HTTP

What Every Web Developer Should Know About HTTP

K. Scott Allen

Ode to Code

HTTP

What Every Web Developer Should Know About HTTP

K. Scott Allen

Ode to Code

What Every Web Developer Should Know About HTTP

By K. Scott Allen

http://odetocode.com/blogs/scott

http://twitter.com/OdeToCode

Second Edition

Table of Contents

Chapter 0: Introduction

HTTP is the protocol that enables us to buy microwave ovens from Amazon.com, reunite with an old friend in a Facebook chat and watch funny cat videos on YouTube. HTTP is the protocol behind the World Wide Web. It allows a web server from a data center in the United States to ship information to an Internet café in Australia where a young student can read a web page describing the Ming dynasty in China.

In this book, we'll discuss 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 the basics including resources, messages, connections and security as it relates to HTTP.

We'll start by looking at resources.

Chapter 1: Resources

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 almost never, then I might open my web browser and enter `http://food.com` in the address bar to go to the food.com web site and search for recipes. My web browser understands this syntax and 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, we just want to focus on the address – `http://food.com` .

Resource Locators

The address `http://food.com` is what we call a URL – a uniform resource locator. It represents a specific resource on the web. In this case, the resource is the home page of the food.com web site. Resources are those places where I want to interact on the web. Images, pages, files and videos are all resources.

The Internet compiles billions, if not trillions, of places for people to explore. In other words, there are trillions of resources. Each resource will have a URL I can use to find it. `http://news.google.com` is a different place than `http://news.yahoo.com` . These are two different names, two different companies, two different web sites and therefore, two different URLs. Of course, there also will be different URLs inside the same web site. `http://food.com/recipe/broccoli-salad-10733/` is the URL for a page with a broccoli salad recipe, while `http://food.com/recipe/grilled-cauliflower-19710/` is still at food.com, but is a different resource describing a cauliflower recipe.

We can break the last URL into three parts:

1. `http` , the part before the `://` , is what we call the **URL scheme** . The scheme describes *how* to access a particular resource, and in this case, it tells the browser to use the hypertext transfer protocol. Later we'll also talk about a different scheme – https, which is the secure HTTP protocol. You

might run into other schemes too, such as ftp for the file transfer protocol and mailto for e-mail addresses.

Everything after the `://` will be specific to a particular scheme. A legal HTTP URL may not be a legal mailto URL – those two aren't really interchangeable (which makes sense because they describe different types of resources).

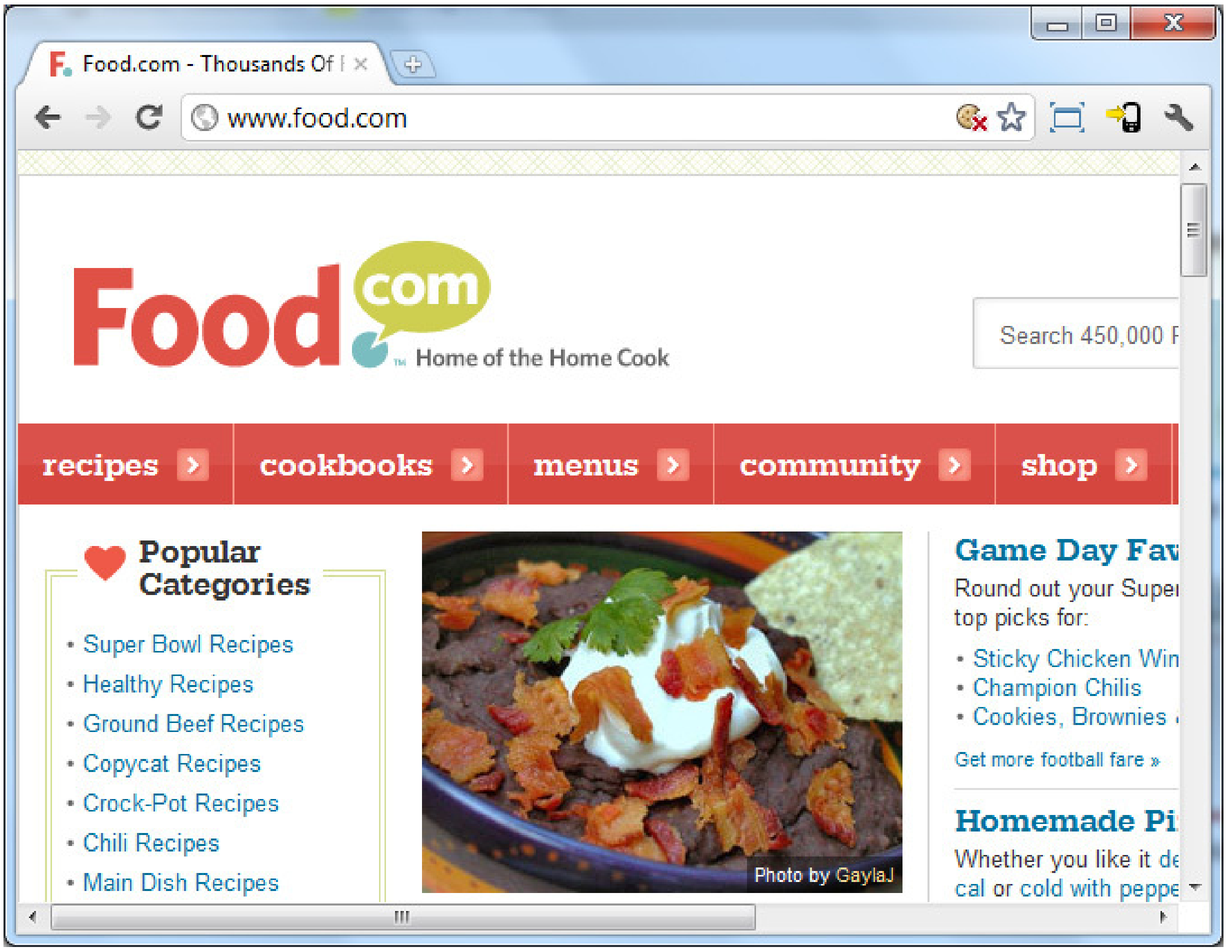
2. `food.com` is the **host** . This host name tells my browser the name of the computer hosting the resource. My computer will use the Domain Name System (DNS) to translate `food.com` into a network address, and then it will know exactly where to send the request for the resource. You also can specify the host portion of a URL using an IP address.

3. `/recipe/grilled-cauliflower-19710/` is the **URL path** . The food.com host should recognize the specific resource being requested by this path and respond appropriately.

Sometimes a URL will point to a file on the host's file system or hard drive. For example, the URL `http://food.com/logo.jpg` might point to a picture that really does exist on the food.com server. However, resources can be dynamic. The URL `http://food.com/recipes/brocolli` probably does not 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 using ASP.NET, PHP, Perl, NodeJS, Ruby on Rails, or some other web technology that knows how to respond to incoming requests by creating HTML for a browser to display.

In fact, these days many web sites try to *avoid* having any sort of real file name in their URL. For starters, file names are usually associated with a specific technology, like the .aspx file extension for Microsoft's ASP.NET technology. Many URLs will outlive the technology used to host and serve them. Secondly, many sites want to place keywords into a URL (like having /recipe/broccoli/ in the URL for a broccoli recipe). Having these keywords in the URL is a form of search engine optimization (SEO) that will rank the resource higher in search engine results. Its descriptive keywords, not filenames, that are important for URLs these days.

Some resources also will lead the browser to download additional resources. The food.com home page will include images, JavaScript files, CSS style sheets and other resources that will all combine to present the "home page" of food.com.



Ports, Query Strings and Fragments

Now that we know about URL schemes, hosts and paths, let's also look at a URL with a port number:

```
http://food.com:80/recipes/broccoli/
```

The number 80 represents the **port number** the host is using to listen for HTTP requests. The default port number for http is port 80, so you generally see this port number omitted from a URL. You only need to specify a port number if the server is listening on a port other than port 80, which usually only happens in testing, debugging or development environments. Let's look at another URL.

```
http://www.bing.com/search?q=broccoli
```

Everything after the question mark is known as the **query** . The query, also called the **query string** , will contain information for the destination web site to use or interpret. There is no formal standard for how the query string should look because it is technically up to the application to interpret the values it finds, but you'll see the majority of query strings used to pass name/value pairs in the form name1=value1&name2=value2.

For example:

```
http://foo.com?first=Scott&last=Allen
```

There are two name value pairs in the above. The first pair has the name "first" and the value "Scott". The second pair has the name "last" with the value "Allen". In our earlier URL (`http://www.bing.com/search?q=broccoli`), the Bing search engine will see the name "q" associated with the value "broccoli". It turns out the Bing engine looks for a “q” value to use as the search term. We can think of the URL as the URL for the resource that represents the BIng search results for broccoli.

Finally, one more URL:

```
http://server.com?recipe=broccoli#ingredients
```

The part after the number sign is known as the **fragment** . The fragment is different from the other pieces we've explored so far because unlike the URL path and 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. Specifically, the fragment is typically used to identify a specific HTML element in a page by the element's ID.

Web browsers will typically align the initial display of a web page, such that the top of the element identified by the fragment is at the top of the screen. As an example, the URL `http://odetocode.com/Blogs/scott/archive/2011/11/29/programming-windows-8-the-sublime-to-the-strange.aspx#feedback` has the fragment value of "feedback". If you follow the URL, your web browser should scroll down the page to show the feedback section of a blog post on my site. Your browser retrieved the entire resource (the blog post), but focused your attention to a specific area – the feedback section. You can imagine the HTML for the blog post looking like the following (with all the text content omitted):

```
<div id="post">

...

</div>

<div id="feedback">

...

</div>
```

The client makes sure the element with an ID of feedback is at the top.

If we put together everything we've learned so far, we know a URL is broken into the following pieces:

```
<scheme>://<host>:<port>/<path>?<query>#<fragment>
```

A Quick Note on Case

The path of a URL is case sensitive, meaning `http://food.com/Broccoli` is a different resource than `http://food.com/broccoli` , because one uses an uppercase B and one uses a lowercase b.

However, many web sites will try to make URLs behave as if they are case-insensitive. Thus, `/broccoli` and `/Broccoli` will reach the same resource. Web sites will do this to avoid broken links and allow users to find content even if they mistakenly type an uppercase character. Some web servers, notably the web server on Microsoft Windows operating systems, are case insensitive by default.

URL Encoding

All software developers who work with the web should be aware of character encoding issues with URLs. The official documents describing URLs go to great lengths to make URLs as usable and interoperable as possible. A URL should be as easy to communicate through e-mail as it is to print on a bumper sticker and affix to a 2001 Ford Windstar. For this reason, the Internet standards define **unsafe characters** for URLs. For example, the space character is considered unsafe because space characters can mistakenly appear or disappear when a URL is in printed form .

Other unsafe characters include "#" (because it is used to delimit a fragment), and "^" (because it isn't always transmitted correctly through all network devices). In fact, RFC 3986 (the "law" for URLs), defines the safe characters for URLs to be the alphanumeric characters in US-ASCII, plus just a few special characters (like ":" and "/").

Fortunately, you can still transmit unsafe characters in a URL, but all unsafe characters must be percent encoded (a.k.a. URL encoded). %20 is the encoding for a space character (where 20 is the hexadecimal value for the US-ASCII space character).

As an example, let's say you wanted to create the URL for a file named "^my resume.txt" on someserver.com. The legal, encoded URL would look like:

```
http://someserver.com/%5Emy%20resume.txt
```

Both the "^" and space characters have been percent encoded. Most web application frameworks will provide an API for easy URL encoding. On the server side, you should run your dynamically created URLs through an encoding API just in case one of the unsafe characters appears in the URL.

Resources and Media Types

So far, we've focused on URLs and simplified everything else. But, what does it mean when we enter a URL into the browser? Typically, it means we want to retrieve or view some resource. There is a tremendous amount of material to view on the web, and later, we'll also see how HTTP also enables us to create, delete and update resources. For now, we'll stay focused on retrieval.

We haven't been very specific about the types of resources we want to retrieve. There are thousands of different resource types on the Web – images, hypertext documents, XML documents, video, audio, executable applications and Microsoft Word documents.

For a host to serve a resource properly and for a client to display a resource properly, the parties involved must be specific and precise about the type of the resource. Is the resource an image? Is the resource a movie? We wouldn't want our web browsers to try to render a PNG image as text, and we wouldn't want them to try to interpret hypertext as an image.

When a host responds to an HTTP request, it returns a resource and specifies the **content type** (also known as the media type) of the resource. We'll see the details of how the content type appears in an HTTP message in the next chapter.

