HTTP Succinctly

by Scott Allen

Syncfusion | Technology Resource Portal

HTTP Succinctly

By
Scott Allen

Foreword by Daniel Jebaraj

Syncfusion®
Deliver innovation with ease®

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Edited by

This publication was edited by Daniel Jebaraj, vice president, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Scott Allen is a founder and principal consultant with OdeToCode LLC.

Scott has more than 20 years of commercial software development experience across a wide range of technologies. He's successfully delivered software products for embedded, Windows, and web platforms. He's also developed web services for Fortune 50 companies and firmware for startups.

Scott is available for consulting through OdeToCode LLC. Scott also offers training classes in the following areas:

- C#
- Test-Driven Development
- ASP.NET MVC
- HTML 5, JavaScript, and CSS 3
- LINQ and the Entity Framework

You can reach Scott via email at scott@OdeToCode.com.

http://odetocode.com/blogs/scott

http://twitter.com/OdeToCode

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

—Scott Allen

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 webpage describing the Ming dynasty in China.

In this book we'll look at HTTP from a software developer's perspective. Having a solid understanding of HTTP can help you write better web applications and web services. It can also help you debug applications and services when things go wrong. We'll be covering all 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 website 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 website. Resources are things I want to interact with on the web. Images, pages, files, and videos are all resources.

There are billions, if not trillions, of places to go on the Internet—in other words, there are trillions of resources. Each resource will have a URL I can use to find it. **`http://news.google.com`** is a different place than **`http://news.yahoo.com`**. These are two different names, two different companies, two different websites, and therefore two different URLs. Of course, there will also be different URLs inside the same website. **`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 look at a different scheme, HTTPS, which is the secure HTTP protocol. You might run into other schemes too, like FTP for the file transfer protocol, and mailto for email addresses.

   Everything after the **`://`** will be specific to a particular scheme. So, 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 the browser the name of the computer hosting the resource. The 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 can also 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 also 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 built using ASP.NET, PHP, Perl, 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 websites 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 .aspx 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. Descriptive keywords, not file names, are important for URLs these days.

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

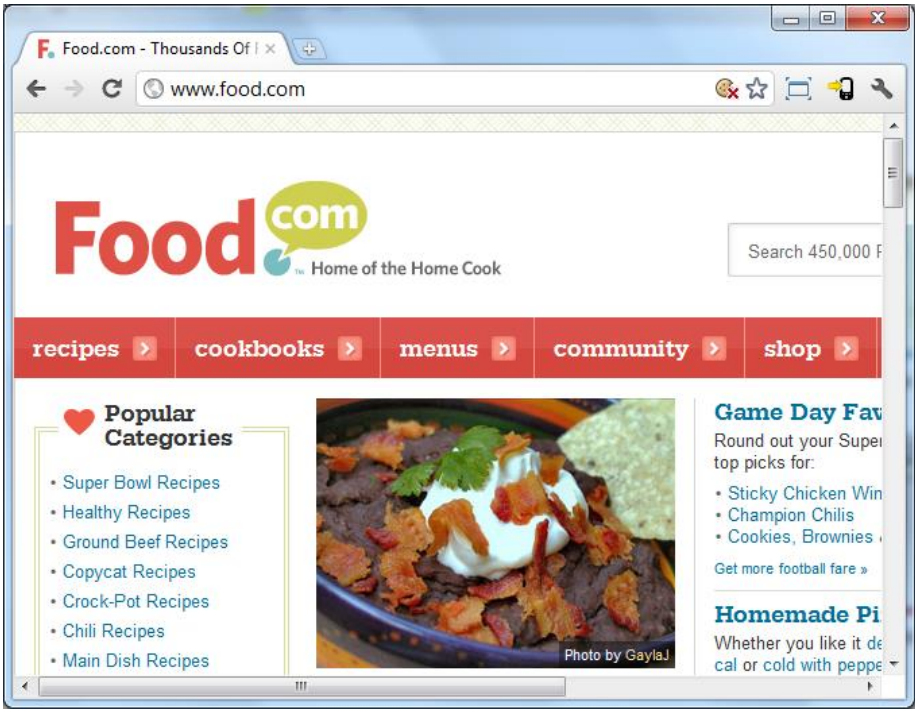


Figure 1: food.com home page

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**, contains information for the destination website to use or interpret. There is no formal standard for how the query string should look as 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 this example. 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 # sign is known as the **fragment**. The fragment is different than the other pieces we've looked at 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 webpage 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 "feedback". If you follow the URL, your web browser should scroll down the page to show the feedback section of a particular blog post on my blog. 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 the “feedback” ID 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>
```

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 email 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 (is that one space or two spaces on your business card?).

Other unsafe characters include the number sign (#) because it is used to delimit a fragment, and the caret (^) 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 a few special characters like the colon (:) and the slash mark (/).

Fortunately, you can still transmit unsafe characters in a URL, but all unsafe characters must be percent-encoded (aka 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, Microsoft Word documents, and countless more.

In order for a host to properly serve a resource, and in order for a client to properly display a resource, the parties involved have to 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 rendering a PNG image as text, and we wouldn't want them to try interpreting hypertext as an image.

When a host responds to an HTTP request, it returns a resource and also 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 email 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 webpage, 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, and 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 in JPG format. Microsoft documents Internet Explorer (IE) as first looking at the content type tag specified by the host. If the host doesn't provide a content 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 content type and can't guess the content 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 webpages, 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, and this is why the media type settings are important.

However, media types aren't just for hosts. Clients 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, 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. The media types are not only for the host to tag outgoing resources, but also for clients to specify the media type they want to consume.

The client specifies what it will accept in the outgoing request message. Again, we'll see details of this message in Chapter 2, but imagine a request to **`http://food.com/`** saying it will accept a representation in the German language. It's up to the server to try fulfilling the request. The host might send a textual resource that is still in English, which will probably disappoint a German-speaking user, but this is why we call it content negotiation and not 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?" In order for the stranger to respond with the correct time, a few things have to be in place. First, the stranger has to understand your question, because if he or she does not know English, he or she might not be able to make any response. Secondly, the stranger will need access to a watch or some other time-keeping device.

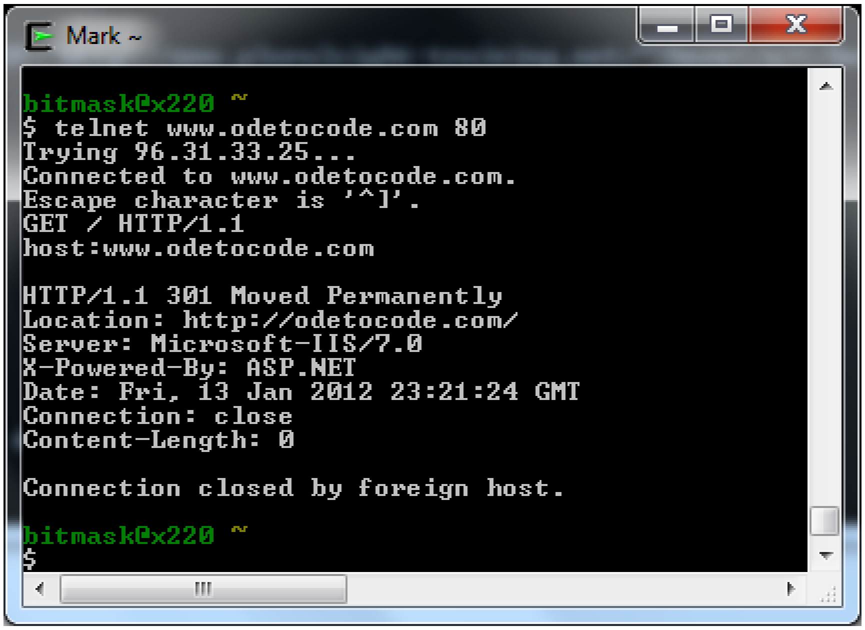
This airport analogy is similar to how HTTP works. You, the client, need a resource from some other party (the resource being information about the time of day). So, you make a request to the other party using a language and vocabulary you hope the other party will understand. If the other party understands your request and has the resource available, it can reply. If it understands the request but doesn't have the resource, it can still respond and tell you it doesn't know. If the other party doesn't understand what you are saying, you might not get 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 these request and response messages so that everyone who speaks "HTTP" will understand each other and be able to exchange resources (or when a resource doesn't 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 request—it's just a command in plain ASCII text and formatted according to 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.

The following figure is a screenshot of a Telnet session that connects to odetocode.com on port 80, makes an HTTP request, and receives an HTTP response.



