that your Live Server will use a different IP address because every computer has a few IP addresses that refer to it at any given moment. So, know matter what IP address you have showing up here, you need to configure your SmartyStreets website key to use it. So, take note of that address, and then over on SmartyStreets you can add a host to your website key. So, my key has a host already of 127.0.0.1 you may have that but need a different IP address, and you can do that by simply clicking add host here then typing in whatever IP address you need. For example, If I was at 10.0.0.5, and then you click the checkbox over here. And now we've got two different hosts associated with this key. If one of those hosts is not actually relevant for you, you can just use the garbage can icon, confirm that with the delete button, and just have the host that corresponds to the server that you're using. And now that website key will work when the request comes from the associated address. Now, over on the page that I opened using Live Server, I'm going to open up my console, and notice that I have an error. So, I never actually swapped in my own authorization information into the URL, and I'm opening this from a different source then the SmartyStreets website, so that authorization information that they provide on their website isn't valid here. Now, checking out the Fdocumentation notice that we have two different sections of authorization information of API keys. The first is the website keys for client-side code. The second is secret keys for server-side code. We need the client-side credentials, so since I'm here I'm going to copy that key again, and then I'm going to follow this client-side code link to learn more about how to use this. So, in this documentation page it shows us a sample request, and notice that I have an auth-id key, but not an auth-token key because in this case you only need that one key value pair. So, I want to change the auth-id key in my request to my website key, and I want to get rid of that auth-token key. So, I'm going to scroll up, I'm going to hit products and APIs, and then scrolling down, this is the demo once again for the endpoint we're using. I'm going to use the Belmont Massachusetts sample data set, but anyone's going to work here, scroll back up. And so first I'm going to select the contents of the auth-id field, I'm going to paste in my own which I just copied. And then I can get rid of the auth-token field because I don't need it for a website request, so I can just use that minus button over here. And I'm once again going to get rid of the default street2, and zip code which don't have a value here. And now I have a fully ready to use URL for my request. So I'm going to click in here, I'm going to copy that, and then back in my code, in my JavaScript, I'm just going to replace the previous URL within the quotes with the one I just copied. So, now I have the auth-id key with my specific key. I still have all those other key value pairs for the data I'm looking for, but I don't have that extraneous auth-token key, and I'm using my own API keys and not the sample ones provided by SmartyStreets. So, I'm going to save my changes here, and my Live preview app automatically refreshes the view in the browser. Going back to the browser, I can see in the console that now I'm all good, I have the data I want. So, let's take a little closer look at this. I am going to move this into its own window, and then I'm going to increase the font size a little bit, so it's easier to read, and so I can see that I have amongst all this, the zip code and the plus four that I was looking for. And so this is the data, again, for the Belmont, Massachusetts sample request. But I know now that I have a starting point that gets me data back, and I can then customize this from the data that I get from my form.

### **Handle XHR success and failure**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] My code, so far, creates and sends a request and then logs whatever's in the response to the console. Now, eventually, I want to be working with the response data to change what users see in the browser window. But, until I get there, I at least want to update my code so it differentiates between a successful response and a response containing an error message. And I'll break those out into separate functions to keep my concerns separate so the AJAX request code can be changed without affecting the code to work with the response. So first, I'm going to break out my code for analyzing the response into a new function and I'll call that response method. And that's going to take the HTTP request parameter. And so, within that function, I want to check a couple properties of that request. Now, the ready state property will tell us if the request is completed. That's the case when the value's four. So, I will check if HTTP request .ready state, triple equals four, and then only if that's the case, I want my code to examine my status property. This property gives me the HTTP status code. That's a three-digit number like 200, 404, 550, it summarizes the response. If the code is 200, that corresponds to a successful response. Otherwise, there's some issue. So, I'm going to call an update UI success function that I'll create in a bit and I'm going to pass that HTTP request .response text and that is the text that we receive, the body of the response. Now, if the status is not 200, I'm going to do an else and I'm going to call a different function that I'm about to create, update UI error, and I will pass that HTTP request.status, so that's status value, and then I will concatenate a colon and a space, that needs to be a plus. And then I will concatenate HTTP request.response text and in this case, when I don't actually have have a successful response body, that response text is going to contain the text of any error and that'll let me work with exactly what's wrong. So, now I just need to define these new functions, so I'll do const update UI success and that is going to take a parameter we'll call data, and for now, I'm simply going to console.log data and likewise, const update UI error and that's going to take a parameter I'll call error and we'll console.log that error parameter. Now, saving my changes, going to switch to the HTML and go live in my browser and opening up my console, and I've got a successful response. And so, with those functions split out, everything's working like it did before. So the final change I need to make to my code then is down here in the create request function. And I need to take out this if/else section. Like I mentioned earlier, when I first set things up, each time you want to view files from a different folder, you need to make sure you've closed and restarted live server. Otherwise, you'll be viewing the files from the previous folder in the browser instead of the current files. So I need to go down here, and the live server icon has changed to just show the port and the slash symbol and clicking that will dispose of that server. Now we've got go live again and so I can click the HTML file and then click go live to open that back up. And on ready state change, I simply want to call response method and a pass it the HTTP request object. So I'm going to save that, then switching over to the HTML file, and going live in the browser, and I'm going to open up my console and so I have my response text logged just like I did before. Now, going back to my code, I want to test that error function. So I'm going to go up to my URL. I'm going to copy it and duplicate it. I'm going to comment out the first one so that I have a known good URL and then I'm going to hack apart my URL here a little bit. I'm just going to go to the auth ID key and for that auth ID value, I'm just going to take out the last couple digits, so it's going to be a invalid auth ID. I'm going to save that, then, going back to my browser, I've got an error and this is my update UI error function logging the HTTP request status and the status text. So that, as a developer, just for this development process, I can see exactly what error I got and the error text pretty straightforwardly. And so, going back and just ironing things out, I'm going to take out that broken version of my variable. I'm going to uncomment the good version of the variable. I'm going to save that. Back to my console one more time. I've got my data back, so great. Now I have success and error code working and I also have them separated out into their own independent functions.

### **Challenge: Build an XHR request**

Selecting transcript lines in this section will navigate to timestamp in the video

(upbeat music) - [Instructor] It's time for a challenge. Ready to try your hand at getting information from an API? Another aspect that we want to change on Explore California form page is to add information about national parks in California. The National Park Service has an API that provides park information. So you can get this data using an Ajax request. Check out the developer guide and the API documentation at NPS.gov. Then create a request that returns information on national parks in California. You can reuse the functions you already created in this chapter, so just focus on actually constructing and implementing the URL. This is a longer challenge that might take you around 30 minutes. If you don't have time for that right now, feel free to come back to it later on. When you're done, join me in the next video, and I'll go over how I approached it.

### **Solution: Build an XHR request**

Selecting transcript lines in this section will navigate to timestamp in the video

(upbeat electronic music) - [Instructor] The National Park Service API is free for a reasonable volume of requests, but it requires use of an API key, so we've got a register link right here, and I click that, and just have to fill in first name and my email. And you can tell 'em how you want to use it, but you don't have to, so we'll just sign up. And so, we get back an API key. Again, I'm going to copy that and save it. So then, I'm going to switch over to the documentation. We have a bunch of endpoints listed on this page, including a parks endpoint that gives us park-related data, so that seems like a good candidate. And so, if I click the blue area here, we've got some information about the parameters, and there's a try it out option. But in order to use that, I need to authorize on this page using my API key, so I can just paste in the API key I just got, and now I am authorized so that any test requests I make on this page are going to use my API key. And we see these locks meaning that I'm authorized; these icons have changed. So now, I want to try this out and what I want is Parks for California, so reading through here, I can specify state code and get back parks from that state. So I will add item; the two-character state code for California is CA, and then I'm going to scroll down and hit this big execute button. And then the request is submitted, and the results. First, I get a curl statement with that URL, and I get a 200 response from the server and a response body showing all this beautiful information about parks. So, great! I can get the data I need and so, I'm going to go ahead and grab this URL, already includes my API key, and take everything but the quotes in the URL itself. I'm going to copy that, and then back over in my JavaScript file, I am going to change my existing URL to specifically, smartyUrl, then I'm going to create a new constant called parksUrl, and I am going to paste in as a string the URL that I just copied. An so, this includes my API key, it includes my endpoint, and it includes this key value pair for the state code to get data specifically about parks in California. Now, one of the great things about learning XHR is that once you've written the code once, you can pretty much reuse the same pattern. So, you may have discovered here, you don't really need to write any new code. You've already created a reusable set of functions, so all you have to do is pass in the new URL to the create request function, and you get data handled. So, I'm going to do a couple things here. I changed my URL variable to smartyUrl, so I'm going to change that, and then I'm going to comment this out. Since I only have 250 requests this month, I don't want to submit a request to SmartyStreets every time I load my page. So I'm going to comment that out, and then I'm just going to call createRequest again, and this time I'm going to pass it parksUrl, and I'm going to save those changes. I'm going to go to my html file, and I'm going to start up my live server, and open up my console. So, the parks API can take a few seconds between when you make the request and you get the data back, but notice I've got tons of data here. It is data about parks, so it looks like we got what we needed. And then I just want to double-check errors as well, so going back into my JavaScript file, I'm going to duplicate that parksUrl variable. I'm going to comment out one of them, and I'm going to take out a couple characters from that API key value. I'm going to save that, and then back in my browser, I got a 403 forbidden error. And I can see that my error-handling function is giving me the error code, followed by the error text, which is API\_KEY\_INVALID, and I get a specific message from NPS telling me exactly what's going on. That's super helpful. And so, going back here, deleting that broken URL, uncommenting the original good URL, and saving those changes, and then back in my browser once again, I get good data back, so we're all good. We've now got data coming in from the National Parks Service with just the addition of a new URL and a function call. So we've now got data from the National Parks Service using just a new URL and a function call

### **What is the DOM?**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Educator] The Document Object Model or DOM is the way that a browser or other user agent represents the contents of a web document. The Dom serves as the structure by which developers can reference and manipulate elements in the browser window using JavaScript. The elements in the DOM are structured in a hierarchy. It can be useful to think of the structure of the DOM as a tree with the elements higher in the tree containing the elements lower down. Browser developer tools provide an interface for viewing and interacting with the DOM for the current document. In the browser window, right-click an item. On the menu that opens, you'll see an option like Inspect in Chrome or Inspect Element in Firefox. When you click that item, the developer tools open and display the Elements pane which shows the DOM tree for the current document. The contents of the Elements pane look a lot like a standard HTML document with nested elements attributes and even content. However, it's important to understand that the DOM is not the same as the HTML file that a webpage is based on. We can use the developer tools to explore the difference. So here on duckduckgo.com which is a search engine, I'm inspecting the search box itself, and the Elements pane shows the code for the input box I right clicked with a highlight over it and I can see that it's nested in form element which is nested in a div. And if I kept scrolling, I could explore the whole tree. So keep your eye on the form tag. And as I move my mouse over the text box, notice that an additional class name is added to the form tag. So we have search dash dash adv and search dash dash focus. And if I go over it again, now we also have search dash dash hover which is indicating that my mouse pointer is hovering over that input box, over that form. So changing class names is a pretty common way to change styles in response to user interactions using JavaScript. And so we can see that the code in the DOM tree is a reflection of exactly what's in the browser window at current moment and it doesn't necessarily map to the original HTML that was loaded. The Elements tab in the browser tools also gives us access to the CSS that's applied to an element. So I still have that Input element selected and I can scroll through the Styles Tab down here and it shows me each style rule from the CSS that applies to this element. And it shows every declaration from those rules and the declarations that have been superseded or aren't recognized are crossed out. So this is a super useful tool for understanding the current state of a web document in the browser window. The DOM is especially important when working with Ajax because we don't want to just receive remote data, we also want to use that data as part of the users interaction. And that often means changing what's shown in the browser window. So once we've received data back from an AJAX request, we'll use Dom properties and methods to change the contents of the browser window itself.