To specify content types, HTTP relies on the Multipurpose Internet Mail Extensions (MIME) standards. Although MIME was originally designed for e-mail communications, HTTP uses MIME standards for the same purpose, which is to label the content in such a way that the client will know what the content contains.

For example, when a client requests an HTML web page, the host can respond to the HTTP request with some HTML that it labels as "text/html". The "text" part is the primary media type, the "html" is the media subtype. When responding to the request for an image, the host will label the resource with a content type of "image/jpeg" for JPG files, "image/gif" for GIF files, or "image/png" for PNG files. Those content types are standard MIME types and are literally what will appear in the HTTP response.

A Quick Note on File Extensions

You might think that a browser would rely on the file extension to determine the content type of an incoming resource. For example, if my browser requests "frog.jpg" it should treat the resource as a JPG file, but treat "frog.gif" as a GIF file. However, for most browsers the file extension is the last place it will go to determine the actual content type.

File extensions can be misleading, and just because we requested a JPG file, doesn't mean the server has to respond with data encoded into JPG format. Microsoft documentation for Internet Explorer says IE will first look at the MIME type specified by the host. If the host does not provide a MIME type, IE will then scan the first 200 bytes of the response trying to guess the content type. Finally, if IE doesn't find a type and can't guess the type, it will fall back on the file extension used in the request for the resource. This is one reason why the content type label is important, but it is far from the only reason.

Content Type Negotiation

Although we tend to think of HTTP as something used to serve web pages, it turns out the HTTP specification describes a flexible, generic protocol for moving high fidelity information. Part of the job of moving information around is making sure all the parties involved know how to interpret the information.

Media types are not just for hosts. The client can play a role in what media type a host returns by taking part in a **content type negotiation** .

A resource identified by a single URL can have **multiple representations** . Take, for example, the broccoli recipe we mentioned earlier. The single recipe might have representations in different languages (English, French and German). The recipe could even have representations in different formats (HTML, PDF, XML and plain text). It's all the same resource and the same recipe but different representations.

The obvious question is – which representation should the server select? The answer is in the content negotiation mechanism described by the HTTP specification. When a client makes an HTTP request to a URL, the client can specify the media types it will accept. Again, we'll see details of the request in chapter 2, but imagine a browser request to http://food.com/ saying it will accept a representation in the German language. It's up to the server to try to fulfill the request. The host might send a textual resource that is still in English, which will probably disappoint a German speaker, but this is why we call it content negotiation and not a content ultimatum.

Web browsers are sophisticated pieces of software that can deal with many different types of resource representations. Content negotiation is something a user would probably never care about, but for software developers (especially web service developers), content negotiation is part of what makes HTTP great. A piece of code written in JavaScript can make a request to the server and ask for a JSON representation. A piece of code written in C++ can make a request to the server and ask for an XML representation. In both cases, if the host can satisfy the request, the information will arrive at the client in an ideal format for parsing and consumption.

Where Are We?

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. We've learned about URLs, URL encoding and content types. It's time to see what these content type specifications look like as they travel across the wire.

Chapter 2: Messages

In this chapter, we'll look inside the messages exchanged in an HTTP transaction. We'll learn about message types, HTTP headers and status codes. Understanding what is inside an HTTP message is vitally important for developers who work on the web. Not only will you build better applications by responding with the right types of messages, but you'll also be able to spot problems and debug issues when web applications aren't working.

Requests and Responses

Imagine walking up to a stranger in an airport and asking, "Do you know what time it is?” For the stranger to respond with the correct time, a few things must be in place. First, the stranger must understand your question because if she does not know English, she might not be able to make any response. Second, the stranger will need access to a time keeping device.

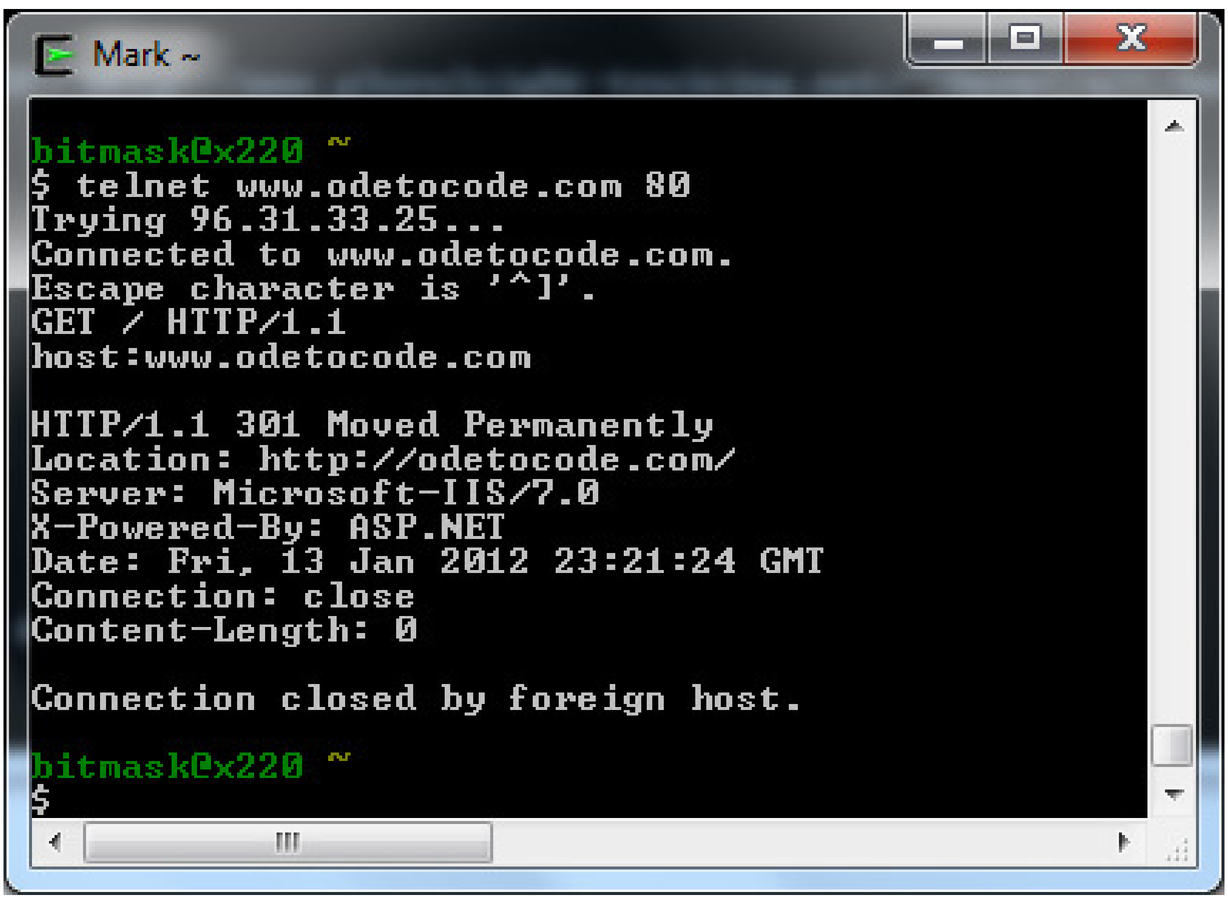
This airport analogy is like how HTTP works. You, the client, need a resource from a server. The resource you are requesting is the current time of day. You make a request to the server using a language and vocabulary you hope the server will understand. If the server understands your request and has the resource available, the server can reply with the current time of day. If the server understands the request, but does not have the resource available, the server can still respond and tell you there is a problem. If the server does not understand what you are saying, you might not see any response!

HTTP is a request-and-response protocol. A client sends an **HTTP request** to a server using a carefully formatted message that the server will understand. A server responds by sending an **HTTP response** that the client will understand. The request and the response are **two different message types** that are exchanged in a **single HTTP transaction** . The HTTP standards define what goes into the request and response messages so that everyone who speaks "HTTP" will understand each other and can exchange resources. When a resource does not exist, a server can still reply and let you know.

A Raw Request and Response

A web browser knows how to send an HTTP request by opening a network connection to a server machine and sending an HTTP message as text. There is nothing magical about the message. The message is a command in plain ASCII text and formatted per the HTTP specification. Any application that can send data over a network can make an HTTP request. You can even make a manual request using an application like Telnet from the command line. A normal Telnet session connects over port 23, but as we learned in the first chapter, the default network port for HTTP is port 80.

Below is a screen shot of a telnet session that connects to odetocode.com on port 80, makes an HTTP request and receives an HTTP response.



The telnet session starts by typing:

```
telnet www.odetocode.com 80
```

This command tells the operating system to launch the telnet application and tells the telnet application to connect to www.odetocode.com on port 80.

Once telnet connects, we can type out an HTTP request message. The first line is created by typing the following text, followed by the enter key:

```
GET / HTTP/1.1
```

The above information will tell the server we want to retrieve the resource located at "/" (the root resource or the home page), and we will be using HTTP 1.1 features. The next line we type is:

```
Host: www.odetocode.com
```

This host information is a required piece of information in an HTTP 1.1 request message. The technical reason to do this is to help servers that support multiple web sites to deliver the message to the correct site. Web servers and networking infrastructure can be expensive, and for small web sites, we don't want to run a single web site on a single server. We want to run multiple web sites on a single server to save money. Both www.odetocode.com and www.odetofood.com might live on the same physical server. The host information in the message will help the web server direct the request to the proper web application.

After typing the above two lines we can press the enter key twice to send the message to the server. What you see next in the telnet window is the HTTP response from the web server. We'll go into more details later, but the response says that the resource we want (the default home page of www.odetocode.com), has moved. It has moved to the location odetocode.com. It's up to the client now to parse this response message and send a request to odetocode.com instead of www.odetocode.com if it wants to retrieve the home page. Any web browser will go to the new location automatically.

These redirect messages are common. In this scenario, the redirect ensures all the requests for resources from OdeToCode go through odetocode.com and not www.odetocode.com. The redirect forces search engines to see one true URL for a given resource, what we call the canonical URL, because having a canonical URL will improve search result rankings for a given resource.

Now that we have seen a raw HTTP request and response, let's dig into specific pieces.

HTTP Request Methods

The "GET" word typed into the telnet session is one of the primary **HTTP methods** . Every request message must include an HTTP method. The method describes the purpose of the request. The purpose of an HTTP GET request is to read, fetch, or retrieve a resource. You could GET an image (GET `/logo.png`), or GET a PDF file, (GET `/documents/report.pdf`), or any other retrievable resource the server might hold. A list of common HTTP methods appears below.

Method	Description
GET	Retrieve a resource
PUT	Store a resource
DELETE	Remove a resource
POST	Update a resource
HEAD	Retrieve the headers or metadata for a resource

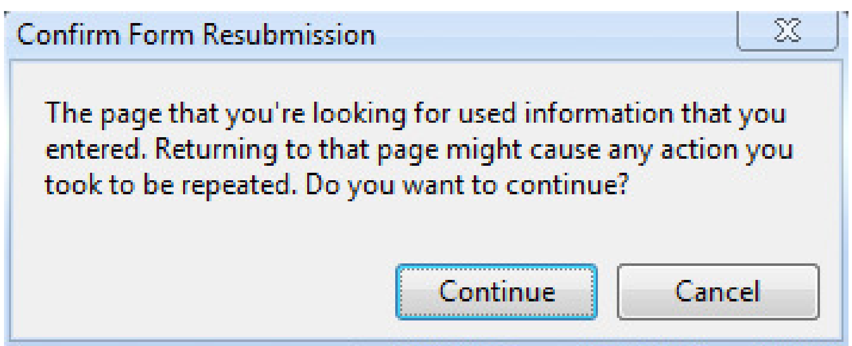
Of these five methods, two are the primary workhorses of the web: **GET and POST** . A web browser issues a GET request when it wants to retrieve a resource, like a page, an image, a video or a document. GET requests are the most common type of request.

A web browser sends a POST request when it has data to send to the server. For example, clicking "Add to Cart" on a site like amazon.com will POST information to Amazon about what we want to purchase. A `<form>` element on a web page generally initiates a POST request. A form contains the information you fill out inside <input> elements, like for address and credit card information.

GET and Safety

One part of the HTTP specification talks about "safe" HTTP methods. **Safe methods** , as the name implies, do not do anything dangerous like destroy a resource, submit a credit card transaction, or cancel an account. The GET method is one of the safe methods. A GET should only retrieve a resource and not alter the state of the resource. Sending a "GET" request for a JPG image doesn't change the image, it only fetches the image for display. In short, there should never be a side effect to a GET request.

