HTTP Resources

Introduction

Scott Allen: Hi, this is Scott Allen and this module is the first module in a series describing the Hypertext Transfer Protocol or what we call HTTP. HTTP is the protocol that lets me search for microwave ovens and buy one from Amazon.com. It's also the protocol that lets me reunite with old friends on a Facebook chat or when there's nothing good on TV I can go to YouTube and watch videos of cats doing funny things. All of these things happen on the web where HTTP defines what is possible. It's a protocol that allows a web server from a data center in the United States to ship information to an Internet cafe in Australia where a student can read a web page describing the Ming Dynasty in China. In this course, we'll look at HTTP from a software developer's perspective. Having a solid understanding of HTTP can help you write better web applications and web services. It can also help you debug applications and services when things go wrong. We'll be covering all of the basics including resources, messages, connections and security as it relates to HTTP. This module is going to focus on resources.

Uniform Resource Locators

Perhaps the most familiar part of the web is the HTTP address. When I want to find a recipe for a dish featuring broccoli, which is always never, I might open my web browser and enter http://food.com in the address bar. So I can go to food.com and search for recipes. My web browser understands that syntax and it knows it needs to make an HTTP request to a server named food.com. We'll talk later about what it means to make an HTTP request and all the networking details involved. For now I just want to focus on this address, http://www.food.com. It's what we call a URL, Uniform Resource Locator. It represents a specific resource on the web. In this case, the URL locates a resource that is the home page of the food.com website. Resources are things I want to interact with on the web; images, pages, files and videos. These are all resources. There are billions if not trillions of places to go on the Internet. In other words, there are trillions of resources. Each resource will have a URL I can use to find it. News.google.com is a different place than news.Yahoo.com; two different names, two different companies, two different websites so, therefore, two different URLs because they're two different resources. Of course you can have different resources and different URLs inside the same website. Food.com/recipes/broccolisalad is the URL for a page with a broccoli salad recipe while food.com/recipe/grilledcauifllower that's still at food.com but it's a different resource describing a different recipe. We can break the URL for that last resource into three parts. First there's the HTTP part, the part before the colon slash, slash. It's what we call the URL scheme. It describes how to access a particular resource and in this case it tells the browser to use the Hypertext Transfer Protocol. Again, we'll provide more low-level details on this protocol later. We'll also look at a different scheme HTTPs, which is the security HTTP protocol. You might run into other schemes on the Internet like ftp for the File Transfer Protocol and mail to for email addresses because URLs are used for other protocols besides just HTTP. Everything after colon slash, slash will be specific to a particular scheme. So, a legal HTTP URL may not be a legal mail to URL. Those two really aren't interchangeable which makes sense because they describe different types of resources. The first part after colon slash, slash is food.com and that is the host. This part literally tells my browser which computer on the Internet is hosting the resource. My computer will use the domain name system to look up an address for food.com, turn that into a network address and then it will know exactly where to send a request. You can also specify that host portion using the IP address directly but most people want to use a friendly name like food.com instead of 204.78.50.82. The last part of the URL is the URL path. The food.com host should recognize what specific resources requested by this path and respond appropriately. A path looks very hierarchical like a file system path and sometimes a URL will point to a real resource that is on the host's file system or hard drive. For example, the URL food.com/logo.jpeg might point to a jpeg file that really does exist on the food.com server; however, resources can also be dynamic. The URL food.com/recipes/broccoli probably doesn't refer to a real file on the food.com server. Instead, some sort of application is running on the food.com host that will take that request and build a resource using content from a database. The application might be built using asp.net, php, pearl, ruby on rails or some other web technology that knows how to respond to incoming requests by sending back HTML that a browser can display. In fact, these days many websites try to avoid having any sort of real file name in the URL. For starters file names are usually associated with a specific technology but many URLs will outlive their technology that is used to host and serve them. Secondly, many sites want to place key words into a URL like having recipe and broccoli in the URL for a broccoli recipe resource. Having those key words in the URLs is a form of search engine optimization that will rank the resource higher in search engine results. Its descriptive key words not file names that are important for URLs these days. Some resources will also lead the browser to download additional resources. This food.com page will include images, JAVA script files, cascading style sheets and other resources that all combine together to present the recipe that we're viewing. If you view the HTML source code to this page, you'll see script tags, image tags and style tags inside that will point to additional URLs. So in building one web page like that a browser will typically make multiple HTTP requests to retrieve all the resources needed for that one page to display properly in the browser.

HTTP and IIS

Let's look at a quick demonstration with IIS just to make this a little more concrete. IIS or Internet Information Services is the web server that can run on Windows machines. What I have to open is the IAS manager which is showing me that there is a default website configured to run on this machine. If I go into the basic settings for this, it will show me that it maps to the physical path C:/inetpub/wwwroot. Now, IIS knows how to respond to incoming HTTP requests and one of the things it can do is actually look in this physical path to see if the resource exists on the file system that it can use to respond to an incoming request. So, inside of that physical path I have a folder called "test" and inside of that folder I have a file called "test.text" that just has plain text inside of it. So this is not paper text, this is not HTML, this is just a plain text file with the word hello in it and in order to reach it, I'm going to have to craft a URL to reach that file. I know the scheme that I want is HTTP, I know the host that I want is going to be local hosts. That's the machine that this is executing on. So let me open up a web browser and go to http:localhost/test/test.text and that returns me the resource and displays the contents of that file which is just the string hello. If I were to ask for something else like test.asp, IAS will come back and say sorry the resource that you're looking for has been removed or it's not there, I cannot find it, but let's actually change that file into an asp file. So, if you're not familiar with asp, it was one of the early web technologies that Microsoft released that allowed you to build dynamic web pages. So I'm going to rename test.text to test.asp and then we'll edit test.asp and do something very simple, which is to say response.rate and we'll still just say hello and that was the syntax you could use to write out strings or the results of calculations that you put together in an asp file and that's still not HTML; that's still just writing out the text hello. If we wanted to really produce HTML, we'd need an HTML tag and a body tag and all of that stuff, but let's just see if our request works now after I make sure I save this file. We'll do a quick refresh and now we still get the hello. So asp is just an example of a technology that you can use to dynamically build a response to an incoming HTTP request for URL. We'll look at some more details of the interaction that just happened as we progress through this course.

Ports, Queries, and Fragments

Now we know a little more about URLs or URLs as some people call them and we know that a URL consists of a scheme, a host and a URL path but there's an additional piece of information in this URL, the colon 80 and the number 80 represents the port number that the host is going to use to listen for HTTP requests. The default port number for HTTP is port 80. So you generally see this port number omitted from an URL. I would not need to use colon 80 to reach any web server that's listening on the default port because the browser will just assume you mean port 80 unless something else is specified. If, however, I went into IAS, which also listens to port 80 by default and I configured it to listen on port 8,080, then I would need to put that port number in the URL in order to reach that test.asp resource that we were experimenting with earlier. So you only need to specify the port number if the server is listening when a port other than the default port, port 80, and that usually only happens in testing, debugging and development environments because most commercial websites don't want a port number in their URL, it just makes the URL that much harder to remember and it makes the URL a little bit longer. Let's look at another URL. This one still has a scheme and a host and URL path, of course, but it has another optional piece on the end which is known as the query. Everything after the question mark is the query or query string some people will call it. The query string will contain information for the website that it can take and it has to interpret to figure out what resource you want. There's actually no formal standard for how this query string should look. It's technically up to the application to interpret whatever it finds there but you'll see the majority of query strings in use are used to pass mean value pairs. So, for example, the query string in this particular URL has two parameters. The first one has the name first name with the value Scott, the second one has the name last name with the value Allen and so if we go back to the bing URL at the top, the bing search engine will see the name Q and it turns out that is the query string parameter that it looks for to figure out what you're trying to search for and we can think of that URL as the URL for the resource that represents the bing search results for a particular type of tree that happens to grow in Southeastern Brazil. Finally, one more URL. This particular URL, again, still has a scheme, still has a host, those are required, and has a URL path and then it also has this piece at the end while along the hash sign it is known as the fragment. The fragment is different than the other pieces we've looked at so far because unlike the path in the query string the fragment is not processed by the server. The fragment is only used on the client and it identifies a particular section of a resource that the client should navigate to or focus on. Web browsers will typical align the initial display of a web page such that the element identified in the fragment will be at the top of the screen. Let me show you an example. We will actually open up the Wikipedia article for the Brazilian grape tree, and we'll see that there's a table of contents here. Notice that the URL at this point does not have a fragment inside of it, but if I click on cultural aspects all I'm really doing is navigating within that resource, this single particular article and I can even take that link and hand it to someone else and if they follow that link they, too, will be taken to that particular spot so that this particular section of the resource appears on the page instead of starting at the very top of the page and if we look at the source code to this or inspect the element, we would indeed see that the ID of this dom element that's in the browser matches the text that is in the fragment of the URL. So fragments have nothing to do with reaching the server or how the server interprets particular URL; they're strictly used on the client side, but you will see them in URLs so I wanted to point them out. If you've been following along so far, then you'd be able to look at this URL and know that it has a scheme, which is HTTP, and that's required; a host name called host, that's also required; a port of 8,080; a URL path called path; a query string, q equals query; and finally a fragment.

Query Strings and ASP

I wanted to do a quick demonstration of reading the query string from an active server page just so you can get a feel of what this would look like from a web programming perspective. Here I had the browser pointed to the test.asp file that we were working with earlier and right now it's passing in a query string that says name equals Scott. So how could I take that query string and print out a message here inside of the HTML that would say hello, Scott? With an asp the first thing I might do is declare a variable called name and use that to hold the value that I get from asking for query string sub name. Basically on active server pages you could have it parse out the value of a particular query string variable that you give it and we'll just write that like this. Declare it then assign it and then declare a message. We'll say that the message is equal to hello comma and then we'll add on the name value, but before I do that I'm going to make sure I HTML encode that value just in case some malicious user comes to this page and puts JAVA script in the query string trying to get that to show up on the page. HTML encode will make sure that that shows up just as text on the page and here in the div I can now put the message. So, I'll save that file, refresh the browser and that works.

URL Encoding

All software developers who work with the web should be aware of character and coding issues with URLs. The official standards describing URLs go to great lengths to make sure URLs are as useable and interoperable as possible. A URL should be as easy to communicate through email as it is to put on a billboard or a bumper sticker or a business card. For this reason, the standards to find unsafe characters for URLs and unsafe character is a character that should not appear in an URL. For example, the space character is considered unsafe because spaces are hard to read. They can mistakenly appear/disappear when a URL is in printed form. Other unsafe characters include the pound sign because it's used to delimit a fragment. That doesn't mean that you cannot use a pound sign in a URL; it just means that the pound sign can only be used in its reserved position which is to delimit a fragment. Another unsafe character is the caret because it isn't always transmitted correctly through the network. RFC 3986, which is the Internet standard or the law for URLs, defines the safe characters as basically being the printable US ASCII characters. The lower case alphabet, the upper case alphabet, the digits and then just a few special characters like dollar sign, underscore, asterisk, the parentheses and a comma. Unfortunately, you can still transmit unsafe characters in a URL, but they need to be percent encoded or URL encoded; two different terms but it's the same outcome. Percent encoding is the process of taking a character like the space character and a URL and replacing it with a percent 20; 20 is the hexadecimal value for the space character in the US ASCII character set and so a percent encoding is basically taking that hexadecimal value, putting a percent in front and then replacing a character in the URL with that percent encoded value. As an example, if you really did want to have Scott_Allen in a URL, it would need to be encoded so that it would come out to be Scott&20Allen. Almost every web application framework will make this really easy. They all have APIs for URL encoding. On the service side, you should run dynamically created URLs through the encoding algorithm just in case one of the unsafe characters will appear in the URL.

Content Types

So far we've focused on URLs and simplified everything else, but what does it really mean when we enter a URL into the browser? Typically it means we want to retrieve or view some resource. There's a tremendous amount of materials to view on the web and also we'll later see how HTTP enables us to create, delete and update resources but for now we're going to stay focused on retrieval. We haven't been very specific about the types of resources that we want to retrieve. There are thousands of different resources on the web. There are images, there are hypertext documents, xml documents, video files, audio files, executable applications, PDF documents and Word documents. In order for a host to properly serve a resource and in order for the client to properly display a resource, the parties involved have to be very specific and precise about the type of resource. Is the resource an image or a movie? We wouldn't want our web browsers to try and render a JPEG image as text and we wouldn't want them to take text and try to interpret it as an image. So when a host responds to an HTTP request, it returns a resource and also specifies the content type. This is also known as the media type of the resource. We'll see the details of how this appears in an HTTP message in the next module. The content type that a server will specify rely on the Multi-purpose Internet Mail Extensions or MIME standards. Although MIME was originally designed for email communications it worked so well that HTTP uses these standards for the same purpose, which is to label the content in a way that the client will know what the content is. So, when the client requests an HTML web page, the host can respond to the request with some HTML that it labels as text/html. Text is the primary media type; HTML is the sub type. When responding to a request for an image, the host can label the resource with a content type of image/jpeg or /gif or /png for png files. Those content types are standard MIME types and are literally what will appear, that text will appear in the HTTP response and location where the client can parse it. So for a long time I used to believe that a browser determined the type of content it was receiving just by looking at the file extension in the URL, but it turns out it doesn't work that way at all. In fact for many browsers, the file extension is the last place the browser will go to determine the content type that it's receiving. The first place it will go is the MIME type that the server returns. To demonstrate that real quick let's go back into Internet Information Services. Inside of the IAS manager for any given website I can go in and configure the MIME types that IAS will use. The MIME types literally are the content types or the MIME types that it will return when it's serving up a file from the file system. You can see there's quite a few MIME types that are registered in here including .PDF, which has a MIME type of application/PDF. That should tell the client that what it's receiving is a PDF file and that's how it should interpret the content. It just so happens that in the directory, the test directory that we were using earlier, I've put a file in here slides.PDF. So it's a static file and I'd like to be able to open a browser and just be able to request logohost/test/slides/pdf and have that display in the browser. You can see that that is working quite well, but let's do a couple of experiments. First, let me come in and rename slides.PDF to be slides.foo. Just some arbitrary extension and I'll come in and try to request slides.foo and IAS will refuse to serve this file because it doesn't know what the MIME type is. We need to put an entry into that MIME type configuration to say .foo should map too and in this case it could be application/pdf, but now let's go back and see what happens if we do something wrong so I'll rename slides.foo back to slides.pdf and if I request slides.pdf everything is working again and now to prove that the browser is relying on this content type to figure out what the content is, let's go in and edit this MIME type and say that when you serve up a PDF file the MIME type should be text/html. So the browser is going to be told that what it's receiving is HTML even though there's a .pdf in the URL. So let me do a hard refresh and now what we're seeing is that the browser is trying to display the contents of that PDF file as HTML and it's not working out so well. So, having incorrect or missing MIME types mapped in your server configuration and this is true for IAS, it's true for Apache, it's true for nearly every web server, it can cause problems in your website. For instance, one case I ran across recently was video files not being served because the correct MIME types were not registered on a server.