```
bitmask@x220 ~
$ telnet www.odetocode.com 80
Trying 96.31.33.25...
Connected to www.odetocode.com.
Escape character is '^]'.
GET / HTTP/1.1
host:www.odetocode.com

HTTP/1.1 301 Moved Permanently
Location: http://odetocode.com/
Server: Microsoft-IIS/7.0
X-Powered-By: ASP.NET
Date: Fri, 13 Jan 2012 23:21:24 GMT
Connection: close
Content-Length: 0

Connection closed by foreign host.

bitmask@x220 ~
$
```

Figure 2: Making an HTTP request

The Telnet session starts by typing:

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

Please note that the Telnet client is not installed by default on Windows 7, Windows Server 2008 R2, Windows Vista, or Windows Server 2008. You can install the client by following the procedure listed at http://technet.microsoft.com/en-us/library/cc771275(v=ws.10).aspx.

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 then pressing **Enter**:

```
GET / HTTP/1.1
```

This information will tell the server we want to retrieve the resource located at "/" (i.e. 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 websites, i.e. both www.odetocode.com and www.odetofood.com could be hosted on the same server, and the host information in the message will help the web server direct the request to the proper web application.

After typing the previous two lines we can press **Enter** 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 types of "redirects" are common, and in this scenario the reason is to make sure all the requests for resources from OdeToCode go through odetocode.com and not www.odetocode.com (this is a search engine optimization known as URL canonicalization).

Now that we've 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 one of the HTTP methods, and the method tells the server what the request wants to do. An HTTP **`GET`** wants to get, fetch, and 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 operators is shown in the following table.

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

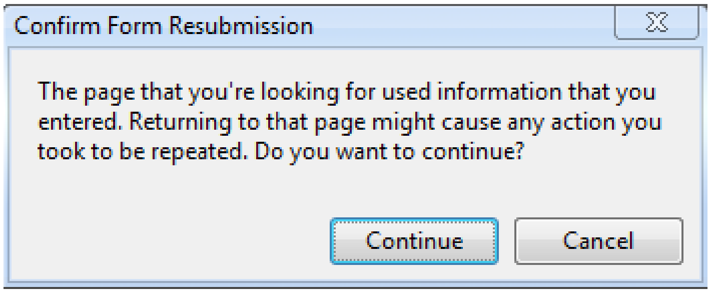
Of these five methods, just two are the primary workhorses of the web: **`GET`** and **`POST`**. A web browser issues a **`GET`** request when it wants to retrieve a resource, like a page, an image, a video, or a document. **`GET`** requests are the most common type of request.

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. **`POST`** requests are typically generated by a **`<form>`** on a webpage, like the form you fill out with **`<input>`** elements for address and credit card information.

GET and Safety

There is a part of the HTTP specification that talks about the "safe" HTTP methods. **Safe methods**, as the name implies, don't do anything "unsafe" like destroy a resource, submit a credit card transaction, or cancel an account. The **`GET`** method is one of the safe methods since it 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 does some other special operation. Web browsers typically treat **`GET`** and **`POST`** differently since **`GET`** is safe and **`POST`** is unsafe. It's OK to refresh a webpage retrieved by a **`GET`** request—the web browser will just reissue the last **`GET`** request and render whatever the server sends back. However, if the page we are looking at in a browser 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.



Confirm Form Resubmission

The page that you're looking for used information that you entered. Returning to that page might cause any action you took to be repeated. Do you want to continue?

Figure 3: Refreshing a POST request

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 submitting an order), the server will process the information and respond with an HTTP redirect (like the redirect we saw in the Telnet window) telling the browser to **`GET`** some other resource. The browser will issue the **`GET`** request, the server will respond with a "thank you for the order" resource, and then the user can refresh or print the page safely as many times as he or she would like. This is a common web design pattern known as the **`POST`/Redirect/`GET`** pattern.

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

Common Scenario—GET

Let's say you have a page and want the user to click a link to view the first article in this series. In this case a simple hyperlink is all you need.

```
<a href="http://odetocode.com/Articles/741.aspx">Part I</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/741.aspx HTTP/1.1
Host: odetocode.com
```

Scenario—POST

Now imagine you have a page where the user has to fill out 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 tells the browser that the HTTP method to use is **`POST`**, and the path to **`POST`** is **`/account/create`**. The actual HTTP request the browser makes will look something like this.

```
POST http://localhost:1060/account/create HTTP/1.1
Host: server.com
firstName=Scott&lastName=Allen
```

Notice the form inputs are included in the HTTP message. This is very similar to how parameters appear in a URL, as we saw in Chapter 1. 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 telling the user that the account has been created. Doing so will leave the user viewing the result of a **`POST`** request, which could lead to issues if he or she refreshes the page—it might try to sign them up a second time!

2. Respond with a redirect instruction like we saw earlier to have the browser issue a safe **`GET`** request for a page that tells the user the account has been created.

3. Respond with an error, or redirect to an error page. We'll take a 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 on this form is **GET**, not **POST**. That's because 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://localhost:1060/search?term=love HTTP/1.1
Host: searchengine.com
```

Notice instead of putting the input values into the body of the message, 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 send it in an email. The user could also refresh the page as many times as he or she would like, again because the **GET** operation for the search results is a safe operation that won't 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 a 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 pointing inside of 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 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 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 the database. The **/account/create** resource 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 (most commonly 1.1, which has been around since 1999). 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** (remember, in HTTP 1.1 **host** is a required header). Headers contain useful information that can help a server process a request. For example, in Chapter 1 we talked about resource representations and how the client and server can negotiate on the best representation of a resource (content negotiation). If the client wants to see a resource in French, for example, it can include a header entry (the **Accept-Language** header) requesting French content.

```
GET http://odetocode.com/Articles/741.aspx HTTP/1.1
Host: odetocode.com
Accept-Language: fr-FR
```

There are numerous 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. The client or server can include a **Date** header indicating when it created the message.

```
GET http://odetocode.com/Articles/741.aspx HTTP/1.1
Host: odetocode.com
Accept-Language: fr-FR
Date: Fri, 9 Aug 2002 21:12:00 GMT
```

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

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

Header	**Description**
Referer	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 (Internet Explorer 6 versus Internet Explorer 9 versus Chrome, etc.).
Accept	Describes the media types the user agent is willing to accept. This header is used for content negotiation.