An HTTP POST is *not* a safe method. A POST typically changes something on the server – it updates an account, submits an order or transfers money between two bank accounts. Web browsers typically treat GET and POST differently because GET is safe and POST is not safe. It's OK to refresh a web page retrieved by a GET request. When you press refresh, a web browser will reissue the last GET request and re-render the resource the server returns. However, if the page is the response of an HTTP POST request, the browser will warn us if we try to refresh the page. Perhaps you've seen these types of warnings in your web browser.



Because of warnings like this, many web applications always try to leave the client viewing the result of a GET request. After a user clicks a button to POST information to a server, like when submitting an order, the server will process the information and respond with an HTTP redirect, just like the redirect we saw in the Telnet window earlier. The redirect tells the browser to GET some other resource. The other resource will typically display the result of the previous POST operation. For example, the resource might show a confirmation number for the order and a “thank you” message. The user can refresh or print the page safely. The practice of redirecting after a POST is a common web design pattern known as the POST/Redirect/GET (PRG) pattern.

Now that we know a bit more about POST and GET, let’s talk about some common scenarios and understand when to use the different methods.

Common Scenario - GET

Let's say you have a web page and want the user to click on a link to view another page. In this case, a simple hyperlink is all you need.

```
<a href="http://odetocode.com/articles/365.aspx">Table
Variables</a>
```

When a user clicks on the hyperlink in a browser, the browser issues a GET request to the URL specified in the `href` attribute of the anchor tag. The request would look like this:

```
GET http://odetocode.com/articles/365.aspx HTTP/1.1

Host: odetocode.com
```

Common Scenario – POST

Now imagine you have a page where the user has to fill in information to create an account. Filling out information requires <input> tags, and we nest these inputs inside a <form> tag and tell the browser where to submit the information.

```
<form action="/account/create" method="POST">

  <label for="firstName">First name</label>
  <input id="firstName" name="firstName" type="text" />

  <label for="lastName">Last name</label>
```

```
    <input id="lastName" name="lastName" type="text" />

    <input type="submit" value="Sign up!"/>

</form>
```

When the user clicks the submit button, the browser realizes the button is inside a form. The form's `method` attribute tells the browser that the HTTP method to use for submission is POST, and the path to POST is /account/create. The actual HTTP request the browser makes will look something like this.

```
POST http://servername.com/account/create HTTP/1.1

Host: servername.com


firstName=Scott&lastName=Allen
```

Notice the form inputs are included in the body of the HTTP message. This is like how query string parameters appear in a URL, as we saw in Chapter I. It's up to the web application that receives this request to parse those values and create the user account. The application can then respond in any number of ways, but there are three common responses:

1) Respond with HTML to show the successful account creation. An immediate response will leave the user viewing the result of a POST request, which could lead to issues if he refreshes the page. A refresh might try to sign him up a second time!

2) Respond with a redirect message, as we saw earlier, and force the browser to issue a safe GET request for a page that tells him about the successful account creation.

3) Respond with an error, or redirect to an error page. We'll look at error scenarios a little later in the book.

Forms and GET Requests

A third scenario is a search scenario. In a search scenario, you need an <input> for the user to enter a search term. It might look like the following.

```
<form action="/search" method="GET">

   <label for="term">Search:</label>
   <input id="term" name="term" type="text" />
   <input type="submit" value="Sign up!"/>

</form>
```

Notice the `method` attribute on this form is GET, not POST. A search is a safe retrieval operation, unlike creating an account or booking a flight to Belgium. The browser will collect the inputs in the form and issue a GET request to the server:

```
GET http://searchengine.com/search?term=love HTTP/1.1

Host: searchengine.com
```

Instead of putting the input values into the body of the message, as we saw with a POST, the inputs go into the query string portion of the URL. The browser is sending a GET request for `/search?term=love` . Since the search term is in the URL, the user can bookmark the URL or copy the link and share it in an e-mail or tweet. The user could also refresh the page as many times as she likes because the GET operation for the search results is a safe operation that will not destroy or change data.

A Word on Methods and Resources

We've talked quite a bit about resources as physical resources on the file system of the server. Quite often, resources like PDF files, video files, image files and script files *do* exist as physical files on the server. However, the URLs for many modern web applications don't truly point to files. Technologies like ASP.NET and Ruby on Rails will intercept the request for a resource and respond however they see fit. They might read a file from a distant database and return the contents in the HTTP response to make it appear as if the resource really existed on the server itself.

A good example is the POST example we used earlier that resulted in a request to `/account/create` . Chances are great, there is no real file named "create" in an "account" directory. Instead, something on the web server picks up this request, reads and validates the user information and creates a record in a database.

The resource `/account/create` is virtual and doesn't exist. However, **the more you can think of a virtual resource as a *real* resource, the better your application architecture and design will adhere to the strengths of HTTP** .

HTTP Request Headers

So far, we've seen a raw HTTP request and talked about the two popular HTTP methods – GET and POST. But as the telnet output demonstrated, there is more to an HTTP request message than just the HTTP method. A full HTTP request message consists of the following parts:

```
[method] [URL] [version]

[headers]


[body]
```

The message is always in ASCII text, and the start line always contains the method, the URL and the HTTP version . The last section, the body section, can contain data, like the account sign-in parameters we saw earlier. When uploading a file, the body section can be quite large.

The middle section, the section where we saw `Host: odetocode.com` , contains one or more **HTTP headers.** `Host` is the only required header in HTTP 1.1, but there can be many other optional headers included in a request. Headers contain useful information that can help a server process a request. For example, in chapter 1, we talked about how a resource can have multiple representations, and how the client and server can negotiate on the best representation of a resource. If the client wants to see resource in French, the client can include an `Accept-Language` header requesting French content.

```
GET http://odetocode.com/Articles/741.aspx HTTP/1.1

Host: odetocode.com

Accept-Language: fr-FR
```

`Accept-Language` is one of numerous standard headers defined by the HTTP specification. Some of the headers are general headers that can appear in either a request or a response message. An example is the `Date` header. A client or a server can include a Date header indicating a message creation date.

```
GET http://odetocode.com/Articles/741.aspx HTTP/1.1

Host: odetocode.com

Accept-Language: fr-FR

Date: Fri, 9 Aug 2014 21:12:00 GMT
```

Everything but the host header is optional, but when a header does appear, it must obey the standards. For `Date` , the HTTP specification says the value of the header has to follow the RFC822 format for dates.

Some of the more popular request headers appear in the table below.

Header	Description
Referrer	When the user clicks on a link, the client can send the URL of the referring page in this header.
User-Agent	Information about the user agent (the software) making the request. Many applications use the information in this header, when present, to figure out what browser is making the request (Edge versus Chrome versus Firefox).
Accept	Describes the media types the user-agent is willing to

	accept. This header is used for content negotiation.
Accept-Language	Describes the languages the user-agent prefers.
Cookie	Contains cookie information, which we will look at in a later chapter. Cookie information generally helps a server track or identify a user.
If-Modified-Since	Will contain a date of when the user-agent last retrieved (and cached) the resource. The server only has to send back the entire resource if it's been modified since that time.

A full HTTP request might look like the following.

```
GET http://odetocode.com/ HTTP/1.1

Host: odetocode.com

Connection: keep-alive

User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64)
Chrome/16.0.912.75 Safari/535.7

Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*
/*;q=0.8

Referer: http://www.google.com/url?&q=odetocode

Accept-Encoding: gzip,deflate,sdch

Accept-Language: en-US,en;q=0.8

Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
```

As you can see, some headers contain multiple values separated by commas, like the Accept header in the above example. The Accept header is listing the MIME types my browser prefers to work with. My browser likes HTML, but can also work with XHTML and XML. The `*/*` entry near the end of the header is my browser's way of saying "I'll ultimately take any type of content and figure out what to do."

Also, notice the appearance of "q" in some of the headers.

```
Accept-Language: en-US,en; q=0.8
```

The q value is always a number between 0 and 1. The q value represents the **quality value** or "relative degree of preference" for a value. The default is 1.0. Higher numbers indicate a higher preference. The `Accept-Language` header above is telling the server that U.S. English is the preferred language. Because there is no q value specified for `en-US` , `en-US` would have the default quality value of 1. If U.S. English is not possible, then any type of English will work, although the preference has a lower value at 0.8.

The Response

An HTTP response has a similar structure to an HTTP request. The sections of a response are:

```
[version] [status] [reason]

[headers]


[body]
```

The full HTTP response to the last full request we listed might look like this (with most of the HTML omitted for brevity).

```
HTTP/1.1 200 OK

Cache-Control: private

Content-Type: text/html; charset=utf-8

Server: Microsoft-IIS/7.0
```

```
X-AspNet-Version: 2.0.50727

X-Powered-By: ASP.NET

Date: Sat, 14 Jan 2012 04:00:08 GMT

Connection: close

Content-Length: 17151

<html>

<head>

  <title>.Net related Articles, Code and
  Resources</title>

</head>

<body>

... content ...

</body>

</html>
```

The opening line of a response starts with the HTTP version, and then the all-important status code and reason.

Response Status Codes

The status code is a number defined by the HTTP specification. Response status codes are an incredibly important part of the HTTP response because they tell the client what happened (or in the case of redirects, where to go next).

All the numbers fall into one of five categories.

Range	Category
100-199	Informational
200-299	Successful
300-399	Redirection
400-499	Client Error
500-599	Server Error

Although we won't detail all of the possible HTTP status codes, the following table shows the most common codes.

Code	Reason	Description
200	OK	The status code everyone wants to see. A 200 code in the response means everything worked!
301	Moved Permanently	The resource has moved to the URL specified in the Location header and the client never needs to request the original URL in the future. We saw an example of this earlier in the book when we used telnet and the server redirected us from www.odetocode.com to odetocode.com.
302	Moved Temporarily	The resource has moved to the URL specified in the Location header. In the future, the client should still request the original URL because the move is only temporary. Applications typically use this response code after a successful POST operation to redirect a client to a resource the client can retrieve with GET (the PRG pattern we talked about earlier).
304	Not Modified	This is the server telling the client that the resource hasn't changed since the last time

		the client retrieved the resource. The client can use a cached copy of the resource.
400	Bad Request	The server could not understand the request. The request probably used an incorrect syntax or sent inappropriate data.
403	Forbidden	The server refused access to the resource. A 403 might happen because a user does not have permissions to view a protected or top-secret resource. Access denied!
404	Not Found	A popular code meaning the server could not locate the resource. A 404 might happen because of a typo in the URL.
500	Internal Server Error	The server encountered an error in processing the request. A 500 code could mean there is a programming error in the logic of the web application but can also happen because something the web application needs, like a database, is offline and not available.
503	Service Unavailable	The server will currently not service the request. This status code can appear when a server is throttling requests because the server is under heavy load.

HTTP Status Codes versus Your Application

Remember that the HTTP status code is a code to indicate what is happening at the HTTP level. It does not necessarily reflect what happened inside your web application. For example, imagine a user submits a sign-up form to the server but forgot to fill out the field for his last name. If your application requires a last name, your application will fail to create an account for the user. This does not mean you must return an HTTP error code indicating failure. You probably want the opposite to happen. You want the application to return some content to the client with a 200 (OK) status code. The content will tell the user they forgot to provide a last name during sign-up. From an application perspective, the request was a failure, but from an HTTP perspective, the HTTP transaction was a success.

On the other hand, maybe you are writing a web service instead of a web application. Instead of using HTTP to send HTML to browsers for humans to read, you are using HTTP to send XML or JSON data to another application. In this scenario, if a program sends data to create a user account but does not include the user's last name, you might want to reply with a 400 status code and include information to specify the missing field. Web services and web APIs generally operate closer to the HTTP level than a web application for web browsers.

Response Headers

A response includes header information that gives a client additional data the client can use to process the response. For example, the server will specify the content type of a resource using a MIME type, as we talked about in chapter 1. In the following response, the content type is html, and the character set used to encode the HTML is UTF-8. The headers can also contain information about the server, like the type and version of the software used to create the web site.

```
HTTP/1.1 200 OK

Cache-Control: private

Content-Type: text/html; charset=utf-8

Server: Microsoft-IIS/7.0

X-AspNet-Version: 2.0.50727

X-Powered-By: ASP.NET

Date: Sat, 14 Jan 2012 04:00:08 GMT

Connection: close

Content-Length: 17151
```

```
<html>

<head>

  <title>.Net related Articles, Code and
  Resources</title>

</head>

<body>

... content ...

</body>

</html>
```

The response headers appearing in a response will often depend on the type of response. For example, a redirection response needs to include a `Location` header that tells the client where to go next.

```
HTTP/1.1 301 Moved Permanently

Server: Microsoft-IIS/7.0

X-AspNet-Version: 2.0.50727

X-Powered-By: ASP.NET

Location: http://odetocode.com
```

There are a number of headers devoted to caching and performance optimizations. ETag, Expires and Last-Modified all provide information about the cacheability of a response. An ETag is an identifier that will change when the underlying resource changes, so comparing ETags is an efficient way for a client to know if a cached resource is stale. An Expires header tells a client how long to cache a resource. We’ll return and look at caching in more detail later.