### **Select elements with vanilla JavaScript**

### Selecting transcript lines in this section will navigate to timestamp in the video

### - [Instructor] Before you can make changes to an element in the DOM, you first have to select it. The DOM interface supported by browsers includes several methods for selecting one or more elements, but a few methods are particularly common. querySelector takes a CSS selector as an argument and selects a single element. querySelectorAll is similar to querySelector except that if your CSS selector matches multiple elements, all of them are returned as a collection. There are also older methods such as getElementById, which selects the element with the ID value you pass. In general, querySelector and querySelectorAll are more versatile, but you may see getElementById in legacy code. Looking at our Explore California form, remember our first goal is to collect address, city, and state info, and then use that to retrieve and display zip info. So we'll need to reference those four form fields in the DOM. Now, when you use the method like querySelector, the browser needs to do some work to search the DOM, which is a tiny hit on performance. But if we know we want to reference an element more than once while our app is running, we want to save a reference to the element so we're only looking it up once. So in the HTML code for the contact form, I'll start by searching for address. And it takes me to the code for the address field with the city, state, and zip just below. And the easiest way to get references to these input elements will be with their ID values, and each one has one, address, city, state, and zip. So back in my JavaScript file, I'm going to create new constants just below my URL variables. So const addressField. Now, the querySelector method is a method of the document object, so to use it, we have to say document.querySelector. And then in the parens, I pass the CSS selector as a string. So that's going to be a quote, hash for an ID, address. So that gives me a reference the input box where users will type the address. And likewise, const cityField equals document.querySelector hash city. And the same for state. At this point, I could even just copy that, paste it, change city to state, and change city to state. Now sometimes we have legacy code that uses a library like jQuery to select elements. Back in the day, jQuery was a lot easier than using DOM manipulation with the older methods. But these days, if you're only using jQuery for DOM manipulation, then it's pretty straightforward to convert everything to vanilla JavaScript and remove jQuery as a dependency. So let's first write a jQuery statement to select the zip field. So const dollar zipField equals dollar paren quote hash zip. Notice it's really similar. All we're doing is replacing the dollar sign with document.querySelector and leaving out the dollar sign in the variable name. So I'll comment this out and leave it there for reference, and then I'll rewrite it using vanilla JavaScript. So const zipField equals document.querySelector paren quote hash zip, and now we have references saved for all four of the DOM elements that we'll be working with in the form.

### 

### 

### 

### **Request data in response to an event**

### Selecting transcript lines in this section will navigate to timestamp in the video

### - [Instructor] Where DOM manipulation really shines is when your code changes the view in response to an event whether that's a user event or a browser event. My plan for this form is to respond to a user leaving either the address, city or state field. Leaving a field triggers the blur event and when a user leaves one of those fields, my code is going to check if all three of them have content. And if they do, then I want to fire off a request to SmartyStreets, sending those three pieces of info and getting back data on that address including the nine-digit zip code. So the first thing I'm going to do is to define a function called checkCompletion that I'll call each time a user leaves one of those fields. So down here underneath the createRequest function, I'm going to say const checkCompletion. I can check the value of a form field using the value property of the referenced element. So I'm going to check if addressField.value and I want to see if that is anything other than empty. So I'm going to use bang equal equal and empty quotes, so as long as that value is not equal to an empty string or falsey, but I also want to check whether city field and state field are also empty. So I'm going to use an and logical operator here and on a new line cityField.value bang equal equal empty string and stateField.value bang equal equal empty string. So if all three of those conditions are true, that is to say if there is data in all three of those fields, then I actually want to build out a request to SmartyStreets using the information that the user entered. Now scrolling up to my smartyURL, I actually need to break this up because this is a URL that has hard coded street, city and state information but it gives me the basic structure for what that URL should look like. So I'm just going to duplicate my smartyURL. I'm going to comment out the original. So I have it there for reference in case I make a mistake. And then in the new one, I want to cut everything after the candidates value. So starting with the ampersand before street going all the way through MA and I'm just going to delete that. So this is my base URL for this request. And then back to my checkCompletion function, I'll create a local request URL variable that concatenates smartyURL with the values that we got. So I'm going to say const requestURL equals smartyURL plus the string ampersand street equals, which is the key that we use in the URL to reference the street value, plus addressField.value, just the value that the user entered in the form, plus the string ampersand city equal plus cityField.value plus the string ampersand state equals plus stateField.value. And to make this a little more readable, I'm going to break this up over multiple lines and indent. So we've got our three key value pairs each on a separate line. I'm going to put a semicolon at the end of my const statement. And then finally after that const, I just want to call createRequest and pass it this requestURL that I just created.

### 

### **Add an event listener**

### Selecting transcript lines in this section will navigate to timestamp in the video

### - [Instructor] Now I have a function that will actually check if the address field, city field and state field are all filled, and then submit my request. So the next thing I need to do to finish this up is to make sure that that function is called in response to the blur event on each of these three fields. Now, I want to call my check completion function in response to the blur event on three different elements. To do that with JavaScript I use the add event listener method, which takes two arguments, an event type as a string, and either a function name or an inline anonymous function. So first I want to add an event listener to the element I referenced with the address field variable. So addressField.addEventListener, and I'll pass the event name blur as a string, and then specify checkCompletion as the function. And I want to do the same thing for the other two fields, because if a user changes info in any of these fields, the query needs to be resubmitted. So I can do cityField.addEventListener, blur, checkCompletion, and stateField.addEventListener, blur, checkCompletion. Now I'm going to save my work. Go back to the HTML file and start up my live server. And in my form, I'm going to enter an address, and we'll just say 1600 Pennsylvania Avenue. And city, Washington, state, DC. Now, when I tab out, that is the blur event. And now I'm going to open up my console. And up here I have my park information coming in from my other request. And down here I can see for 1600 Pennsylvania Avenue, I have data coming back from Smarty Streets. So I've written code to access data from the DOM, and then use that to create an Ajax request.

### 

### 

### **Work with JSON data**

### Selecting transcript lines in this section will navigate to timestamp in the video

### - Once you have the response from your AJAX request, you have to dig into it and find the specific data you need. JavaScript Object Notation, or JSON, is by far the most common format used to exchange data on the modern web. JSON data is transmitted as a string, so the first thing you need to do is parse it into the data format it encodes. Although the word object is in the name, JSON data could parse out either as an object or as an array, which is technically just another type of object in JavaScript. To parse JSON, you use the parse method of the JSON object which is built into JavaScript. Now that we have a string of data back from our web service, to go any further in our code we'll need to parse that. So, in my updateUISuccess function I want to create a new variable called parsedData and I'm going to set that equal to JSON.parse and I'm going to grab the value of data, which is that JSON string that came in from my API call. So JSON, all caps, is the name of the object that we use to work with, JSON in JavaScript, and parse is a method that takes a JSON string and renders it as a JavaScript object or array. And then, I just need to replace data in my console.log statement with parsedData. And then, saving those changes and then starting up my live server from the html document. Going to open up my console and then to test this out we'll do 1600 Pennsylvania Avenue again in Washington DC Tab out of DC And now, notice that this first log statement is from the National Parks request. The second one is from my address request. So now, the response, both from the Park's array and from the SmartyStreets address request is no longer just a big string of JSON. Instead, I get an array containing an object for my address information, and I get an object for the parks information. Now when I open a data URL using the address bar, the response is displayed in the browser window, and I can work with that with a browser extension like JSON formatter. But modern browsers also make it easy to explore data in the console. So, for instance, I can click the triangle here to expand the next level in this array and start to explore the structure. So this array contains a single element with an index number of zero. And so I'll click the triangle for that element to open that up. And we have a nested data structure, which is super common in AJAX responses. An array containing an object or multiple objects, an object containing an array containing multiple objects, and so I can keep drilling down by simply clicking triangles. And so, for instance, we have a components property. And this is where I have each of the individual pieces of the address information broken out, each with a different property name. And so here I have the data I need. We got a zip code property, which is the five digit zip code. We've got a plus4 code property, which gives me those last four digits. So I put those two together with a dash between them, I'll have the nine digit zip code that I'm aiming for. Now one of the challenges in accessing data return from an AJAX request is building out the combination of square bracket notations and dot notation that's necessary to reference each specific piece of data that you want from the response. But this is another place that browser developer tools can really help. But notice that when I hover over the name of the data I want, the developer tools display the path to that specific piece of data, and I can even copy this. So if I right click, I can say copy property path. Now keep in mind, this is the path through this chunk of data and this chunk of data in my code has the name parsedData. That is the variable that's storing this reference. So going back to my JavaScript code, in my updateUISuccess, to create that zip code I can say let zip equal and then I want to reference parsedData, parsedData, and then I'm going to paste in that reference. So notice that Chrome adds in these extra quotes which I don't need. So I'm going to take those out because in the square brackets here all I need is the number zero for the index. So parsedData is an array. I want the first item in that array which is an object. Within that object, I want the components property. And within that components object I want the zip code property. So now I have those first five digits saved in the zip variable, and I can actually make that a const cause I'm not going to need to change it. And then one more constant I'm going to create plus four as a variable name. This is also going to be part of parsedData and I can go back to my browser plus four code I'm going to right click here, copy that property path and back in my editor I'm going to paste that in, once again, take out those quotes and then I'm going to create another console.log statement for the data that I'm actually going to want to stick in the webpage. And that's going to be the zip variable plus a string containing simply a dash, which is how you separate the two pieces of the zip code, plus the value of plus four. And so if I've done this right, I should get those two pieces of data logged to the console with a dash between them. So I'm going to save these changes, go back to my browser and one more time, we'll try 1600 Pennsylvania Avenue, try the city of Washington, state DC, tab out of that and there is my nine digit zip code. So notice I've got an error up here, and that's because I've just written very specific code for SmartyStreets while I still have a request going out to Parks. The easiest way to fix that right now is just going to be to comment out this call to the Parks URL and that will ensure that we don't get that error anymore. And so one more time, 1600 Pennsylvania Avenue, Washington, DC. Tab out of that, so I get my data back, and then I get that nine digit zip code. So now I have that data I need, and it happens in response to a user entering address city and state data.