Header	Description
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, like the **Accept** header. The **Accept** header is listing the MIME types it likes to see, including HTML, XHTML, XML, and finally */* (meaning I like HTML the best, but you can send me anything (*/*) and I'll try to figure it out).

Also notice the appearance of "**q**" in some of the headers. The **q** value is always a number from 0 to 1 and represents the **quality value** or "relative degree of preference" for a particular value. The default is 1.0, and higher numbers indicate a higher preference.

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 request starts off 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 and 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 will detail 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 check this URL again. We saw an example of this earlier when we used Telnet and the server redirected us from www.odetocode.com to odetocode.com to give search engines a canonical URL.
302	Moved Temporarily	The resource has moved to the URL specified in the **`Location`** header. In the future, the client can still request the URL because it's a temporary move. This type of response code is typically used after a **`POST`** operation to move a client to a resource it can retrieve with **`GET`** (the **`POST`**/Redirect/**`GET`** 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, so it can just use a locally cached copy.
400	Bad Request	The server could not understand the request. The request probably used incorrect syntax.
403	Forbidden	The server refused access to the resource.
404	Not Found	A popular code meaning the resource was not found.
500	Internal Server Error	The server encountered an error in processing the request. Commonly happens because of programming errors in a web application.

Code	Reason	Description
503	Service Unavailable	The server will currently not service the request. This status code can appear when a server is throttling requests because it is under heavy load.

Response status codes are an incredibly important part of the HTTP message because they tell the client what happened (or in the case of redirects, where to go next).

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 doesn't necessarily reflect what happened inside your application. For example, imagine a user submits a sign-in form to the server, but didn't fill out the Last Name field. If your application requires a last name it will fail to create an account for the user. This doesn't mean you have to return an HTTP error code indicating failure. You probably want quite the opposite to happen—you want to successfully return some content to the client with a 200 (OK) status code. The content will tell the user a last name was not provided. From an application perspective the request was a failure, but from an HTTP perspective the request was successfully processed. This is normal in web applications.

Response Headers

A response includes header information that gives a client metadata it can use to process the response. For example, the content type will be specified as a MIME type, as we talked about in Chapter 1. In the following response we can see the content type is HTML, and the character set used to encode the type is UTF-8. The headers can also contain information about the server, like the name of the software and the version.

```
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 that appear 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.

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 `ETag`s is an efficient way to know if something needs to be refreshed. An `Expires` header tells a client how long to cache a particular resource. We'll return and look at caching in more detail later.

Where Are We?

In this chapter we've learned that HTTP messages always come in pairs. First there is the request, and then there is the response. The information in these messages is all in readable text, and there are lots of tools you can use to inspect HTTP requests being made on your machine. Fiddler is one such tool if you are running Windows (http://fiddler2.com). It's easy to use, and you can see the raw HTTP requests being made, including all of the headers.