See It for Yourself!

There are many tools available for inspecting HTTP messages. The developer tools in most web browsers, like the developer tools in Google Chrome, can show you the HTTP messages exchanged between the browser and the server. On Windows, you might try Fiddler (http://www.telerik.com/fiddler). Fiddler is easy to use and will allow you not only to see raw HTTP request and response messages, but also create and modify requests.

Where Are We?

In this chapter, we've learned that HTTP messages always come in pairs. There is the request, and there is the response.

The HTTP specification designs the messages to make sure both parties in an HTTP transaction understand what they are receiving. The first line of an HTTP message is always explicit about its intent. In a request message, the URL and HTTP method appear first and identify what should happen to a resource. In a response, the status code will indicate if the server processed the request successfully or if there was an error.

We also have headers moving in both directions that provide even more information about the request and response.

In the next chapter, we'll learn a little more about how these messages travel across the network.

Chapter 3: Connections

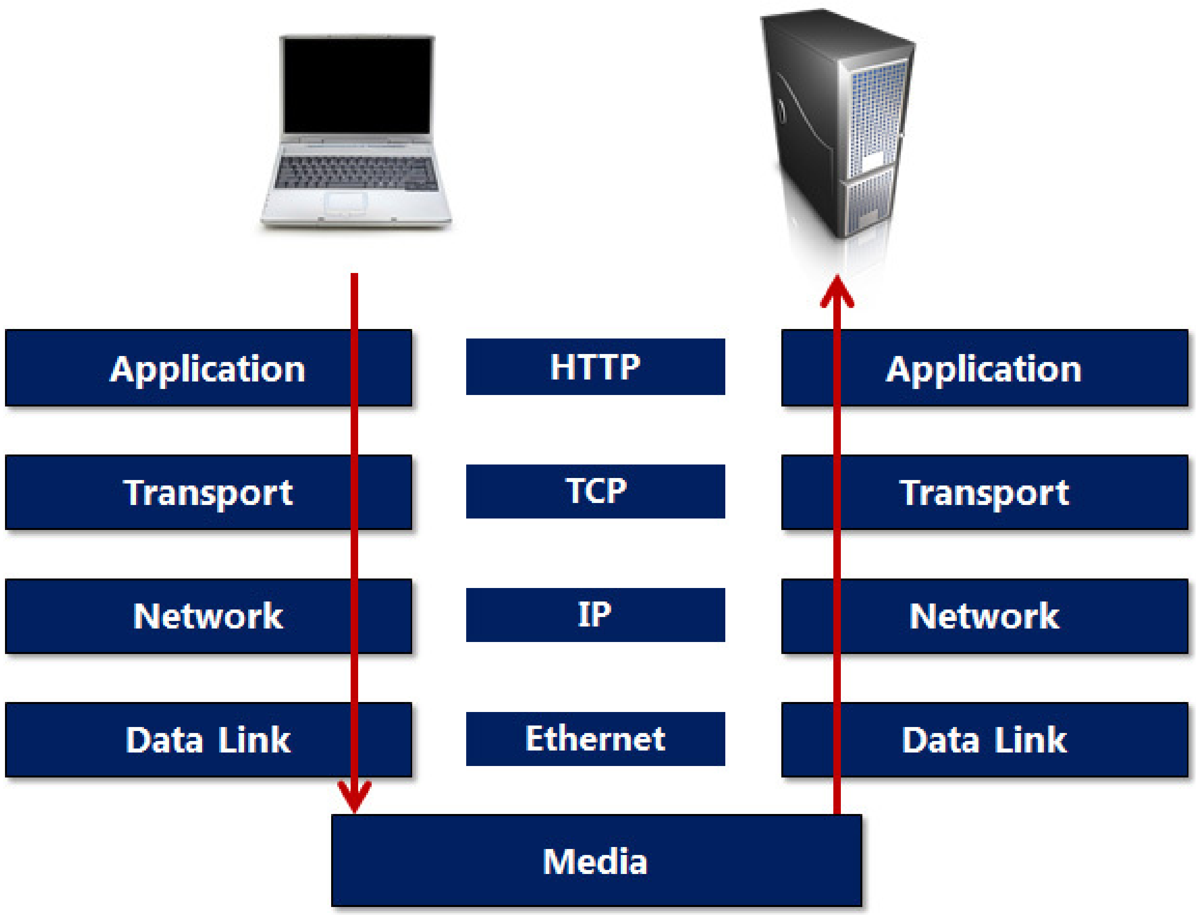
In chapter 2, we analyzed HTTP messages and saw examples of the textual commands and codes that flow from the client to the server and back in an HTTP transaction. How does the information in these messages move through the network? When does a web browser open a connection? When does a browser close a connection? We will answer these questions in this chapter, but first, we'll need to understand some of the network abstractions underneath HTTP.

A Whirlwind Tour of Networking

Network communication, like many applications, consists of layers. Each layer in a **communication stack** is responsible for a specific and limited number of responsibilities.

HTTP is what we call an **application layer protocol** , because HTTP 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 IIS or Apache.

We saw how HTTP messages allow the browser to request resources from the server. The HTTP specifications don't say anything about how the messages traverse the network and reach the server. The transmission of an HTTP message is the job of a lower layer protocol. A message from a web browser has to travel down a series of network layers. When the message arrives at the web server it travels up through a series of layers to reach the web service process.



The layer underneath HTTP is a **transport layer protocol** . Almost all HTTP traffic travels over **TCP** (short for Transmission Control Protocol), although this is not required by HTTP. When a user types a URL into the browser, the browser first extracts the host name from the URL (and port number, if any), and opens a **TCP socket** by specifying the server address (derived from the host name) and port (which defaults to 80).

Once an application has an open socket it can begin writing data into the socket. The only thing the browser needs to worry about is writing a properly formatted HTTP request message into the socket. The TCP layer accepts the data and guarantees the delivery of the message to the other server. TCP will automatically resend any information that might get lost in transit, and therefore TCP is what we call a *reliable protocol* .

In addition to error detection, TCP also provides flow control. The flow control algorithms in TCP will ensure the sender does not send data too fast for the receiver to process the data. Flow control is important in this world of varied networks and devices.

In short, TCP provides services vital to the successful delivery of HTTP messages, but it does so in a transparent way so that most applications don't need to worry about TCP. As the last picture shows, TCP is just the first layer beneath HTTP. After TCP at the transport layer comes IP as a network layer protocol.

IP is short for **Internet Protocol** . While TCP is responsible for error detection, flow control, and overall reliability, IP is responsible for taking pieces of information and moving them through the various switches, routers, gateways, repeaters, and other devices that move information from one network to the next and all around the world. IP tries hard to deliver the data at the destination, but IP does not guarantee delivery because guaranteed delivery is TCP's responsibility.

IP requires every device to have an address, an IP address, which looks like 208.192.32.40. IP is also responsible for breaking data into packets (often called datagrams), and sometimes fragmenting and reassembling these packets so they are optimized for a network.

Everything we've talked about so far happens inside a computer, but eventually these IP packets must travel over a piece of wire, a fiber optic cable, a wireless network or a satellite link. This is the responsibility of the **data link layer** . A common choice of technology at this point is **Ethernet** . Low-level protocols like Ethernet are focused on 1s, 0s and electromagnetic signals.

Eventually, the signal reaches the server and enters the machine through a network card where the same process happens in reverse. The data link layer delivers packets to the IP layer, which hands over data to TCP, which can reassemble the data into the original HTTP message sent by the client and push the message into the web server process. The web service process receives the message because the web server listens for messages on a TCP socket.

HTTP over TCP and IP is a marvelous piece of engineering all made possible by standards.

Quick HTTP Request with Sockets and C#

If you are wondering what it looks like to write an application that will make HTTP requests, then the following C# code is a simple example of what the code might look like. This code does not have any error handling, and tries to write any server response to the console window (so you'll need to request a textual resource), but it works for simple requests.

```
using System;

using System.Net;

using System.Net.Sockets;

using System.Text;


public class GetSocket

{

   public static void Main(string[] args)

   {

      var host = args[0];

      var resource = args[1];

      var result = GetResource(host, resource);

      Console.WriteLine(result);

   }
```

```
private static string GetResource(string host, string
resource)

{

    var hostEntry = Dns.GetHostEntry(host);

    var socket = CreateSocket(hostEntry);

    SendRequest(socket, host, resource);

    return GetResponse(socket);

}

private static Socket CreateSocket(IPHostEntry
hostEntry)

{

    const int httpPort = 80;

    foreach (var address in hostEntry.AddressList)

    {

        var endPoint = new IPEndPoint(address,
        httpPort);

        var socket = new Socket(

                endPoint.AddressFamily,

                SocketType.Stream,

                ProtocolType.Tcp);

        socket.Connect(endPoint);

        if (socket.Connected)
```

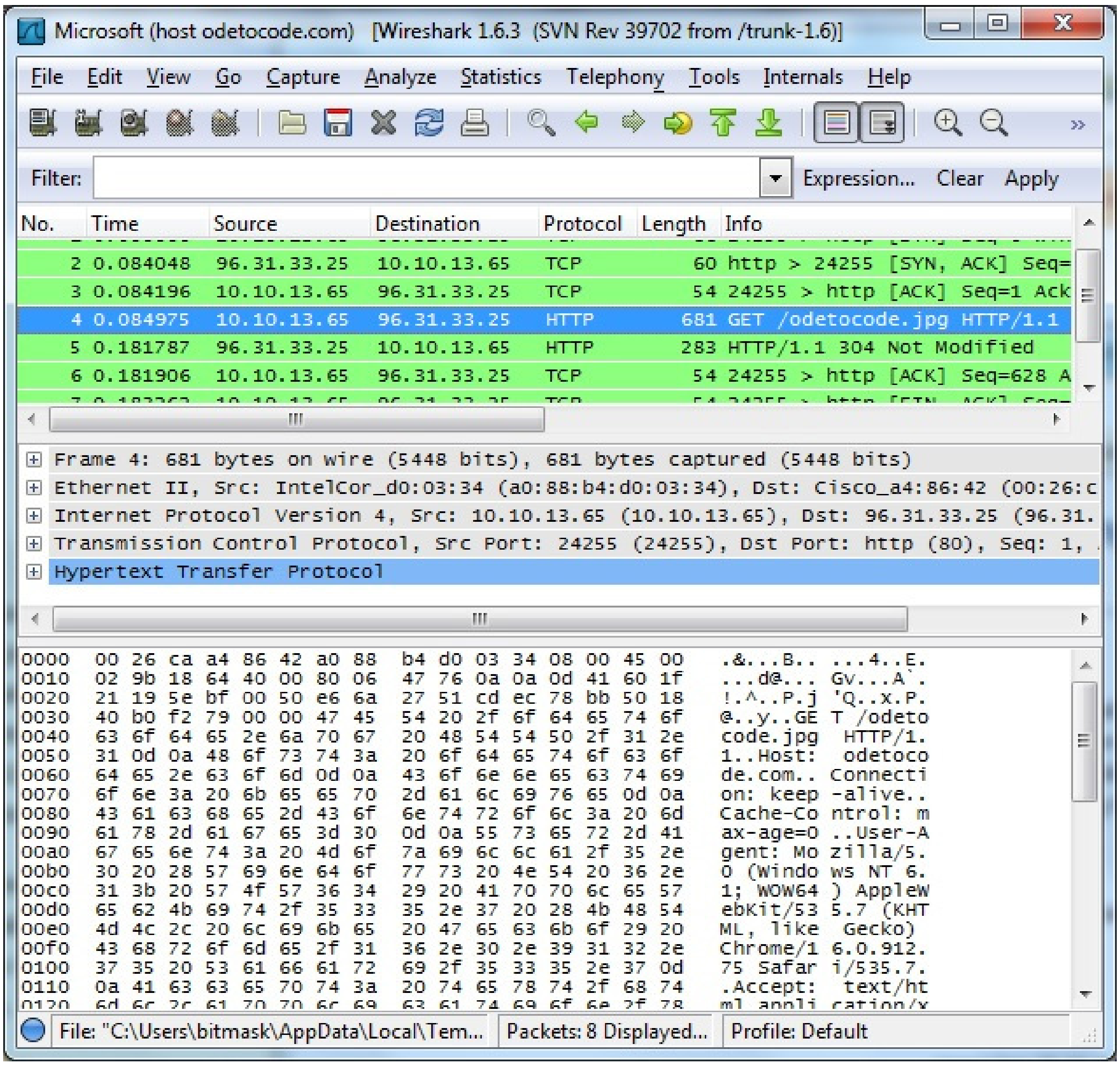
```
        {
            return socket;
        }
    }
    return null;
    }

                    private static void
                                    SendRequest(Socket
                                    socket, string host,
                                    string resource)
    {
        var requestMessage = String.Format(
            "GET {0} HTTP/1.1\r\n" +
            "Host: {1}\r\n" +
            "\r\n",
            resource, host
        );

        var requestBytes =
        Encoding.ASCII.GetBytes(requestMessage);
        socket.Send(requestBytes);
    }
```

```
    private static string GetResponse(Socket socket)
    {
        int bytes = 0;
        byte[] buffer = new byte[256];
        var result = new StringBuilder();

        do
            {
            bytes = socket.Receive(buffer);
            result.Append(Encoding.ASCII.GetString(buffer,
            0, bytes));
        } while (bytes > 0);

        return result.ToString();
    }
}
```

