CODETOOLS & TIPS

HTTP: The Protocol Every Web Developer Must Know - Part 2

by Pavan Podila 29 Apr 2013

Difficulty: Intermediate Length: Long Languages: English ▼

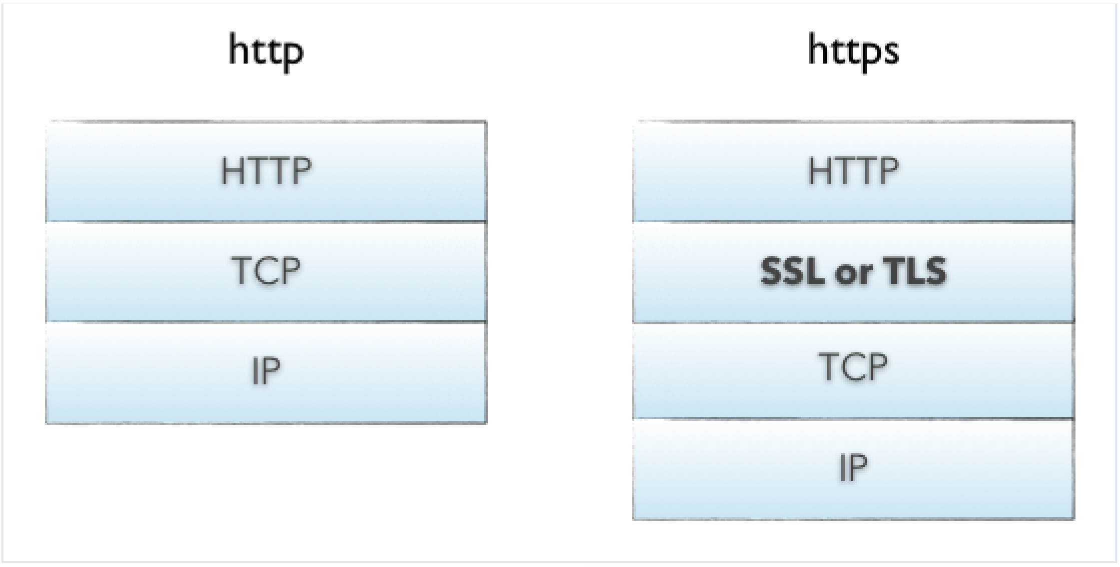
Tools & Tips Web Development HTTP

In my previous article, we covered some of HTTP's basics, such as the URL scheme, status codes and request/response headers. With that as our foundation, we will look at the finer aspects of HTTP, like connection handling, authentication and HTTP caching. These topics are fairly extensive, but we'll cover the most important bits.