Messages are all about making sure both parties in a 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 to identify what should happen to a particular resource. In a response the status code will indicate how the request was processed. 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 looked at HTTP messages and saw examples of the text commands and codes that flow from the client to the server and back in an HTTP transaction. But, how does the information in these messages move through the network? When are the network connections opened? When are the connections closed? These are some of the questions this article will answer as we look at HTTP from a low-level perspective. But first, we'll need to understand some of the abstractions below HTTP.

A Whirlwind Tour of Networking

To understand HTTP connections we have to know just a bit about what happens in the layers underneath HTTP. Network communication, like many applications, consists of layers. Each layer in a **communication stack** is responsible for a specific and limited number of responsibilities.

For example, HTTP is what we call an application layer protocol because it allows two applications to communicate over a 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. But, the HTTP specifications don't say anything about how the messages actually cross the network and reach the server—that's the job of lower-layer protocols. A message from a web browser has to travel down a series of layers, and when it arrives at the web server it travels up through a series of layers to reach the web service process.

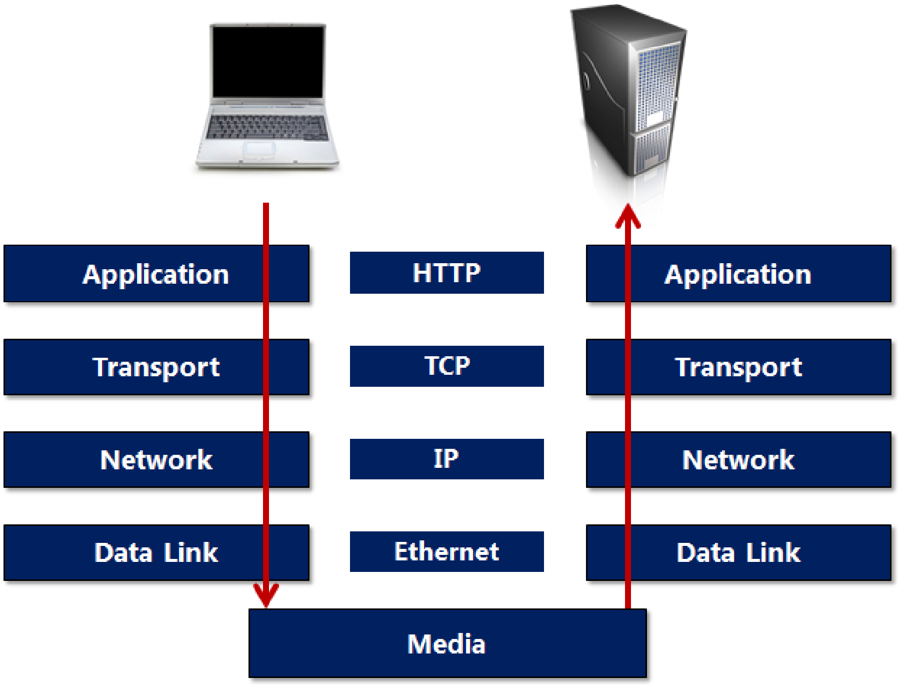


Figure 4: Protocol layers

The layer underneath HTTP is a **transport layer protocol**. Almost all HTTP traffic travels over **TCP** (short for Transmission Control Protocol), although this isn't 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 ensures the message gets delivered to the server without getting lost or duplicated. TCP will automatically resend any information that might get lost in transit, and this is why TCP is known as 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 previous figure 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 it doesn't guarantee delivery—that's TCP's job). IP requires computers to have an address (the famous IP address, an example being 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 particular network segment.

Everything we've talked about so far happens inside a computer, but eventually these IP packets have to 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**. At this level, data packets become frames, and low-level protocols like Ethernet are focused on 1s, 0s, and electrical signals.

Eventually the signal reaches the server and comes in through a network card where the process is reversed. 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 it into the web server process. It's a beautifully engineered piece of work all made possible by standards.

Quick HTTP Request with Sockets and C#

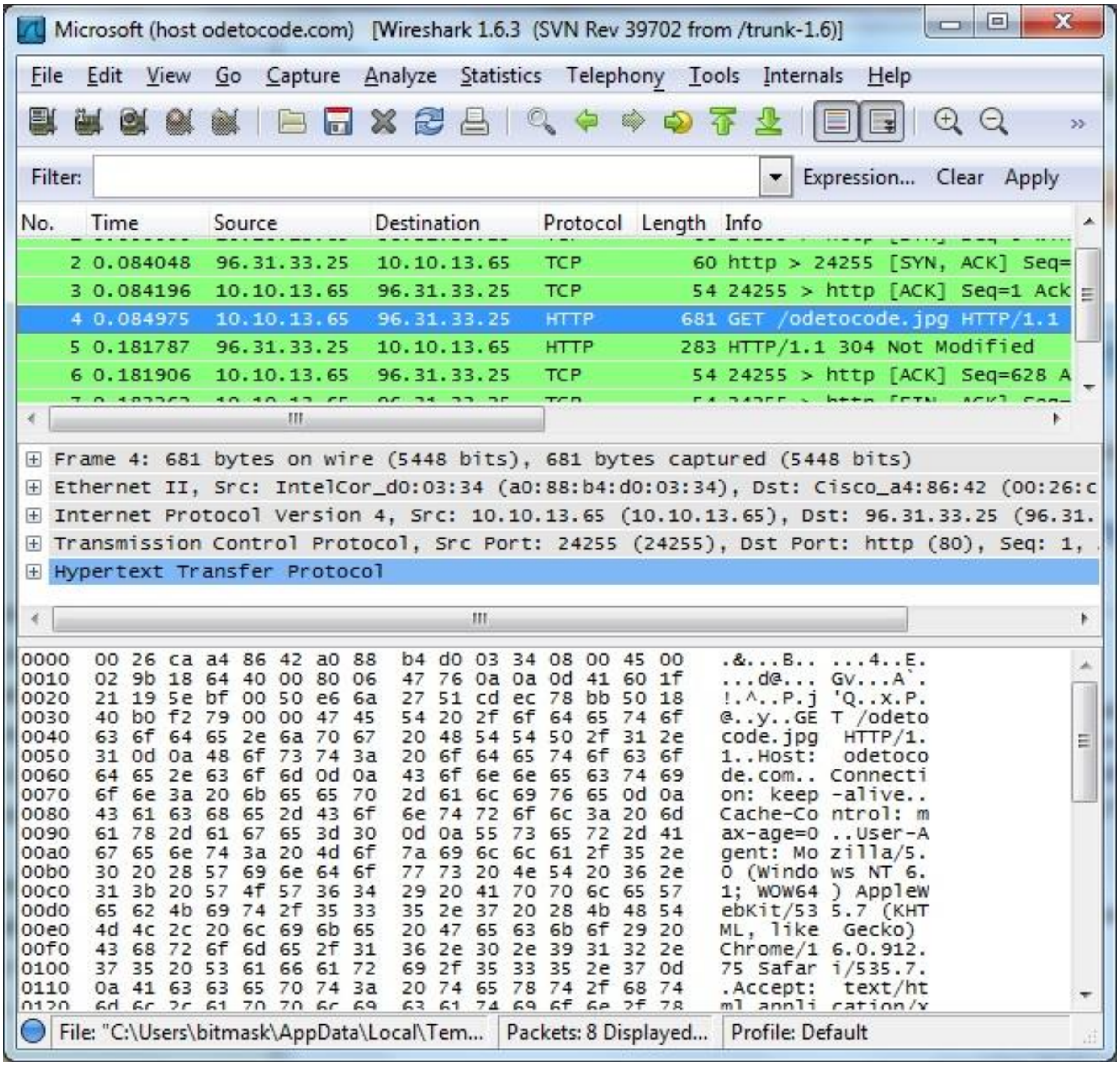
If you are wondering what it looks like to write an application that will make HTTP requests, then the following C# code is a simple example of what the code might look like. This code does not have any error handling, and tries to write any server response to the console window (so you'll need to request a textual resource), but it works for simple requests. A copy of the following code sample is available from https://bitbucket.org/syncfusiontech/http-succinctly. The sample name is sockets-sample.

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

public class GetSocket
{
    public static void Main(string[] args)
    {
        var host = "www.wikipedia.org";
        var resource = "/";

        Console.WriteLine("Connecting to {0}", host);

        if(args.GetLength(0) >= 2)
        {
            host = args[0];
            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 address (using **`Dns.GetHostEntry`**), and formulate a proper HTTP message with a **`GET`** operator and **`Host`** header. The actual networking part is fairly easy, because the socket implementation and TCP take care of most of the work. TCP understands, for example, how to manage multiple connections to the same server (they'll all receive different port numbers locally). Because of this, two outstanding requests to the same server won't get confused and receive data intended for the other.

Networking and Wireshark

If you want some visibility into TCP and IP you can install a free program such as Wireshark (available for OSX and Windows from 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 the client and server before the actual HTTP messages start flowing. You can also see the TCP and IP headers (20 bytes each) on every message. The following figure shows the last two steps of the handshake, followed by a **`GET`** request and a **`304`** redirect.