Notice how the program needs to look up the server IP address using `Dns.GetHostEntry` , and formulate a proper HTTP message with a GET method and Host header. The actual networking part is easy because the Socket class and TCP take care of most of the work. TCP understands how to manage multiple connections to the same server, so you can have two concurrent requests to the same server using two separate sockets. You do not need to worry about one socket getting confused and receiving data intended for a different socket.

Networking and Wireshark

If you want some visibility into TCP and IP, you can install a free program like Wireshark. Wireshark is available for OS/X and Windows from http://www.wireshark.org/ . Wireshark is a network analyzer that can show you every bit of information flowing through your network interfaces. Using Wireshark, you can observe TCP handshakes, which are the TCP messages required to establish a connection between client and server before the actual HTTP messages start flowing. You can also see the TCP and IP headers on every message. Below is a screen shot showing the last two steps of the handshake, followed by an HTTP GET request and a 304 redirect response.



With Wireshark, you can see when HTTP connections are established and closed. The important part to take away from all of this is not how handshakes and TCP work at the lowest level, but that HTTP relies almost entirely on TCP to take care of all the hard work and TCP involves some overhead, like handshakes. Thus, the performance characteristics of HTTP also rely on the performance characteristics of TCP, and this is the topic for the next section.

HTTP, TCP, and the Evolution of the Web

In the very old days of the web, most resources were textual. You could request a document from a web server, go off and read for five minutes, then request another document. The world was simple.

For today's web, most web pages consist of more than a single resource. Every page in a web application has one or more images, one or more JavaScript files, and one or more CSS style sheets. It's not uncommon for the initial request for the home page of a web site to spawn off 30 to 50 additional requests to retrieve all the other resources associated with a page.

In the old days, it was also simple for a browser to establish a connection with a server, send a request, receive the response and close the connection. If today's web browsers opened connections one at a time and waited for each resource to download before starting the next download, then the web would feel very slow. The Internet is full of latency. Signals must travel long distances and wind their way through different pieces of hardware. There is also some overhead in establishing the underlying TCP connection. As we saw in the Wireshark screen shot there is a 3-step handshake to complete before an HTTP transaction can begin.

The evolution from simple documents to complex web pages has required some ingenuity in the practical use of HTTP.

Parallel Connections

Most web browsers will not make requests in a serial one-by-one fashion. Instead, web browsers can open multiple, **parallel connections** to a server. For example, when downloading the HTML for a page the browser might see two `<img>` tags in the page, so the browser will open two additional HTTP connections in parallel and download the two images concurrently.

The number of parallel connections depends on the browser and the browser's configuration.

For a long time, we considered two connections as the maximum number of parallel connections a browser would create. We considered two as the maximum because the most popular browser for many years was Internet Explorer 6, and IE6 would only allow two simultaneous connections to a single host. Internet Explorer was only obeying the rules spelled out in the HTTP 1.1 specification, which states:

A single-user client SHOULD NOT maintain more than two connections with any server or proxy.

To increase the number of parallel downloads, many web sites use some tricks. For example, the two-connection limit is *per host* , meaning a browser like IE6 would happily make two parallel connections to www.odetocode.com *and* two parallel connections to images.odetocode.com. By hosting images on a different host, web sites could increase the number of parallel downloads and make their resources load faster. Behind the scenes, the DNS records might ultimately point all four requests to the same IP address, but the two-connection limit is per host name, not per IP address.

Things are different today. Most browsers will use a different set of heuristics when deciding on how many parallel connections to use. Most browsers today will open at least six concurrent connections.

The question to ask is this - how many connections are too many? Parallel connections will obey the law of diminishing returns. Too many connections can saturate and congest the network, particularly when mobile devices or unreliable networks are involved. In addition, a single server can only accept a finite number of connections. If 100,000 browsers simultaneously create 100 connections to the same web server, the web server can start to experience failures and reject or miss incoming requests.

Fortunately, parallel connections are not the only performance optimization.

Persistent Connections

In the early days of the web, a web browser would open and close a connection for each individual HTTP transaction. This implementation was per HTTP's idea of being a stateless protocol, meaning the server does not have to remember the state of any client that connects. A server doesn't have to remember, for example, the preferred language of a client. All the information a server needs to know about a client will be inside each incoming HTTP request message, including the client's preferred language.

As the number of requests per page grew, so did the overhead generated by TCP handshakes and the in-memory data structures required to establish each TCP socket. To reduce this overhead and improve performance, the HTTP 1.1 specification suggests that clients and servers should implement **persistent connections** and make persistent connections the default type of connection.

A persistent connection stays open after the completion of one request-response transaction. This behavior leaves a browser with an already open socket it can use to continue making future 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, making persistent connections perform better over time. The slow start strategy of TCP helps to avoid sending more data than a network can handle by starting transmission at a slow rate and gradually increasing the speed if the network is proving itself capable.

Persistent connections reduce memory usage, reduce CPU usage, reduce network congestion, reduce latency, and generally improve the response time of a page. However, like everything in the world of computers there is a trade-off.

As mentioned earlier, a server can only support a finite number of incoming connections. The exact number depends on the amount of memory available, the configuration of the server software, the performance of the application, and many other variables. It is difficult to give an exact number, but if you talk about supporting thousands of concurrent connections on a single machine, you'll want to start testing to see if the machine and the application will support the load.

Many servers have a default configuration to limit the maximum number of concurrent connections far below the point where the server will fail. The configuration is a security measure to help prevent denial of service (DoS) attacks. A malicious person can initiate a DoS attack by creating programs that open thousands of persistent connections to a server. By overwhelming the server with connections, the malicious person can inhibit the server from responding to real customers.

Thinking along the lines of a security vulnerability, we also have to wonder how long a persistent connection should stay open. In a world where all computers had an infinite amount of memory, CPU power, and network bandwidth, the persistent connection could stay open for as long as the web browser was running. However, because a server only supports a finite number of connections, most web servers will close a persistent connection if it is idle for some period. The default Apache configuration says to close a persistent connection if the connection is idle for 5 seconds. The only way to see a forced closing is through a network analyzer like Wireshark.

In addition to closing persistent connections aggressively, web servers usually allow you to disable persistent connections. Disabling persistent connections is a common practice with shared servers offered by inexpensive webhosts like GoDaddy. Shared servers sacrifice performance to allow as many connections as possible. Because persistent connections are the default connection style with HTTP 1.1, a server that does not allow persistent connections must include a `Connection` header in every HTTP response. Below is an example.

```
HTTP/1.1 200 OK
```

```
Content-Type: text/html; charset=utf-8

Connection: close

Content-Length: 17149
```

The `Connection: close` header is a signal to the browser that the connection will not be persistent and the browser should close the connection immediately. The browser will not be able to make a second request on the same connection.

Pipelined Connections

Parallel connections and persistent connections are both in wide use by clients and server. Future specifications will allow for **pipelined connections** . In a pipelined connection, a browser can send multiple HTTP requests on a connection before waiting for the first response. Pipelining allows for a more efficient packing of requests into packets and can reduce latency .

Where Are We?

In this chapter, we've analyzed HTTP connections and talked about some of the performance optimizations made possible by the HTTP specifications. Now that we have delved deep into HTTP messages and even examined the connections and TCP support underneath the protocol, we'll take a step back and look at the Internet from a wider perspective in the next chapter.

Chapter 4: Web Architecture

In the first chapter, we talked about resources, but mostly focused on URLs. Resources are the centerpiece of HTTP. Now that we understand HTTP messages, methods and connections, we can return to look at resources in a new light. In this chapter, we'll talk about the true essence of working with resources when building web applications and web services.

Resources Redux

Although it is easy to think of a web resource as being a file on a web server's file system, thinking along these lines disrespects the true power behind the idea of the resource.

Many web pages do require physical resources on a file system – JavaScript files, images, and style sheets. However, consumers and users of the web don't care much for these background resources. Instead, they care about resources they can interact with, and more importantly, resources they can *name* . Resources like:

- The recipe for broccoli salad

- The search results for "Chicago pizza"

- Patient 123's medical history

There are entire applications built around resources like these. The common theme in the list is how each item is significant enough to identify and name. If we can identify a resource, we can also give the resource a URL for someone to locate the resource. A URL is a handy thing to have. Given a URL, you can locate a resource, but you can also hand the URL to someone else by embedding the URL in a hyperlink or sending it in an e-mail.

It's also interesting to think about what you *can not* do with a URL.

A URL can not restrict a client or a server to a specific type of technology. Anyone can speak the language of HTTP. A client program written in C++ can talk to a web application written in Ruby. Python can talk to Java. iOS applications can talk to ASP.NET servers.

A URL can not force the server to store a resource using any specific technology. The resource could be a document on the file system, but a web framework could also respond to the request for the resource and build the resource using information stored in files, stored in databases, retrieved from web services or derived from the current time of day.

A URL can not specify the representation of a specific resource, and a resource can have multiple representations. As we learned earlier, a client can request a representation using headers in the HTTP request message. A client can request a specific language or a specific content type. If you ever worked with a web application that allows for content negotiation, you've seen the flexibility of resources in action. JavaScript can request Patient 123's data in JSON format, C# can request the same resource in XML format, and a browser can request the data in HTML format. They all work with the same resource but using three different representations.

There is one more thing a URL can not do – it can not 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. It's the job of the HTTP request message to describe this intention using one of the HTTP standard methods. As we talked about in chapter 2, there are a limited number of standard HTTP methods, including GET, POST, PUT and DELETE.

When you start thinking about resources and URLs as we are in this chapter, you start to see the web as part of your application and as a flexible architectural layer you can use for building applications and services. For more insight into this line of thinking, see Roy Fielding's famous dissertation titled *Architectural Styles and the Design of Network-based Software Architectures* . The dissertation is the paper that introduces the "representational state transfer" (REST) style of architecture and goes into detail about the ideas and concepts in this section and the next. The

article resides at
http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

The Visible Protocol - HTTP

So far we've been focused on what a URL *can't do* , when we should be focused on what a URL *can do* . Or rather, focus on what a URL + 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 these benefits because a URL is just a pointer, or a unit of indirection, between a client and server application. The URL itself doesn't dictate a specific resource representation, technology implementation or the client intention. Instead, a client can express the desired intention and representation in an HTTP message.

An HTTP message is a simple, plain text message. The beauty of the HTTP message is how both the request and the response are fully self-describing. A request includes the HTTP method (what the client wants to do), the path to the resource and additional headers providing information about the desired representation. A response includes a status code to indicate the result of the transaction, but also includes headers with cache instructions, the content type of the resource, the length of the resource, and possibly other valuable metadata.

All the information required for an HTTP transaction is contained in the HTTP request and response messages, and the information is visible and easy to parse. As messages move between the client and the server, several services can act on the message to add value.

Adding Value

An HTTP message moves from the memory space in one process on one machine to the memory space in one process on another machine. As the message moves, the message can pass through different pieces of software and hardware that interact with the message.