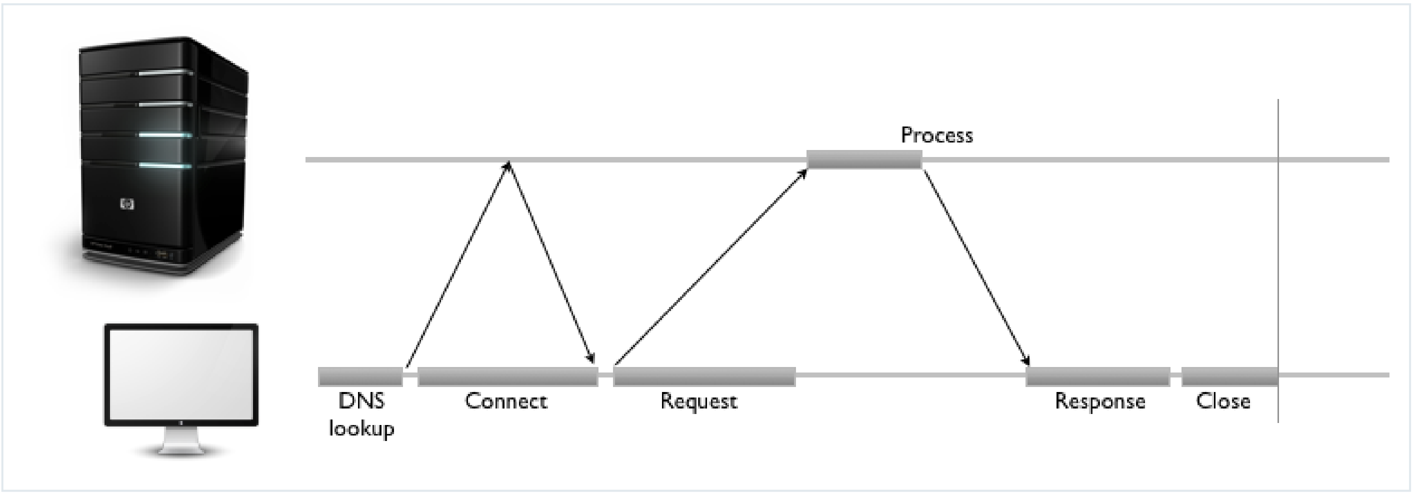
HTTP Connections

A connection must be established between the client and server before they can communicate with each other, and HTTP uses the reliable TCP transport protocol to make this connection. By default, web traffic uses TCP port 80. A TCP stream is broken into IP packets, and it ensures that those packets always arrive in the correct order without fail. HTTP is an application layer protocol over TCP, which is over IP.

HTTPS is a secure version of HTTP, inserting an additional layer between HTTP and TCP called TLS or SSL (Transport Layer Security or Secure Sockets Layer, respectively). HTTPS communicates over port 443 by default, and we will look at HTTPS later in this article.



An HTTP connection is identified by `<source-IP, source-port>` and `<destination-IP, destination-port>`. On a client, an HTTP application is identified by a `<IP, port>` tuple. Establishing a connection between two endpoints is a multi-step process and involves the following:



- resolve IP address from host name via DNS
- establish a connection with the server
- send a request
- wait for a response
- close connection

> *The server is responsible for always responding with the correct headers and responses.*

In HTTP/1.0, all connections were closed after a single transaction. So, if a client wanted to request three separate images from the same server, it made three separate connections to the remote host. As you can see from the above diagram, this can introduce lot of network delays, resulting in a sub-optimal user experience.

To reduce connection-establishment delays, HTTP/1.1 introduced **persistent connections**, long-lived connections that stay open until the client closes them. Persistent connections are default in HTTP/1.1, and making a single transaction connection requires the client to set the `Connection: close` request header. This tells the server to close the connection after sending the response.

In addition to persistent connections, browsers/clients also employ a technique, called **parallel connections**, to minimize network delays. The age-old concept of parallel connections involves creating a pool of connections (generally capped at six connections). If there are six assets that the client needs to download from a website, the client makes six parallel connections to download those assets, resulting in a faster turnaround. This is a huge improvement over serial connections where the client only downloads an asset after completing the download for a previous asset.

Parallel connections, in combination with persistent connections, is today's answer to minimizing network delays and creating a smooth experience on the client. For an in-depth treatment of HTTP connections, refer to the Connections section of the HTTP spec.

Server-side Connection Handling

The server mostly listens for incoming connections and processes them when it receives a request. The operations involve:

- establishing a socket to start listening on port 80 (or some other port)
- receiving the request and parsing the message
- processing the response
- setting response headers
- sending the response to the client
- close the connection if a `Connection: close` request header was found

Of course, this is not an exhaustive list of operations. Most applications/websites need to know who makes a request in order to create more customized responses. This is the realm of **identification** and **authentication**.

Advertisement

Identification and Authentication

> *HTTP is an application layer protocol over TCP, which is over IP.*

It is almost mandatory to know who connects to a server for tracking an app's or site's usage and the general interaction patterns of users. The premise of identification is to tailor the response in order to provide a personalized experience; naturally, the server must know who a user is in order to provide that functionality.

There are a few different ways a server can collect this information, and most websites use a hybrid of these approaches:

- **Request headers**: `From`, `Referer`, `User-Agent` - We saw these headers in Part 1.
- **Client-IP** - the IP address of the client
- **Fat Urls** - storing state of the current user by modifying the URL and redirecting to a different URL on each click; each click essentially accumulates state.
- **Cookies** - the most popular and non-intrusive approach.

Cookies allow the server to attach arbitrary information for outgoing responses via the `Set-Cookie` response header. A cookie is set with one or more *name=value* pairs separated by semicolon (;), as in `Set-Cookie: session-id=12345ABC; username=nettuts`.

A server can also restrict the cookies to a specific `domain` and `path`, and it can make them persistent with an `expires` value. Cookies are automatically sent by the browser for each request made to a server, and the browser ensures that only the `domain`- and `path`-specific cookies are sent in the request. The request header `Cookie: name=value [; name2=value2]` is used to send these cookies to the server.