Content Negotiation

Although we tend to think of HTTP as something that's used to serve up web pages and images, it turns out the HTTP specification describes a very generic protocol for moving information around in an interoperable way. Part of the job of moving information around is making sure everyone knows how to interpret the information and that's why those content type settings and MIME mappings they're so important to the web, but media types aren't just for hosts. Clients can also play a role in what media type a host returns by taking part in a content type negotiation. A resource that's identified by a single URL can have multiple representations. Take, for example, the broccoli recipe that we were looking at earlier. A single recipe might have representations in different languages like English versus French versus German. It could also have representations that differ by format HTML versus PDF versus plain text versus xml. It's all the same resource and the same recipe just different representations. The question that comes to mind then is which representations should the server use and the answer to that is in the content negotiation mechanism described by the HTTP specification. So when a client makes an HTTP request to a server, the client can specify the media types that it will accept. Media types then are not only for the host to use to tag outgoing resources but they're also available for clients to specify the media that they want to consume. The client specifies what it will accept in the outgoing request message and, again, we'll see the details of that message in the next module, but imagine this request going out to the food server saying I want HTML and, oh, by the way, I also want this in French. Now it could turn out that the server doesn't have HTML for that recipe available. It only has a PDF and it will send that back or it could turn out that it does have HTML but only an English version and that might disappoint the user but that's why we call it content negotiation and it's not an ultimatum. We can actually see content negotiation at work with languages. For instance, if I go to google.com with my default configuration, everything is showing up in English. I have the Google search button and the I'm feeling lucky button, but now let me go into my Internet options and change the languages and move French to the top of this language preference list. These languages that I want actually go out to the server and by putting French first on top I'm advertising to the server that I prefer resources to be in French where possible. After doing that if I refresh, you'll notice most of the text on the page here has changed over to French text. If I go back into Internet options, I move English up the server is going to respect this again and put everything into English. So, web browsers are pretty sophisticated pieces of software and they can deal with many different types of resource representations. This content negotiation is something a user would probably never care about except for purchase the language settings but for software developers, you and me, especially people who develop HTTP web services content negotiation is part of what makes HTTP great. A piece of code written in JAVA script can make a request to the server and ask for a JSON representation because that's easy to parse in JAVA script. Meanwhile, a piece of code written in C++ can make a request to the same server to the same URL and ask for an XML representation of a resource. In both cases if the host can satisfy that request, the information arrives at the client in an ideal format for parsing and consumption.

Conclusion

In this module, we learned that the web and HTTP are both all about resources. We have URLs to locate those resources and MIME type to specify the representation of those resources. All of this was designed to just make things work. So a Linux server can communicate with a PC client and vice versa. At this point, we've gotten about as far as we can go without getting into the nitty gritty details of what an HTTP message looks like. You're probably already wondering what these content type specifications look like when they go across the network wire. We'll dig into those details in the next module.

HTTP Messages

Introduction

Hi, this is Scott Allen, and in this module we're going to look inside the messages exchanged in an HTTP transaction. We're going to learn about methods, message types, HTTP headers, and status codes. Understanding these concepts is important for developers who work on the Web. Not only will you be able to build better applications by responding with the right types of messages, but you're also going to be able to spot problems and debug issues, when Web applications aren't working.

Message Types

Imagine walking up to someone in an airport and asking, Do you know what time it is? In order for that person to respond with the correct time, a few things have to be in place. First, the person you've asked has to understand your question, because if they don't know English, they might not be able to make any response. Secondly, the person you've asked will need access to a watch or some sort of timekeeping device. This airport analogy is similar to how HTTP works. You're the client, and you need a resource from some other party, the resource being information about the time of day. So you make a request to the other party, using a language and vocabulary that you understand and you hope they will, too. If the other party does understand your request and has the resource available, they can reply. If they don't understand the request, then you might not get any response. The HTTP specification -- specifically, the HTTP/1.1 specification -- it defines the language, so that everyone on the Web - - all the clients and all the servers -- they can understand each other. It defines the messages being exchanged on the Web and what they should look like and what they contain. And it turns out there's two types of messages. HTTP is a request and response protocol, so the first type of message is the HTTP request. That's what the client sends to the server, and they carefully format that message so that the server will understand it. A server responds by using a different type of message, the second type of message, which is an HTTP response. And again, that message will be formatted so that the client will understand. It's formatted according to the HTTP/1.1 specification. The request and the response are two different message types, but they get exchanged inside of a single HTTP transaction. The standards define what goes into those messages, so that everyone who speaks HTTP will understand each other and be able to exchange resources. Or, when the resource doesn't exist, the HTTP response can contain an error message, that the client will understand that that resource didn't exist.

A Manual Request

A Web browser knows how to send an HTTP request by opening a network connection to a server or machine and sending out that request message. For instance, if I come into a browser and ask to go to www.odetocode.com and pull down an image, odetocode.JPEG, the browser sends off that request and displays the JPEG. Notice one thing that happened here is the URL changed from www to just odetocode.com/odetocode.jpg. Now there's nothing magical about this request. It's just a command in plain ASCII text, and it's formatted to the HTTP specification. And any application that can send data over the network -- pretty much any of those applications can make an HTTP request. You can even make a request manually, using an application like Telnet, from the command line. Now, Telnet has been around for decades. It's a very old application. And what we can try to do is Telnet to odetocode.com. The problem is that Telnet, by default, tries to use port 23, and there's nothing listening on port 23 on odetocode.com. And we learned in the first module of this course that the default HTTP port is port 80. So that connection didn't work. Fortunately, you can pass a port number to Telnet and tell it to connect to port 80, and now we are connected. Let's see if the server understands plain text messages. So I'll type out: Can I have odetocode.jpg? And I get a response. The response is telling me that I made a bad request. Essentially, it didn't understand the natural English that I typed in here. Didn't send a proper HTTP request message. It wasn't formatted according to the specification. Fortunately for us, I know how to make a proper HTTP request. So let me connect again to odetocode.com, port 80. And this time I'm going to enter an HTTP message, a proper message. So first I'm going to type what I want to do, which is get; and the resource I want to get, which is odetocode.jpg; and the protocol that I'm using, which is HTTP/1.1. On the next line, I need to type some additional information that is required in every HTTP message, and that is the host that I'm trying to connect to, which is www.odetocode.com. This host information is required, and it's because a server like this can support multiple websites. This server could support odetocode.com and odetofood.com and www.odetocode.com. They could all be different websites. Obviously, odetocode and odetofood would be two different websites. And the server doesn't really know that I'm trying to connect to odetocode.com, even though I typed that into the command window. That has to be in the message. It has to parse that out and figure out where to send this message, which site. So with all that in place, I can press the Enter key twice, and this time we get back a different response. Now, we will be breaking down the pieces of this request and response, as we move through this module. But let me give you the high-level details of what this response is telling me. It's telling me that the resource I want exists -- this JPEG image -- but it's moved somewhere else. It's moved permanently to another location. And the response says that that resource is now at odetocode.com. Not www.odetocode.com, just odetocode.com. So if I'm implementing a Web browser, it's up to me to parse that response, realize that I've been redirected -- that something has moved, and then issue another request to that proper location that's specified in the response. These types of redirects are common, and the reason is to make sure that all requests for a particular resource go through a single URL. In other words, we don't want content coming from www.odetocode.com and just odetocode.com. We want everything to go through odetocode.com. This is a search engine optimization technique known as URL canonicalization. Now that I know the resource has moved to a different location, let's try this once again. I'm going to Telnet to www.odetocode.com, port 80, just to prove that the host name that I'm using to connect and the host that I specify in the HTTP message really are two different things. This piece that I'm typing into the Telnet command just gets looked up as an IP address, and all the server knows is that we're trying to connect to it on this port. So now let's try to get odetocode.jpg, using the HTTP/1.1 protocol. Host, now, is just odetocode.com, not www.odetocode.com. And I'll submit that request. And what we get back is a bunch of binary data that represents the JPEG. And if I scroll up, we will see that that content -- that JPEG image -- was part of the HTTP response message. The response message tells us that the request went okay; that the content that is being returned -- remember the mime types that we talked about in the first module -- the content type is image/JPEG; and then there's some additional information in this response, some additional headers that specify things like when this particular resource was last modified, that allows the client to do some caching -- we'll talk about that more -- and things like the content length and the date. This is just information -- additional information that the client can use, about the response. And so, what we've done here in the Telnet window -- which is issue a request for that JPEG, get redirected, resend the request to a different host and actually pull down the data -- that's exactly what happened here in the browser window, when we initially went to www.odetocode.com for this image. And the browser saw that redirect, reissued the request to odetocode.com, and it lets us know that it did that, by putting the new URL to the resource here in the address bar.

HTTP Methods

Let's talk about the request that I sent to the server in that last clip. The first word that I typed into the Telnet session was get, and get is one of the primary HTTP methods. Every request message has to include one of the available HTTP methods. And the method tells the server what the request wants to do. So get wants to do what it sounds like it wants to do, it wants to get, or in fact, to retrieve a resource. I can get an image or get a PDF file, an HTML page, or any other resource that the server might hold. Some of the common HTTP methods are shown in this table here. We have get, to retrieve a resource; post, to update a resource; put, to store or add a resource; and then there's delete, to remove a resource. There's also a head method, which is asking the server just for the headers that describe a resource. We'll talk more about headers later. Now, of these five methods, just the first two are the primary workhorses of the Web, get and post. So even though the HTTP specification lists a number of legal methods -- even more than what we see here -- it turns out that the HTML specifications only use get and post. So put and delete are almost never used in Web applications. Now, if you're writing an HTTP Web service, you might want to use these other methods to add and delete resources, but you'll have to be careful, because there are even some server-side technologies and pieces of hardware on the network that will not process put and delete messages. So primarily, what you use to get work done with HTTP is get and post. A Web browser issues a get request when it wants to retrieve a resource, like a page, an image, a video, or a document. And a get request is probably the most common type of request. It's basically used to read data. A Web browser sends a post request, when it has data it wants to send to the server, perhaps for an update scenario. For example, if I go to Amazon.com and click Add to Cart, that's going to issue a post request to Amazon, to describe what I want to purchase. Post requests are typically generated by a form element on a web page, like the form that you fill out with input elements for address and credit card information. I'll show you some specific scenarios with form tags, but first we have to really understand the primary difference between get and post.

Safe Methods

There is a part of the HTTP specification that talks about safe HTTP methods. Safe methods are methods that you read and view resources from a Web server. Unsafe methods are methods that let you change resources on a Web server. The get method is one of the safe methods, since it should only retrieve a resource and not alter the state of that resource. So sending a get request for a JPEG image doesn't change the image. It just fetches the image for display. We say that a get operation on the Web should never have a side effect on the server. Contrast that with an HTTP post. This is not a safe method. It typically changes something on the server. Post is what we use to process a credit card transaction, update an account, submit an order, or perform some other operation that may be destructive or constructive. For this reason, Web browsers typically treat get and post differently, since get is safe and post is unsafe. Let me demonstrate the difference in a Web browser, and then we'll dig into some code. When I go into the Web browser and tell it to request signup.cshtml, it issues a get request to the server, brings back some HTML and displays this form that might be a sign-up form to create an account. And I can refresh this page as many times as I want. It's okay to repeatedly send a get request. It's not going to change anything. However, once I fill this form out and I submit it, that has to be a post operation, and what the browser is showing me now is the return message that included some HTML from that post operation. And now, if I try to refresh, I'll get a warning. The browser knows I just performed an unsafe operation. I'm trying to refresh it. And that might cause me to submit a duplicate credit card transaction. It might try to create two accounts, instead of one account. There are many scenarios where undesirable things will happen, if I click that Continue button to resubmit that post operation. Let's look at some code and see if we can improve this scenario.