```
No.  Time      Source        Destination   Protocol Length Info
   2 0.084048  96.31.33.25   10.10.13.65   TCP          60 http > 24255 [SYN, ACK] Seq=
   3 0.084196  10.10.13.65   96.31.33.25   TCP          54 24255 > http [ACK] Seq=1 Ack
   4 0.084975  10.10.13.65   96.31.33.25   HTTP        681 GET /odetocode.jpg HTTP/1.1
   5 0.181787  96.31.33.25   10.10.13.65   HTTP        283 HTTP/1.1 304 Not Modified
   6 0.181906  10.10.13.65   96.31.33.25   TCP          54 24255 > http [ACK] Seq=628 A
```

```
⊞ Frame 4: 681 bytes on wire (5448 bits), 681 bytes captured (5448 bits)
⊞ Ethernet II, Src: IntelCor_d0:03:34 (a0:88:b4:d0:03:34), Dst: Cisco_a4:86:42 (00:26:c
⊞ Internet Protocol Version 4, Src: 10.10.13.65 (10.10.13.65), Dst: 96.31.33.25 (96.31.
⊞ Transmission Control Protocol, Src Port: 24255 (24255), Dst Port: http (80), Seq: 1,
⊞ Hypertext Transfer Protocol
```

```
0000   00 26 ca a4 86 42 a0 88  b4 d0 03 34 08 00 45 00   .&...B.. ...4..E.
0010   02 9b 18 64 40 00 80 06  47 76 0a 0a 0d 41 60 1f   ...d@... Gv...A`.
0020   21 19 5e bf 00 50 e6 6a  27 51 cd ec 78 bb 50 18   !.^..P.j 'Q..x.P.
0030   40 b0 f2 79 00 00 47 45  54 20 2f 6f 64 65 74 6f   @..y..GE T /odeto
0040   63 6f 64 65 2e 6a 70 67  20 48 54 54 50 2f 31 2e   code.jpg  HTTP/1.
0050   31 0d 0a 48 6f 73 74 3a  20 6f 64 65 74 6f 63 6f   1..Host:  odetoco
0060   64 65 2e 63 6f 6d 0d 0a  43 6f 6e 6e 65 63 74 69   de.com.. Connecti
0070   6f 6e 3a 20 6b 65 65 70  2d 61 6c 69 76 65 0d 0a   on: keep -alive..
0080   43 61 63 68 65 2d 43 6f  6e 74 72 6f 6c 3a 20 6d   Cache-Co ntrol: m
0090   61 78 2d 61 67 65 3d 30  0d 0a 55 73 65 72 2d 41   ax-age=0 ..User-A
00a0   67 65 6e 74 3a 20 4d 6f  7a 69 6c 6c 61 2f 35 2e   gent: Mo zilla/5.
00b0   30 20 28 57 69 6e 64 6f  77 73 20 4e 54 20 36 2e   0 (Windo ws NT 6.
00c0   31 3b 20 57 4f 57 36 34  29 20 41 70 70 6c 65 57   1; WOW64 ) AppleW
00d0   65 62 4b 69 74 2f 35 33  35 2e 37 20 28 4b 48 54   ebKit/53 5.7 (KHT
00e0   4d 4c 2c 20 6c 69 6b 65  20 47 65 63 6b 6f 29 20   ML, like  Gecko)
00f0   43 68 72 6f 6d 65 2f 31  36 2e 30 2e 39 31 32 2e   Chrome/1 6.0.912.
0100   37 35 20 53 61 66 61 72  69 2f 35 33 35 2e 37 0d   75 Safar i/535.7.
0110   0a 41 63 63 65 70 74 3a  20 74 65 78 74 2f 68 74   .Accept:  text/ht
0120   6d 6c 2c 61 70 70 6c 69  63 61 74 69 6f 6e 2f 78   ml,appli cation/x
```

Figure 5: Using Wireshark

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 webpages require more than a single resource to fully render. Every page in a web application has one or more images, one or more JavaScript files, and one or more CSS files. It's not uncommon for the initial request for a home page to spawn off 30 or 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 fully download before starting the next download, the web would feel very slow. The Internet is full of latency. Signals have to travel long distances and wind their way through different pieces of hardware. There is also some overhead in establishing a TCP connection. As we saw in the Wireshark screenshot, there is a three-step handshake to complete before an HTTP transaction can begin.

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

Parallel Connections

Most user agents (aka web browsers) will not make requests in a serial one-by-one fashion. Instead, they 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 parallel connections to download the two images simultaneously. The number of parallel connections depends on the user agent and the agent's configuration.

For a long time we considered two 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—Internet Explorer (IE) 6—would only allow two simultaneous connections to a single host. IE was only obeying the rules spelled out in the HTTP 1.1 specification, which states:

> *A single-user client SHOULD NOT maintain more than 2 connections with any server or proxy.*

To increase the number of parallel downloads, many websites use some tricks. For example, the two-connection limit is *per host*, meaning a browser like IE 6 would happily make two parallel connections to www.odetocode.com, *and* two parallel connections to images.odetocode.com. By hosting images on a different server, websites could increase the number of parallel downloads and make their pages load faster (even if the DNS records were set up to point all four requests to the same server, because the two-connection limit is *per host name, not IP address*).

Things are different today. Most user agents will use a different set of heuristics when deciding how many parallel connections to establish. For example, Internet Explorer 8 will now open up to six concurrent connections.

The real question to ask is: 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. Thus, having too many connections can hurt performance. Also, a server can only accept a finite number of connections, so if 100,000 browsers simultaneously create 100 connections to a single web server, bad things will happen. Still, using more than one connection per agent is better than downloading everything in a serial fashion.

Fortunately, parallel connections are not the only performance optimization.

Persistent Connections

In the early days of the web, a user agent would open and close a connection for each individual request it sent to a server. This implementation was in accordance with HTTP's idea of being a completely stateless protocol. 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 user agent with an already open socket it can use to continue making requests to the server without the overhead of opening a new socket. Persistent connections also avoid the slow start strategy that is part of TCP congestion control, making persistent connections perform better over time. In short, persistent connections reduce memory usage, reduce CPU usage, reduce network congestion, reduce latency, and generally improve the response time of a page. But, like everything in life there is a downside.

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's difficult to give an exact number, but generally speaking, if you talk about supporting thousands of concurrent connections, you'll have to start testing to see if a server will support the load. In fact, many servers are configured to limit the number of concurrent connections far below the point where the server will fall over. The configuration is a security measure to help prevent denial of service attacks. It's relatively easy for someone to create a program that will open thousands of persistent connections to a server and keep the server from responding to real clients. Persistent connections are a performance optimization but also a vulnerability.

Thinking along the lines of a vulnerability, we also have to wonder how long to keep a persistent connection open. In a world of infinite scalability, the connections could stay open for as long as the user-agent program was running. But, because a server supports a finite number of connections, most servers are configured to close a persistent connection if it is idle for some period of time (five seconds in Apache, for example). User agents can also close connections after a period of idle time. The only visibility into connections closed is through a network analyzer like Wireshark.

In addition to aggressively closing persistent connections, most web server software can be configured to disable persistent connections. This is common with shared servers. 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 has to include a **Connection** header in every HTTP response. The following code is an example.

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Server: Microsoft-IIS/7.0
X-AspNet-Version: 2.0.50727
X-Powered-By: ASP.NET
Connection: close
Content-Length: 17149
```

The **Connection: close** header is a signal to the user agent that the connection will not be persistent and should be closed as soon as possible. The agent isn't allowed to make a second request on the same connection.

Pipelined Connections

Parallel connections and persistent connections are both widely used and supported by clients and servers. The HTTP specification also allows for **pipelined connections**, which are not as widely supported by either servers or clients. In a pipelined connection, a user agent 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, but it's not as widely supported as parallel and persistent connections.

Where Are We?

In this chapter we've looked at 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.

Chapter 4 Web Architecture

In the first chapter we talked about resources, but mostly focused on URLs and how to interpret a URL. However, 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 architecting 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 capability of the resource abstraction. Many webpages 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

All of these resources are the types of resources we build applications around, and 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 around. Given a URL you can locate a resource, of course, but you can also hand the URL to someone else by embedding the URL in a hyperlink or sending it in an email.

But, there are many things that you can't do with a URL. Rather, there are many things a URL cannot do. For example, a URL cannot restrict the client or server to a specific type of technology. Everyone speaks HTTP. It doesn't matter if your client is C++ and your web application is in Ruby.

Also, a URL cannot force the server to store a resource using any particular 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 derive the resource from the current time of day.

A URL can't even specify the representation of a specific resource, and a resource can have multiple representations. As we learned earlier, a client can request a particular 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 cannot do—it cannot say what a user wants to *do* with a resource. A URL doesn't say if I want to retrieve a resource or edit a resource. 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 build on. 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". This dissertation is the paper that introduces the representational state transfer (REST) style of architecture and goes into greater 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 and 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 in part because you can think of a URL as a pointer, or a unit of indirection, between a client and server application. Again, the URL itself doesn’t dictate a specific resource representation, technology implementation, or the client’s 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.

Because all of the information required for a transaction is contained in the messages, and because the information is visible and easy to parse, HTTP applications can rely on a number of services that provide value as a message moves between the client application and the server application.

Adding Value

As an HTTP message moves from the memory space in a process on one machine to the memory space in a process on another machine, it can move through several pieces of software and hardware that inspect and possibly modify the message. One good example is the web server application itself. A web server like Apache or IIS will be one of the first recipients of an incoming HTTP request on a server machine, and as a web server it can route the message to the proper application.

The web server can use 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 applications on the server don't need to worry about logging because the server is configured to log all messages.

Likewise, when an application creates an HTTP response message, the server has a chance to interact with the message on the way out. Again, this could be a simple logging operation, but it could also be a direct modification of 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 100-KB resource and turn it into a 25-KB 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. Compression is an added value provided by the web server software itself.