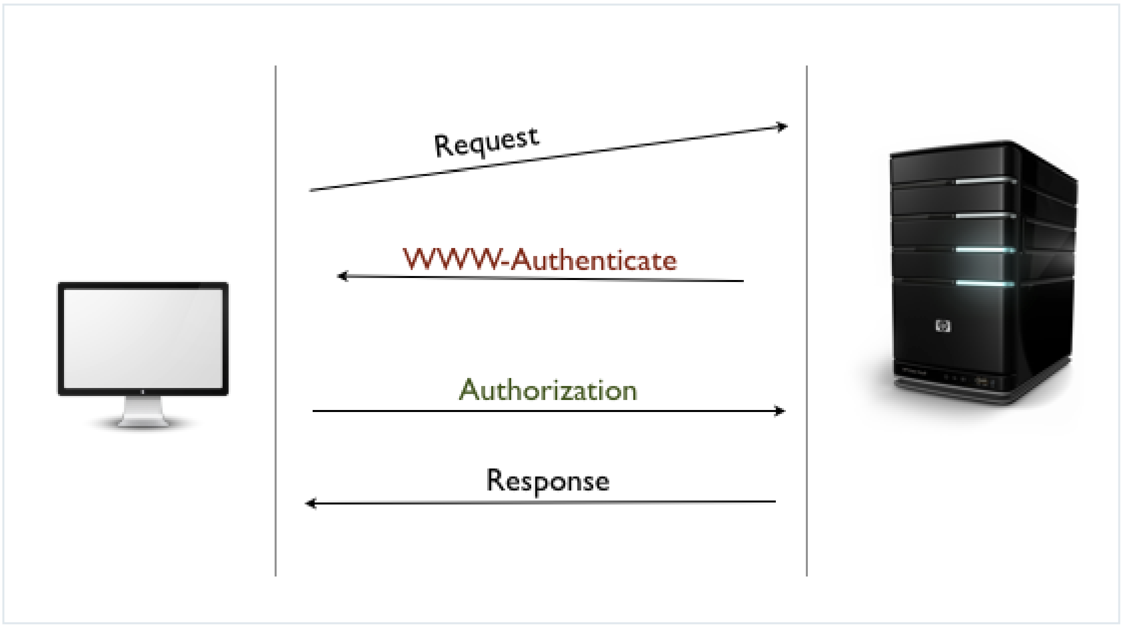
> *The best way to identify a user is to require them to sign up and log in, but implementing this feature requires some effort by the developer, as well as the user.*

Techniques like OAuth simplify this type of feature, but it still requires user consent in order to work properly. Authentication plays a large role here, and it is probably the only way to identify and verify the user.

Authentication

HTTP does support a rudimentary form of authentication called **Basic Authentication**, as well as the more secure **Digest Authentication**.

In Basic Authentication, the server initially denies the client's request with a `WWW-Authenticate` response header and a `401 Unauthorized` status code. On seeing this header, the browser displays a login dialog, prompting for a username and password. This information is sent in a base-64 encoded format in the `Authentication` request header. The server can now validate the request and allow access if the credentials are valid. Some servers might also send an `Authentication-Info` header containing additional authentication details.



A corollary to Basic-Authentication is **Proxy Authentication**. Instead of a web server, the authetication challenge is requested by an intermediate proxy. The proxy sends a `Proxy-Authenticate` header with a `407 Unauthorized` status code. In return, the client is supposed to send the credentials via the `Proxy-Authorization` request header.

Digest Authentication is similar to Basic and uses the same handshake technique with the `WWW-Authenticate` and `Authorization` headers, but Digest uses a more secure hashing function to encrypt the username and password (commonly with MD5 or KD digest functions). Although Digest Authentication is supposed to be more secure than Basic, websites typically use Basic Authentication because of its simplicty. To mitigate the security concerns, Basic Auth is used in conjunction with SSL.

Secure HTTP

https://mail.google.com/mail/u/0/?shva=1#inbox

The HTTPS protocol provides a secure connection on the web. The easiest way to know if you are using HTTPS is to check the browser's address bar. HTTPs' secure component involves inserting a layer of encryption/decryption between HTTP and TCP. This is the Secure Sockets Layer (SSL) or the improved Transport Layer Security (TLS).

SSL uses a powerful form of encryption using RSA and public-key cryptography. Because secure transactions are so important on the web, a ubiquitous standards-based Public-Key Infrastructure (PKI) effort has been underway for quite sometime.

Existing clients/servers do not have to change the way they handle messages because most of the hard work happens in the SSL layer. Thus, you can develop your web application using Basic Authentication and automatially reap the benefits of SSL by switching to the `https://` protocol. However, to make the web application work over HTTPS, you need to have a working digital certificate deployed on the server.