### 

### 

### **Modify form values with vanilla JavaScript**

### Selecting transcript lines in this section will navigate to timestamp in the video

### - [Instructor] My form has event listeners and I have references to the data that I need from the response. So, at this point, my finals step for the form is to actually put the nine-digit zip code data into the form. To add data to a form field, I can use the same value property of the element reference that I used to get the values from the other input elements. I have a reference to the zip code input element saved with the variable name zip field. So, down in my update UI success function, I can comment out my console.log statements, and I'm just going to take out both of them because I don't really need them anymore, and instead, I can specify zipfield.value equals and then I can just copy the same expression that I console.logged previously, zip and a hyphen and plus four. And so, in this case, I'm saying instead of getting the value from the zip field, I am specifying a new value, which is then placed in that form field. So I'm going to save my changes. I'm going to go live with my HTML document. I'm going to open the console up and then, using my sample address, tabbing out of that final field and there we go. So the request is sent, the response is parsed, and that nine-digit zip code is added to the zip field, which is exactly what I was aiming to do. So far, I've been customizing this update UI success function for my form data, but code is currently written to use all the same functions for both my address request and my national park request. But both of those sets of data are going to require very different UI updates, so I need to make a few updates to my code to keep the main XHR code usable for any request while ensuring that any response gets a customer handler and that starts with update UI success, which is now customized just for the smarty streets request. So, I'll rename this function smarty update UI success and then I'll create a new success handler for parks data, so const park update UI success, which is a function that takes a data parameter and we'll simply console.log data. Right now, the update UI error function is the same for both, but eventually, I'll want to do some different things with the UI. So, I'll change that to smarty update UI error and then I'll make a copy of that and I'll change the name to park update UI error. Then in my response method function, I want to take a couple more parameters, succeed and fail. These are going to be the names of the correct success and failure functions for the requests that I'm making. And then, in my if/else structure, I'll update the function names to reference those succeed and fail parameters. So instead of specifically calling the function name, I will reference the function that was passed as a parameter and likewise for the fail. And then, in the create request function, I also want to add succeed and fail parameters and I need to add those as arguments in the response method function call here. So we'll just pass those right along. Now, the check completion function is custom for the smarty streets request, so I can customize the create request function call, passing along smarty update UI success and smarty update UI error. That needed to be a success. So now that should cascade through all my function calls and all the rest of the functions should work correctly and end up with the appropriate success and fail callbacks. And then for the national parks request, which is currently just a simple create request function call, I'll uncomment that statement and I'll add park update UI success and park update UI error as additional arguments. Then I'm going to save my work. Check in the browser. And so I have my park data automatically requested and logged, so that means the success handler's working just fine there and then testing out an address request from the form, that's working as well. So now, I have custom UI functions for the AJAX requests while still sharing the main code to create the request and check the response.

### 

### **Solution: Customize code for an Ajax proxy**

### **Selecting transcript lines in this section will navigate to timestamp in the video**

### **(electronic music) - [Instructor] To create my proxy end point for SmartyStreets, I need to do three things. In my index.js file, the first thing I need to do is duplicate the apiOptions variable. And then I need to rename the new copy, because I need a different copy for each of my end points. So I'm going to, instead of saying API, I'm going to say smartyOptions. And then to configure this, I need to change the target value so it's not for the National Park Service, but it's for SmartyStreets, and that's us-street.api.smartystreets.com. Then I need to come up with a different API path, and so instead of NPS, I'm going to use streets. Finally, I need an API key. So I know that in my SmartyStreets url, if I take a look at that code, the key in the query string that SmartyStreets is expecting is auth-id, so I know I can swap that in here as the key name, auth-id. And then I need to replace this with whatever name I choose in Heroku. So right now, I'm just going to decide to call this STREETS\_APIKEY, STREETS\_APIKEY, and I'm actually going to copy that, and then over in the dashboard for my app I'm going to go to settings, I'm going to reveal my config vars, I copied that new key name and I'm going to paste it in here, STREETS\_APIKEY, and then for the value I'm going to grab that key which I've saved, paste it in here, click the add button to save it. So now I have that SmartyStreets API key saved on the back end as a config var in Heroku. And so now, I have my query string key, I have a reference to the key value that's saved on Heroku, and so that takes care of smarty options. There's two more things I need to do. I need to actually create the proxy, and so I'm just going to copy this apiProxy variable, and again I need to rename it. So I'm going to call this streetsProxy, and I need to reference smartyOptions, so I'll paste that variable name in here. And then the last thing I need to do is duplicate this app.use statement. So in my smartyOptions I chose /streets as my endpoint, so down here I need to say /streets, and the name of my proxy is streetsProxy. So what this means is that when a request comes in, the endpoint /streets, it gets routed to streetsProxy, streetsProxy uses this framework to create a proxy using smartyOptions, and smartyOptions defines how requests that come in to that endpoint should be handled. So I'm going to save my changes, now I need to do one more thing, I need to actually change my front end code. And so over in app.js, first what I want to do is copy and paste my smartyUrl, keep it around for reference, and then I'm going to copy the beginning of that parksUrl up through the .com, and I'm going to swap that in through the .com here. Remember the endpoint that I chose was /streets, so I need to append that here. And then I don't need my auth-id or my API key anymore, so I can take off all of that, can take off the end of that quote, and then I don't need that ampersand because I've got a question mark here. So I'm just sending the key value pair candidates=10, my other key value pairs are getting appended once that form is completed, and then again the proxy will take care of adding the API key. So I'm saving that updated URL. Just for proof of concept I can also comment out my API key in the front end code. So I'm going to save that, and so switching to my terminal, I'm going to do a git status, and I have made some changes to index.js, so I'm going to do a git add ., git commit -m 'added streets endpoint', and then a git push heroku master. And again as long as I haven't made any syntax errors in my code, any logic errors, this should build correctly. Build did succeed, I see right there, which is always a good thing to see. Got my bash prompt back. So now I will go back to Visual Studio Code, go to contact.htm, and go live. And this time I need to actually submit an address, and I get my zip code back. So opening my console up, and let me go back and reload that... And I can see my street address request here. Again, I have a request to the proxy with all of my data there, and looking through all of this, the API key is nowhere to be seen, but I got my data back just like I did before, and so I have successfully extended my proxy server with a second endpoint, so that it can handle requests for both National Park Service data and SmartyStreets data.**

### 

### **Challenge: Modify the DOM**

### Selecting transcript lines in this section will navigate to timestamp in the video

### (techno upbeat music) - It's time for another challenge. Ready to try your hand at adding content to the DOM? To complete this challenge, write code that adds information for the first park in the return data to the DOM. You'll add the full park name as the content of the a element in the special section. You'll add the park url as its href value. Then you'll show the park description as the content of the p element. Update the function call to respond to the DOMContentLoaded event on the Window object. For an extra challenge, update your code to display data for a random park in the results rather than always displaying the first one. This challenge might take you around 20 minutes. If you don't have time for that right now, feel free to come back to it later on. When you're done, join me in the next video and I'll go over how I approached it.

### 

### **Solution: Modify the DOM**

### Selecting transcript lines in this section will navigate to timestamp in the video

### (upbeat electronic music) - [Narrator] To get the part data into the dom, the first thing I need is references to a couple more dom elements, and we've done this before. So, after parkSection, I'm going to create a new const called parkName, and that's going to be document.querySelector, and it's going to be #specials h2 a. And then we also need one for the description, so I'm going to call this parkDesc. That's going to be document.querySelector, and that's just #specials, space, p. And then, in the parkUpdateUISuccess function, I want to parse data variable, storing the parsed version of the data. So I will start with a const parsedData. And that's JSON.parse data. And then we'll change our console.log to parseData. And so, just saving this, starting my viewer, checking out the console, and after a moment, I should get a response. And there we go. And so now I can see the data, which is an object, and if I start digging in, I can see that object has this property that has a lot more data in it. So in the data property, I've got an array. And each one of these arrays stores an object. And each one of these objects contains information about a single park. This is a super common way that you'll see data come in from a lot of API's, although not all of them. An object, containing a property, that contains an array and each element in that array is itself an object containing data. So, the first thing I need is the full name of the park. And so we're just starting by working with the first element in the array, which is index zero. And so we've got a property called fullName. If I hover over that, there is the reference. So I'm going to copy that property path and back in my editor, I can use my parkName reference and set the text content. So, parkName.textContent equals and remember I'm working with parsedData dot and then I can paste in my reference. Again, I can take out those quotes that Chrome sticks in there. And that is a reference to the fullName value, which I'm now sticking into the element reference by the parkName variable as its text content. And then similarly, for the url, I'm going to go through here and find there is a url property, so again I'm going to right-click, copy that property path and back in my editor parkName.href, which is the attribute we use to set the target of a link, it's going to be parsedData dot and then paste in my reference, take out those quotes and I've got the url. And finally for the description, that is in the parkDesc variable. So it's parcDesc again dot textContent, cause we're just putting in some text and switching back over to our data, we have a description property. So I'm going to copy that property path and that's going to be parsedData dot and then take out those quotes. And we have the description. And so saving that, turning to the browser and now notice we have a heading with a link. When we hover over it, we can see that the pointer turns to a hand. At the bottom of the screen, we have the link, the remote url that that links to and then we've also got the description displayed here. So, we're great! Now the other thing I want to change is that I want this ajax request to happen only after the dom finishes loading. And that's the dom content loaded event on the window object. So, scrolling down, I'm going to comment out my existing createRequest function call. And instead I want to use an EventListener. And so again, instead of that being in response to something happening on a specific element, it is on the window object itself. I can add an EventListener. And the event is DOMContentLoaded and that's got to be DOM all caps. And then, that's going to be a function. And so, in this case, I can't simply call my createRequest function because I need to pass in some arguments. So I'm going to put in an in-line function and I'll just use an arrow function for that. So within this function, I can call createRequest and I can actually pass in arguments. So, parkUrl, parkUpdateUISuccess for the success handler and parkUpdateUIError for the error handler. And then saving that one more time. I'm going to check back in my browser. And my url variable is parksUrl not parkUrl and that's why that function call didn't work. Changing that and saving that again, switching back and there's my data. So everything works like it did before but now we're prioritizing other things before making this ajax request. And that's a pretty standard practice. Now for the bonus, I want to randomize which park has its info displayed. And so that's up here in the parkUpadateUISuccess function. And so, I'm going to create a random number. So I'm going to do const number. So I want to get the length of the array, so parseData.data.length, and I want to multiply that by a random number, so I can do math.random times parseData.data.length. And so that's going to give me a random number that is between zero and the length of the array. But, it's not necessarily going to be an integer and in fact, it probably won't. So I'm going to stick all of that in parense and I'm going to use that as an argument for Math.floor. And that rounds down to the next lowest integer. And so that neatly takes numbers based on the length of the array and rounds them to numbers that correspond to the index values, which start at zero rather than one. So now I have that random number and so then everywhere I've hard-coded as zero, I just need to replace that with number. And so that every time this function runs, I have a new number and a new choice of data from the results. So I've saved that, I'm going to go back to the browser, now I've got Muir Woods instead. And so every time I load, I should get a different park displayed here as well.