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 actually 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 these messages, the proxy can inspect the messages and potentially take some additional actions.

One of the clients I work for uses a proxy server to capture all HTTP traffic leaving the office. They don't want employees and contractors spending all their time on Twitter and Facebook, so HTTP requests to those servers will never reach their destination and there is no tweeting or Farmville inside the office. This is an example of one popular role for proxy servers, which is to function as an access control device.

However, a proxy server can be much more sophisticated than just dropping messages to specific hosts—a simple firewall could perform that duty. A proxy server could also inspect messages to remove confidential data, like the **`Referer`** 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, a topic we'll look at in the next article.

The proxy I'm describing in the previous 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.

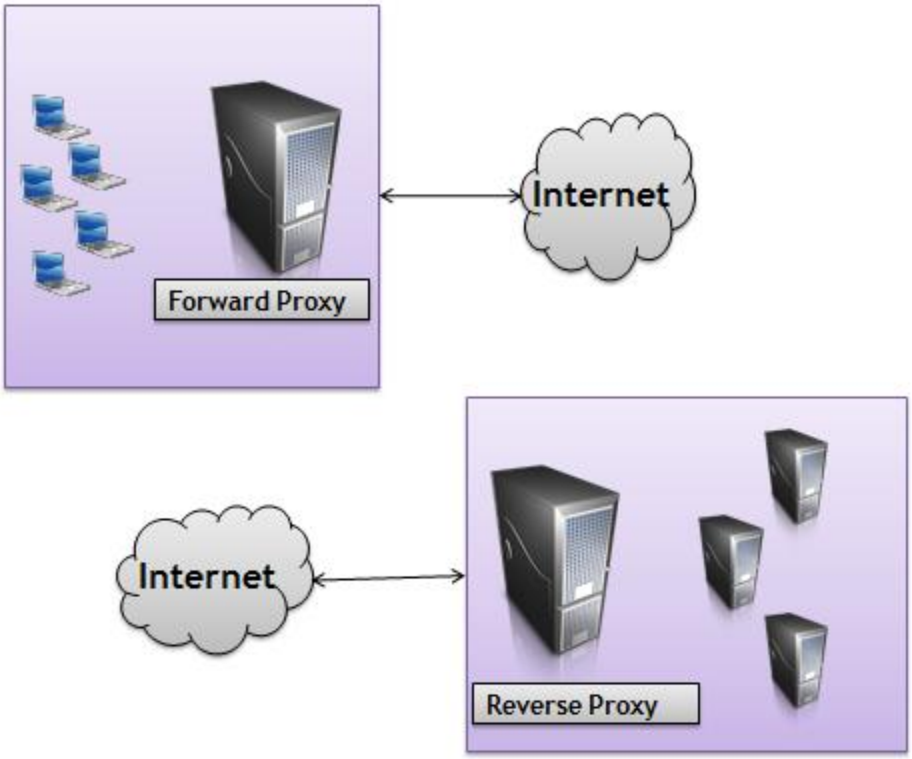


Figure 6: Forward and reverse proxies

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 is layered-in via a proxy. That's the beauty of HTTP.

Some other popular proxy services include the following.

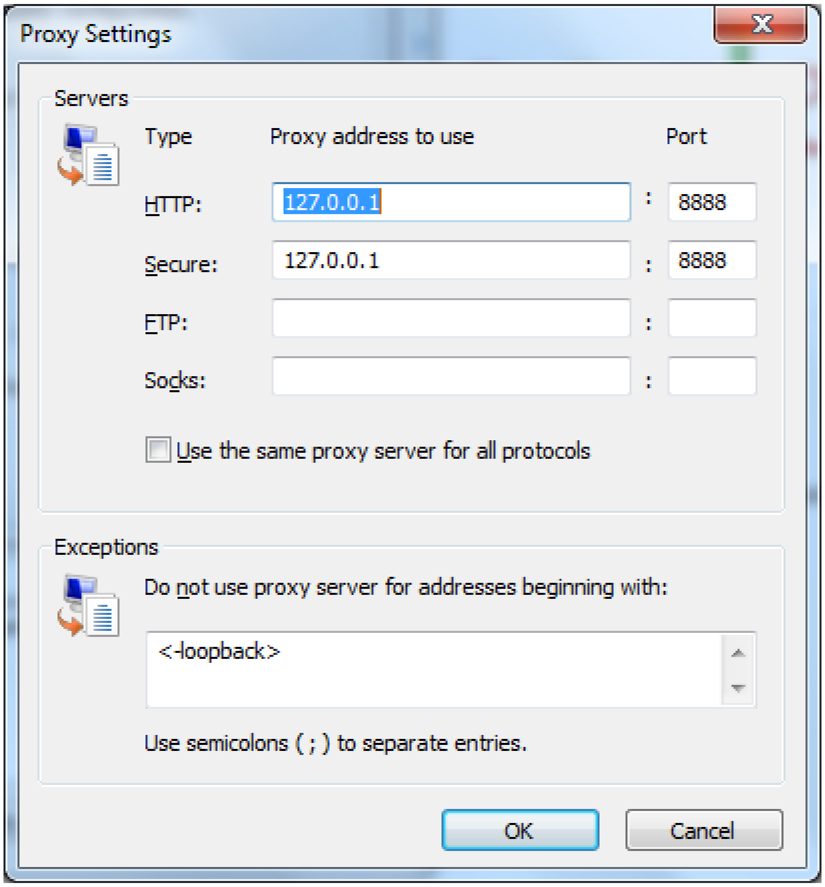
Load balancing proxies can take a message and forward it to one of several web servers on a round-robin basis, or by knowing which server is currently processing the fewest number of requests.

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 can store copies of frequently accessed resources and respond to messages requesting those resources 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. Fiddler, a tool mentioned in the previous chapter, is an HTTP debugger that allows you to capture and inspect HTTP messages. Fiddler works by telling Windows to forward all outgoing HTTP traffic to port 8888 on IP address 127.0.0.1. This IP address is the loopback address, meaning the traffic simply goes directly to the local machine where Fiddler is now listening on port 8888. Fiddler takes the HTTP request message, logs it, forwards it to the destination, and also captures the response before forwarding the response to the local application. You can view the proxy settings in Internet Explorer (IE) by going to **Tools**, **Internet Options**, clicking on the **Connections** tab, and then clicking the **LAN Settings** button. Under the **Proxy Server** area, click on the **Advanced** button to see the proxy server details.



Proxy Settings

Servers

Type	Proxy address to use		Port
HTTP:	127.0.0.1	:	8888
Secure:	127.0.0.1	:	8888
FTP:		:	
Socks:		:	

☐ Use the same proxy server for all protocols

Exceptions

Do not use proxy server for addresses beginning with:

<-loopback>

Use semicolons (;) to separate entries.

Figure 7: Proxy settings in Internet Explorer

Proxies are a perfect example of how HTTP can influence the architecture of a web application or website. There are many services you can layer into the network without impacting 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 time and time again for each request. Or, a proxy server or a client can cache the representation locally and reduce the amount of time and bandwidth required for a full retrieval.

Caching can 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.

The first thing to know is that there are two types of caches.

A **public cache** is a cache shared among multiple users. A public cache generally resides on a proxy server. A public cache on a forward proxy is generally 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 generally caching the resources that are popular on a specific website, like popular product images from Amazon.com.