Certificates

Just as you need ID cards to show your identity, a web server needs a digital certificate to identify itself. Certificates (or "certs") are issued by a Certificate Authority (CA) and vouch for your identity on the web. The CAs are the guardians of the PKI. The most common form of certificates is the X.509 v3 standard, which contains information, such as:

- the certificate issuer
- the algorithm used for the certificate
- the subject name or organization for whom this cert is created
- the public key information for the subject
- the Certification Authority Signature, using the specified signing algorithm

When a client makes a request over HTTPS, it first tries to locate a certificate on the server. If the cert is found, it attempts to verfiy it against its known list of CAs. If its not one of the listed CAs, it might show a dialog to the user warning about the website's certficate.

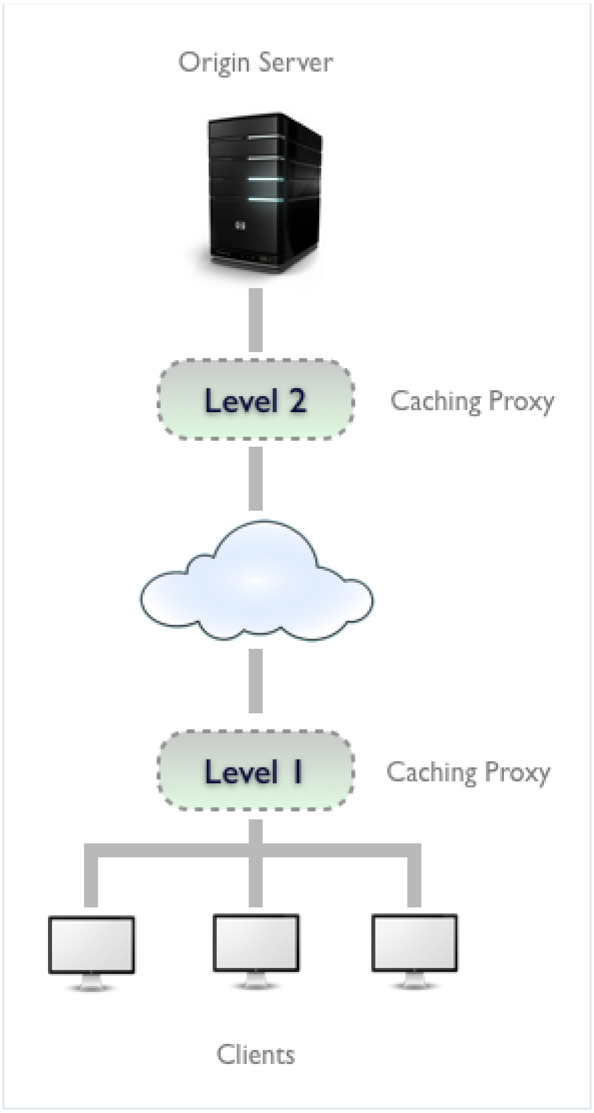
Once the certificate is verified, the SSL handshake is complete and secure transmission is in effect.

HTTP Caching

It is generally agreed that doing the same work twice is wasteful. This is the guiding principle around the concept of HTTP caching, a fundamental pillar of the HTTP Network Infrastructure. Because most of the operations are over a network, a cache helps save time, cost and bandwidth, as well as provide an improved experience on the web.

Caches are used at several places in the network infrastructure, from the browser to the origin server. Depending on where it is located, a cache can be categorized as:

- **Private**: within a browser, caches usernames, passwords, URLs, browsing history and web content. They are generally small and specific to a user.
- **Public**: deployed as caching proxies between the server and client. These are much larger because they serve multiple users. A common practice is to keep multiple caching proxies between the client and the origin-server. This helps to serve frequently accessed content, while still allowing a trip to the server for infrequently needed content.



Cache Processing

Regardless of where a cache is located, the process of maintaining a cache is quite similar:

- **Receive** request message.
- **Parse** the URL and headers.
- **Lookup** a local copy; otherwise, fetch and store locally
- Do a **freshness check** to determine the age of the content in the cache; make a request to refresh the content only if necessary.
- Create the **response** from the cached body and updated headers.
- **Send** the response back to client.
- Optionally, **log** the transaction.

Of course, the server is responsible for always responding with the correct headers and responses. If a document hasn't changed, the server should respond with a `304 Not Modified`. If the cached copy has expired, it should generate a new response with updated response headers and return with a `200 OK`. If the resource is deleted, it should come back with `404 Not Found`. These responses help tune the cache and ensure that stale content is not kept for too long.