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

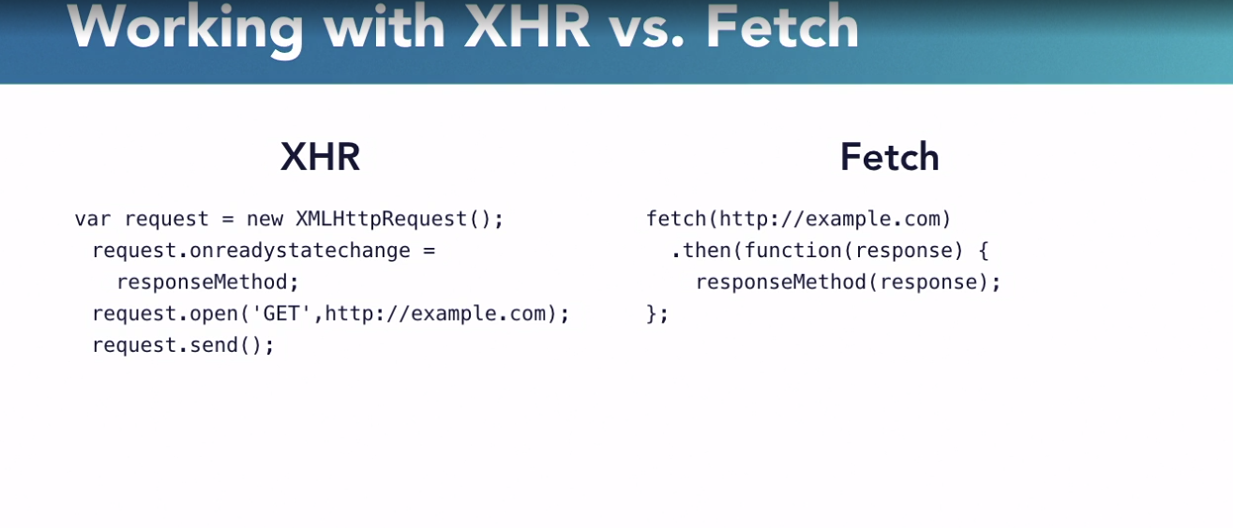
### 

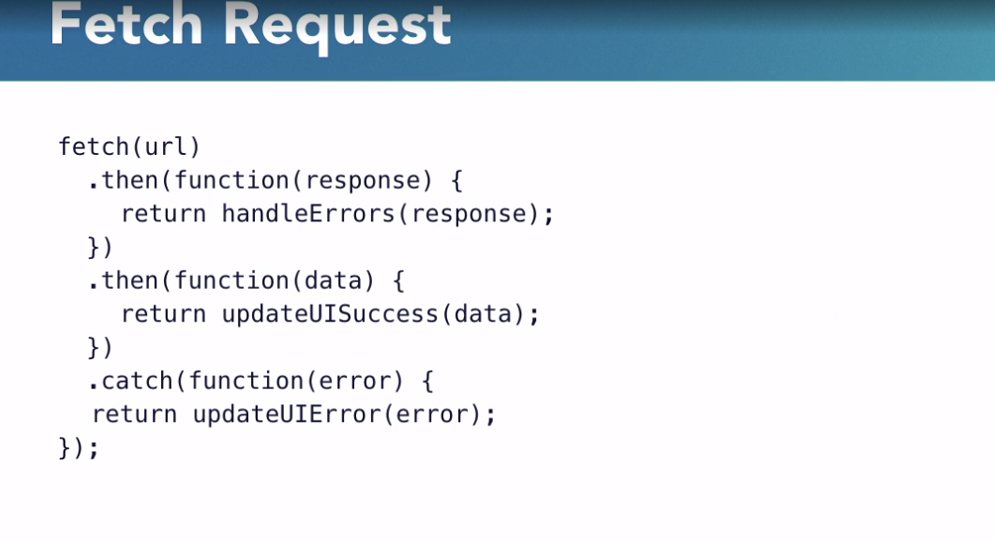
### 

### **Understand the Fetch API**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Libraries like jQuery include convenience methods that abstract a way the need to work with XHR objects directly. But until recently, a streamlined solution was missing from Vanilla JavaScript. Today, the Fetch API fills this void. One of the biggest challenges that XHR presents in all but the most basic use cases is what's known as, colloquially, as callback hell. This is the situation where you want a series of things to happen in sequence. The callback-based solution for this in JavaScript is generally to create a series of nested function calls each calling another so the result of the deepest nested function must be determined before the next function in the chain is executed and so on, up to the stop of the stack. Nested callbacks can be a challenge to read in code and can create a call stack that presents debugging challenges. The Fetch API does two things. First, it performs some of the basic AJAX functions without requiring you to code them. You pass a URL to the Fetch method and then opening the connection and sending the request are performed behind the scenes without the need for additional statements. You can pass the Fetch method an optional second argument containing parameters, but by default, it assumes basic settings, such as get, for the HTTP method. The result of the Fetch statement is returned in a standardized response object. Once the request has happened and that response object is generated, you can easily specify what happens in the next and in subsequent steps asynchronously using a syntax first introduced in another recent JavaScript feature, promises. Promises provide a way to easily specify code that should run in sequence. Rather than create nested callbacks, promises allow you to chain the results of running code using the keyword, then. Promises also pass along standardized objects and even include their own streamlined way to handle errors. You can use the promise constructor in your code in a wide variety of situations. The Fetch API uses the fetch keyword, but adapts the syntax of promises including the super useful then method. This allows us to simplify XHR code into something as straightforward as this. And in particular, while more complex XHR code often requires you to interpret nesting in the code, you can read Fetch code in a linear fashion from top to bottom and you can understand the order in which all of the code will be executed. The modern jQuery syntax for AJAX requests is also based on promises, so if you've seen that, you'll notice the similarities right off. In fact, you can rewrite jQuery dollar.get code by replacing dollar.get with Fetch, then replacing each instance of .done with .then, replacing the fail method with the catch method, and then adding an error-handling step. Each step in a Fetch chain takes a function as an argument and returns a value. For this reason, it's super common to see Fetch code written even more concisely using arrow functions. This code is identical to the Fetch chain shown previously, but uses fewer lines of code. Fetch is supported by all modern browsers. We can look on caniuse.com to see its current level of support. Can I use is a great resource for understanding the level of browser support for newer features in HTML, CSS, and JavaScript. So, in the search box, I can simply type in fetch and get a chart showing the current level of support for Fetch. I find it more useful to click the usage relative button which shows the weighted bars that indicate how much traffic comes from browsers that support this feature and those are indicated in green and in red, we see that percentage of traffic from browsers that are not supported. And so, we can see here, the biggest culprit is IE 11, the last version of Internet Explorer, which has still got over 2% global use in this chart and then little dribs and drabs of older IE, some older versions of Safari are still out there, both on Mac OS and iOS. And we have Opera Mini, which does have a significant user base. The Android browser is still out there a little bit. And so we have a summary up here that shows us that not quite 92% of traffic, by their reckoning, supports Fetch. That means that we have a hole of 8% if we don't support alternatives. So, as with a few other modern web development tools, we need to use a polyfill for full backward compatibility. So Fetch is pretty exciting and makes AJAX code easier to create and maintain using Vanilla JavaScript.





### **Create a fetch request**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] My form page for Explore California currently uses an XHR request to get remote data. I want to update that to fetch to make it easier to read and to maintain. To do this, I need to replace two functions, createRequest creates an XHR object and responseMethod works with the properties of the XHR object. I'll start by commenting these out. So there's responseMethod, and there's createRequest. And next, I want to recreate the createRequest function using fetch. So that'll be const createRequest = function takes url, succeed and fail as parameters, got to spell that right, C-C-E-E-D, and then I'm going to call fetch, pass it my URL, which is a parameter, and because I'm going to be chaining then to this, I'm going to do that on a separate line. So this is pretty common so that it's clear that .then is chained to the fetch that's above it. And then I need to pass in a function, and I'm going to use arrow function syntax. So, the parameter for this function is (response) and I'm going to start by simply console.log.(response). I'm using the same function name as before, so this will still get called for the National Parks API when the dom finishes loading, but all the function does for now is log the response. So we're taking a first step and we'll check it out. So I'm going to save those changes, start my live server and open my console. So I have the response logged to the console, and it's an object, so I can dig into it a little bit. And we have some details here, we have some header information telling us about the response, that it was okay. But looking at the body, not actually able to access the data there automatically. So fetch doesn't actually allow me to write code to access the value of that body, which is the response data. So instead, I need to use a method, .json, on the response object itself, and I have to return that value and pass it to another .then method. So this is part of the underlying architecture of Promises. And so back in my file, I'm still getting a response parameter, and then, since I want to return response.json, I'm just going to take advantage of the implicit return in an arrow function, and simply say response.json, and in this case .json is all lower case, and then, another .then, which is getting that data as a parameter, so I'm just going to use data as my parameter name, and this time, I will console.log value of that data parameter. I'll save that, and back in my browser, I now have the data being logged, which is actual park data. So with those two steps, first simply returning response.json, and then actually working with that data, I now have access to the data just like I did with HXR. And so now, instead of simply console.logging, I can actually return the succeed function call and pass it data, so I don't need those curly braces here anymore. I'm going to save that change, and back in my browser, so I have an error, and back in my browser, I am sending parsed data to the succeed method, but if I look in the succeed method for my parkUpdateUI, it is starting by parsing the data. But I already have parsed data. So in this case, I'm simply going to comment out parsedData here, and I'm going to change my parameter name to parsedData to match the data that I actually have coming in. So now all the rest of my code is expecting to work with parsed data, and that's what is actually being supplied as the parameter here. And now, back on my page, we can see that the data's been received, and we have the dom actually being manipulated, and that data for a random park being added to the dom. Recall that fetch is a feature that's supported by modern browsers, but not by older ones. And although we can use a tool like Babble to transpile a number of Yes Six features into Yes Five compatible code, that's not the case with fetch. Instead, we need to include a polyfill in our code, which is a separate set of JavaScript functions that check for the feature and recreate it as a method of the Window object if it's not already present. Github makes a popular and concise fetch polyfill. Now, as you look through the documentation here, one thing they make clear is that in order to use this, you need to also include a Promise polyfill, which reproduces that underlying architecture that fetch is based on. I've already included the files for both this fetch polyfill and the recommended Promise polyfill in the start files for this course. So, returning to my contact.html file, I need to go to bottom where the script element referencing app.js is located. Now, because my code uses the fetch keyword, I need to include the fetch polyfill in my .html document before my app.js file. And because the fetch polyfill uses Promises, I need to include the Promises polyfill before the fetch polyfill. If these are out of order they won't work in older browsers. So, before the app.js script element, I'm going to add another script element, src="\_scripts/promise.min.js" and then after that promise polyfill, another script element, srs="\_scripts/ and this time, fetch.umd.js" and now saving this, and going back to my browser, and everything still works just like it did. But if I were to transpile, meaning that, for instance, my const would be replaced with vars, and if I opened this in an older browser like Internet Explorer, now I'd be sure that that fetch request would work, no problem.