A **private cache** is dedicated to a single user. Web browsers always keep a private cache of resources on your disk (these are the “Temporary Internet Files” in IE, or type **`about:cache`** in the address bar of Google Chrome to see files in Chrome’s private cache). Anything 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 out of the cache), are unfortunately complicated and mired by some legacy behaviors and incompatible implementations. Nevertheless, I will endeavor to point out some of 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 being deprecated in HTTP 1.1. **`Pragma`** is another example of a header used to control caching behavior, but it too is really 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 browser can cache the response. A value of **`no-cache`** means nobody should cache the response. There is also a **`no-store`** value, meaning the message might contain sensitive information and should not be persisted, but should be removed from memory as soon as possible.

How do you use this information? For popular shared resources (like the home page logo image), you might want to use a **`public`** cache control directive and allow everyone to cache the image, even proxy servers.

For responses to a specific user (like the HTML for the home page that includes the user’s name), you’d want to use a private cache directive.

Note: In ASP.NET you can control these settings via **`Response.Cache`**.

A server can also specify a `max-age` value in the `Cache-Control`. 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
```

Notice the `Cache-Control` allows public and private caches to cache the file, and they can keep it around for more than 315 million seconds (10 years). They also use an `Expires` header to give a specific date of expiration. If a client is HTTP 1.1 compliant and understands `Cache-Control`, it should use the value in `max-age` instead of `Expires`. Note that this doesn't mean Flickr plans on using the same CSS file for 10 years. When Flickr changes its design, it’ll probably just use a different URL for its updated CSS file.

The response also includes a `Last-Modified` header to indicate when the representation was last changed (which might just be the time of the request). 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 agent 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 it doesn’t have any inherent meaning. An `ETag` is often created using a hashing algorithm against the resource. If the resource ever changes, the server will compute a new **ETag**. A cache entry can be validated by comparing two **ETag**s. If the **ETag**s are the same, nothing has changed. If the **ETag**s 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 it all possible. In the next chapter we’ll talk about a few of the subjects we’ve skirted around, 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 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. There is nothing in the HTTP protocol that requires a server to retain information about an HTTP 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 previous chapter, services like caching, are all made possible (or at least 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 couldn't 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 log in before allowing the user to view his or her account-related resources. As each stateless request arrives for a private resource, the application wants to ensure the user was already authenticated. Another example is when the user wants to open an account and fills out forms in a three-page wizard. The application will want to make sure the first page of the wizard is complete before allowing the user to submit the second page.

Fortunately, there are many options for storing state in a web application. One approach is to embed state in the resources being 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 best for short-lived state (like the state required for moving through a three-page wizard). Embedding state in the resource keeps all the state inside of HTTP messages, so it 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). This option is required for state that has to be around a long time. Let's say the user submits a form to change his or her email address. The email 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, a file, or call a web service to let someone else take care of saving the address.

For server-side storage, many web development frameworks like ASP.NET also 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. Data stored in a session is scoped to an individual user (actually, to the user's browsing session), and is not shared among multiple users.

Session storage has an easy programming model and is only good for short-lived state, because eventually the server has to assume the user has left the site or closed the browser and the server will discard the session. Session storage, if stored in memory, can negatively impact scalability because subsequent requests must go to the exact same server where the session data resides. 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 more reliable techniques.

Identification and Cookies

Websites that want to track users will often turn to **cookies**. Cookies are defined by RFC6265 (http://tools.ietf.org/html/rfc6265), and this RFC is aptly titled "HTTP State Management Mechanism". When a user first visits a website, 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 website has placed some sort of unique identifier into the cookie, then the site can now track a user as he or she makes 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 than my cookie, but cookies do not authenticate users. An authenticated user has proved his or her 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 requests are made to the site.

Secondly, since 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 each URL a site gives to a user contains the proper identifier, and the URLs become much larger (which is why this technique is often called 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 in this sample. The three areas are delimited by semicolons (;). First, there are one or more name–value pairs. These name–value pairs are delimited by a dollar sign ($), and look very similar to how query parameters are formatted into a URL. In the example cookie, 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 later to talk about domain and path.

A website can put any information it likes into a cookie, although there is a size limitation of 4 KB. However, many websites only put in a unique identifier for a user, perhaps a GUID. A server can never trust anything stored on the client unless it is cryptographically secured. Yes, it is possible to store encrypted data in a cookie, but it's usually easier to store an ID.

```
HTTP/1.1 200 OK
Set-Cookie: GUID=00a48b7f6a4946a8adf593373e53347c;
            domain=.msn.com; path=/
```

Assuming the browser is configured to accept 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 quickly look up any associated user data from an in-memory data structure, database, or distributed cache. You can configure most web application 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 have to 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, it's worth pointing out that the **firstName** and **lastName** data stored in the session object is **not going into 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 goes into an in-memory data structure and stays alive for 20 minutes. 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 can open up 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). An ASP.NET session identifier looks like the following, which makes it harder to guess what someone else's session ID would look like.

```
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 website. If the other website sends the malicious script to its users, the malicious script can modify, or inspect and steal cookie information (which can lead to session hijacking, or worse).

To combat this vulnerability, Microsoft introduced the **HttpOnly** flag (seen in the last **Set-Cookie** example). The **HttpOnly** flag tells the user agent to not allow script code to access the cookie. The cookie exists for "HTTP only"—i.e. 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 are destroyed when the user closes the browser. **Persistent cookies** can outlive a single browsing session and a user agent will store the cookies to disk. You can shut down a computer and come back one week later, go to your favorite website, and a persistent cookie will still be there for the first request.

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

```
Set-Cookie: name=value; expires=Monday, 09-July-2012 21:12:00 GMT
```

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

Cookie Paths and Domains

So far we've said that once a cookie is set by a website, the cookie will travel to the website with every subsequent request (assuming the cookie hasn't expired). However, not all cookies travel to every website. The only cookies a user agent should send to a site are the cookies given to the user agent 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 could only open up 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 also 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 the **`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 user agent will only deliver the cookie to www.server.com. The domain in the previous 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 cannot 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 user agent should reject the cookie.

The **`path`** attribute can restrict a cookie to a specific resource path. In the previous 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 websites 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 have been vulnerable to XSS attacks as we've mentioned earlier, and also receive bad publicity when sites (particularly advertising sites) use **third-party cookies** to track users across the Internet. Third-party 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 websites, when sending a page resource back to the client, will 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 websites 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 they are enabled by default).

Two of the biggest downsides to cookies, however, are how they interfere with caching and how they transmit data with every request. Any response with a **`Set-Cookie`** header should not be cached, at least not the headers, since this can interfere with user identification and create security problems. Also, 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). Since we know there are lots of devices that 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, he or she 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 the user. We'll discuss authentication features next.

Authentication

The process of authentication forces a user to prove her or his identity by entering a username and password, or an email and a PIN, or some 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 section we'll briefly look at the top 5: basic, digest, Windows, forms, and OpenID. Of these five, only two are “official” in the HTTP specification—the basic and digest authentication protocols. We'll look at these two first.