One example is the web server software itself. A web server like Apache or Windows IIS will be one of the first recipients of an incoming HTTP request on a server machine. The web server will inspect information in a message, like the URL or the host header, when deciding where to send a message. The server can also perform additional actions with the message, like logging the message to a local file. The web applications on the server don't need to worry about logging because the server is configured to log all messages.

Likewise, when a web application creates an HTTP response message, the server has a chance to interact with the response on the way back out to the network. The server could log the operation, but the server could also modify the message itself. For example, a server can know if a client supports gzip compression because a client can advertise this fact through an accept-encoding header in the HTTP request. Compression allows a server to take a 100kb resource and turn it into a 25kb resource for faster transmission. 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.

Applications don't have to worry about logging HTTP transactions or compression, and this is all thanks to the self-descriptive HTTP messages that allow other pieces of infrastructure to process and transform messages. This type of processing can happen as the message moves across the network, too.

Proxies

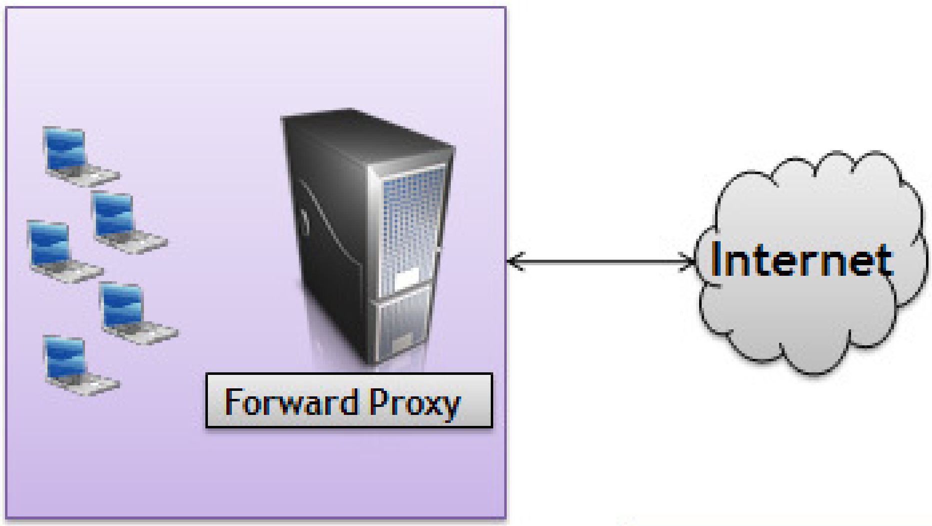
A **proxy server** is a computer that sits between a client and server. A proxy is mostly transparent to end-users. You think you are sending HTTP request messages directly to a server, but the messages are going to a proxy. The proxy accepts HTTP request messages from a client and forwards the messages to the desired server. The proxy then takes the server response and forwards the response back to the client. Before forwarding messages, the proxy can inspect the messages and potentially take some additional actions.

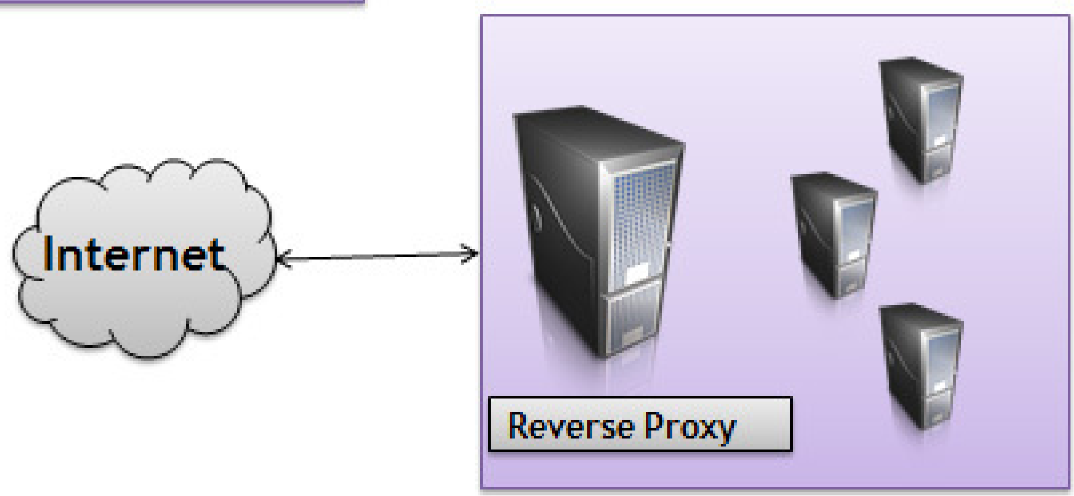
One of my clients uses a proxy server to capture all HTTP traffic leaving the office. They don't want employees and contractors spending any time on Twitter or Facebook, so HTTP requests to those servers will never reach their destination. The proxy also prevents interaction with popular file sharing web sites like Dropbox, and the red-light district of the Internet.

However, a proxy server can be much more sophisticated than just dropping messages to specific hosts. A proxy server could also inspect messages to remove confidential data, like the referrer headers that point to internal resources on the company network. An access control proxy can also log HTTP messages to create audit trails on all traffic. Many access control proxies require user authentication to ensure only authorized users have external network access.

The proxy I'm describing in the above paragraph is what we call a **forward proxy** . Forward proxies are usually closer to the client than the server, and forward proxies usually require some configuration in the client software or web browser to work.

A **reverse proxy** is a proxy server that is closer to the server than the client and is completely transparent to the client.





Both types of proxies can provide a wide range of services. If we return to the gzip compression scenario we talked about earlier, a proxy server has the capability to compress response message bodies. A company might use a reverse proxy server for compression to take the computational load off the web servers where the application lives. Now, neither the application nor the web server has to worry about compression. Instead, compression is a feature that layers into an architecture via a proxy. That's the beauty of HTTP.

Some other popular proxy services include the following.

Load balancing proxies can take a message and forward it to one of several web servers on a round-robin basis, or by knowing which server is currently processing the fewest number of requests.

SSL acceleration proxies can encrypt and decrypt HTTP messages, taking the encryption load off a web server. We'll talk more about SSL in the next chapter.

Proxies can provide an additional layer of **security** by filtering out potentially dangerous HTTP messages. Specifically, messages that look like they might be trying to find a cross-site scripting (XSS) vulnerability or launch a SQL injection attack.

Caching proxies will store copies of frequently accessed resources. The proxy can respond to messages requesting a cached resource by returning the cached resource directly. We'll go into more detail about caching in the next section.

Finally, it is worth pointing out that a proxy doesn't have to be a physical server. There are software proxies that can plug into an operating system or web browser. These proxies typically offer logging, debugging, and diagnostic services.

Proxies are a perfect example of how HTTP can influence the architecture of a web application or web site. You can layer many services into the network without changing the application. The one service we want to examine in more detail is caching.

Caching

Caching is an optimization made to improve performance and scalability. When there are multiple requests for the same resource representation, a server can send the same bytes over the network 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 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 the HTTP message headers facilitates additional layers and services. 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 of a specific company, or the users of a specific Internet service provider. A public cache on a reverse proxy is commonly caching the resources that are popular on a specific web site, like popular product images from Amazon.com.

A **private cache** is a cache for a single user. Web browsers always keep a private cache of resources on your disk. In Internet Explorer, the cache is the "Temporary Internet Files" you'll see Windows mention. In Google Chrome, you can type `about:cache` in the address bar to see files in Chrome's private cache. Any resource a browser has cached on the file system can appear almost instantly on the screen.

The rules about what to cache, when to cache and when to invalidate a cache item (that is, kick it from the cache), are unfortunately complicated and mired by some legacy behaviors and incompatible implementations. Nevertheless, I will endeavor to point out the things you should know about caching.

In HTTP 1.1, a response message with a 200 (OK) status code for an HTTP GET request is cacheable by default (meaning it is legal for proxies and clients to cache the response). An application can influence this default by using the proper headers in an HTTP response. In HTTP 1.1 this header is the **Cache-control** header, although you can also see an **Expires** header in many messages, too. The Expires header is still around and is widely supported despite the HTTP 1.1 specification deprecating the header. **Pragma** is another example of a header used to control caching behavior, but it too is only around for backward compatibility. In this book, I'll focus 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. A value of private means only the user's browser can cache the response. A value of no-cache means no one should cache the response. There is also a **no-store** value, meaning the message might contain sensitive information and not only should the message not be cached or saved, but the browser should remove the message from memory as soon as possible.

How do you use this information in a web application?

For popular resources like the company logo, you might to use a public cache control directive and allow everyone to cache the image, even proxy servers. For responses customized to a specific user, you'd want to use a private cache directive.

Every web application framework will allow you to create In ASP.NET you can control cache settings for a response message using the `Response.Cache` property.

A server can also specify a **max-age** value in the `Cache-control` header. The `max-age` value is the number of seconds to cache the response. Once those seconds expire, the request should always go back to the server to retrieve an updated response. Let's look at some sample responses.

Here is a partial response from Flickr.com for one of the Flickr CSS files.

```
HTTP/1.1 200 OK

Last-Modified: Wed, 25 Jan 2012 17:55:15 GMT

Expires: Sat, 22 Jan 2022 17:55:15 GMT
```

`Cache-Control: max-age=315360000,public`

The cache control allows for both public and privates caches, and a cache can store the file for 315 million seconds, which is 10 years. The above message also uses an `Expires` header to give a specific date of expiration. If a client is HTTP 1.1 compliant and understands `Cache-Control` , the client should use the value in `max-age` instead of `Expires` .

The cache headers do not mean Flickr plans on using the same CSS file for 10 years, but the header does mean Flickr is using an aggressive caching strategy. When Flickr does change their design, they'll use a new and different URL for the updated CSS file.

The Flickr response also includes **a Last-Modified** header to indicate when the resource representation last changed. Cache logic can use this value as a **validator** , or a value the client can use to see if the cached representation is still valid. For example, if the browser decides it needs to check on the resource it can issue the following request.

```
GET … HTTP/1.1

If-Modified-Since: Wed, 25 Jan 2012 17:55:15 GMT
```

The `If-Modified-Since` header is telling the server the client only needs the full response if the resource has changed. If the resource hasn't changed, the server can respond with a 304 – Not Modified message.

```
HTTP/1.1 304 Not Modified

Expires: Sat, 22 Jan 2022 17:16:19 GMT

Cache-Control: max-age=315360000,public
```

The server is telling the client, "go ahead and use the bytes you already have cached."

Another validator you'll commonly see is the ETag.

```
HTTP/1.1 200 OK

Server: Apache

Last-Modified: Fri, 06 Jan 2012 18:08:20 GMT

ETag: "8e5bcd-59f-4b5dfef104d00"

Content-Type: text/xml

Vary: Accept-Encoding
```

```
Content-Encoding: gzip

Content-Length: 437
```

The ETag is an opaque identifier, meaning the ETag does not have any inherent meaning. An ETag might be a timestamp, a GUID, or a value computed by running a hashing algorithm against the resource. All the client knows is that if the resource ever changes, the server will compute a new ETag value for the resource. The client can validate a cache entry by comparing two ETags. If the ETags are the same, nothing has changed. If the ETags are different, it's time to invalidate the cache.

Where Are We?

In this chapter, we covered some architectural theory as well as practical benefits of HTTP architecture. The ability to layer caching and other services between a server and client has been a driving force behind the success of HTTP and the web. The visibility of the self-describing HTTP messages and indirection provided by URLs makes the magic of the web possible. In the next chapter, we'll talk about topics like authentication and encryption.

Chapter 5: State and Security

In this last chapter, we will look at the security aspects of HTTP, including how to identify users, how HTTP authentication works and see why some scenarios require HTTPS (secure HTTP). Along the way, we are also going to learn a bit about how to manage state with HTTP.

The Stateless yet Stateful Web

HTTP is a stateless protocol, meaning each request-response transaction is independent of any previous or future transaction. Unlike some other communication protocols, HTTP does not require a server to retain information about an HTTP request or the client making the request. All the server needs to do is generate a response for every request. Every request will carry all the information a server needs to create the response.

The stateless nature of HTTP is one of the driving factors in the success of the web. The layered services we looked at in the last chapter, services like caching, are all made possible (and easier) because every message contains all the information required to process the message. Proxy servers and web servers can inspect, transform and cache messages. Without caching, the web could not and would not scale to meet the demands of the Internet.

However, most of the web applications and services we build on top of HTTP are highly stateful.

A banking application will want a user to login before allowing him to view account related resources. As each request arrives for a private resource, the application will verify the user's identity and ensure he has an authorization to see the resource.