GET and POST Scenarios

Let's start off with a very simple scenario. Here I have an ASP.NET Web page, written using the Razor syntax, that is just going to display a link to a sign-up page. And when I run this in the browser and the browser sees that URL, it has to issue a get request to pull down the HTML. And since that's a safe operation, I can click Refresh as many times as I want. And even when I follow that link, the browser gets presented with a new URL, and now I have a form that I can fill out. But once I click the Submit button, that's going to be a post operation. So here, in signup.cshtml, we have inputs, where a user can enter the first name and their last name. These are inside of a form. Method equals post. That's going to allow the browser to collect those inputs together and post them as name value pairs inside the body of the HTTP message. And we'll take a look at exactly what that message looks like in a bit. But for now, just concentrate on the fact that this is a post operation. We're going to post back to the same URL that this came from, so post back to signup.cshtml, which, when it detects that there's a post back, it's going to read those values out, redisplay itself to the user. And it's going to write out the first name and the last name that get posted to it in the form collection. Now, since that's a post operation, and the browser knows that that is not a safe operation, clicking Refresh at this point brings up that warning -- Are you sure you want to do this? -- because you might duplicate something. You might put two charges on your credit card instead of one. How can you fix a problem like this? Well, one solution would be, if there is a post back, let's not try to figure out how to redisplay this page to the user. Instead, let's save off the values that the user has entered, put them in the database. In this case, I'm just going to put them in a couple server-side session variables, and those are implemented with HTTP cookies, which we'll take a look at in a later module. But for now, just know that the first name and the last name are saved off in the server, in a place where I can get them, after we do a response.redirect, and redirect the browser to a different page, signedup.cshtml. So now, when I save this, it will come out and we'll go back and issue the initial get request for this page. Now, when I submit this page, then we should come into here, save the values, and immediately redirect the browser to signedup.cshtml. And when you do a response.redirect, the browser has to issue a get request for that URL. So now I'm signed up, and I'm also sitting on a page that is the result of a get operation. So I can refresh everything again. This post and redirect to a get is a fairly common pattern these days on the Web. It's called post-redirect-get -- or PRG for short -- and it's a pattern that you specifically implement, to avoid that warning, if the user tries to refresh the result of a post operation. In this case, what we did was send the user over to signedup.cshtml, which is just going to retrieve the values for first name and last name from the database, or from a Web service, or in this case from a session object, and display that on the screen. Now, one thing I want to point out is that not every form requires method equals post. When you use method equals post, the values that are inside of here get tunneled into the HTTP message. But you can also have a form, method equals get, and there's a significant difference between the two. So here is a search form. This has a form with a method equals get. It has an input, where the user can type what they're searching for. But when they click the Submit button, what the browser's going to do is issue a get request to the URL specified by this action attribute, which is results.cshtml. And instead of taking any inputs that I have inside of this form and putting them into the message body, it's going to put them into the URL, into the query string instead. So let's try that out. Let me go to search.cshtml, and let's do a search for food. And when I click Submit, results.cshtml knows I am searching for food. How does it know that? Because this get request forced the browser to issue a get request and put the input into the query string. So the results.cshtml?Q=food, the Q comes from the name of this input. And then, all results.cshtml has to do is look at request.query string, to figure out what the user is searching for. What are the significant differences here? Well, I just submitted a form, but created a get request, and I can refresh this as many times as I want. I can even take this URL, paste it into an e-mail or to a document and give it to someone else, so they can click on it and issue a get request to see these search results. That's significantly different than post. In a post operation, the inputs don't go into the URL. Again, they go into the HTTP message. Searching is inherently a safe operation. I'm really just viewing search results. Creating an account is not a safe operation. I want to have a form with a method equals post, to perform that create operation. Again, we'll come back a little bit later, once we've learned a few more things about HTTP messages, and figure out exactly what this request looks like with a post operation.

Request Messages

So far, we've seen a raw HTTP request and we talked about two popular HTTP methods, get and post. But as this Telnet session demonstrated, there's more to an HTTP request message than just the HTTP method. A full request message will consist of the following parts: There's the method; and then the URL; and finally, the version. This message is always in ASCII text, by the way, and this first line is what we call the start line. The HTTP version that you'll see in the start line is typically going to be 1.1, because that's the standard that's been around since 1999. After the start line, there can be one or more headers, and then there can be a body to the HTTP message. For get requests, you typically don't see a body. You just see the start line and then one or more headers. And I keep saying one or more, because, remember, the host header is a required header. Headers generally contain useful information that can help a server process a request. For example, in Part 1, we talked about resource representations and how the client and server can negotiate on the best representation of a resource. That's what we called content negotiation. If the client wants to see a resource in French, for example, this is where it will include a header entry that says the language I want to accept is French. Then there are numerous other headers that are defined by the HTTP specifications. Some of the headers are general headers that can appear in a request or a response. One example of that is the date header that you see here. And that allows the client or the server to include a header that indicates when it created this message. Now, everything but the host header is optional. But when a header does appear, it has to obey the standards. So the HTTP specification points to another standard that describes the standard format for dates. Here are some of the other popular request headers that you might see on an outgoing request. One is the Referer header -- and yes, it's misspelled in the standard. The Referer will contain the URL of the referring page, so when the user clicks on a link, the client can send the URL of the page that referred the user to the resource that it's getting. With the Web browsers, there's typically also a user agent header that describes what user agent is making this request. On the server, you can parse that user agent and figure out if the user is using Internet Explorer 6 or Chrome or Firefox. There's an Accept header that describes the media types that the user agent will accept, and the Accept Language that describes the preferred language. You'll also see Cookie headers in an HTTP request. We'll talk about cookies in a later module. And then, If-Modified-Since is one of these headers that you'll commonly see. The If-Modified-Since header tells the server when the user agent last retrieved this resource. And so, if the resource hasn't changed, the server does not have to send the entire resource back to the client. It can just tell it, yes, you have the most recent version. If-Modified-Since is quite commonly used to retrieve images, because the browser can aggressively cache images and improve performance, by not retrieving an entire image, if it already has a copy in the local file system. So when you put all these together, a full HTTP request message might look like this. We're going to get the root resource, using HTTP/1.1. We'll have a Host header, because that's required. We want the connection to stay alive. We'll talk about that in the next module. Here's the user agent header that says this is essentially Chrome, version 16. Chrome will happily accept HTML text, also XHTML. At the very end of that Accept header we're basically saying, give me anything that you have and I'll try to figure out what it is. There is a Refer, so we landed on this page as the result of a Google search.

And then there's some more Accept headers that describe the preferred language, the preferred character set, and the encoding that Chrome understands. So since it understands gzip, the server can compress certain resources, to send them down to the client. Notice that some of these headers can contain multiple values, so the Accept header says text/HTML comma, some other mime type comma, some other mime type. And you'll also notice this Q value. The Q value is always a number from zero to one, and it represents the relative quality value, or what the specification calls the relative degree of preference. So something with a Q value of 0.9 is just slightly more preferred than something with a Q value of 0.8, but far more preferred than something that has a Q value of 0.3.

Response Messages

After the request, you should receive a response. And a response message is similar to a request message. It has a start line that includes a version -- that comes first here in the response, though. Then the all-important status code, which we'll detail in just a minute. And then a reason, which is essentially a textual description of the status. Then you can have response headers and a response body, which might be HTML or image content or anything like that. A full HTTP response might look like this. We're responding with a message that follows the HTTP/1.1 specification. The status code is a 200, which is a good status code. That means the HTTP transaction was successful. And then there are some headers. There's a Cache Control, which essentially describes how this response can be cached. Private means it's a private response for an individual user, so it's okay for the browser to cache this response. But if there's any hardware in between the server and the client, it shouldn't try to cache this for multiple users. There's the content type, which is the mime type that describes the type of response you're receiving -- what's in the body -- and what character set it is encoded with. And you can have it a date, you can have a content length, which allows the client to know when it is has received the end of this message. And there's also some extensions that you see in this response. ASP.NET and IIS like to plug in their own headers, just to advertise the fact that this website is running on Microsoft technology. Since the content type in this response is text/HTML, we should expect to find HTML in the body of the response.

Status Codes

The status code that we saw in that previous response is very important, because it tells the client the result of the response. Did it work? Did it not? Is there something else to do? Every status code that is standard falls into one of these five categories. If the status code is in the range of 100 to 199 inclusive, it's an informational status code. The 200 range is the good range. That means the request was successful. Anything that starts with a 3 is going to be some sort of redirection, either a permanent redirect or a temporary redirect. 400 means the client did something wrong. It might have been a bad request. It might have been a request for a resource that doesn't exist. And 500 and up means something went wrong on the server. There was an application logic error, an unknown exception, the server is on fire. And here are some of the common status codes that you'll see. Status code 200 -- reason OK -- is the best one. That means everything was successful. 301 is the redirection that you issue from the server, when you want the browser to go to another location to find the resource and never check the original location. That's done for search engine optimization quite a bit. That's what we saw demonstrated in the beginning with our Telnet session. We had a 301 redirect. That's slightly different than a 302 redirect. That means the resource has moved, but it's okay to check that location again, to see if the resource is there some time later. I'll show you a specific example of where a 302 redirect happens here, in just a minute. 304 means you've requested a resource and you've said, if it hasn't changed since such and such a date, with the If-Modified-Since header, then don't make me pull down the full resource. In that case, if the server detects that, indeed, the resource hasn't changed, it'll just send back a 304, not modified, which means use whatever you have cached locally. It's still good. The 400 series of errors include 400, bad request. That's what happened if you perhaps send an HTTP request that's not using the proper syntax. 401 is an unauthorized request. The client might need to provide some credentials, before they can access this resource. We'll talk about security in a later module. 403 is just a flat-out refusal. You've tried to gain access to something that we do not want you to have access to, and there's no possibility to get to this resource. And then there's the infamous 404, which means you've requested something that isn't there. The server couldn't locate it. 404 is considered a client error. The two most popular server errors are 500, meaning internal server error. That could be because you had an exception inside of your application code, and the server was unable to complete the response. It could also be a bug or a problem in the server software itself. And then there's a 503 status -- service unavailable. Some services will return a 503, when they are under heavy load and cannot handle any additional connections, cannot process any additional requests. And it's essentially telling the client, we're having a problem, please try again later. And now that we know a lot more about the request message and the response message, let's actually use a tool that can show us real requests and responses in a browser-server interaction.

HTTP Fiddler

In this section, we're going to use a tool called Fiddler. Fiddler is a free tool that you can download. It runs on Windows. And what it will do is intercept all the HTTP traffic between your machine and some distant server, or even a server that's running on the same machine, and allows you to inspect every HTTP request. And it's actually a lot more advanced than that. You can build your own requests. You can write some scripts that manipulate requests. But we're just going to capture some requests and see what some real HTTP messages look like. I already have Fiddler up and running. On the left-hand side we'll see HTTP transactions as they occur. Currently, it's configured just to capture traffic from Web browsers. I can toggle that by clicking the button down here. And on the right-hand side, we'll be able to inspect the request message -- and we're going to do that with a raw view -- and also inspect the response message. We'll get a raw view of that, too. So let's jump into the website we were looking at earlier and go to the default.cshtml file. That's the file that had an anchor tag here, to present a link, so that the user could click and go to the signup page. I've also added an image to this page, so we'll see how that behaves. I'm going to open this in Internet Explorer, since we've already seen some of the request and response headers in the slides, and they were from Chrome. So we'll see how IE is a little bit different. And now the site is up and running in IE. We've made a request to localhost/default.cshtml. If we look over here in Fiddler, we can see there were two requests that were sent off for that page. This third one came from another browser instance that's running. I'm going to delete that, by selecting it and hit the Delete key. We just really want to inspect these first two requests. The first request was a get request for default.cshtml. You can see my accept language is in there. You can see IE basically has an Accept header that says give me anything. And of course, it has the required Host header at the end. And down here, in the response, what we see is a 200 okay response. That means everything worked. We can see a couple X- headers that ASP.NET and IIS inserted. And we can also see the HTML that makes up the page. The second request was when the browser encountered the image and said, ah, I need to go out and fetch DuPont.jpg. So it issues a get request for DuPont.jpg. And down here, in the response, it's mostly binary, so we can't really view this in Fiddler. We can see that the content type was set to image/JPEG. There was a last-modified header sent to the client. That's very important. What the client can do now is it realizes the date at which that picture was created. It can save that photo. It can save the date. And it will never need to download that again, as long as it has it on the local file system. In fact, if we come back to the browser and I issue a refresh, you'll see, again, we have two requests that go out. The first one was for default.cshtml. And again, we got the HTML back. The second one was for the JPEG image, but this time IE set up a header that said, basically, only give me this if it was modified since April 2nd, 2010. And the browser responded with an HTTP message that has a 304 status code, says essentially that resource has not been modified, so feel free to use your local copy, instead of me pushing all those ones and zeros to you again. Now let's take a look at what happens when we go to the sign-up page, and I'm going to fill out my first name and my last name -- and the capital letters are okay -- and hit Submit Query. And I've been signed up. Let's see what happened since our last interaction. First, there was a request for signup.cshtml. The browser returned the HTML that included the form tag and the input tags, where I could fill out my first name and last name. And then, the next step was the browser did a post. So the form method was equal to post. When I hit that Submit button, the browser had to put together a post request message. You can see that this request message actually includes a body, and the body is the main value pair, so the inputs that were in that form. So first name equals Scott, and last name equals Allen. When that request arrived at the server, we did some work inside of the logic of the application in the C# source code to do some things with the database or a session variable. And then we did a response.redirect. And a response.redirect generates an HTTP response from the server, with a status code of 302. If you remember, 302 is that temporary redirect. So we're telling the browser, well, you wanted this signup.cshtml resource, but really what you need to do -- just this one time -- is instead go to this location, signedup.cshtml. So that's a temporary redirect, much different from the 301 status, which is, you wanted this resource, but you need to always go to this other place, and don't come back here again. 302 is just temporary, because what we're trying to do -- if you remember from the beginning of this module -- is implement that post-redirect-get pattern, so that the user is left on a page that is displayed as the result of a get request. And that's this get request, the last request that was sent out to the URL that was specified in the 302 response. And now, as a user, I can happily refresh that page. All it's doing is sending off a get request and displaying my sign-up results that have been saved somewhere -- persisted in a database, perhaps. Now, to be thorough, let's also see what happens if I request a resource that doesn't exist. Let's just ask for something.cshtml. That's the 404 response -- resource not found. And let's also flip over into Web matrix. And in the default page, I'm going to add an expression for a C# variable that doesn't exist, and save that file, and see what happens when we come into default.cshtml. And this is a compilation error that displays in the browser. And over here, in Fiddler, we can see that that returned an HTTP status code of 500, which is internal server error -- something went wrong. Now, one quick note on this 500 status code. An HTTP status code of 500 means there was an error on the server. But it's really indicating something at the HTTP level. It doesn't necessarily reflect what's happening inside your application. For instance, imagine that sign-up form that we have, if the user didn't fill out the last name field. The application probably requires the last name field to be present, or it's not going to be able to create some account. That doesn't mean that you have to return an HTTP error code indicating failure, like an HTTP 500 server error or an HTTP 400 bad request, because, really, things are working at the HTTP transaction level. It's just that the user didn't give you all the information that you need for the business operation to complete. So probably, what you want to do there is return some HTML with a normal 200 okay status code, or just add some markup to the page, to tell the user that, sorry, we couldn't create the account. You forgot to provide a last name. This field is required. Please do that and try to submit -- and click the Submit button again. From an application perspective, the request was a failure. But from an HTTP perspective, the request was successfully processed. So, those types of scenarios are normal in Web applications.