### **Customize a fetch request**

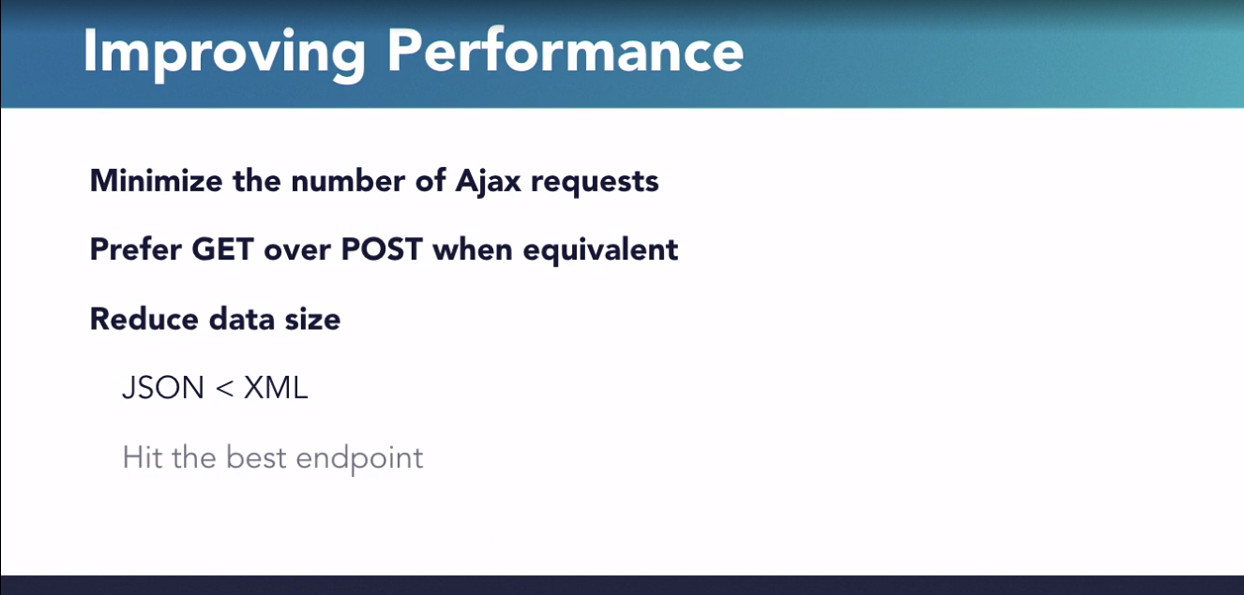
Selecting transcript lines in this section will navigate to timestamp in the video

- [Insructor] Fetch allows you to create a well-organized request with a minimum of code. But sometimes APIs require a more finely grained HTTP request than what fits in a URL. By far the most common instance of this, is APIs that require one or more HTTP headers as part of your request. Fetch supports a second argument after the URL, which is an initialization object containing one or more arguments. This can include the HTTP method, your credentials, or a number of other options, including headers. To include headers, you use the headers properties name within the init object. And then specify an object as its value. This object includes each header name and value as a key value pair. Our HTTP request to SmartyStreets has been working fine, but if you read through the documentation, you'll notice that SmartyStreets prefers each request to include two specific HTTP headers. If they decide to enforce this down the line, my request might result in an error in the future, so I want to add these headers to my request, just to square everything away as neatly as I can. At the top of my app.js file, I'm going to create a new constant called smartyInit. This will store the object will pass as the second parameter to Fetch, so I'm going to create an object data type. Go ahead and close that now. The only custom option I need is headers, so I'll add a headers property, then as its value, I need to specify an object with each of those two specified headers as properties, and I can just copy those straight from the website to ensure that I don't make any typing mistakes. So there's the first one. Now notice that because this property name has a hyphen in it, I actually need to treat that as a string here. And then of course, I need to treat the value as a string as well. So then the other header is host. So again, pasting that in. This time I only need to quote that value. I always like to end with a comma, and there we go. So now, I have these two headers saved in a variable, and then I can use that variable to customize my Fetch request. To implement this, first I'll go down to create request, and I'll add a fourth parameter, which will be init. Then I'm going to update my Fetch invocation to add that parameter, so now I'm passing the URL and init parameters when I call Fetch. Then when I move down to check completion, I need to add my variable as the fourth argument in the create request function call. So after smartyUpdateUIError, I want to specify smartyInit. So I'll save my work, fire up my live server, open my console. And notice that my park request still works because it's got a null value for that init object, and it doesn't make a difference in the request. But I will go ahead and try out a Smartystreets request And it looks like I have an error, and that's pointing me to line 23 in my code. And that's where I am trying to, again, like I did down in my parkUpdateUISuccess, I'm trying to treat JSON as something I need to parse into JSON. So I'm going to comment out that parse data declaration, and I'm going to change my parameter name to parseData, which is used throughout the function and that should get all the data that I'm providing aligned with the data that's expected in this function. So going back to my page, let's try this request one more time. So there's the zip code. So I'm all good, I have my Fetch request customized to meet the criteria laid out in the API documentation by using that init object.

### **Structure Ajax requests for performance**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Narrator] No matter whether you're building a native XHR object or using Fetch, the way you structure an Ajax request impacts its performance. There are a few important considerations when creating a request that can help ensure that your app is as responsive as possible. First off, you should aim to minimize the number of Ajax requests in your app. While it may be tempting to limit yourself to retrieving only the data you need in a particular part of your code, it's important to balance that against the performance hit you take for each Ajax request. When you know you'll need additional data from the same source, or even if it's very likely, then you should go ahead and get everything in a single request rather than repeating requests for smaller bites of data. Second, GET requests are faster that POST requests. When you have a choice, use GET. Each of these verbs has its own specific use case: GET for retrieving data, and POST for creating a resource. You may find that you're hitting a single endpoint using the same method for different requests even when you're simply retrieving data. You can improve the performance of your Ajax requests by identifying these unneeded POST requests and recoding them as GET. Finally, you can reduce the impact of your Ajax requests on performance by reducing the size of the data you're sending and receiving. One way to do this is to use the leanest data format that your app and your endpoint can deal with. Obviously, plain text is free of structural characters and using plain text enables you to minimize the size of the data stream. However, most requests and responses need some sort of structured data. Among popular formats in use, JSon has the lowest overhead. XML has much higher overhead because of the high ratio of structuring characters, tags, to data. Of course, if XML is required for a specific endpoint or application, there's not much you can do. If a data source offers a choice, JSon is highly preferable over XML. If your Ajax requests use XML because your app is written to accept it, it may be useful to perform a cost-benefit analysis of rewriting your app to accept a data format with lower overhead. Another way to reduce data size is to make sure you're hitting the right endpoint. If you're requesting user profile information, for instance, but the endpoint you're hitting returns full user information, including an array of recent activity, it's worth looking for a different endpoint that returns a narrower slice containing less data that you don't need. App development is a constant tug of war between adding features and increasing performance. Following these performance guidelines in your Ajax requests can help keep them from adding bloat to your apps and keep your apps responsive.



### **View requests and responses in the browser**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] When you're working with Ajax requests, it can be useful to examine the actual HTTP request and response headers that are being sent when your code is executed. Modern browser developer tools let you record and review network activity including each HTTP request for the current page. Both Chrome and Firefox dev tools include a network tab. I'm using Chrome so I'll open the dev tools and then I'm going to click network and notice that this list is empty. Browsers want to avoid doing a bunch of work in the background and they want to avoid saving information that takes up memory but that may never be viewed. So dev tools generally don't log network requests until you open the tab. With the tab open, I can go back and reload the page and now I see all the requests logged for this page including the CSS and JS files, fonts, images, and down here is my Ajax request for parks info. So on my screen this is third from the bottom, starts with the word parks followed by a question mark. And when I hover over a request, I see any additional details. If it's cut off, I see the full path. And I can click any request in the name column to see the details. So I'll click that National Parks URL in the list and on the headers tab I have a summary which tells me the URL that I hit and the result which is a 200. Below that is a list of response headers. These are the key value pairs that my browser received from the National Parks Service web service. And then further down, I have a list of request headers. This is the data that my browser sent to the National Parks Service web service when making a request. At the very bottom, I have a summary of my query string parameters which are the key value pairs I encoded into the URL for the request I made. So I've got my API key and I've got the state code. On the preview tab, I can view the response body which is the data I received back in the response, a parsed version. And this is the data that I've already been working with in my code. But if I was having a problem with getting the right data back, this might be a quicker way to examine that than doing a bunch of console.logging. I can watch this in action for my other API. To do that, I'm going to change my view of dev tools so it's docked at the bottom of my screen and I'll drag that up just a little bit and I want to turn off this overview so I just have a list of headers over here. I'm going to scroll to that bottom of that list and then up here in my webpage I'm going to go ahead and generate a request for an address. And as soon as I press Tab, I've got not one but two requests and that just has to do with the way that the remote server is configured for SmartyStreets. And so it's the second request that should actually have the data and I've got that preview tab selected so here's the data I got back and again I've got headers for this. So now that I have this tab open, any new request gets logged here, a tool when you're building and working with Ajax requests to see exactly what's going on behind the scenes.

### **Work with authorization errors**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] When errors happen in an API request, sometimes half the battle is understanding what went wrong. Often the error code and message returned from a failed request is enough to point you in the right direction. But once you have a general idea of the issue, it's also important to understand how to fix it. One common error you might encounter is an Authorization Error. Which corresponds with a 401 HTTP code. When you request results in this error, it generally has to do with your API key. A few things can go wrong with an API key. The first place to check is to ensure that the key in your code matches the credential issued by the web service. A copy and paste error or even erroneously typing or deleting a single character in the wrong place can cause issues here. If the key and your code matches the provided string, another thing to check is that you're using the correct key. Some providers such as SmartyStreets provide multiple authentication strings and it's up to you to read the documentation and understand which needs to be used in which scenario. Ensure you're using the right string. Also check that you've completed all necessary configuration. For instance, SmartyStreets requires you to associate an app address with your key. Even using the correct key in your app will result in a 401 error unless you also associate it with your address. Right now my code's working really well. But let's go ahead and introduce an error and see what that looks like in the console. So I'm going to start with an Authentication Error here. Again, I'm going to copy and paste my Smarty URL and I'm going to take out a couple digits of that auth ID. Saving that then starting up my HTML file. Opening up my console. And let me just switch to a separate window there. And now I'm going to run a zip code request. And so in my console I can see that the error I got out of my fetch is a 401. The error I logged is a 401. And I know that's because I went in and broke that API key. So again, getting that API key right, that auth ID, whatever it's called, is crucial to make sure that your requests work. And I'm just going to fix my code. Uncomment the correct URL. Go back to my page and try one more time to make a request. And everything's fixed now so I'm good. And so making sure that your API key is correct. Making sure that it matches the one you've been provided. And making sure that it's correctly configured are all steps you can take to work with a 401 error.