Cache Control Headers

> *Parallel connections, in combination with persistent connections, is today's answer to minimizing network delays.*

Now that we have a sense of how a cache works, it's time to look at the request and response headers that enable the caching infrastructure. Keeping the content fresh and up-to-date is one of the primary responsibilities of the cache. To keep the cached copy consistent with the server, HTTP provides some simple mechanisms, namely *Document Expiration* and *Server Revalidation*.

Document Expiration

HTTP allows an origin-server to attach an *expiration date* to each document using the `Cache-Control` and `Expires` *response* headers. This helps the client and other cache servers know how long a document is valid and fresh. The cache can serve the copy as long as the *age* of the document is within the expiration date. Once a document expires, the cache must check with the server for a newer copy and update its local copy accordingly.

`Expires` is an older HTTP/1.0 response header that specifies the value as an absolute date. This is only useful if the server clocks are in sync with the client, which is a terrible assumption to make. This header is less useful compared to the newer `Cache-Control: max-age=<s>` header introduced in HTTP/1.1. Here, `max-age` is a relative age, specified in seconds, from the time the response was created. Thus if a document should expire after one day, the expiration header should be `Cache-Control: max-age=86400`.

Server Revalidation

Once a cached document expires, the cache must revalidate with the server to check if the document has changed. This is called *server revalidation* and serves as a querying mechanism for the *stale-ness* of a document. Just because a cached copy has expired doesn't mean that the server actually has newer content. Revalidation is just a means of ensuring that the cache stays fresh. Because of the expiration time (as specified in a previous server response), the cache doesn't have to check with the server for every single request, thus saving bandwidth, time and reducing the network traffic.

> *The combination of document expiration and server revalidation is a very effective mechanism, it and allows distributed systems to maintain copies with an expiration date.*

If the content is known to frequently change, the expiration time can be reduced—allowing the systems to re-sync more frequently.

The revalidation step can be accomplished with two kinds of request-headers: `If-Modified-Since` and `If-None-Match`. The former is for date-based validation while the latter uses Entity-Tags (ETags), a hash of the content. These headers use date or ETag values obtained from a previous server response. In case of `If-Modified-Since`, the `Last-Modified` response header is used; for `If-None-Match`, it is the `ETag` response header.

Controlling the Cachability

The validity period for a document should be defined by the server generating the document. If it's a newspaper website, the homepage should expire after a day (or sometimes even every hour!). HTTP provides the `Cache-Control` and `Expires` response headers to set the expiration on documents. As mentioned earlier, `Expires` is based on absolute dates and not a reliable solution for controlling cache.

The `Cache-Control` header is far more useful and has a few different values to constrain how clients should be caching the response:

- **Cache-Control: no-cache**: the client is allowed to store the document; however, it must revalidate with the server on every request. There is a HTTP/1.0 compatibility header called **Pragma: no-cache**, which works the same way.
- **Cache-Control: no-store**: this is a stronger directive to the client to not store the document at all.
- **Cache-Control: must-revalidate**: this tells the client to bypass its freshness calculation and always revalidate with the server. It is not allowed to serve the cached response in case the server is unavailable.

- **Cache-Control: max-age**: this sets a relative expiration time (in seconds) from the time the response is generated.

As an aside, if the server does not send any `Cache-Control` headers, the client is free to use its own heuristic expiration algorithm to determine freshness.

Constraining Freshness from the Client

Cachability is not just limited to the server. It can also be specified from the client. This allows the client to impose constraints on what it is willing to accept. This is possible via the same `Cache-Control` header, albeit with a few different values:

- **Cache-Control: min-fresh=<s>**: the document must be fresh for at least *<s>*seconds.
- **Cache-Control: max-stale** or **Cache-Control: max-stale=<s>**: the document cannot be served from the cache if it has been stale for longer than *<s>* seconds.
- **Cache-Control: max-age=<s>**: the cache cannot return a document that has been cached longer than *<s>* seconds.
- **Cache-Control: no-cache** or **Pragma: no-cache**: the client will not accept a cached resource unless it has been revalidated.

HTTP Caching is actually a very interesting topic, and there are some very sophisticated algorithms to manage cached content. For a deeper look into this topic, refer to the Caching section of the HTTP spec.

Summary

Our tour of HTTP began with the foundation of URL schemes, status codes and request/response headers. Building upon those concepts, we looked at some of the finer areas of HTTP, such as connection handling, identification and authentication and caching. I am hopeful that this tour has given you a good taste for the breadth of HTTP and enough pointers to further explore this protocol.

References

- RFC 2616, HTTP specification
- HTTP Definitive Guide