Summary

In this module, we learned that HTTP messages always come in pairs. First there is the request, then there is the response. The information inside of these messages is all in readable text. And there are lots of tools that you can use to inspect HTTP requests being made on your machine. Fiddler is one such tool. That's the tool I demonstrated. But most browsers also have developer tools that you can plug in, to give you a view of the HTTP requests that are coming and going. That includes Firefox, Chrome and Internet Explorer. The messages that we've looked at, they've all been engineered, to make sure that both parties understand what they're receiving. The first line of that message is always explicit about its intent. There's either an HTTP method there in a request that expresses what that message wants to do -- get something or post something. And there's a response with a status code on that first line that says, did this work out or not, or what do I do next. In the next module, we're going to go one step lower and actually take a look at when these connections are opened to send these messages and how they travel across the network.

HTTP Connections

Introduction

In the previous module, we looked at HTTP messages, and we saw examples of the textual command that flowed between the client and server in an HTTP transaction. But, how do those messages actually move through the network? When are the network connections opened? When are the network connections closed? Those are the types of questions that we'll be answering in this module, as we look at HTTP from a lower level perspective. We're going to look at network protocols like the transmission control protocol and use a tool to analyze the network during an HTTP transaction. We're also going to get a feel for what it might be like to write the code in a web browser that makes an HTTP request.

Whirlwind Networking

To understand HTTP connections, we have to know just a bit about what happens in the layers underneath the HTTP. Network communication protocols, the things that move information around the internet, they're like most business applications, they consist of layers, each layer in a communications stack is responsible for a specific and very limited number of responsibilities. For example, HTTP is what we call an application layer protocol because it allows two applications to communicate over the network. Quite often, one of the applications is a web browser and the other application is a web server, like IAS or Apache. And, we saw how HTTP messages allow the browser to request resources from the server, but those HTTP specifications don't say anything about how the messages actually move across the network and reach the server. That's the job of lower layer protocols. A message from a web browser has to travel down through a series of layers and when it arrives at the web server, it travels up through a series of layers to reach the web server process. So, the layer underneath of HTTP is what we call a transport layer protocol. Most all HTTP traffic travels over TCP, which is short for transmission control protocol, although that's not technically required by HTTP. When a user types a URL into the browser, the browser first has to extract the host name from the URL and the port number, if there is any, and it opens a TCP socket, by specifying that server address, which was derived from the host name and the port, which as we saw, will default to port 80, then it just starts writing data into the socket. We're actually going to see code that does this in a little bit. All the browser needs to worry about is writing the proper HTTP message into the socket. The TCP layer accepts that data and ensures that the data gets delivered to the server without getting lost or duplicated. TCP automatically resends any information that might get lost in transit. The application doesn't have to worry about that and that's why TCP is known as a reliable protocol. In addition to this error detection, TCP also provides flow control, meaning TCP will ensure the sender does not send data too fast for the receiver to process that data. Flow control is very important in this world where we have different kinds of networks and devices. So, in short, TCP provides many of the vital services that we need for the successful delivery of HTTP messages, but it does so in a transparent way and most applications don't need to worry about TCP at all, they just open the socket and write data into it. But TCP is just the first layer beneath HTTP. After TCP at the transport layer comes the IP as a network layer protocol. IP is short for internet protocol. And so while TCP is responsible for error detection, flow control, and overall reliability, IP is responsible for taking pieces of information and moving them through all the switches, routers, gateways, repeaters, and all of these other devices that move information from one network to the next and all around the world. IP tries very hard to deliver the data at the destination, but it doesn't guarantee delivery, that's TCP's job. To do its work, IP requires computers to require an address, which is the famous IP address, an example would be 208.192.32.40, that's an IP version four address. IP is also responsible for breaking data into packets, which sometimes we call them datagrams, and sometimes it needs to fragment and reassemble those packets so they're optimized for a particular network segment. Now, everything we've talked about so far happens inside a computer, but eventually those IP packets have to travel over a piece of wire or fiber optic cable or wireless network, or over a satellite link, and that's the responsibility of the datalink layer. A common choice of technology, at this point, is Ethernet. With Ethernet, these IP packets become frames and protocols like Ethernet become very focused on ones and zeros and electrical signals. Now, eventually that signal reaches the server and it comes in through a network card where the process is reversed, the datalink layer delivers the packet to the IP layer, which hands it over to TCP, which can reassemble the data into the original HTTP message sent by the client and eventually push it into the web server process. It's all a beautifully engineered piece of work. It's all made possible by standards. (Pause)

Programming Sockets

I thought it would be interesting to write a little C sharp application that kind of behaves like a web browser. It's going to kind of behave like a web browser in the sense that I'm going to give it a URL and it's going to go off and use some of the lower level classes in dot net to retrieve that resource and display it here on the screen. So, it's called console application four because it's a very simple exercise, we're not going to do a lot of error checking, we're only going to be able to display textual resources, but we will get to see how to make an HTTP request and process the response from a low level. And, we're not going to use any of these high level classes in dot net like web request, which makes this really easy, we're going to use the lower level stuff like sockets. So, I already have some code written in this application. One of the parts that is written is that we're going to assume that the user passes the URL to the resource that they want to retrieve as a command line parameter, and we're going to grab that parameter and initialize a new URI class with it. Now, URI stands for uniform resource identifier and it's a little more generic than a URL because it can identify a resource by either name or location, but a URL is a URI, therefore when the user types in a URL, we can stick it in the URI class and this class is helpful because it helps us parse apart that strain. And the three basic steps to this program are to grab the URL, go out and retrieve the resource identified by the URL and then we'll write it out to the screen. So, again, just textual resources. So, the second step here, get resource, what we're going to need to do is, just like a web browser, find out some information about what the user is trying to connect to. For instance, we're going to have to find out what the host is, and the URI makes this very easy because I can just go to the host property and it's going to parse out odetocode.com. We can also find what resource they want to connect to. That's going to be in the path inquiry, and if you remember the first module in this course, we talked about the path portion of the URL, the query string portion of the URL, we're going to bring both of those in and put it in this resource, which will be a string typed variable. And, now we need to find out more information about the host. We need to find out what the IP address is of the host, we need an IP address in order to be able to create a socket that will connect to that host. And this is fairly straight forward. What I am going to do is pull out the host entry, by connecting to a DNS server and this is very easy because it's just a static method on the DNS class. I give this method the host name, it's going to return a host entry which can be one or more addresses that are available for this server. So, we're going to take this entry and we're going to pass it into a method called create socket, and it's our job inside of here to connect to HTTP port 80 and actually open up a socket where we can exchange HTTP messages. So, it would also be good if I took the URL that the user typed in and check to see if they used a port other than 80, but again this is a simple program, we're just going to assume that they went for an 80. So, I'm going to need an end point to connect to that server, and there is a class within the dot net framework called IP end point that I can use to, basically, construct a data structure that describes what I want to connect to. And, you can see it needs an address and it needs a port number. So, the address I'm going to get from the host entry, and again you can have multiple IP addresses available for a server. Since this a quick and simple program, I'm just going to use subzero or just just give me the first address that's the address that we're going to use, and we're going to connect using port 80. And once I have an end point, I can construct a socket. First, I have to specify the address family, this would be something like does it use IP version four or IP version six, the end point will tell us that. I want it to be a streaming socket, and I want it to use a protocol that we've talked about already and that's the transmission control protocol, TCP. Now, the socket has everything that it needs to connect, so we can tell it to connect, and tell it to connect to the end point that we've constructed earlier. And that's a block and call by the way, so the code will be stuck on that line until it actually makes that connection to the server or fails or times out, and we can check if it actually did connect by walking up to the connected property on the socket and if that return is true we have a a real socket available, I'll go ahead and return that socket for someone else to use. Now back here in get resource, we've implemented create socket. When we get a socket what we have to do is send the HTTP request message to that host for that particular resource. So, if we look at send request then this bit of code should look relatively familiar in the sense that it looks like an HTTP request message. The only type of message we're going to send, the only operator we're going to use is the get operator, we want to use the HTTP one dot one specification, we have to fill in the path and query string here of what we want, we're going to fill in the host name so that we have the required host header and then the request message basically has to end with two consecutive carriage return line feeds. Now, we have to take this message, which is a which is in a string representation and turn it into bytes to actually write it into the socket, and that's easy enough with encoding.asking.getbytes that's going to turn it into the bytes using an asking encoding, and this is the beautiful part right here, all we've really been worried about is formulating the message. We can walk up to the socket and say here's the data that you have to send, now everything is on you, you make sure it gets delivered, it doesn't get duplicated, it doesn't get lost. Sure, there might be a network error and the whole operation might fail, but if something strange happens and a few bits get changed on the way over to the other server because of electrical interference, then TCP will detect that and just automatically resend data for me, I don't have to worry about it. Again, this is a blocking call so it will sit on this line of code until the send finishes. There's also asynchronous versions that you can use and that would be better if you were writing a real application, but once we've sent the data, then we'll come back up here, once we've sent off the HTTP request, then we wait for the HTTP response, we're going to listen for that response on the same socket. So, if I scroll down to this bit of code, then it looks like I threw in an extra curly brace here for some reason, but it's basically having a buffer where we can stuff information into, we're going to tell the socket to receive as much data as this buffer will hold, but we're not sure if we need 256 bytes or if we need one million bytes, so we're just going to set up a loop and keep refilling that buffer, which will be full of bytes, and each time the buffer fills up, we'll use the encoding class again to take the data that's inside of that buffer and convert it into a string, I'll pin it here to a string builder, and we're going to keep doing that until the socket says well we did a receive but didn't receive anything the response is finished. And, at that point, we can take the result, turn it into a string, and so now if I do a build, we should be able to execute this program and there it returns some HTML from the server. So really not a lot of code to write at this low level, and of course we haven't built a really robust web browser, we've just built something that can get a resource, a textual resource. But I just want you to think of all the beautiful things that are happening behind things like socket.send the error detection, the flow control, the fragmentation and reassembly of packets, moving electrical signals across wires and then across the country and processing all this, it's all made very very easy by the network protocols that are in place and the programming API that they have specified and that is implemented by the dot net framework. (Pause)

Handshakes with a Shark

If you want some visibility into TCP and IP you can install a program like Wire Shark. This is a free program for OSX and Windows. In the last module we used Fiddler to examine HTTP messages that were being exchanged between the client and the server, but Wire Shark goes much deeper than this. You can examine every bit of information that's flowing through your network interfaces. Using Wire Shark, we're going to be able to see the TCP handshake, these are the TCP messages that are required to establish a connection between the client and the server and that happens before the actual HTTP messages can start to flow. You can also observe TCP and IP headers. They add 20 bytes each on top of every message, and what I'd like to do is take a look at the program that we just wrote while Wire Shark is running to see what gets exchanged. I've configured the application to run in the debugger now, the command line argument is going to specify a URL that says just get the root resource for www.odetocode.com, now if you remember from module two, everything on odetocode.com make sure that it gets and redirects resources to make sure they come from odetocode.com and not www.odetocode.com so this request should generate a simple redirect response from the server. So, let's get started with Wire Shark. The first thing I'm going to do is specify some capture options. In the capture options I'm going to use a filter here to say only capture stuff between this computer and the host odetocode.com and it's going to be able to figure out what the IP address is for that host and capture all the traffic. I don't want to capture everything, because when you run Wire Shark without any filter you'll find out that your network card is probably busier than you thought it was with all the little services and synchronization and chat windows, they're all connecting to something. So, let's start Wire Shark and let me come into Visual Studio and start our program with the debugger. So, we're at the point where we are about to connect with a socket, let me just step over that line of code and we see that there are three new entries behind us here in Wire Shark. We'll come back and look at them later. Now, let me advance to the point where we actually send off the get request, this is the HTTP request message, I will step over that. We have three new entries that popped up here in Wire Shark and let's just run to completion where we read the results, and finally just exit the application. And, let me stop the capture, just so we don't get any more messages that are captured. All right, not let's drill into what we have. The first message here is a message that was sent from 192.168.1.134, which is the private IP address of my computer here in the local area network at the Allen Estate in Western Maryland. That message was sent from me to 96.31.33.25, which I'm going to assume is the IP address of the server hosting odetocode.com and it was sent using the TCP protocol, there's no HTTP involved yet. Here in the bottom of the window you can see the nitty gritty details of what was sent using the transmission control protocol and also what TCP put into the message, what did the IP protocol put into the message, what happened at the Ethernet level. And I'm not going to go into all the details of sequence numbers and datagram headers and the like because we're primarily focused on HTTP and what I really want you to take away from this discussion is that we exchanged three messages before the HTTP traffic started to flow. So, three messages, this is known as the TCP handshake, the three step TCP handshake, it's the handshake protocol to make sure both the server and the client are in agreement about how to communicate. It's not until the handshake completes that we start sending HTTP messages. So, this was the message that I sent out that was the get request or host www.odetocode.com you can see that that get message, that HTTP request message is layered into a TCP message, which also IP adds its own headers here and so does Ethernet and that's what the layered communications stack, that's how it does its job. It encapsulates and surrounds data from a higher level inside of information that it uses to do, let's say error detection at the TCP level or routing at the IP level. So, that was the outgoing request message, here was the incoming response messages that says, essentially, this is a status code 301, the resource is moved permanently, my computer acknowledged that and then the final line that is in red is a bit of a concern, and it sort of indicates that I didn't write my HTTP client correctly. What happened is my client expected the server to keep that socket open and stay connected, but something must have happened on the server side and it closed the socket. I was expecting it to be open, at the TCP level this generated a reset, and that actually leads nicely into the next discussion about HTTP connections and HTTP performance. As you can see, HTTP relies almost entirely on TCP to take care of all the hard work, and TCP does involve some overhead, like the handshakes that we can see here in Wire Shark. And thus the performance characteristics of HTTP, they're mostly also going to rely on the performance characteristics of TCP and that's what we're going to talk about next, and we'll also talk about why that red line appears, why did the server close the connection on me.