### **Work with malformed requests**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Narrator] As you're first building out your request code, it's not uncommon to receive errors due to malformed requests. This type of error results from an error in either the data you're sending, or the syntax of your request. A few common issues are often at the root of malformed requests. One of these is simply an invalid endpoint. That can include forgetting HTTP or HTTPS at the start of the URL, or not providing the full path for the endpoint, or even using an outdated endpoint for which support has ended. Another place to look is in the separator characters in the query section of the URL. Forgetting the question mark at the start of the query string, or omitting the ampersand between key value pairs, or the equal sign separating a key from a value can all result in un-parsable data, or in data that's parsed by the endpoint in a different way than you intended. Finally it's important to check that the keys and values you're sending in the query string map to what's expected by the endpoint. That may be using an incorrect key, for instance address one rather than street one, or sending data that hasn't been parsed correctly before being added to the query string. The developer tools can be particularly useful in troubleshooting this type of error. The network tab lets you view the malformed request, including the parsed query parameters. So let's test this out with our code I'm going to go to the parks URL and save a good copy. And comment... And then I'm going to start by taking out the question mark that separates the endpoint from the first parameter, which is stateCode. So I'm going to save that change, and go live with my page in the browser. And then in my developer tools, I've got a 403 forbidden error, and so lets dig in on that a little bit. We want to go to the network tab, I'm going to move my tools to the bottom of the screen, to make them easier to view. And I'm going to go ahead and reload this page, to get all of these requests logged. So, in red, I see this request that didn't work, 403. And that looks like my national parks service request, so I'm going to click that, there's the results. Here are the response headers, the request headers, and I don't actually have any parsed query parameters down here. I look at the preview, there's an error code that says the API key is missing. I can dig in a little bit on that, and it says it doesn't have an API key. So going back to my code, that can be a little hard to determine what's up if I didn't know what I'd done. Cause clearly I did provide the API key, but because I never actually ended, never actually made a distinction between the endpoint and the query string, it thinks this is all part of the end point, and that I haven't actually provided that query parameter. So putting that question mark back in is going to solve that issue. Now what happens if I take the equals sign out? Now instead of specifying a value for the state code key, and saying that I want values for California, it's going to get this blob of text so let's see how it deals with that. So I'm going to save that, back in my browser, I do have a park showing up. So let's look at our parks request here, which is the bottom of my list, here I can see already that missing equals sign. And so, I got a bunch more parks returned than I usually do. This is in the preview tab. If I go back to headers, everything is okay. If I scroll to the bottom, my query string parameters. I have a key called stateCodeca with no value, rather than stateCode key with a value of ca. And so it looks like the API has just ignored that all together, and is giving me information on every park there is, which is not what I was looking for. And so sticking that equals sign back in there is going to fix that. Finally let's see what happens if I leave out that ampersand. So I'll save that change, go back to the browser, and it looks like I have a 403 error again at the bottom of my list, so I'm going to take a look at that. I have a forbidden, and I have a single query string here stateCode, with this really long value with the API key. My guess is that the response is going to say I don't actually have an API key. Right, because it wasn't provided in a way that was parsable. So all of these characters are super important, and if you miss just one, it can cause all sorts of havoc with the data you get back. And so I want to make sure that I get my code back to a working state. So I'm going to delete that parks URL that I was messing up. Going to un-comment the good one that I saved, and I'm going to save those changes, then back in my browser I do get park back, and when I look at the data, I get 33 responses, so, looks like we're all good here, and the code's working again. So one of the first places to look when you're getting an error when you're first constructing your API request is in that URL that you're providing to make sure that everything is formatted correctly.

### **Work with errors in Fetch**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Fetch is both easier to write and easier for other developers to read than XHR. However, because the underlying promises pattern is a little different than the way JavaScript code has traditionally been written, Fetch has a few gotchas that can result in errors, especially when you're using Fetch for the first time. It's super common to try to access the data too soon. Remember that the first .then method needs to return the payload, and then the second .then method can work with the data. The architecture of Fetch keeps the data off limits to your code in the first .then method. Another place where Fetch can trip you up is forgotten return statements. Every chained .then method needs to use return in order to pass data to the next method. You can instead manipulate variables in the parent scope, but it's better practice to simply pass the data along the chain. Finally, sometimes you'll see no result from a Fetch request, including no errors logged in the consol, even if you've included a .catch method. Remember that Fetch considers a request to have failed, only if there's something like a network error that prevents it from being sent, or a response from being received. In all other cases, it's important for you to provide the error handling logic yourself. Otherwise, responses other than 200 may result in no data to be rendered, but will not throw an error. Let's check this out ourselves. I'm here in the create request function and I'm going to start by just commenting out this first then. This is the first then that takes that response object, parses out the response .json and passes that along to the next .then. Let's just check out what happens if we skip that step and try to parse that response object with the succeed handler. Going live in my browser, and I'm opening up my developer tools. I have a response object logged here in my success handler, and again, when I look at the body property, it says invoke property getter, which is the browser's way of reminding me I need to use response .json in order to get that data out. I've got information about that actual response, but then when I try and use it in my success code, it's just telling me that it doesn't actually have access to the data. Going back to that file, I will uncomment that. That .then method is calling the handle errors function and passing along the response object. What happens if I take out the error handling code here? We're simply just returning response .json. Then, if I go up to my parks URL and I mess something up. I'm going to make a copy of that, comment out the original, and then we'll try something like taking out the question mark, which should result in the service not recognizing that we've actually sent along an API key. Checking back in the browser, I get a 403 error, but notice that I get an error object logged in my success handler, and my success handler is trying to read the length property. Even though the browser has gotten an error message from the remote server, my code has not actually thrown an error, and the Fetch method has actually continued to send that data along the chain. Leaving that error in my URL, I'm going to scroll back down to the handle errors function and I'm going to put that error handling content back in. Again, I'm just checking to see if there's an okay in that response, and that'll only happen if I have a response in the 200 range. With that saved, going back to my browser, and now, I've still got that error being thrown by the browser and I've also got my code recognizing there's an error and throwing that exception, having that exception handled and having the success callback skipped. That's exactly what I want my error handling code to do, and that's exactly why we need to write our own error handling code. Finally, I'm going to go back to my URL, take out the one that I broke, uncomment the good one and then go back to the browser and make sure, there's the data. There's the DOM manipulation. I have everything back to a working state, and now I have my Fetch request set up in such a way that I can accommodate issues that come up.



### **Provide error fallbacks**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] You can make your Ajax error handling code more useful by accounting for specific cases that are significant to your app and ensuring that your code can deal appropriately with each of them. Many types of Ajax errors can result in the requested data not being available to incorporate into the DOM. In this case, it's important to consider how you can meet the needs of your end users without remote data. For the Explore California form, the zip code data is just an extra, if the zip input box doesn't get autocompleted, users can still use the form and just fill it out manually, so there's no extra fallback required there. Now, the National Park section is already hidden by default and if the data isn't returned for whatever reason, that section of the page remains hidden, so we don't exactly have an issue. However, we could improve the user experience by including some default data that's displayed if we don't get a response. For instance, we could include the data for one of the parks in the code for the page and setup our code to use that data if we can't get live data back from the National Park service. This would require someone to check, maybe, once or twice a year to ensure that the data is up-to-date, but it would make this feature of our page a little more bulletproof. So, the first thing I want to do, in my error handler, right now I am throwing some text and that's getting handled by logging into the console, but JavaScript actually lets me create an error object that will capture a little more information. And so, here on line 66, all I'm going to do is update this a little bit to throw a new error, and that's capital E, and then I'm going to add extra parens to make sure that it's treating my concatenated string as a single string. So then, this text will be included with a new error object that's generated and that object will be logged to the console. So I'm going to go up to my parksUrl, I'm going to copy, duplicate, comment out the original and then, break my API key by deleting a couple characters, I'll save that and then go back to my browser, open the console. So, I have a 403 Forbidden error, and notice here, from line 45, this is where I consoled out logged in my error handling function. But in this case, instead of just getting 403 Forbidden, which was the text I was passing before, I have the indication that this is an error object and I also have a stack trace showing what was in process when this error happened. So that can be a little more useful to me, as a developer, when I actually run into errors. Now, I'm going to fix my API key, I'm going to delete the broken one, put back the good one and then, I'm getting my request working again. Now, what I'd like to do is actually grab the data for, say, the first park that comes through in my request, copy that into my code and then, setup some code that uses that as a default if I don't get data back from a request when the page loads. So, opening this up, what I'm going to do here is right-click zero in the data array, and we have this option Store as global variable. And so, this is going to grab the contents of element zero in the data array, and if I scroll to the bottom, the browser has created a new variable called temp1 that just stores this single entry in my array. And so, now I can use the Copy command at the console, I can pass it the name of that variable, temp1, and that actually copies that data to the clipboard. So now, switching back over to my code, I'm going to create, right after the parksUrl, a new variable called parksFallback. And, as the value, I'm simply going to paste in the code I just copied, that is an object, going to put a semicolon at the end. Now, this is the full data for one park and out of this data, I only need three properties, I need the description, the full name and the URL. So, I'm going to delete all the properties that I will never need, to trim this up a little bit. So, we've got description, down here is full name, so I'll delete everything before that, right after that is URL, and then I can take out the rest before the closing curly brace. So now, I have an object containing information that I need for just one park and that's saved in my code. And so, to make use of that, in my parkUpdateUIError, I have my error being logged to the console, but I also then want to do a little DOM manipulation, and, in particular, up here from parkUpdateUISuccess, I just need a few lines of code. This is the last, let's see, one, two, three, four, five statements, I'm going to copy, I'm going to stick those down here before the end of parkUpdateUIError and then, I need to update these references a little bit. So, all of these should refer to that new variable I just created, called parksFallback, so instead of parsedData.data[number], it's just going to be parksFallback, same for this one and same for this one, and these two should be exactly the same. And so, if I save this, then I scroll up and once again, I need to break my parksUrl, so again, I'm going to duplicate that, I'm going to make one of them a comment, I'm going to break that API key by deleting a couple characters and then, switching over to my browser, I have my error object logged, but looking in the browser, there's the information for Alcatraz. So, my error handling code was able to go and recognize I did not get a successful response, but then, it was able to use the data that I'd saved in the file as a backup to display something to enhance the page. So, we've implemented a couple useful features in our code to deal with potential errors in a graceful way. And then, I just need to make sure I go back to my code and fix my URL, make sure everything works, I'm going to take out that broken parksUrl, I'm going to uncomment the original one, save that change, and then, going back to my browser, and now, I've got the data back, I've even got a different park being displayed.

### **Provide error fallbacks**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] You can make your Ajax error handling code more useful by accounting for specific cases that are significant to your app and ensuring that your code can deal appropriately with each of them. Many types of Ajax errors can result in the requested data not being available to incorporate into the DOM. In this case, it's important to consider how you can meet the needs of your end users without remote data. For the Explore California form, the zip code data is just an extra, if the zip input box doesn't get autocompleted, users can still use the form and just fill it out manually, so there's no extra fallback required there. Now, the National Park section is already hidden by default and if the data isn't returned for whatever reason, that section of the page remains hidden, so we don't exactly have an issue. However, we could improve the user experience by including some default data that's displayed if we don't get a response. For instance, we could include the data for one of the parks in the code for the page and setup our code to use that data if we can't get live data back from the National Park service. This would require someone to check, maybe, once or twice a year to ensure that the data is up-to-date, but it would make this feature of our page a little more bulletproof. So, the first thing I want to do, in my error handler, right now I am throwing some text and that's getting handled by logging into the console, but JavaScript actually lets me create an error object that will capture a little more information. And so, here on line 66, all I'm going to do is update this a little bit to throw a new error, and that's capital E, and then I'm going to add extra parens to make sure that it's treating my concatenated string as a single string. So then, this text will be included with a new error object that's generated and that object will be logged to the console. So I'm going to go up to my parksUrl, I'm going to copy, duplicate, comment out the original and then, break my API key by deleting a couple characters, I'll save that and then go back to my browser, open the console. So, I have a 403 Forbidden error, and notice here, from line 45, this is where I consoled out logged in my error handling function. But in this case, instead of just getting 403 Forbidden, which was the text I was passing before, I have the indication that this is an error object and I also have a stack trace showing what was in process when this error happened. So that can be a little more useful to me, as a developer, when I actually run into errors. Now, I'm going to fix my API key, I'm going to delete the broken one, put back the good one and then, I'm getting my request working again. Now, what I'd like to do is actually grab the data for, say, the first park that comes through in my request, copy that into my code and then, setup some code that uses that as a default if I don't get data back from a request when the page loads. So, opening this up, what I'm going to do here is right-click zero in the data array, and we have this option Store as global variable. And so, this is going to grab the contents of element zero in the data array, and if I scroll to the bottom, the browser has created a new variable called temp1 that just stores this single entry in my array. And so, now I can use the Copy command at the console, I can pass it the name of that variable, temp1, and that actually copies that data to the clipboard. So now, switching back over to my code, I'm going to create, right after the parksUrl, a new variable called parksFallback. And, as the value, I'm simply going to paste in the code I just copied, that is an object, going to put a semicolon at the end. Now, this is the full data for one park and out of this data, I only need three properties, I need the description, the full name and the URL. So, I'm going to delete all the properties that I will never need, to trim this up a little bit. So, we've got description, down here is full name, so I'll delete everything before that, right after that is URL, and then I can take out the rest before the closing curly brace. So now, I have an object containing information that I need for just one park and that's saved in my code. And so, to make use of that, in my parkUpdateUIError, I have my error being logged to the console, but I also then want to do a little DOM manipulation, and, in particular, up here from parkUpdateUISuccess, I just need a few lines of code. This is the last, let's see, one, two, three, four, five statements, I'm going to copy, I'm going to stick those down here before the end of parkUpdateUIError and then, I need to update these references a little bit. So, all of these should refer to that new variable I just created, called parksFallback, so instead of parsedData.data[number], it's just going to be parksFallback, same for this one and same for this one, and these two should be exactly the same. And so, if I save this, then I scroll up and once again, I need to break my parksUrl, so again, I'm going to duplicate that, I'm going to make one of them a comment, I'm going to break that API key by deleting a couple characters and then, switching over to my browser, I have my error object logged, but looking in the browser, there's the information for Alcatraz. So, my error handling code was able to go and recognize I did not get a successful response, but then, it was able to use the data that I'd saved in the file as a backup to display something to enhance the page. So, we've implemented a couple useful features in our code to deal with potential errors in a graceful way. And then, I just need to make sure I go back to my code and fix my URL, make sure everything works, I'm going to take out that broken parksUrl, I'm going to uncomment the original one, save that change, and then, going back to my browser, and now, I've got the data back, I've even got a different park being displayed.