Alternatively, when a user wants to open a banking account she might need to fill out information across three distinct web pages. The application will want to make sure the first page is completed and valid before allowing her to submit the second page of the application.

Fortunately, there are many options for storing state in a web application. One approach is to embed state in the resource transferred to the client, so that all the state required by the application will travel back on the next request. This approach typically requires some hidden input fields and works the best for short-lived state, like the state required for moving through a three-page application process. Embedding state in the resource keeps all the state inside of HTTP messages, so this approach is a highly scalable approach, but it can complicate the application programming.

Another option is to store the state on the server (or behind the server), and this options is required for state that must be around a long time. Let's say the user submits a form to change her e-mail address. The e-mail address must always be associated with the user, so the application can take the new address, validate the address and store the address in a database or file, or call a web service to let someone else take care of saving the address.

For server side storage, many web development frameworks provide access to a **user session** . The session may live in memory or in a database, but a developer can store information in the session and retrieve the information on every subsequent request from the same user. Session storage has an easy programming model but is only good for short-lived state. Eventually, the server must assume the user has left the site or closed his browser and the server will discard the session. Session storage, if stored in memory, can affect scalability negatively because subsequent requests must go to the exact same server where the session data reside. Some load balancers help to support this scenario by implementing **sticky sessions** .

You might be wondering how a server can track a user to implement session state. If two requests arrive at a server, how does the server know if these are two requests from the same user or if there are two different users each making a single request?

In the early days of the web, server software might have differentiated users by looking at the IP address of a request message. These days, however, many users live behind devices using Network Address

Translation, and for this and other reasons, you can have multiple users effectively on the same IP address. An IP address is not a reliable technique for differentiating users.

Fortunately, there are techniques that are more reliable.

Identification and Cookies

Web sites that want to track users will often turn to **cookies** . Cookies are defined by RFC6265, and this RFC is aptly titled *HTTP State Management Mechanism* . When a user first visits a web site, the site can give the user's browser a cookie using an HTTP header. The browser then knows to send the cookie in the headers of every additional request it sends to the site. Assuming the web site has placed some sort of unique identifier into the cookie, then the site can now track a user as they make requests and differentiate one user from another.

Before we get into more 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 from my cookie, but cookies do not necessarily authenticate users. An authenticated user has proved their identity, usually by providing credentials like a username and password. The cookies we are talking about so far just give us some unique identifier to differentiate one user from another, and track a user as they make requests to the site.

Secondly, because cookies can track what a user is doing they raise privacy concerns in some circles. Some users will disable cookies in their browsers, meaning the browser will reject any cookies a server sends in a response. Disabled cookies present a problem for sites that need to track users, of course, and the alternatives are messy. For example, one approach to **cookieless sessions** is to place the user identifier into the URL. Cookieless sessions require that every URL a site gives to a user must contain the proper identifier, and the URLs become much larger (which is why we call this technique the "fat URL" technique).

Setting Cookies

When a website wants to give a user a cookie, it uses a Set-Cookie header in an HTTP response.

```
HTTP/1.1 200 OK

Content-Type: text/html; charset=utf-8

Set-Cookie: fname=Scott&lname=Allen;

            domain=.mywebsite.com; path=/

...
```

There are three areas of information in the cookie shown above. First, there are one or more name-value pairs in a cookie. These name value pairs are delimited by a dollar sign ($), and look similar to query parameters are formatted into a URL. In the example above, the server wanted to store the user’s first name and last name in the cookie. The second and third areas are the domain and path respectively. We'll circle back to talk about domain and path.

A web site can put any information it likes into a cookie, although there is a size limitation of 4KB. Many web sites only put in a unique identifier for a user, perhaps a GUID. A server can never trust any datum on the client unless the server encrypts and signs the data, so it’s usually easy to give the client an ID and keep session data on the server.

```
HTTP/1.1 200 OK

Set-Cookie: GUID=00a48b7f6a4946a8adf593373e53347c;

            domain=.msn.com; path=/
```

Assuming the browser is accepting cookies, the browser will send the cookie to the server in every subsequent HTTP request.

```
GET ... HTTP/1.1

Cookie: GUID=00a48b7f6a4946a8adf593373e53347c;
```

```
...
```

When the ID arrives, the server software can look up any associated user data from an in-memory data structure, database, or distributed cache. You can configure most web frameworks to manipulate cookies and automatically look up session state. For example, in ASP.NET the `Session` object exposes an easy API for reading and writing a user's session state. As developers, we never worry about sending a Set-Cookie header, or reading incoming cookies to find the associated session state. Behind the scenes, ASP.NET will manage the session cookie.

```
Session["firstName"] = "Scott"; // writing session state

...

var lastName = Session["lastName"]; // reading session state
```

Again, I'll mention that the firstName and lastName data stored in the session object is **not living in the cookie** . The cookie only contains a session identifier. The values associated with the session identifier are safe on the server. By default, the session data go into an in-memory data structure and stay alive for 20 minutes by default in ASP.NET. When a session cookie arrives in a request, ASP.NET will associate the correct session data with the Session object after finding the user's data using the ID stored in the cookie. If there is no incoming cookie with a session ID, ASP.NET will create one with a Set-Cookie header.

One security concern around session identifiers is how they open the possibility of someone hijacking another user's session. For example, if I use a tool like Fiddler to trace HTTP traffic I might see a Set-Cookie header come from a server with "SessionID=12" inside. I might guess that some other user already has a SessionID of 11, and create an HTTP request with that ID just to see if I can steal or view the HTML intended for some other user. To combat this problem most web applications will use large random numbers as identifiers (ASP.NET uses 120 bits of randomness), which makes it harder to guess what someone else's session ID would look like. A real ASP.NET session identifier looks like the following.

```
Set-Cookie: ASP.NET_SessionId=en5yl2yopwkdamv2ur5c3z45;

          path=/; HttpOnly
```

HttpOnly Cookies

Another security concern around cookies is how vulnerable they are to a cross-site scripting attack (XSS). In an XSS attack, a malicious user injects malevolent JavaScript code into someone else's web site. If the other web site sends the malicious script to their users, the malicious script can modify, or inspect and steal cookie information.

To combat this vulnerability, we often use the **HttpOnly** flag. You can see the HttpOnly flag in the last Set-Cookie example. The HttpOnly flag tells the browser to prevent access to the cookie from JavaScript. The cookie exists only to travel out in the header of every HTTP request message. Browsers that implement HttpOnly will not allow JavaScript to read or write the cookie on the client.

Types of Cookies

The cookies we've seen so far are **session cookies** . Session cookies exist for a single-user session, and the browser will destroy the cookie when the user closes his browser. **Persistent cookies** can outlive a single browsing session and a browser will store the cookies to disk. You can shut down a computer and come back one week later, go to your favorite web site, and a persistent cookie will still be there for the first request.

The only difference between the two cookie types is that a persistent cookie needs an Expires value.

`Set-Cookie: name=value;` **`expires=Monday, 09-July-2017 21:12:00 GMT`**

A session cookie can explicitly add a `Discard` attribute to a cookie, but without an `Expires` value the browser should discard the cookie in any case.

Cookie Paths & Domains

So far, we've said that once a cookie is set by a web site, the cookie will travel to the web site with every subsequent request, assuming the cookie hasn't expired. However, not all cookies travel to every web site. The only cookies a browser should send to a site are the cookies given to the browser by the same site. It wouldn't make sense for cookies from amazon.com to be in an HTTP request to google.com. This type of behavior would create additional security and privacy concerns. If you set a cookie in a response to a request to www.server.com, the resulting cookie will only travel in the requests to www.server.com.

A web application can change the **scope** of a cookie to restrict the cookie to a specific host or domain, and even to a specific resource path. The web application controls the scope using **domain** and **path** attributes.

```
HTTP/1.1 200 OK

Set-Cookie: name=value; domain=.server.com; path=/stuff

...
```

The domain attribute allows a cookie to span sub-domains. In other words, if you set a cookie from www.server.com, the browser will deliver the cookie only to www.server.com. The domain in the above example also permits the cookie to travel to any URL in the server.com domain, including images.server.com, help.server.com and just plain server.com. You can not use the domain attribute to span domains, so setting the domain to .microsoft.com in a response to .server.com is not legal and the browser should reject the cookie.

The path attribute can restrict a cookie to a specific resource path. In the above example, the cookie will only travel to a server.com site when the request URL is pointing to `/stuff` , or a location underneath `/stuff` , like `/stuff/images` . Path settings can help to organize cookies when multiple teams are building web applications in different paths.

Cookie Downsides

Cookies allow web sites to store information in the client and the information will travel back to the sites in subsequent requests. The benefits to web development are tremendous, because cookies allow us to keep track of which request belongs to which user. But, cookies do have some problems which we've already touched on.

Cookies are vulnerable to XSS attacks and receive bad publicity when sites, particularly advertising sites, use **third-party cookies** to track users across the Internet. Third-parties cookies are cookies that get set from a different domain than the domain in the browser's address bar. Third-party cookies have this opportunity because many web sites will, when sending a page resource back to the client, include links to scripts or images from other URLs. The requests that go to the other URLs allow the other sites to set cookies.

As an example, the home page at server.com can include a <script> tag with a source set to bigadvertising.com. This allows bigadvertising.com to deliver a cookie while the user is viewing content from server.com. The cookie can only go back to bigadvertising.com, but if enough web sites use bigadvertising.com, then Big Advertising can start to profile individual users and the sites they visit. Most web browsers will allow you to disable third-party cookies, but third-party cookies are on by default.

Two of the biggest downsides to cookies, however, are in how they interfere with caching and in how they transmit data with every request. Any response with a Set-Cookie header should not be cached, at least not the headers because caching can interfere with user identification and create security problems. In addition, keep in mind that anything stored in a cookie is visible as it travels across the network (and in the case of a persistent cookie, as it sits on the file system). Because we know there are many devices who can listen and interpret HTTP traffic, a cookie should never store sensitive information. Even session identifiers are risky because if someone can intercept another user's ID they can steal the session data from the server.

Even with all these downsides, cookies are not going away. Sometimes, we need state to travel over HTTP, and cookies offer this capability in an easy, mostly transparent manner. Another capability we sometimes need is the ability to authenticate user.

Authentication

The process of authentication forces a user to prove her identity by entering a user name and password, or an e-mail and a pin, or other type of credentials.

At the network level, authentication typically follows a challenge / response format. The client will request a secure resource, and the server will challenge the client to authenticate. The client then needs to send another request and include authentication credentials for the server to validate. If the credentials are good, the request will succeed.

The extensibility of HTTP allows HTTP to support various authentication protocols. In this chapter, we'll briefly look at the top five: basic, digest, Windows, forms and OpenID. Of these five, only two are officially in the HTTP specification – the basic and digest authentication protocols. We'll look at these two first.

Basic Authentication

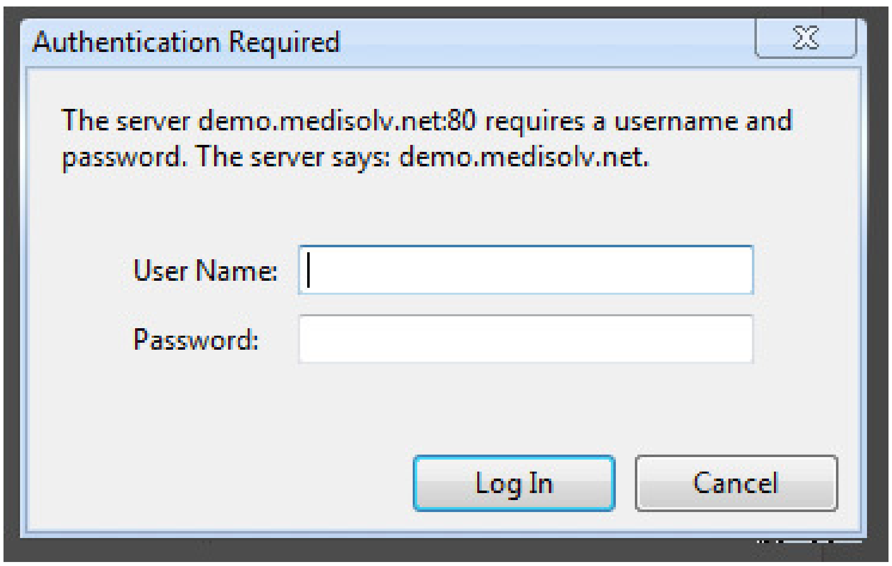
With Basic authentication, the client will first request a resource with a normal HTTP message.

```
GET http://localhost/html5/ HTTP/1.1

Host: localhost
```

Web servers will let you configure access to specific files and directories. You can allow access to all anonymous users or restrict access so that specific users or groups can access a file or directory. For the request above, let's imagine the server is configured to allow only certain users to view the /html5/ resource. In that case, the server will issue an authentication challenge.

```
HTTP/1.1 401 Unauthorized

WWW-Authenticate: Basic realm="localhost"
```

The 401 status code tells the client the request is unauthorized. The WWW-Authenticate header tells the client to collect the user credentials and try again. The realm attribute gives the client a string it can use as a description for the protected area. What happens next depends on the client, but most browsers can display a UI for the user to enter credentials.