On the Evolution of HTTP

In the early days of the web when we had the original HTTP specification, most resources were textual and you would go out with your computer and you would request a document from a web server, the web server would give it back to you, and you could go off and read for five minutes before maybe you'd click a link on that document and request another one. The world was very simple then, and it was really easy for a browser to open a connection to a server, send a request, receive the response, and then just close the connection. The idea was, why do we keep connections open if we only need them once every five minutes? But for today's web, most web pages require more than a single resource to fully render. Every webpage I go to is going to have one or more images, one or more JavaScript files, one or more CSS style sheets, it's not uncommon to request a webpage and have that spawn off 30 or 50 or 100 additional HTTP requests to retrieve all the resources associated with that page. So, if today's web browsers were to open connections one at a time like this and wait for each resource to fully download before starting the next download, then the web would feel very very slow because the internet's fully of latency, signals have to travel long distances and wind their way through different pieces of hardware, and as we saw on Wire Shark, there's also some overhead in establishing a TCP connection, the three step handshake. Let me demonstrate what happens if I'm running Fiddler and I go out and I refresh the Wire.com homepage. So, there's the initial request and as it's pulling that down, it's discovering other resources to download, things are filling in slowly now, you can see the requests rolling by. A lot of them have a 304 status code, because they are for images that have already been downloaded and cached by IE, but now we're up to over 100 requests 120 requests to fully render the Wire.com homepage. So, this evolution from simple documents to complex pages has required some ingenuity in the practical use of HTTP.

Parallel Connections

Most user agents, aka web browsers will not make requests in a serial one by one fashion, instead they can open multiple, parallel, connections to a server. So, for example, when downloading the HTML for a page, the browser might see two image tags in that page, so it can open two connections to download the images simultaneously. Hopefully that will cut the amount of time needed to display the images in half, but it's not always perfect like that, and the exact number of parallel connections that a browser will make depends on the browser and how it's configured. For a long time we considered two as the maximum number of parallel connections a browser would create. We considered two the max because the most popular browser from any year is Internet Explorer six, able to only allow two simultaneous connections to a single host, and to be fair, Internet Explorer six was really just following the HTTP specification, which says, a single user client should not maintain any more than two connections with a server. But, a lot of people found ways to work around this limitation, or at least perceived limitation, to increase the number of parallel downloads. So, for example, this two connection limitation is per host, per host name, meaning IE six would happily make two connections to www.odetocode.com and two connections to images.odetocode.com, so by hosting images on a different server, then you can have four parallel requests, and that different server just needed to be a different host name. Ultimately, your DNS records could point all four requests to the same physical server, but IE six was just using that two connection limit per host name, it would happily open four connections in that scenario. A lot of people also figured out how to go into the registry and make IE six support more connections. Now, things are a bit different today, and most modern web browsers will use a different set of heuristics when deciding on how many parallel connections to establish. So, for example, in IE eight you can now have up to six concurrent connections per host. And the real question then is, well if six is better than two then why don't we just open 100 parallel connections? Well connections and parallel connections, they're they're going to obey the law of diminishing returns. If you have too many connections open it can saturate and congest the network, particularly when you're dealing with mobile devices and unreliable networks. So, having too many connections can hurt performance, and also a server can only accept a finite number of connections. So, if 100,000 browsers simultaneously create 100 connections to a single web server, I'm sure that bad things are going to happen. Still, using more than one connection per agent is better than downloading everything in a serial fashion and parallel connections are not the only performance optimization in HTTP.

Persistent Connections

In the early days of the web, it was easy for a browser to open and close a connection for each request it sent to a server and that is literally create a new socket, connect it, send a request, get a response, and close the socket. That was in line with HTTP's idea of being a completely stateless protocol. But, as we've seen, the number of requests per page has grown and the overhead generated by TCP handshakes and the in memory data structures required to establish each socket connection, it's not trivial. So, to reduce the overhead and improve performance, the HTTP 1.1 specification suggests that implementation should implement persistent connections and actually persistent connections are the default type of connection in HTTP 1.1. A persistent connection stays open after the completion of one request response transaction. That leaves the browser with an already open socket it can use to continue making requests to the server, without the overhead of opening a new socket. Persistent connections also avoid the slow start strategy that is part of TCP congestion control and that's going to make persistent connections perform better over time. So, in short, these persistent connections that we have today with HTTP, they typically reduce memory usage, reduce CPU usage, reduce network congestion, reduce latency, they generally improve the response time of a page, but like everything in software there is always a downside. A server can only support a finite number of connections, the exact number depends on the amount of memory, the configuration of the server, the performance of your application. There's a whole host of variables there. So, it's difficult to give an exact number, but generally speaking, if you're talking about supporting thousands of concurrent connections, you're going to have to start testing to see if a server will support that load. Many servers are configured to limit the number of concurrent connections far below the point where the server will just fall over. And that configuration is as much a security measure as anything else. It helps to prevent a denial service attack because it's relatively easy for someone to create a program or a script that will just open thousands of persistent connections to a server and not do anything with them, or send a minimal amount of data over over the connections, so persistent connections are performance optimization, but some people also see them as a vulnerability. So, thinking along those lines of persistent connections possibly being a vulnerability, we've talked about them remaining open, but for how long? In a world where you have infinite scalability, you can keep the connections open as long as you want, but because a server supports a finite number of connections, most servers will be configured to close a persistent connection if it's idle for some period of time. For example, in the most recent Apache release, I know the default time out is five seconds. User agents can also close connections after a period of idle time. If you want visibility into actual physical connections that are being opened and closed you can use a network analyzer like Wire Shark. In addition to aggressively closing persistent connections, most web server software can also be configured to not enable persistent connections. That's common with shared servers. Shared servers are sacrificing performance because they're hosting hundreds of websites on the same machine, they're sacrificing performance to allow as many connections as possible, and because persistent connections are the default connection style with HTTP 1.1, a server that does not allow a persistent connection has to include a connection header in every HTTP response. That response header is connection close, that's a signal to the client that the connection will not be persistent and it should be closed as soon as possible, the agent is not allowed to make a second request on the same connection. That's something I should have checked for in my HTTP client, I should have seen if there was a connection close header, and immediately closed the socket after receiving the data. Where can we see the difference? Well let's go to Fiddler and have Fiddler running and capturing and let's actually make a request to odetocode.com and if we inspect that one of the headers that we'll see in the response is the connection close header because it is hosted on a shared website. And, just for comparison, if we go to Pluralsight.com, then in the response that comes back, we will not see connection close, it's going to allow a persistent connection. One additional optimization that I want to mention is the pipeline connection. Now, persistent and parallel connections are both widely used and supported by clients and servers, but the HTTP specification also allows for pipeline connections, which are not as widely supported by either servers or clients. In a pipeline connection a user agent can send multiple HTTP requests on a single connection and send those off before it even waits for the first response. Pipelining allows for more efficient packing of requests and to packets and can reduce latency, but like I say, it's just not as widely supported as parallel and persistent connections. (Pause)

Summary

In this module we took a look underneath the HTTP messages and got an idea of how messages actually move across the internet. We talked about some of the performance optimizations that are made possible by the HTTP specifications and we wrote a little program to open a socket, sent off an HTTP request, and received an HTTP response. In the next module we're going to take a step back and look at the internet from a wider perspective and talk a little more about the hardware that is out there on the network that can influence our HTTP messages and HTTP transactions.

HTTP Architecture

Introduction

Hi, this is Scott Allen, and this module is about HTTP and Web architecture. In the first module of this course, we talked about resources, but I mostly focused on URLs and how to interpret a URL. But resources are really the centerpiece of HTTP. And now that we understand HTTP messages, methods, and connections, we can return to look at resources in a new light. In this module, I want to show you the essence of working with the resources and messages and how the architecture of the Web really works.

Resources Redux

It's really easy to think of a Web resource as being a file on the Web server's file system. But thinking along those lines really disrespects the true capability of the resource abstraction. Many Web pages do require physical resources on a file system: JavaScript files, images, stylesheets. However, consumers and users of the Web don't really care for those background resources. Instead, they care about the resources they can interact with and -- more importantly -- the resources they can name; for instance, resources like the recipe for Beef Wellington, the search results for deep dish pizza, and Patient 123's medical history. All of these resources are the types of resources that we build applications around, and the common theme in the list is how significant each of these items is, if they're significant enough that we want to identify them and name them. And as soon as we can identify a resource, we can also give the resource a URL for someone to locate the resource. And a URL is a handy thing to have around. Given a URL, I can locate a resource, of course. But I can also hand the URL to someone else, by embedding it in a hyperlink or sending it in an e-mail. But there's many things I cannot do with a URL. Or, rather, there are many things that a URL cannot do. For instance, a URL cannot restrict the client or the server to a specific type of technology. Everybody speaks HTTP. It doesn't matter if your client is in Ruby and your server application is written in C++. It doesn't matter. Also, a URL cannot force the server to store the resource using any particular technology. The resource could be a document on the file system, but a Web framework could also respond to an incoming request for that resource and build it, using information stored in files, stored in databases, retrieved from other Web services, or simply derive the resource from the current time of day. Another thing a URL cannot do is specify the representation of a specific resource. And a resource can have multiple representations. There could be one in HTML, one in PDF, one in English, one in French. I demonstrated this in the second module. We sent different accept language headers to a server, and we watched it respond in different languages. I also wanted to show you what happens when you send different accept types; that is, specify the representation that the client is willing to receive. To do that, we'll go back to the console mode application that we wrote, in the last module, to make HTTP connections using sockets. And first, let's just demonstrate that we can go to Pluralsite.com, and that returns an HTTP response of 301. That's the permanent redirect, saying, no, you don't want to go here, you want to go to http://www.pluralsite-training.net. But I have modified this program slightly. What I've done is when we send off the HTTP get request, I'm putting an accept header in here that says I want to accept application/XML. Now, the server might not have an XML representation for any particular resource. In fact, it turns out here that the server returned the content type of text/HTML. And sometimes, as a client, you have to deal with what you're given. But there are Web services out there, and Web applications, that respond to that accept header appropriately. For example, Netflix has a Web service out there, and it's on odata.Netflix.com/V2/catalog. What this Web service will do is respond with, What can I get to in Netflix? So if I send off that request, what I'll get back is XML, just like I requested. It sends back a content type of application/XML. And what we're looking at is the catalog, or the things I can get to inside of this Web service. I can get to titles. I can get to people. I can get to the different languages. But now let me go back into Visual Studio and change this accept header ever so slightly. Instead of application/XML, I want application/json. And I'll rebuild the program. And now there's nothing in this URL, odata.Netflix.com, that says I want a json representation, so I can use it from JavaScript. That's part of the content negotiation that is in the request headers. So when we run the application, what I get back now is a content type of application/json. This is something that's very easy to parse and consume from JavaScript. I might use the XML, if I'm querying this from C# or C++ or Java. But for some of the dynamic languages and languages that support json serialization and deserialization -- like JavaScript -- then json is a perfect choice. Now, there's one more thing that a URL cannot do. It cannot say what a user wants to do with a resource. A URL doesn't say if I want to retrieve a resource or edit a resource. That's the job of the HTTP request message, to describe the intention of the user. And it does that using one of the HTTP standard methods. And as we talked about in Part 2, there's a limited number of those methods. The most popular ones are get and post, but there's also put and delete. Now, when you start thinking about resources and URLs, as we are in this module, you start to see the Web as part of your application, and it's a flexible, architectural layer that you build on. For more insight into that line of thinking, see Roy Fielding's famous dissertation entitled {italic}Architectural Styles and the Design{plain} {italic}of Network-Based Software Architectures.{plain} That's the research paper that introduced the representational state transfer style of architecture. And it goes into great detail about the ideas and concepts I'm talking about here in this clip and in the next one.

Architectural Qualities

So far, we've been focused on what a URL cannot do, when we really should be focused on what a URL can do. Or rather, I want to focus on what a URL plus HTTP can do, because they work beautifully together. In his dissertation, Fielding describes the benefits of embracing HTTP. These benefits include scalability, simplicity, reliability, and loose coupling. HTTP offers those benefits, because, in part, you can think of the URL as a pointer, or a unit of indirection, between a client and a server application. Again, the URL itself, it doesn't dictate a specific resource representation. It doesn't dictate the technology implementation. It doesn't dictate the client's intention. Instead, the client expresses the desired intention and representation in an HTTP message. An HTTP message is very simple. It's plain text, as we've seen. The beauty of that is how both the request and the response are fully self-describing. They're standardized. They're easy to parse. The request message includes the HTTP method, which describes what the client wants to do; the path to the resource; and additional headers that provide information about what representation that I want. The response includes a status code to indicate the result of a transaction, but also includes headers with cache instructions, the content type of the resource, the length of the resource, and other valuable metadata. Because all of the information required for this transaction is contained in these messages, and because that information is visible and easy to parse, HTTP applications can rely on a number of services that provide value, as a message moves between the client and server.

Adding Value

As an HTTP message moves from the memory space of a process on one machine to the memory space of a process on another machine, it can move through several pieces of software and hardware that inspect and possibly modify that message. One good example is the Web server application itself. A Web server, like IIS or Apache, will be one of the first recipients of an incoming HTTP message on a server machine; and as a Web server, it's responsible for routing messages to the proper application. So here we had a Web server that was hosting three different sites, using three different technologies. It received an incoming HTTP request message. It needs to peek inside that message, look at a host header, and it can use that to figure out which application should receive and process that message. That's a fairly common scenario and something that's really easy to configure in IIS and Apache. But all these Web servers, they can also perform additional actions with the message, like logging to a local file. So as the Web server is sitting there and a message comes through, it can take that message and record it in a log file, as many details as you want. And likewise, when the application creates the HTTP response message, the server also has a chance to interact with that message on the way out. That could be a simple logging operation, but it could also be a direct modification of the message itself. For example, a server knows that the client supports gzip compression, because a client can advertise that fact through an accept encoding header in the HTTP request. What compression allows you to do is take a 100-kilobyte resource and turn it into a 25-kilobyte resource. That means it's going to transmit much faster. And you can configure many Web servers to automatically use compression for certain content types -- typically, text types. And this happens without the application itself worrying about compression. Compression is an added value provided by the Web server software itself. The applications don't have to worry about logging the HTTP transactions or adding compression. And that's all thanks to the self-descriptive messages that allow pieces of infrastructure to process and transform these HTTP messages. This type of processing can also happen as the message moves across the network, too.