### **Challenge: Fix an Ajax request**

Selecting transcript lines in this section will navigate to timestamp in the video

(peppy music) - [Instructor] It's time for a challenge. Ready to try your hand at debugging an Ajax request? For this challenge, you need to use the begin folder for this video, which contains a broken API request for SmartyStreets. You'll need to troubleshoot and fix the errors. I've separated out the API key portion of the URL into its own variable to make it easier for you to swap in your own key. Be sure to update this value with your own key, or the request won't work. You'll know you're done when the zip code functionality in the form works. This challenge could take around 10 minutes. When you're done, join me in the next video, and I'll go over how I approached it.

### **Solution: Fix an Ajax request**

Selecting transcript lines in this section will navigate to timestamp in the video

(upbeat music) - [Narrator] So the API requests in this file for Smarty Streets is broken. Now, the first thing we need to do is update the API Key. So again, because this was all in a single URL, I broke that out into a variable. Created this smartyAPIKey variable that's going to store the API Key. So you want to replace that value, I went ahead and copied my API KEY, and I'm just going to paste it in there. And so now, that API Key is concatenated with the rest of that URL, to create my request URL. So, I'm going to save that, I'm going to go to my HTML file and we'll just start by going live and checking out what things look like. So, I'm going to run a test query on Smarty Streets, and in my console. So, right here in the console I have an error, and it's specifically an error that says name not resolved. Now if I look over in the Network tab, I go back and I reload my page, and I try this again, I have a failed request, and I can see there's a problem with the auth id. But I'm not even getting an error message here, so that tells me that there's something going on a little more upstream. And this name not resolved suggests that there's an issue with the endpoint itself. So, going back to my code, I want to check out the URL here. And so normally what I would do in a case like this is actually go in and look at the documentation, maybe even copy that URL from the documentation. One thing I can take advantage of here is that we have our init variable which gives these headers, that Smarty Streets wants us to use. And we have a Host value here, that shows the host, the endpoint that we're supposed to be hitting. So the endpoint is us-street.api.smartystreets.com. Up here, we've got us-streets, plural, .api.smartystreets.com. So this is the sort of thing that can happen, if you're typing something from memory, or if you're just getting ahead of yourself, confusing things while you're typing. So I'm going to go in here and take this s out, so that it matches the host value down here. I'm going to save that, go back to my browser, and I'm going to try another request. And in the console, this time I do not get an issue with the URL itself, I get a 401 error which means that my request was actually submitted to a service, and got a response. So, that's progress. Now I'm going to go over to the Network tab again, and notice I've got my two requests, the first one went through fine. We always have that two requests with Smarty Streets. The second gave me a 401. If I look at my response it says unauthorized. So that's the text that's corresponding with this 401. And so again scrolling down, I can look at those query string parameters. And just looking at those here I can see an issue. Cause I've got my state, my city, my street, just like I entered in the form. But my auth-id has candidates equals 10 stuck at the end, and that's clearly, not part of my auth-id. So, back in my code, I have this question mark auth-id equals then the value. And after each value, before the next key, I'm supposed to have an ampersand. So, I'm going to stick that back into my query string, go back to my browser, run my query again, and there's my zip code. If I switch over and look at my console, everything looks great. Got my park API request log there, and I've got a status of 200 on both of my responses from SmartyStreets. So, again when we're working with errors in an API request, they can come from a whole bunch of different directions. And it's really important to just start with one issue, chase it down, go on to the next, and then finally get everything ironed out, and get your API request working.

### **Why are proxy servers necessary?**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Building out a basic Ajax request in front end code is a common practice when developing an app. It's a quick and easy process for trying out the request, writing code to work with the response and building out your app around that. But in many cases before you deploy an app for public use, it's necessary to set up a proxy server to handle your Ajax requests. So why are proxy servers necessary? A lot of APIs require the use of credentials like API keys. Without including a key in your request, the server returns a 401 error indicating that your app isn't authorized to access the requested data. To get authorization, you include your API key as part of your request either in the query string or as a header. This lets the web service associate your request with your account. As long as you're still below any quota or limit on your account, the service sends an OK response with the data included. The problem with this setup is that when you include your API key in the front end code, an end user who understands the basics of HTTP and knows how to open developer tools can access your API key and use it for their own apps. This means their requests will count against your quota potentially costing you money if you're paying for the web service. It also means that if other users misuse your keys, for instance incorporating the data in an app that's outside the terms of service, the service would look to you as the owner of that credential as the responsible party. At a minimum, your credential would be revoked and your account might even be shut down. And this is where proxies come into the picture. A proxy is simply a web server that receives the request, makes some modification to it and then forwards it along. You can set up and configure your own proxy server to store your API credentials and then append them to requests that it receives. You can then rewrite the Ajax requests in your front end code to target your proxy server which will add your credentials and redirect a request to the target API. The proxy also then receives the response from the API and forwards it back to the client. Throughout this process, your private credentials never pass through the browser so users are unable to view and possibly misuse them. Proxies are super common and an important tool for web apps that incorporate Ajax. Note that not all APIs require the use of a proxy. Instead, some services require you to associate your credentials with one or more domains. The service then checks each request that includes your API key to ensure that it originated at your domain and fulfills your request when the API and the origin match. However, the service rejects requests from other origins. Users of your app do have access to the API key in their browsers if they dig into the code. However, they would be unable to reuse your key in their own projects because the origin of the request wouldn't match the authorized list you associated with your account.

### **Tools: Git, GitHub, and Heroku**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] I'm going to create and configure an HX proxy for the Explore California contact page. I'm going to use Heroku to host my proxy, which is a popular service for hosting apps in the cloud. This process is going to draw on a few more advanced developer skills, which I'll assume you have some experience with. I'll be cloning a repo from GitHub. I'll also be using Git at the command line to perform some basic commands. And finally, I'll be working with the Heroku Dashboard. I'll be walking through each of these but if you have questions I'm not answering here, I encourage you to find another course in the library that focuses on that topic for a deeper dive. You'll need to have the Git command line utility installed on your machine, as well as a free account set up with heroku.com. So if you need to take care of any of that, go ahead and do it now.

### **Set up an app on Heroku**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Heroku allows you to create and deploy apps which it hosts, builds, and executes. A free account on Heroku isn't appropriate for an app that needs to be available 24-7, but it will work fine for our purposes. To get started, I'm already logged into Heroku and on my dashboard, I'm going to create a new app. To do that I'm going to click the New button, and I want an app rather than a pipeline so I click Create New App. Now any app you create on Heroku has to have a unique name throughout all the apps that are on Heroku already. So anything obvious like Ajax Proxy's probably not going to be available. I'm going to try out ec for explore California followed by ajaxproxy. And it tells me it is available, that's great, I'm in the US so I'm going to use that and I'm going to create that. So now I have an app setup and I have a number of different tabs here that I can select to see different options and settings for this app. So the next thing I need to do is actually store my National Park Service API credentials in Heroku cause the whole point here is to get those out of the front end code and in to the back end code. So to do that, I'm going to create what's called a config var which is short for configuration variable. This let's me store a value and give it a name just like a variable in any other Java script code. And then I can reference that name in my apps. So I'll go to the Settings tab, and there's this button here Reveal Config Vars. So those are hidden by default in case there's somebody just happens to be walking behind you when you open the screen, you can make sure there's no one around who shouldn't see your actual config var values before you actually reveal them. So I'm going to click that button, has no config vars yet and we've got a key and a value box here. So I'm going to click in the Key box. I'm going to name my new variable NPS\_APIKEY. So it's really common to name these in all caps because they are essentially constants and then in the value box I've copied my National Park Service API key to the clipboard and I'm just going to paste that in here. And then click the Add button, and now that's saved. So now my back end code can reference this variable name and this API key value will get swapped into my code. And this has the extra advantage that if I ever change or replace my API key, I don't have to dig into my code to do that. I can come right here to my dashboard and just swap it in.