Basic Authentication

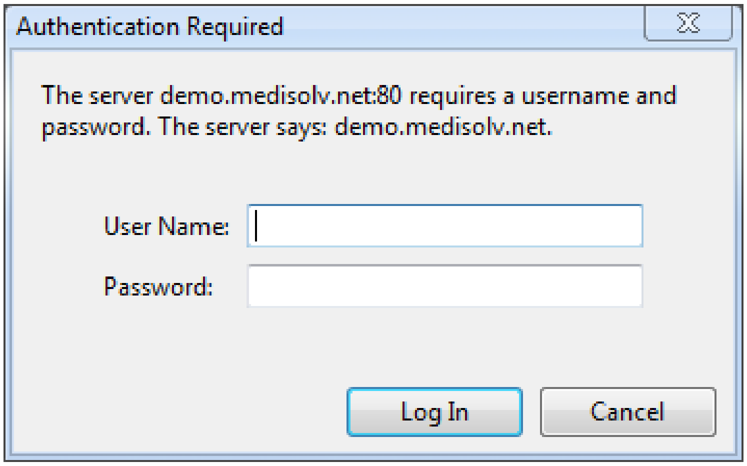
With basic authentication, the client will first request a resource with a normal HTTP message.

```
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 only specific users or groups can access a file or directory. For the previous request, let's imagine the server is configured to only allow 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 user agent a string it can use as a description for the protected area. What happens next depends on the user agent, but most browsers can display a UI for the user to enter credentials.



Authentication Required

The server demo.medisolv.net:80 requires a username and password. The server says: demo.medisolv.net.

Figure 8: Authentication dialog

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 username 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 look at later.

It's up to the server to decode the authorization header and verify the username 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 (which is essentially transmitting the password in plain text). 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 similar to 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="dcd98b7102dd2f0e8b11d0f600bfb0c093",
                         opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

Digest authentication is better than basic authentication when secure HTTP is not available, but it is still far from perfect. Digest authentication is still vulnerable to man-in-the-middle attacks in which someone is sniffing network traffic.

Windows Authentication

Windows Integrated Authentication is not a standard authentication protocol but it is popular among Microsoft products and servers. Although Windows Authentication is supported by many modern browsers (not just Internet Explorer), it doesn't work well over the Internet or where proxy servers reside. You'll find it is common on internal and intranet websites 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/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 HTML).

```
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 challenged by a server, and will do so using the user's credentials that he or she 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 in 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 he or she was 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 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
```

For ASP.NET, the authentication ticket (the **.ASPXAUTH** cookie value) is encrypted and hashed to prevent tampering. However, without secure HTTP the cookie is vulnerable to being intercepted, so session hijacking is still a potential problem. Yet, forms authentication remains popular because it allows applications complete control over the login experience and credential validation.

OpenID

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 is an open standard 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 has to verify his or her username 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 its “Federated Login for Google Account Users” webpage (https://developers.google.com/accounts/docs/OpenID).

While OpenID offers many potential benefits compared to forms authentication, it has faced a lack of adoption due to complexity in implementing, debugging, and maintaining the OpenID login dance. We have to hope the toolkits and frameworks continue to evolve to make the OpenID approach to authentication easier.

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. 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 is also known as HTTPS (because it 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 it has to use an explicit port that is 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).

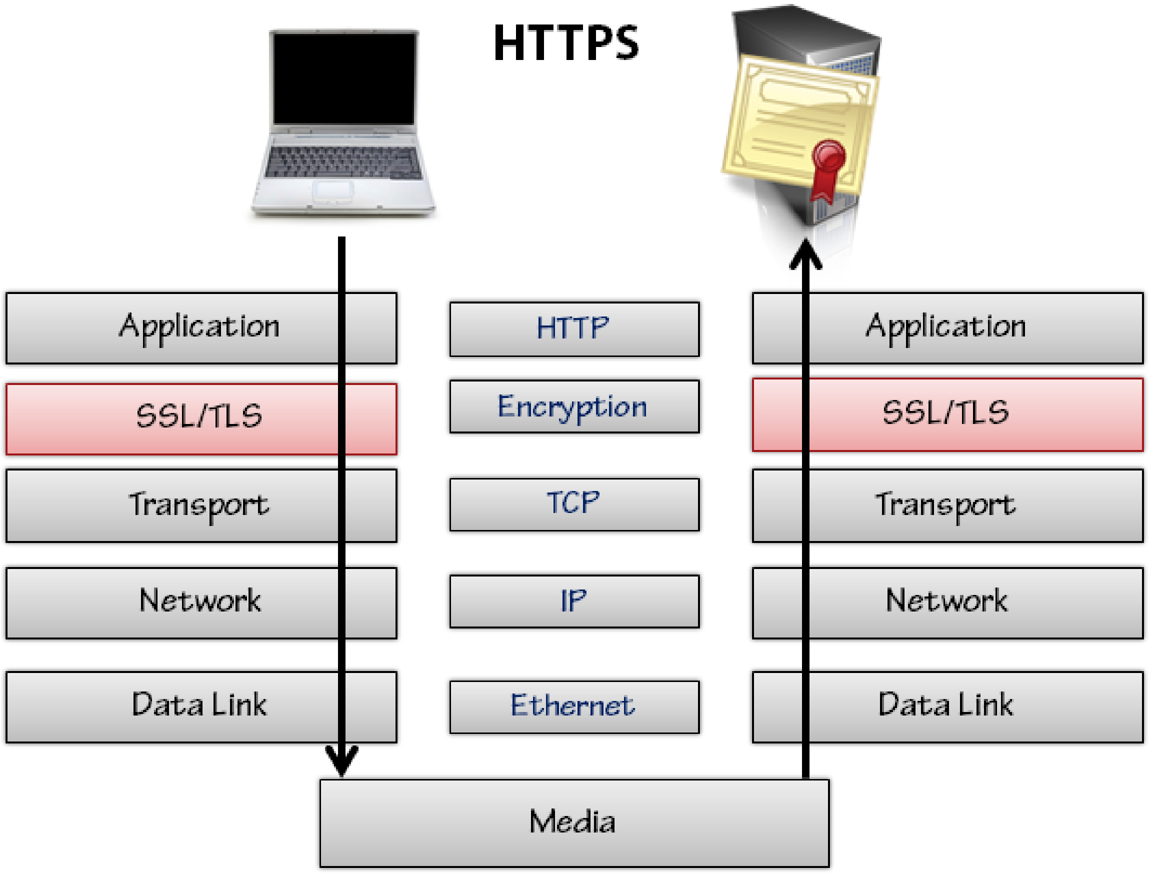


Figure 9: Secure HTTP protocol layers

HTTPS requires a server to have a cryptographic certificate. The certificate is sent to the client during setup of the HTTPS communication. The certificate includes the server's host name, and a user agent can use the certificate to validate that it is truly talking to the server it thinks it is talking to. The validation is all made possible using public key cryptography and the existence of certificate authorities, like Verisign, that will sign and vouch for the integrity of a certificate. Administrators have to purchase and install certificates from the certificate authorities.

There are many cryptographic details we could cover, but from a developer’s perspective, the most important things to know about HTTPS are:

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

- **The server is authenticated to the client thanks to the server certificate.** If you are talking to mybigbank.com over HTTPS, you can be sure your messages are really going to mybigbank.com and not someone 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 previously if they need to know the user’s identity. HTTPS does make forms-based authentication and basic authentication more secure since all data is 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, client-side certificates are generally not used on the open Internet since not many users will purchase and install 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 related to performance. 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, but 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 looked at 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.