Proxies

These self-describing, visible HTTP messages allow us to use proxy servers. A proxy server is a server that sits between a client and a server. A proxy is mostly transparent to the end user. So you think you're sending a request directly to a server, but the HTTP request message is actually going to the proxy, which will take that message and forward it to the server that you want it to get to. It can also, then, wait on the response from the server and forward it back to the client. But before forwarding either of those messages, the proxy can also inspect the message and potentially take some initial actions. For example, one of the clients I work for uses a proxy to capture all HTTP traffic leaving the office. They don't want employees and contractors spending their time on Twitter and Facebook, so HTTP requests to those servers will never reach their destinations, and there's no tweeting or Farmville inside the office. That's an example of one popular role for a proxy server, which is to function as an access control device. But a proxy server can actually be much more sophisticated than just dropping messages that are trying to get to specific hosts. Any firewall could do that. A proxy server can also inspect messages, to remove confidential data, like strip out the Referer header from HTTP messages, if that Referer points to an internal resource inside the company network. An access control proxy can also log all the HTTP messages, create audit trails on traffic. And many of these access control proxies require a user to log in, before they can access the Web. That's a topic we'll look at in the next module. The proxy I'm describing here is what we would categorize as a forward proxy. A forward proxy is usually closer on the network to the client than it is to a server, usually one or two network hops away from any particular client. And forward proxies usually require some configuration in the client software or Web browser to work. The idea is that the forward proxy is providing some services, to benefit just the users in a particular location, not the Internet as a whole; so the users at a specific company or in a specific office building, or the customers of a single Internet service provider. Another category of proxy servers is the reverse proxy. A reverse proxy is a proxy server that's usually closer to a server than it is to the client, and these are usually completely transparent to the client. A reverse proxy exists to provide some benefit to a specific website, and it can indirectly benefit all the users on the Internet, because all the requests coming to servers for that website are coming through the reverse proxy. Now, both these types of proxies -- forward proxies and reverse proxies -- they can provide a wide range of services. For example, if we return to that gzip compression scenario we talked about earlier, a proxy server has the capability to compress response message bodies. A company might use a proxy server for compression, to take some of the load off of the server where the application actually lives. Now, neither the application or the Web server software itself has to worry about compression. Instead, it's a feature that's layered in, via a proxy server. That's the beauty of HTTP.

Proxy Services

Proxies can perform a wide range of services. For example, load balancing. This is where a proxy takes incoming messages and distributes them to one of several Web servers on a round-robin basis, or by knowing which server is currently processing the fewest number of requests. For example, the proxy server here at the foodchopper.com might take a request from the Internet and send it to Web2. Then when the next request arrives, since the previous request went to Web2 for processing, it might send this request to Web1, in hopes that the load will stay even across those servers. Some proxies and load balancers can even look at the servers to see how much CPU and memory they're using and route messages to the servers with more resource headroom. A proxy server can also direct requests to different servers, depending on the content types. For example, a company might put all their images and static assets on a server optimized for serving those types of assets. And all dynamically-generated content on servers optimized for PHP or ASP.NET or RubyOnRails, that's another common proxy operation. The proxy server can make sure that those requests get directed to the right server. There's also proxy servers that implement SSL acceleration. This is where the proxy server actually does the encryption and decryption of HTTP messages, taking that load off of the Web servers. We'll talk more about SSL in the next clip. Proxies can also add an additional layer of security, by filtering out potentially dangerous HTTP messages. Specifically, some proxies can look for messages that might have cross-site scripting attacks or SQL injection attacks embedded inside of them. And finally, caching proxies will store copies of frequently-accessed resources and respond to messages requesting those resources directly. That typically improves performance. We'll go into more detail about caching in the next clip. But before we leave here, I just want to point out that proxies do not have to be a piece of hardware. For example, the tool we were using previously, Fiddler, it has the ability to intercept HTTP requests, and it will do that by installing itself as a proxy on the machine. So here you can see it is picking up requests to Google. And the way it does that, you can see it, if you go to the right location, to see how the proxy server is configured in Internet Explorer. And I'll just warn you that you might want to bring out a pencil to write this down, because it's under Tools, Internet Options. You have to go to Connections, click on LAN Settings, and then go to the Advanced section. So what happens, when you launch Fiddler, is it goes in to Windows and figures things in such a way that all HTTP traffic will need to go through this proxy, 127.0.0.1. That turns out to be the loopback IP address, which is essentially the address of this machine that's local host. So all the outgoing HTTP requests are going to my machine on port 8888. That's where Fiddler will sit and listen for an incoming HTTP request, log it, and pass it along to the server, wait for the response, log the response, and pass that back here to Internet Explorer or Chrome, or any other Web browser on the system. Once we close all this and we close Fiddler, you'll notice that suddenly the proxy connection goes away.

Caching

We talked about proxy servers possibly caching information. Caching is an optimization, to improve performance and scalability. When there are multiple requests for the same resource representation, a server can send the bytes over the network time and time again, for each request. Or a proxy server or a client can cache the representation locally and reduce the amount of time and bandwidth required for a full retrieval. Caching can help reduce latency, help prevent bottlenecks, and allow a Web application to survive, when every user shows up at once to buy the newest product or see the latest press release. Caching is also a great example of how the metadata in an HTTP message facilitates additional layers and services. The first thing to know is that there are two types of caches. A public cache is a cache shared among multiple users. A public cache generally resides on a proxy server. A public cache on a forward proxy is usually caching the resources that are popular in a community of users, like the users from a specific company, or the uses of a specific Internet service provider. A public cache on a reverse proxy is generally caching the resources that are popular on a specific website, like popular product images from Amazon.com. Those are public caches. A private cache is dedicated to a single user. Web browsers always keep a private cache of resources on your disk. These are the temporary Internet files in Internet Explorer, or type about:cache in the address bar of Google Chrome, to see the files in its private cache. Anything a browser has cached on the file system can appear almost instantly on the screen. The browser doesn't even have to send off a request. The rules about what to cache, when to cache, and when to invalidate the cache -- that is, kick an item out of the cache, because it's no longer fresh or up-to-date -- they are a little bit complicated and mired by some legacy headers and behaviors. But allow me to point out some of the things that you should know. First of all, with HTTP 1.1, clients and proxies generally want to cache a response that has a 200 okay status code and that is the response to an HTTP get request. Remember, we talked about safe and unsafe methods in an earlier module, and get is a safe method. It's not supposed to change state on the server, and we can send off as many get requests as we like, without messing up the application. Put, post, and delete are considered unsafe, because we use them to change state on the server. We use a post request to submit a credit card transaction, change a profile, log in to a site. Most everyone will avoid caching these types of requests, because bad things can happen. I thought I ordered the 20-piece knife set with the bonus woodcutting board, but it turns out I had a cached response, and the transaction didn't really go through. Now, an application or server can influence the cache settings by using the proper HTTP headers and a response. In HTTP 1.1, this header is the cache control header, although you can also see an expires header in many messages. The expires header is still around and widely supported, despite being deprecated in HTTP 1.1. Pragma is another example of a header commonly used to control caching behavior, but it, too, is really only around for backward compatibility. So we are going to focus in on cache control. An HTTP response can have a value for cache control of public, private, or no cache. A value of public means public proxy servers can cache the response. This response is for anyone. A value of private means the response is really targeted to a single user. So only private caches should keep those; that is, caches in the Web browser. And of course, no cache is telling everyone in the world that they shouldn't cache this response. There's also a no store value, meaning the message might contain some sensitive information and it shouldn't be persisted at all. It should be removed from memory as soon as possible. Now that you know this, how would you use this type of information? Well, for popular requests for shared resources, like a homepage logo, you might want to use a public cache control directive, to allow everyone to cache the image, even proxy servers. For requests that are going to a specific user, like the HTML for the home page and that HTML include to the user's name, you want to use a private cache directive. You don't want other users to have the wrong user name. In ASP.NET, you can control these settings via response.cache. So there's response.cache.setcachability. Set it to public or private or no cache. And there's also an expiration that you can set, because once an item goes into the cache, it may not want to live there forever. You might just want to cache something for 10 seconds, because information changes every 10 seconds. Or you might want to cache it for 10 years, because it's an image and you don't expect it to change. Here is an HTTP response for an image, and you can see it's using a cache control of private, and the expiration is set by the max age value. Max age is specified in seconds. This particular cache control setting is saying that this is good for at least 31 million seconds. That would be about 10 years. And notice there's another header here, Last-Modified. The browser can use that piece of information as a validator. That is, how can I validate that this thing I have in the cache is still good. Well, it could go to the server and request that image again and say, Hey, here's the last date that I have for the image. Has it changed since then? And if it has changed, the server can send the new image. If it hasn't changed, the server can send a special response that we looked at earlier, the 304 response that says this content wasn't modified. What you have in the cache is good. Let's actually go out and use Fiddler and see some of these requests and response headers.

Fiddling with the Cache

Here I have Fiddler running in the background and capturing HTTP requests and responses. I have Internet Explorer open. I'm just going to press F12, to go into the Internet Explorer developer tools, and I'm going to use them to clear the browser cache entirely. Sometimes you need to restart the browser after doing that, but at this point the browser shouldn't have any resources that it has cached locally. And now that that's set up, we will go to MSDN.Microsoft.com and take a look at some of the requests and responses that we got. So first off, here is the homepage for MSDN.Microsoft.com. You can see that the cache control for this response for this resource is private, so it's tailored for a specific individual -- that would be me -- and the person that made the request. I also want to point out that the response was compressed. I know that, because the content encoding is set to gzip. I also know that, because Fiddler will tell me the response was encoded. And if I click here, it will actually decompress and decode the response, so I can look at the HTML and everything that Microsoft.com sent back to me. And let's look at a later request. Something in that home page had a link or pointed to another resource rad.MSN.com. I'm assuming that this is some sort of advertisement, some sort of script that will bring up an ad. Notice that the cache control is set to no cache, must revalidate. That's being very explicit and saying, please do not cache this resource. If my server is serving up scripts that put ads on a page, then I want to know every time that the browser is hitting that page and requesting that script. I don't want the browser to just use a cached version of it, because when they actually make the request, that's something that counts as a hit for that advertisement, so I can earn, like, a quarter of a quarter of a quarter of one penny. There's also a Pragma header -- a Pragma header that says no cache. This is part of what makes caching confusing in HTTP, is that there's so many different headers from different periods in time. Pragma is a very old header that's been around forever, but this is basically just the server trying to be as interoperable as possible, trying to express this to many people as possible, that this response is not supposed to be cached. It also has an Expires header. That Expires header is set way back in time -- January, 1990 -- so everyone should be able to see that, well, this request that we got, it's already out of date. The next time we need it, we'll have to make another request. And let's look at a later request. This one is for some sort of JavaScript resource, because the accept type says it once, application/JavaScript. And the content type that was received was application/x-javascript. Notice the cache control here is public. So, whereas the home page was tailored for a specific user, this JavaScript file is the same for everyone. So it's okay if a proxy server wants to cache this JavaScript file. It's not going to change for 7,200 seconds. Notice there's a validator here. That's the Last-Modified header. It says that this file hasn't changed since January 23rd, 2012. And there's another validator here that we haven't talked about. That's the ETag. So an ETag is an opaque number. There's no way you can look at that number and deduce when the file was last modified, or anything like that. ETags are commonly generated by doing a hash of the resource. So they're really just a number that can be used for comparison. So in other words, if I want to find out if this resource has changed or not, I could go to the server and say, Hey, do you still have the JavaScript file. By the way, the last ETag that I had for this was this value, 07F2, etc. And the server can look at that ETag, compare it to the current ETag -- it's a really easy comparison -- and say, yes it changed, or no, it didn't change. So, use the result that you already have cached. Now, if we go back into Internet Explorer and I do a refresh, then what we'll start to see are some 304 responses. So let me see if I can find the JS file that it requested. So here's -- here on this page refresh is an outgoing request for that JavaScript file again. And perhaps we didn't get the exact same JavaScript file, because the If-Modified-Since is different. But basically, here's a request that goes out to the server that says, I need broker.JS. And by the way, I only need it if it's been modified since September 14th, 2011. And where does it get September 14th, 2011? It that got that from the last modified header, on some previous request. It also sends up an If-None-Match ETag. So again, multiple headers, multiple validates. It's all to be as interoperable as possible. The implementation is going to pick one of those two to match against. And the response, in this case, comes back and says, 304 not modified. You have the latest version. Go ahead and serve it up from script. By the way, the last modified date is still 14th of September 2011. The ETag is still 099C9 something. This can still be cached publicly. And don't worry about it expiring for at least another 7200 seconds. Now, in this case, if that was, let's say, a 20-kilobyte JavaScript file, then we just saved 20 kilobytes of bandwidth, by not having to transmit the entire file back to the client. And then it's up to the user agent to determine, if I refresh again, whether it needs to send off the request for that JavaScript file again and get a 304, or if it's just going to serve something directly from the local cache. Every user agent has different rules and heuristics that it uses. But they pretty much all try to stick by the standards. When things say that something should be cached for this long, they're not going to try to extend the cache.

Summary

In this module, we talked about some of the hardware and network infrastructure involved in making HTTP work; and also how caching headers, client servers and proxies make it all work together, to make the Web reliable and scalable, scalable to the point where websites can support millions of users a day. Again, I'll come back to the point that all of this is made possible by the HTTP messages and URLs. On one hand, these messages, they contain a tremendous amount of information. They are self-describing. They describe the intent of the operation, what representation of a resource is desired, how long to cache the response, when a resource last changed, and what type of content is being transferred. All of that information allows proxies and server software to add value to the Web in a transparent fashion. There's an enormous number of details that I don't have to worry about, as an application or Web service developer. On the other hand, the messages in URLs are defined to not contain information that limits the benefits of the Web. There's nothing that requires me to work in a specific language or technology. There's nothing that requires me to have a certain type of file system or layout of files on the disc. This is the loosely-coupled, flexible architecture that we always want in an application, and it's all made possible by HTTP.