### **Deploy and test a proxy**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Narrator] Now that my proxy code is configured for my needs. I need to Heroku, which will build, execute and host my proxy. So first off, I want to open up the dashboard for my app. So if you're not already there, click your app in your list of apps and then, I'm going to go over and choose deploy. And the deploy section actually gives me step by step instructions for how to deploy using the command line tools. So the first step, if you haven't already done it, is to download and install those command line tools. There is a link right there. I already have those installed, but if you don't, then go ahead and do that now. Now, I'm pretty sure I'm already logged in, but let me check. So that's a Heroku log in, and so back in my terminal, Heroku log in. And so I'm going to press a key. It's going to launch my browser, and since I'm already logged in to Heroku in my browser. All I have to do is click the log in button and that's going to make sure that I'm logged in my command line as well. And then I need to switch to the folder containing my, proxy code. So that's, cd, ajaxproxy. And then back on that deploy page on my Heroku dashboard. My cloned project is already initializes as you get repository because I downloaded it from GitHub. So all you need to do is copy this command, make sure you don't grab that dollar sign. That's just showing you that there's a command prompt there. And I'm going to paste that in, at my prompt. And so now, this repository has it's remote set to that project, on Heroku. On so next it's just a standard git work flow. First I do a git add . Then I do a git commit, - m, and in quotes I'll say 'initial commit'. And if this is your first time using git on your machine, you may see something like this if not, you won't see it, it's fine. And then finally, I just want to do a git push heroku master. And that is going to push the entire project, up to Heroku. Now once the files have been uploaded Heroku starts building the app and we're seeing the progress of that here. This can take a little bit. Now if there's any errors in your index that GS code, that may cause the build process to fail. So if that happens, you need to go back to that index that GS file in your editor. Find and fix those errors and then just go through the git add, git commit and git push again. To actually push those changes up to Heroku. So once we see that bash prompt, the dollar sign. I've got build succeeded right here. So that tells me I'm all good, and so know my custom code is hosted and running on Heroku and it's listening for HTTP requests. To actually test my code, I need to make a change to my front end ajax code. Right now, my front end code includes a URL, that makes a direct request to the NPSAPI and includes the API key. And that's starred right here in the parksUrl variable. And I need to change both of those things. So, I'm going to grab that parksUrl, variable, copy it, duplicate it, comment out the original, to keep that for reference. And first off, I need to replace the base URL, the end point. I'll take out, developer.nps.gov, and instead I need to insert the URL from my Heroku app. So just that the name of the app.herokuapp.com. So for me that's, ecajaxproxy.herokuapp.com. Bur make sure that you get the name of your app, that you created in Heroku and use that here instead. And then after that, I need to put /nps which is the name of the end point that I created. Finally, I need to take out the API key. Remember that my back end proxy code running on Heroku will append, api\_key, plus the value to the end of my URL before forwarding it to National Park Service. So, here, I'll delete, and append, api\_key= and that entire API key string. So I got stateCode=ca and my closing quote. And then save this changes and then the moment of truth. I'll start up, live server for contact at HTM. And there's a pause, which is a good sign. Waiting to get data back. And it looks like a good I got a random park, rather than that hard coded fallback, so that a good sign. And, opening up the developer tools, and I'm going to check out the network tab. Go ahead and reload the page to get that network information logged. And I will go full-screen, with my developer tools. And down here, in my parks request, you can see that I have a 200. And if I open this up this up, there is my request to my proxy. And looking all they way through here, nowhere in this data is my API key. Here is the data, I got back. Here is that response, and so, the client, which is the browser. Never got the API key, and yet, it did get data back and that's exactly what a proxy is suppose to do.

### **Configure proxy security**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Setting up a proxy to store your API keys and append them to a request keeps your credentials out of the front end which is a major step forward for keeping them secret. However, what if an enterprising user examined your HTTP request to your proxy and duplicated it in their own app? As long as their request URL matched what the proxy expects, the proxy would do its job, append your API key, forward the request and then send along the response back to the app that had asked for it. Although other developers wouldn't have your actual credentials, they'd still be able to take advantage of your access to the target web service, and if you're paying for that service or trying to stay under a small free limit, you'd eventually get a higher bill than you expected or you'd have your requests cut off more quickly when you hit that limit. You can prevent this situation by configuring your proxy to check the origin of each request. In HTTP, the origin identifies where a request came from. An origin includes the protocol, any subdomain, the domain and the port number. A difference between two origins in even one of these factors makes them distinct. Every HTTP request includes the origin, and this value cannot be altered programmatically from a browser. This means you can use the origin as a mechanism for securing your proxy server by including logic to check the origin of each request against a list you provide. This ensures that requests from your app, whose origin you've specified in the backend code, are allowed while requests from other origins are blocked. The Ajax proxy app includes a filter variable that allows you to specify one or more origins that should receive access. The origin of every request is checked against the origin or origins you specify, and all requests from origins not on your list are rejected. The code contains two versions of the return statement. Both are commented out right now. I am currently running the file locally from 127.0.0.1:5500 which is the port, and I want to allow both HTTP and HTTPS versions of that. So first, I'm going to delete the return true statement that I added earlier because I only need one return statement, and I'm going to uncomment this last line which is the return statement that allows me to specify multiple origins. And then for the first origin, I'm going to take out that example URL, and I'm going to enter HTTP 127.0.0.1:5500, and then I'm going to copy that. I'm going to paste it in for the second one, but I'm going to change that HTTP to HTTPS because remember the protocol is part of the origin, and so each of these URLs is considered a distinct and different origin. I'm going to save that. I'm going to go back to my terminal. We'll do a git status just to see where we are, and I do have that one change I just made, so I'll do a git add, do a git add dot. Git is really useful. It's very good at noticing what you may have meant when you make an error and suggesting it. I did a git add dot. Then, I want a git commit -m, and I'm going to say added origins to filter and then a git push heroku master. Again, we're building out that app, getting to watch in real time as that happens, and as long as I don't have any errors pop up here. It says the build succeeded, so I'm all good once I get my Bash prompt back, and there it is. Now I know I didn't break anything. Now I want to test whether this really works. One easy way to change the origin while I'm testing is to change the port that my live server serves from. So, I'm going to go back to Visual Studio Code, and I've killed my live server already. I'm going to go into the configuration and specify a different port. So, I want to open the command palette, on a Mac that's Command + Shift + P, and then I'm going to type settings and json, and that gives me the two options for JSON files where I can enter settings. I don't want the default settings. I want just open settings These are a few things that I've set. After this last line, I'm going to add a comma, go to the next line, and I have to use double quotes 'cause this is JSON, and I want liveServer, with capital S, .settings.port: 6500. Remember, no comma at the end because this is JSON. I'm going to save that and then switching over to contact.htm in my front end code, and I need to kill my live server over here, and I'm going to go live. And, I can see that I'm serving now from port 6500. I can see that I pretty quickly got my backup data display which suggests the request didn't work, so I'm going to check in my console, and I got a 404 error from my proxy, and that is exactly what I expect. My request didn't go through, and that's because my origin now with that different port number doesn't match the origins that I specified in my proxy server configuration. This is exactly what I want to see. So to get my origin back to the correct one, I'm going to go back to my editor. I'm going to end that live server session, and back in my settings.json file, I'm just going to delete that configuration line for liveServer.settings.port, take that comma out of the previous line because, again, JSON doesn't like it if you do that. I'm going to save that file and then back in contact.htm, I'm going to go live with that again. Now, we're at port 5500. I do not have data immediately which is a good sign, and then when it shows up, it is randomized data from the park service, and I have no errors here. I have my data object logged, and so I'm good. So, now I have my proxy secured against casual misuse. Obviously, 127.0.0.1 is an address that anyone could run locally, so this works for me in testing only. But when my app is finished and I'm ready to deploy to a domain, I'll change my proxy settings to use that domain name.

### **Challenge: Customize code for an Ajax proxy**

Selecting transcript lines in this section will navigate to timestamp in the video

(funky music) - [Instructor] Ready for a challenge? It's time to try your hand at configuring an Ajax proxy server. The Explore California contact page includes a second Ajax call to Smarty Streets. Now the API credentials for this service are already locked only to the origins you specify. So it's actually okay to make them available to the client. But because you already have an Ajax proxy running, and you have one set of credentials stored there, it makes some sense organizationally to store your other credentials there as well. Then you always know where to look for the API credentials used in your app. To complete this challenge, you'll add another endpoint to your Ajax proxy server for the Smarty Streets API. You'll need to duplicate the apiOptions variable with a unique name to store the values for the Smarty Streets request. Then you'll need to duplicate the apiProxy variable with another unique name and swap in the name of your new apiOptions duplicate. Finally, you'll need to duplicate the app.use statement to reference a unique endpoint and then call the unique duplicate you made of apiProxy earlier. This challenge might take you about 10 to 15 minutes. When you're done, join me in the next video and I'll work you through how I approached it.

### **Solution: Customize code for an Ajax proxy**

Selecting transcript lines in this section will navigate to timestamp in the video

(electronic music) - [Instructor] To create my proxy end point for SmartyStreets, I need to do three things. In my index.js file, the first thing I need to do is duplicate the apiOptions variable. And then I need to rename the new copy, because I need a different copy for each of my end points. So I'm going to, instead of saying API, I'm going to say smartyOptions. And then to configure this, I need to change the target value so it's not for the National Park Service, but it's for SmartyStreets, and that's us-street.api.smartystreets.com. Then I need to come up with a different API path, and so instead of NPS, I'm going to use streets. Finally, I need an API key. So I know that in my SmartyStreets url, if I take a look at that code, the key in the query string that SmartyStreets is expecting is auth-id, so I know I can swap that in here as the key name, auth-id. And then I need to replace this with whatever name I choose in Heroku. So right now, I'm just going to decide to call this STREETS\_APIKEY, STREETS\_APIKEY, and I'm actually going to copy that, and then over in the dashboard for my app I'm going to go to settings, I'm going to reveal my config vars, I copied that new key name and I'm going to paste it in here, STREETS\_APIKEY, and then for the value I'm going to grab that key which I've saved, paste it in here, click the add button to save it. So now I have that SmartyStreets API key saved on the back end as a config var in Heroku. And so now, I have my query string key, I have a reference to the key value that's saved on Heroku, and so that takes care of smarty options. There's two more things I need to do. I need to actually create the proxy, and so I'm just going to copy this apiProxy variable, and again I need to rename it. So I'm going to call this streetsProxy, and I need to reference smartyOptions, so I'll paste that variable name in here. And then the last thing I need to do is duplicate this app.use statement. So in my smartyOptions I chose /streets as my endpoint, so down here I need to say /streets, and the name of my proxy is streetsProxy. So what this means is that when a request comes in, the endpoint /streets, it gets routed to streetsProxy, streetsProxy uses this framework to create a proxy using smartyOptions, and smartyOptions defines how requests that come in to that endpoint should be handled. So I'm going to save my changes, now I need to do one more thing, I need to actually change my front end code. And so over in app.js, first what I want to do is copy and paste my smartyUrl, keep it around for reference, and then I'm going to copy the beginning of that parksUrl up through the .com, and I'm going to swap that in through the .com here. Remember the endpoint that I chose was /streets, so I need to append that here. And then I don't need my auth-id or my API key anymore, so I can take off all of that, can take off the end of that quote, and then I don't need that ampersand because I've got a question mark here. So I'm just sending the key value pair candidates=10, my other key value pairs are getting appended once that form is completed, and then again the proxy will take care of adding the API key. So I'm saving that updated URL. Just for proof of concept I can also comment out my API key in the front end code. So I'm going to save that, and so switching to my terminal, I'm going to do a git status, and I have made some changes to index.js, so I'm going to do a git add ., git commit -m 'added streets endpoint', and then a git push heroku master. And again as long as I haven't made any syntax errors in my code, any logic errors, this should build correctly. Build did succeed, I see right there, which is always a good thing to see. Got my bash prompt back. So now I will go back to Visual Studio Code, go to contact.htm, and go live. And this time I need to actually submit an address, and I get my zip code back. So opening my console up, and let me go back and reload that... And I can see my street address request here. Again, I have a request to the proxy with all of my data there, and looking through all of this, the API key is nowhere to be seen, but I got my data back just like I did before, and so I have successfully extended my proxy server with a second endpoint, so that it can handle requests for both National Park Service data and SmartyStreets data.

### **Next steps**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Sasha] Thanks so much for joining me in this course. You now have experience working directly with the XHR object that forms the foundation of Ajax on the web, as well as the Fetch API. To dig deeper into some of the JavaScript mechanics that underpin Ajax, check out courses on asynchronous JavaScript or same-origin policies. If you want to learn more about working with a backend, explore courses on Node or Express.js. If you want to dig into the tools available for inspecting code in the front-end, check out courses on browser developer tools. Feel free to follow me online. Now take your new skills and build something amazing. Good luck!