With the credentials in hand, the browser can send another request to the server. This request will include an Authorization header.

```
GET http://localhost/html5/ HTTP/1.1

Authorization: Basic bm86aXdvdWxkbnRkb3RoYXQh
```

The value of the authorization header is the client's user name and password in a base 64 encoding. **Basic authentication is insecure by default** because anyone with a base 64 decoder who can view the message can steal a user's password. For this reason, basic authentication is rarely used without using secure HTTP, which we'll consider later.

It's up to the server to decode the authorization header and verify the user name and password with the operating system, or whatever credential management system is on the server. If the credentials match, the server can make a normal reply. If the credentials don't match, the server should respond with a 401 status again.

Digest Authentication

Digest authentication is an improvement over basic authentication because it does not transmit user passwords using base 64 encoding. Instead, the client must send a **digest** of the password. The client computes the digest using the MD5 hashing algorithm with a nonce the server provides during the authentication challenge. A nonce is a cryptographic number used to help prevent replay attacks.

The digest challenge response is like the basic authentication challenge response but with additional values coming from the server in the WWW-Authenticate header for use in the cryptographic functions.

```
HTTP/1.0 401 Unauthorized

WWW-Authenticate: Digest realm="localhost",

                qop="auth,auth-int",

                nonce="dcd98b7102dd2f0e8b11d0f600bfb0c0
                93",

                opaque="5ccc069c403ebaf9f0171e9517f40e4
                1"
```

Digest authentication is better than basic authentication when secure HTTP is not available, but it is still far from perfect because the MD5 hashing algorithm is a weak form of encryption. Digest authentication is also vulnerable to man in the middle attacks where someone is sniffing network traffic.

Windows Authentication

Windows Integrated Authentication is a standard authentication protocol among Microsoft products. However, many modern browsers and not just Internet Explorer support Windows Authentication. Windows Auth does not work well over the Internet. You’ll find Windows Auth is common on internal and intranet web sites where a Microsoft Active Directory server exists.

Windows Authentication depends on the underlying authentication protocols supported by Windows, including NTLM and Kerberos. The Windows Authentication challenge and response steps are very similar to what we've seen already, but the server will specify NTLM or Negotiate in the Www-Authenticate header (Negotiate is a protocol that allows the client to select Kerberos or NTLM).

```
HTTP/1.1 401 Unauthorized

WWW-Authenticate: Negotiate
```

Windows Authentication has the advantage of being secure even without using secure HTTP and of being unobtrusive for users of Internet Explorer. IE will automatically authenticate a user when a server challenges a request and will do so using the user's credentials that they used to log into the Windows operating system.

Forms-based Authentication

Forms authentication is the most popular approach to user authentication over the Internet. Forms-based authentication is not a standard authentication protocol and doesn't use WWW-Authenticate or Authorization headers. However, many web application frameworks provide some out of the box support for forms-based authentication.

With forms-based authentication, an application will respond to a request for a secure resource by an anonymous user by redirecting the user to a login page. The redirect is an HTTP 302 temporary redirect. Generally, the URL the user is requesting might be included into the query string of the redirect location so that once the user has completed the login, the application can redirect the user to the secure resource they were trying to reach.

```
HTTP/1.1 302 Found
```

```
Location: /Login.aspx?ReturnUrl=/Admin.aspx
```

The login page for forms-based authentication is an HTML form with inputs for the user to enter their credentials. When the user clicks submit, the form values will POST to a destination where the application needs to take the credentials and validate them against a database record or operating system.

```
<form method="post">

  ...

  <input type="text" name="username" />

  <input type="password" name="password" />

  <input type="submit" value="Login" />

</form>
```

Note that forms-based authentication will transmit a user's credentials in plain text, so like basic authentication, forms-based authentication is not secure unless you use secure HTTP. In response to the POST message with credentials, assuming the credentials are good, the application will typically redirect the user back to the secure resource and also set a cookie indicating the user is now authenticated.

```
HTTP/1.1 302 Found

Location: /admin.aspx

Set-Cookie: .ASPXAUTH=9694BAB... path=/; HttpOnly
```

Most authentication cookies will hold an encrypted and hashed value to prevent tampering. However, without secure HTTPS the cookie is vulnerable to interception, so session hijacking is still a potential problem. Still, forms authentication remains popular because it allows applications complete control over the login experience and credential validation.

OpenID and OAuth

While forms-based authentication gives an application total control over user authentication, many applications don't want this level of control. Specifically, applications don't want to manage and verify usernames and passwords, and users don't want to have a different username and password for every website. OpenID and OAuth are standards allowing for decentralized authentication. With OpenID, a user registers with an OpenID identity provider, and the identity provider is the only site that needs to store and validate user credentials. There are many OpenID providers around, including Google, Yahoo, and Verisign.

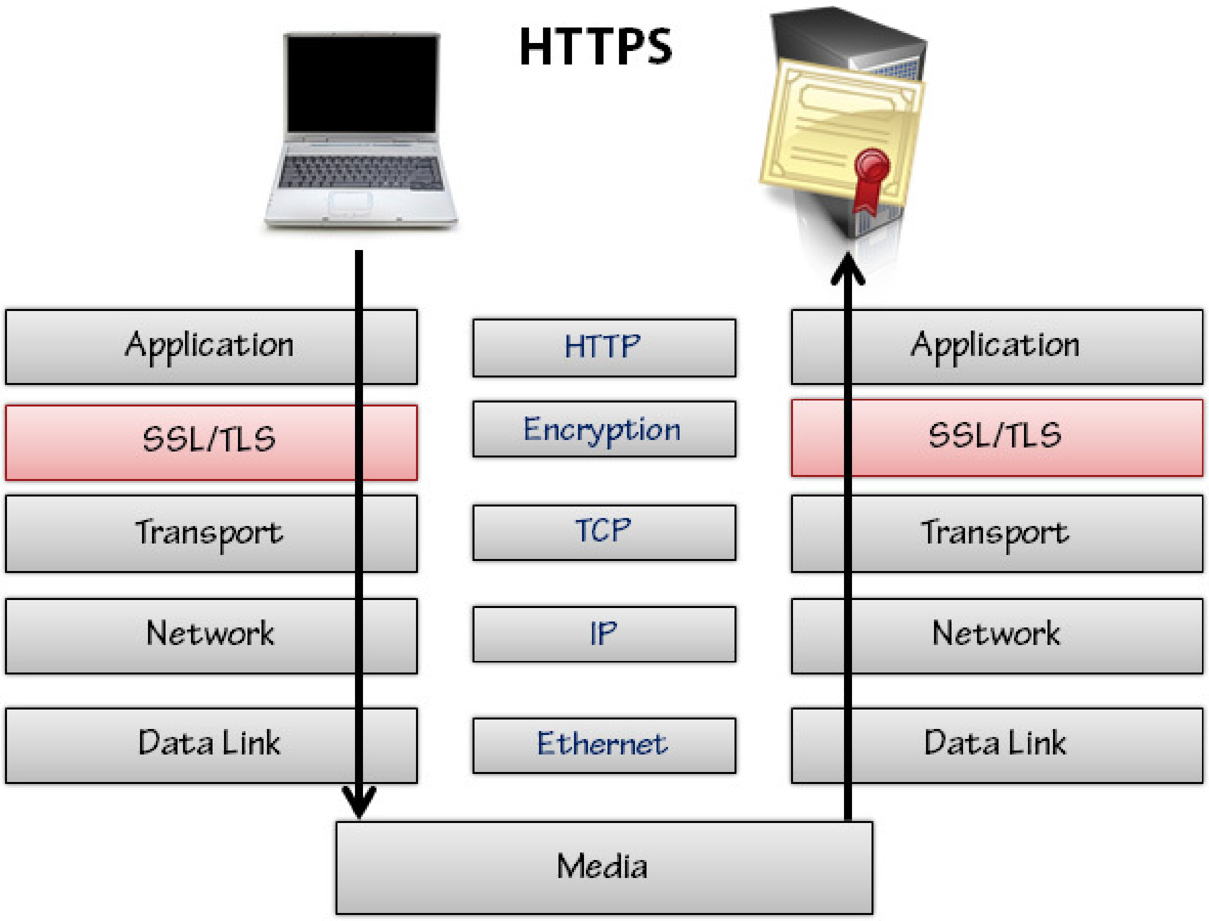
When an application needs to authenticate a user, it works with the user and the user's identity provider. The user ultimately must verify their user name and password with the identity provider, but the application will know if the authentication is successful thanks to the presence of cryptographic tokens and secrets. Google has an overview of the process on their “Federated Login for Google Account Users” web page.

While OpenID and OAuth offer many benefits compared to forms authentication, they do add complexity in implementing and debugging a web application.

Secure HTTP

Previously, we mentioned how the self-describing textual HTTP messages are one of the strengths of the web. Anyone can read a message and understand what's inside it. But, there are many messages we send over the web that we don't want anyone else to see. We've discussed some of those scenarios in this book. We don't want anyone else on the network to see our passwords, for example, but we also don't want them to see our credit card numbers or bank account numbers. Secure HTTP solves this problem by encrypting messages before the messages start traveling across the network.

Secure HTTP, a.k.a HTTPS, uses an https scheme in the URL instead of a regular http scheme. 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 there is an explicit port number present in the URL. HTTPS works by using an additional security layer in the network protocol stack. The security layer exists between the HTTP and TCP layers and features the use of the Transport Layer Security protocol (TLS) or the TLS predecessor known as Secure Sockets Layer (SSL).



HTTPS requires a server to have a cryptographic certificate. The server sends the certificate to the client during setup of the HTTPS communication. The certificate includes the server's host name, and a browser can use the certificate to validate that the server it is talking to is the correct server. The validation happens using public key cryptography and certificate authorities, like Verisign, that will sign and vouch for the integrity of a certificate. Server administrators must purchase and install certificates from the certificate authorities.

From a developer's perspective, the three most important things to know about HTTPS are the following:

- **HTTPS will encrypt all request and response traffic,** including the HTTP headers and message body, and everything after the host name in the URL. This means the path and query string data remain encrypted as well as all cookies. HTTPS prevents session hijacking because no eavesdroppers can inspect a message and steal a cookie.

- **The server certificate authenticates the server to the client** . If you are talking to mybigbank.com over HTTPS, you can be sure your messages are really going to mybigbank.com and not some evil person who stuck a proxy server on the network to intercept requests and spoof response traffic from mybigbank.com.

- **HTTPS does not authenticate the client.** Applications still need to implement forms authentication or one of the other authentication protocols mentioned earlier if they need to know the user's identity.

HTTPS does make forms-based authentication and basic authentication more secure because all data are encrypted. There is the possibility of using client-side certificates with HTTPS, and client-side certificates would authenticate the client in the most secure manner possible. However, most sites on the Internet won't use client-side certificates because most users don't want the hassle and expense of purchasing and installing a personal certificate. Corporations might require client certificates for employees to access corporate servers, but in this case, the corporation can act as a certificate authority and issue employees certificates they create and manage.

HTTPS does have some downsides and most of them are performance-related.

HTTPS is computationally expensive, and large sites often use specialized hardware (SSL accelerators) to take the cryptographic computation load off the web servers. HTTPS traffic is also impossible to cache in a public cache, however, user agents might keep HTTPS responses in their private cache.

Finally, HTTPS connections are expensive to set up and require an additional handshake between the client and server to exchange cryptographic keys and ensure everyone is communicating with the proper secure protocol. Persistent connections can help to amortize this cost.

In the end, if you need secure communications, you'll willingly pay the performance penalties.

Where Are We?

In this chapter, we've discussed cookies, authentication and secure HTTP. If you've read all five chapters I hope you've found some valuable information that will help you as you write, maintain and debug web applications.

End Matter

Thanks for reading. I hope you found the book useful and informative for your everyday work!

-- Scott Allen

About Scott

Scott Allen is a founder and principal consultant with OdeToCode LLC. Find out more at OdeToCode.com , or on Twitter at @OdeToCode .