HTTP Security

Introduction

Hi, this is Scott Allen and in this last module we're going to look at security related topics like using cookies to identify users and manage state, we'll look at some common Web authentication protocols and also the HTTPs protocol which is secure HTTP. I want to start by looking at cookies to see how they help us manage state in the stateless hypertext transfer protocol. (pause)

The Stateful Stateless Web

HTTP is designed as a stateless protocol meaning each request response transaction is independent of any previous or future transaction. There's nothing in the protocol that requires a server to retain state or information about a single HTTP request. All the server needs to do is generate a response for that request and every request carry's all the information a server needs to create the response. This stateless nature is one of the reason's that the Web is so successful because it allows us to build those layered services and add those services, the ones that we looked at in the last module. Those are the services like caching, those are all made possible or at least easier because every message is self descriptive and contains all the information required to process that message. Proxy servers and Web servers can inspect, transform and cache those messages. And without caching the Web couldn't scale to meet the demands of the internet. So while HTTP is stateless most of the applications that we build on top of HTTP are highly stateful. For example, a banking application will want to make sure that a user logs in before allowing them to view their account related resources. So every time one of these stateless requests arrives at the banking Website, the application needs to know a little bit about the user needs to know that they've already authenticated and if they haven't it needs to send them to a login page. Another example of a stateful application is when the user wants to open an account and they need to fill out a four step wizard. The application wants to make sure that the user completed the first step of the wizard successfully before allowing them to get to the second step. Those are going to be independent HTTP transaction but the server needs to know about the state of where the user is inside of that four step wizard. Fortunately there's many options for storing state in a Web application. One approach is to embed state in the resources that are being transferred to the client so that the state required by the application or at least some of that state will travel back on the next request. That approach typically requires some hidden input fields and it works the best for short lived state like tracking the state as you move through a four step wizard. Now if you've used asp.net Web forms you've taken advantage of an approach like this because Web forms has viewstate, that's literally the state of the form when it leaves the server. The state of all the controls on that form get serialized into a single value that gets placed into a hidden input named viewstate and when the user interacts with the page and clicks a button, the viewstate is included in the post to the server which can recreate the Web form just like it was when it last left the server by deserializing that value and then it can apply new changes or data bind some updated data. Embedding state in the resource is essentially maintaining or keeping state inside of HTTP messages and in general that's a very highly scalable approach to the Web to maintaining state but it can complicate application programming. Another option is to store the state on the server or behind the server and that style is required for state that has to be around a long time. So when the user submits a form to change their email address, the email address must always be associated with the user so that application can take the address, validate it and sort into a database or a file or call a Web service to let someone else take care of persisting the address. For server session storage many Web development frameworks like asp.net also provide access to a user session. The session may live in memory or it may live in a database but a developer can store information in the session and retrieve that information on every subsequent request from a particular user. Data stored in the session is scoped to an individual user, actually to that user's browsing session, and it's not shared among multiple users. Session storage usually has a very easy programming model and it's only good for short lived state because eventually the server has to assume that the user left the site or closed the browser and the server will discard that information. In session storage if it's kept in memory it can have some impacts on scalability because subsequent requests must go the exact same server where the session data resides. So if you're in a Web form where you have multiple Web servers, multiple machines that are actually serving the resources for one single Website, you have to make sure that the request always end up at the same machine. Some load balancers help to support that scenario by implementing what we call sticky sessions. I'll show you an example of session state in just a bit but you might already be wondering, how can a server track a user to implement session state? If multiple requests arrive at a server how does the server know if these requests are from the same user or two different users or multiple users? In the early days of the Web, Web server software might have differentiated users by looking at the IP address of request message. These days however, many users live behind devices using network address translation, and for that reason and various other reasons you can multiple users effectively on the same IP address and IP addresses can change. So an IP address is not a reliable technique for differentiating users. Fortunately there are more reliable techniques and they rely on cookies.

Cookies

Websites that want to track users often turn to cookies. Cookies are defined by RFC 6265 and this RFC has the stimulating title of HTTP State Management Mechanism. This document describes how a Website can give the user's browser a cookie using an HTTP header. The browser then knows how to send that cookie and the headers of every additional request that it sends to a site. So assuming a Website has placed some sort of unique identifying into the cookie, then the Website can now track a user as they make requests and differentiate one user from another. Before we get into the details of what cookies look like and how they behave, it's worth noting a couple limitations. First, cookies can identify users in the sense that your cookie is different then my cookie. But cookies by themselves do not authenticate users. An authenticated user has proven their identity usually by providing credentials like a user name and password. The Cookies we're going to look at first just give us some unique identifier to differentiate one user from another and track a user as they make request to a site. Secondly, they do raise some privacy concerns in some circles. Some users will disable cookies in their browsers meaning the browser will reject any cookies that a server gives them. And disabled cookies present a problem for sites that need to track users of course and the alternatives are a little bit messy. For example, one approach to a cookieless session is to place some sort of user identifier into the URL, meaning each and every URL that a site gives to a user must contain the proper identifier and the URLs become much larger. That's why we often call this technique the fat URL technique. When a Website wants to give a user a cookie, it uses a set cookie header in an HTTP response. So here's an incoming request to searchengine.com, someone is searching for lyrics. Searchengine.com wants to track users so in the HTTP response to that message, it's going to have a set cookie header. There are three pieces of information in this particular cookie. The three pieces are delimited by semi colons. First there's a collection of name value pairs and these name value pairs themselves are delimited by a dollar sign. That's very similar to how query parameters are formatted into a URL, we looked at that in the first module. In this example the server must want to store the user's first name and last name in the cookie. The second and third pieces of information are the domain and the path, we'll circle back around and talk about those a little bit later. Now a Website can put any information that it wants into a cookie but many Websites will only put a unique identifier, perhaps a Guid. And there's a couple reasons for this. One is, there is a size limitation in cookies of around four kilobytes and secondly, a server can never really trust anything that it stores on the client unless it's cryptographically secured. So while it is possible to store encrypted data in a cookie, it's usually just easier to store an ID. Assuming the browser is configured to accept cookies then the browser will take that cookie and it's going to send it along in any subsequent request it that it makes to searchengine.com that GUID will be there. And when the ID arrives at the server, the server can use that to look up the associated data for that user from an in memory data structure or from a database or from a distributed cache. You can configure most Web application frameworks to manipulate cookies automatically and look up session state for you. Let's take a look at an example of how this works.

Tracing Sessions and HttpOnly

Back in the module where we talking about HTTP messages, I wrote a little application that would allow a user to sign up for our Website. We weren't really doing a sign up, we were letting the user enter their first name, enter their last name, click a submit button and when the resulting post request came into the server we were just going to save the first name and last name into this session data structure and then redirect the user to another page, signedup.cshtml where I was going to prove that we actually did save information in the session so we would write out first name and last name. And that was just intended to simulate some sort of data access. We'd probably save the first name and last name into the database but we'd still need a way to lookup the user's information, we'd still need a way to identify their session and that's where this session object is useful and it's implemented with cookies. So while Fiddler is running in the background to trace the request, let's actually go in to that form and I'll fill out my name, Scott Allen, click submit and we end up here on signedup.cshtml where it successfully retrieved my first name and last name from the session object. Let's go into Fiddler and take a look at the actual request that were sent. This was the initial get request to the server, give me the form where I can sign up, the response didn't include any special headers here because we weren't using a session as yet. Asp.net will create the session on a lazy basis. Other Web frameworks might do the same or they might do it eagerly. Everyone's a little bit different but the programming api and the implementation is usually pretty similar. Here is the post request where I clicked the submit button, the browser sent up a request to post request that included my first name and last name in the HTTP body, we've seen that before. And here's the response that came back. It was the redirect that we expected in HTTP 302 and here's the set cookie header. Asp.net underscore session ID equals some big jumblation of characters. Now there's several observations that I want to make about that particular header but before I do I just want to show you the next request which was, the browser said oh, I wasn't supposed to go here, I was supposed to go to signedup.cshtml. And if we look in that get request you'll see that the browser sent up the cookie with asp.net underscore session ID that is unique to me. If someone else browses the Website they'll get a different session ID and that's how the server can now differentiate user's and look up the proper session state. One other piece that I want to point out is that if I go to a different browser and this time I'll go to Internet Explorer and if we try to go to the same page, signedup.cshtml, it doesn't know my name. And this is because cookies get set in a browser and yes they are per user but if the user is using different browsers or has cookies disabled that can sort of mess things up. So, there was a cookie that was set in Chrome, there was no Cookie set in IE as yet because I didn't go through signup.cshtml as yet. But let's go back and look at this set cookie header. So first of all I want to point out that first name and last name, that was not data that was stored in the Cookie. Instead the only thing stored in the cookie is some sort of session identifier. First name and last name are stored by default with asp.net in memory on the Web server. The Web server's just using this cookie value to look up the proper data structure in memory. Secondly, we might look at this ID, u3ylzcntnrr, etc. and wonder why it's so complicated. Well one security concern around session identifiers is how they can open up the possibility of someone high jacking some other user's session. So for example, here we are in Fiddler and I can see my asp.net session ID and imagine if my session ID was something like asp.net underscore session ID equals 12. Well then I might take a guess that asp.net is just incrementing some sort of session ID so there's one, two, three and four and I might guess that some other user already has a session ID of 11. Knowing that information I could construct an HTTP request message with an asp.net session ID equals to 11 and just see if I could steal or view that html intended for some other user, find out some other user's name or account number. To combat that problem most application frameworks use large random numbers as identifiers. Asp.net uses 120 bits of randomness and stores it into this string of characters. This just makes it more difficult to guess what someone else's session ID would look like. And note that that still doesn't prevent someone who might be sniffing traffic on my network, picking up my session identifier and using it to high jack my session. The only way to prevent that is to use secure HTTP which we'll talk about in just a bit. The other piece that I wanted to talk about in this set cookie header, first of all there's no domain, we'll talk about what the default domain setting is. I want to point out the HTTPOnly flag here because another security concern around cookies is that they are vulnerable to a cross site scripting attack. In a cross site scripting attack a malicious user injects Java script code into someone else's Website and if the other Website sends that malicious script to their users, a script has the ability to modify and inspect and steal cookie information. So a malicious script could find my asp.net session ID and perhaps use an Ajax request to send it off to some other server where someone's recording these things and then they know my session ID. To stop that sort of problem it was actually Microsoft that introduced this HTTPOnly flag and it's now a standard. And what the HTTPOnly flag tells the browser, the user agent, is that it should not allow script code to access this cookie. This cookie exists only to put into HTTP request and travel in the header of every HTTP request message. So browsers that correctly implement HTTPOnly, and most of them do these days, will not clients like Javascript to read or write this cookie on the client. And that is a very good thing because cross site scripting attacks are very popular these days.

Cookie Paths, Domains, and Persistence

So far all the cookies we've looked at are what we would call session cookies. Don't confuse that with the session object or session data on the server. It's a specific type of cookie that we call a session cookie because it exists for only a single user session. It's get destroyed when the user closes their browser. So in this example we've gone to searchengine.com and it used a set cookie header to give the browser a cookie with a GUID value inside of it and every subsequent request that the browser makes to searchengine.com it's going to pass along that GUID value until the user closes their browser and then the browser simply forgets about that cookie. A persistent cookie is the other type of cookie and it can outlive a single browsing session because the browser, the user agent, will typically store that cookie to the file system to disc. So I can shut down a computer and come back one week later, go to my favorite Website and a persistent cookie would still be around for the first request. The only difference between the two is that a persistent cookie needs an expires value. So what we're looking at right here, I know it's a session cookie because there is no expires value in the cookie. However, this cookie is a cookie that has an expires value. This cookie is going to be around until July 9, 2012. The next piece that I want to talk about is this domain value. I've said that once a cookie is set by a Website, the cookie travels to that Website with every request. However, not all cookies travel to every Website. The only cookies a user agent should send to a site are the cookies that the site gave it. It wouldn't make sense for cookies from Amazon.com to be an HTTP request to Google.com. That type of behavior would only open up additional security and privacy concerns and Google.com really shouldn't understand what's inside of Amazon.com's cookies anyway. So if you set a cookie in a response to www.searchengine.com the resulting cookie should only travel in requests to www.searchengine.com. A Web application can change that a little bit and restrict the cookie to a specific host or domain or even to a specific resource path by using this domain and this path attribute. The domain attribute basically allows a cookie to span sub-domains. In other words, if you set a cookie from www.searchengine.com the browser's only going to deliver that cookie to www.searchengine.com. But if I set a cookie and I say that the domain is dot searchengine.com that allows the cookie to travel to any URL in the searchengine.com domain. That would include images dot searchengine.com and help dot searchenginel.com. So you cannot use this domain attribute to span domains, in other words, if the browser makes a request to searchengine.com and it tries to set a cookie with a domain set to Microsoft.com that's not legal, the user agent should reject the cookie. But if I go to www.searchengine.com it will be allowed to set the cookie domain to dot searchengine.com which is essentially telling the browser don't just send this to the www server, send it to anything that ends with dot searchengine.com. The path attribute, that's another way to restrict a cookie to a specific resource path. So in this example the cookie will travel to basically anything under dot searchengine.com but if we sent that path to something like slash stuff or slash images, that would be telling the browser only send this cookie to something on searchengine.com when the URL path starts with slash stuff or slash images. Path settings can help you to organize cookies when there's multiple teams building Web applications in different paths.

Basic and Digest Authentication

Cookies are good for tracking and differentiating one user from another user but sometimes we need to know an individual user's identity. We need to know exactly who they are. A process of authentication forces a user to prove their identity by entering a user name and a password or an email and a pin or some other type of credentials. With the Web, authentication follows a challenge response format. A client will request a secure resource from the server and the server will challenge the client to authenticate by sending back an HTTP response with a challenge inside of it. The client then needs to send another request and include authentication credentials for the server to validate. If the credentials are good that request will succeed. The extendability of HTTP allows HTTP to support various different authentication protocols. In this module I'm going to briefly look at the top five which include, basic authentication, digest, Windows, forms and open ID. Of these five, only two are official in the HTTP specification, the basic and digest authentication protocols and we'll first talk about basic authentication. With basic authentication the client requests a resource with a normal HTTP message and the Web server, most of them will let you configure access to specific files and directories. You can allow access to all anonymous users or restrict access to the only specific users or groups can access a particular file or directory. For this request, imagine the server's configured to only allow users that have authenticated themselves to view a slash account resource. In this case the server then has taken that anonymous request and returned a challenge saying I need to authenticate, the authentication protocol is the basic authentication protocol and notice the 41 status code, that is telling the client the request is unauthorized. A www dash authenticate header tells the client to collect the user credentials and then try this again. The basic realm attribute, that gives the user agent a techtual description of the protected area. And what happens next depends on the specific browser but most browsers will open up a dialogue that allows the user to enter their user name and password. We'll look at that in just a second. But once that happens the browser can send another request to the server and this request will include an authorization header. And the value of the authorization header is the client's user name and password and with basic authentication the user name and password is just base 64 encoded. That means basic authentication is insecure because anyone who can view that message can find out the user's name and password. So for that reason basic authentication is rarely used without secure HTTP which we'll look at later. But at this point it's up to the server to decode the authorization header, verify the user name and password by checking with the operating system or checking against something that's in a database or whatever credential management system is configured on the server. If the credentials match the server can make a reply and say yes, here's the account resource. If the credentials don't match, the server should respond with a 401 status, you are still unauthorized to view this. (pause) So with my browser I can currently get to the search dot cshtml page that's in my Website. But let me flip over into IAS and for this test directory, for this test application I'm going into the authentication settings and disable anonymous authentication effectively not allowing anonymous users into the site and enable basic authentication. You can see that IAS gives me the warning that SSL is not enabled and credentials will be sent in clear text. That's just a way of saying that since you are not using SSL, aka secure HTTP, that the user name and password will be visible in the message to anyone that is able to view that message. Now let's refresh the page and by the way I have Fiddler running in the background so we'll be able to see this request. And now I'm going to log on as a user that has an account on this machine and now I can get to search dot cshtml. So anonymous access was disabled, I needed to authenticate as a user on this machine in order to be able to get in and let's just take a look at what that looks like in Fiddler. First there was my initial get request to get search dot cshtml. The server challenged that by returning an HTTP 401 status message saying you are unauthorized, please use basic authentication in the local host realm. And we would have seen that local host text up here in the dialogue that popped up for me to login. And so the next request that the browser sent off after I entered in the user name and password was one that said please get search dot cshtml and use this authorization header. Let me just take this value which looks encrypted but it's really just encoded, it's base 64 encoded and we'll paste it into a base 64 decoder and tell it to decode and that's the user name, that's the password. And that's why we say that basic authentication is insecure. Basic authentication really isn't used that much and when it is used it's typically over httbfs. And once the browser has those credentials I can keep accessing this Website. Let me turn capturing back on and I'm just going to refresh search dot cshtml a few times and each time I do that we're going to be sending off a request that includes that authorization header. You can see that each of those 200 requests sent off an authorization header. Digest authentication is another authentication protocol that's included as part of the HTTP specification and it is an improvement over basic authentication because it does not transmit user passwords using base 64 encoding. Instead the client sends a digest of the password and the client needs to compute this digest using an Md5 hashing algorithm with a nonce that the server provides during the authentication challenge that helps to prevent replay attacks. So this is very similar to basic authentication, there's still a www dash authenticate header that the server will send back, it just includes some additional information that the client will need to use in his calculations so they have some cryptic graphic value. And then the client will also send back another request with an authorize header that now includes an encrypted form of the user name and password. And the server again can validate those and let the request through or reject the credentials and say this is still an unauthorized request. So digest authentication is better then basic authentication when secure HTTP is not available but it's still far from perfect because digest authentication is still vulnerable to man in the middle attacks. That's where someone can install it say a malicious proxy server that's looking at HTTP messages as the flow across the network and it sees what your authorization token is using digest authentication. Someone can still steal that piece of information and use it to access the server.

Windows Authentication

Windows integrative authentication is very popular when you have Microsoft servers and Microsoft products. Although it is supported by many modern browsers, not just Internet Explorer, it's just that it does require Windows machine as you Web server and it doesn't work so well over the internet or where proxy servers might reside. So it's commonly used for internal and internet Websites particularly at companies that have Microsoft active directories set up and they're using active directory to manage their users and groups and permissions. Windows authentication behaves very much like basic and digest authentication in the sense that a client makes a request for a resource that has been secured so the server will challenge that request with an HTTP 401 status code reply saying that was unauthorized, please authenticate. And in this case the value of the dub dub dub dash authenticate header will be negotiate. That's a key word, the client will interpret to mean Windows authentication specifically we can negotiate on the protocol because Windows supports a couple different security providers. There's NTLM and there's Kerberos. We can pick which one and agree on it and the next request I'll send along some credential information that will be encrypted. You can decipher that and figure out if the credentials are good or not and allow me in with the next request. And that still comes up in an authorize header. So because things are encrypted Windows authentication has the advantage of being a little more secure even without using secure HTTP and in some cases it can even be unobtrusive. So let's go into IAS and what I'm going to do is disable basic authentication. And enable Windows authentication and I'll also point out that I would have digest authentication available here as an option if I had it installed in IAS. But it does require an active directory server to be available and I don't have that available so it's not an option here. Windows authentication ironically is available without active directory server. What we'll be authenticating against is just the users that are on this machine. So with that in place let's open up Chrome and try to get to this page again. And now I'll be prompted to enter a user name and password and Chrome sort of knows what credentials I'm using here on the Windows machine. I could go ahead and try those or I could enter in the credentials for a different user and that allows me to get to the page. Now I can also come over into Internet Explorer and we go to search dot cshtml and I'm just instantly taken there but that's because in Internet Explorer one of the options that you can set here, if we go into custom level for security, is that it will automatically log me in with my current user name and password for sites that are in a specific zone and that means it's using the same credentials that I used to log in to this machine to access the server which in this case is what I wanted. And that usually works really well in an active directory setting.

Forms Authentication

Forms authentication is the most popular approach to user authentication over the internet. Forms based authentication is not a standard authentication protocol and it doesn't use the www dash authenticate or authorize headers that we've seen so far. However, many Web application frameworks provide some out of the box support for forms based authentication and the application has complete control over how the authentication behaves, how to validate credentials, how the sign in form appears. And that's because with forms based authentication the client will make a request for a secure resource and the server will respond by redirecting the browser to a login page. That's a HTTP 302 temporary redirect. And generally the URL that the user is requesting might be included in the query string of the redirect location so that once the user has completed logging in, the application can then redirect them again back to the secure resource that they were trying to reach. We call this forms authentication because the place where we are redirecting the user to is typically a page that has a form with inputs where the user can enter their user name and their password and then it will have a button to click to do the login. That will submit a post operation to the login destination and the application has to take the credentials that were entered and validate them against the database or the operating system or whatever credential management system it's using. Notice that forms based authentication will transmit a user's credentials in plain text so just like basic authentication forms based authentication does not secure unless you're using HTTPs or secure HTTP. And most Web frameworks once you have entered the proper credentials will respond to that post request with the credentials with another redirect back to the URL that you were trying to get to like slash account and in that response it will also set a cookie. And that cookie will indicate that the user is authenticated. Very commonly that cookie value is going to be encrypted and hashed to prevent tampering. But just remember that without HTTPs that cookie's still vulnerable to being intercepted because everything is being transmitted across the network in plain text. However, forms authentication remains very popular because again it gives you complete control over the login experience. Let me give you an example of what this would look like in terms of user experience. I'm going to try to go to a secure location on github, my inbox. When I press enter it detects that I'm trying to get to a secure resource, it doesn't know who I am, so it's going to redirect me to a login page and in the URL for that login page it's going to have a return to address. So if I'm properly authenticated it should be able to send me back there. And so once I login, it determines that really is me, it's able to send me over to my notifications page. So behind the scenes that was just a couple of HTTP redirects and a login page.

OpenID

Finally I thought I'd give a brief mention about open ID because open ID is slowing gaining some acceptance and here's the problem that it solves. Forms based authentication gives an application complete control over user authentication but many applications do not want that level of control. Specifically when I write an application I'd like to avoid managing and verifying user names and passwords because it's a risk to store user passwords in my database. Most people try to avoid passwords and store just hashed values of passwords but even then, most users don't want to have a different user name and password for every Website that they go to. And it's usually a bad idea to share credentials across multiple Websites. Open ID can solve many of these problems because it's an open standard for decentralized authentication. So with open ID I would go out and register with an open ID identity provider and the identity provider's the only site that needs to store and validate my credentials. There's a lot of providers around now including Google and Yahoo and VeriSign. When an application like stack overflow needs to authenticate a user it works with the user and the identity provider, there's some communication between the application and identity provider directly. There's also a communication between the user and the identity provider directly and the user ultimately has to verify the user name and password with the identity provider. And the application will find out if that was successful or not thanks to the presence of some cryptographic tokens and secrets that are going to be exchanged. So while open ID has a lot of benefits compared to forms authentication, it has faced a lack of adoption due to complexity in implementing, debugging and maintaining open ID and keeping it up and running and understanding how it works in your system. As the toolkits and frameworks continue to evolve, I expect that to make open ID authentication easier and the adoptions going to grow.

Secure HTTP

Finally we'll touch on the topic that we've danced around a few times and that is secure HTTP, also known as HTTPs, also known as SSL or TLS. There's all sorts of different acronyms for this. And basically it all comes down to this. We've talked about how self describing textual messages are one of the strengths of the Web because anyone can read a message and understand what's inside but there's a lot of messages that we need to send that we don't want anyone else to see. We don't want them to see our passwords, we don't want them to see our credit card numbers. Secure HTTP solves this problem by encrypting messages before they start traveling across the network. Secure HTTP is known is known as HTTPs because it uses an HTTPs scheme in the URL instead of a regular HTTP scheme. That's primarily because the default port for HTTP is port 80 and the default port for HTTPs is port 443. The browser will connect to the proper port depending on the scheme unless you've specified an explicit port in the URL. HTTPs works by adding an additional security layer in the network protocol stack. You remember we talked about the network protocol stack when we were looking at HTTP connections and we know that a message coming out of an application that's an HTTP message and has to go through TCP, go through IP, go out across the wire and then come up into the server by reversing through that protocol stack. HTTPs is essentially adding another layer, a secure sockets layer or transport layer security TLS between the application and the transport layers. So before that message even reaches the IP layer and well before it reaches your network card, it has been encrypted and the only thing that can decrypt that message is the other party. HTTPs requires the server to have a cryptographic certificate. That certificate is sent to the client during the set up of HTTPs, during the set up of the communication channel and that certificate includes the server's host name. Now a browser can use that certificate to validate that it is truly talking to the server that it thinks it's talking to. And that validation is all made possible using public key cryptography and the existence of certificate authorities like Bearsign (phonetic) that will sign and vouch for the integrity of certificate. Administrators have to purchase and install certificates from certificate authorities and install them on the Web server for this all to work. There's a lot of cryptographic details that we could cover but from a developer's perspective here's the most important things to know. First of all, all traffic over HTTPs is encrypted in the request and the response. That includes the HTTP headers and the message body and basically everything except the host name. That means that the URL path and the URL query string is encrypted as well as all cookies. So HTTPs prevents session high jacking because no eavesdroppers can inspect a message and steal a cookie. Another thing to know is that the server is authenticated to the client thanks to the server's certificate. If you are talking to bigbank.com over HTTPs you can be sure your messages are really going to bigbank.com and not someone who stuck a proxy server from the network to intercept requests and spoof response traffic from bigbank.com. Another thing to know is that HTTPs does not authenticate the client. So applications still need to implement forms authentication or one of the other authentication protocols mentioned earlier. HTTPs does make forms based authentication and basic authentication more secure since all data is encrypted, even the cookies. And there is the possibility of using what we call client side certificates with HTTPs. And client side certificates would authenticate the client in the most secure manner possible however, client side certificates are generally not used on the open internet since many users will not purchase and install a personal certificate. I've worked for a lot of clients and corporations that require client certificates for employees and contractors to access corporate servers because in that case the corporation can act as a certificate authority and issue employees and contractors their own certificates. Now HTTPs does have some downsides and most of them are related to performance. HTTPs is computationally expensive and large sites often use specialized hardware, we call them SSL accelerators, they help to take all the cryptographic computational load off the Web servers. HTTPs traffic is also impossible to cache in a public cache because once a message is encrypted, it's intended for a single user. However, user agents might keep HTTPs responses in their private cache. And finally, in regards to performance, HTTPs connections are expensive to set up and they require some additional hand shakes between the client and server to exchange cryptographic keys and insure everyone is communicating with the proper secure protocol. Persistent connections that we talked about in the third module, they can help to amortize the cost of setting up a HTTPs connection. But in the end if you need secure communications, then you're willingly going to pay for the performance penalties. Let me just point out that in my browser my communications in my login with github was all done over HTTPs and that's one of the reasons that I really can't use a tool like Fiddler to even intercept these HTTP messages because everything is encrypted as it's leaving the browser. Although there are some tricks you can use to get around it on a local machine. And if I click the lock icon up here, I can get some more information about the encryption. First of all I can see that the certificate that was given to GitHub, Incorporated was issued by DigiCert and that they have verified their location. And that this is all happening with 256 bit encryption using transport layer security. I can see the cryptographic algorithms that are in place and that's all good information. Everything looks good about this server and its certificate. So I can trust that the communication between my browser and github isn't going to be intercepted by anybody.

Summary

In this module we went quickly through some of the most popular authentication mechanisms in use on the Web today and you should know a little more now about the various trade-offs involved. We also talked about cookies and saw some examples of how we can use cookies to track users and track user state on the server. And now this is the last module of the course. I hope you enjoyed the material and that you were able to take away an in-depth knowledge of HTTP and how it works. Thanks for watching.