CODETOOLS & TIPS

HTTP: The Protocol Every Web Developer Must Know - Part 1

by Pavan Podila 8 Apr 2013

Difficulty: Intermediate Length: Long Languages: English ▼

Tools & Tips Web Development HTTP

HTTP stands for Hypertext Transfer Protocol. It's a stateless, application-layer protocol for communicating between distributed systems, and is the foundation of the modern web. As a web developer, we all must have a strong understanding of this protocol.

Let's review this powerful protocol through the lens of a web developer. We'll tackle the topic in two parts. In this first entry, we'll cover the basics and outline the various request and response headers. In the follow-up article, we'll review specific pieces of HTTP - namely caching, connection handling and authentication.

Although I'll mention some details related to headers, it's best to instead consult the RFC (RFC 2616) for in-depth coverage. I will be pointing to specific parts of the RFC throughout the article.

Looking for a Quick Solution?

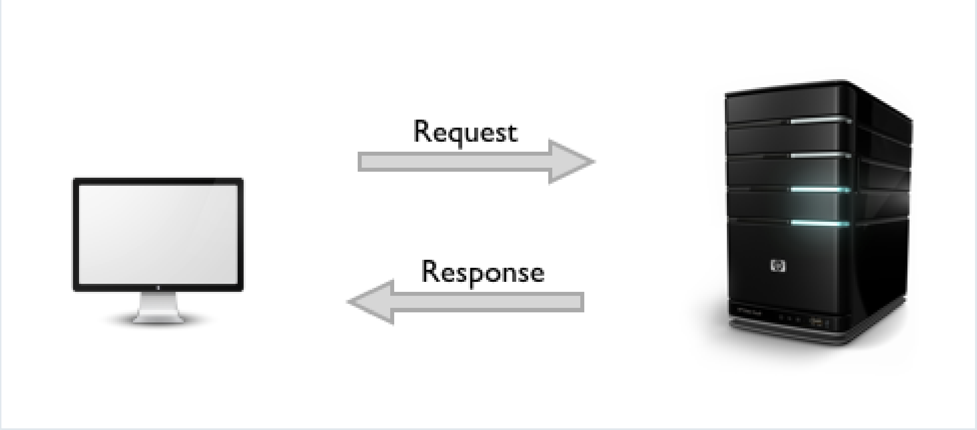
If you're having trouble with an HTTP error in WordPress that you need fixed, you can order an Express HTTP error fix on Envato Studio, and have the error fixed in one day for just $50.

HTTP Basics

HTTP allows for communication between a variety of hosts and clients, and supports a mixture of network configurations.

> *To make this possible, it assumes very little about a particular system, and does not keep state between different message exchanges.*

This makes HTTP a **stateless** protocol. The communication usually takes place over TCP/IP, but any reliable transport can be used. The default port for TCP/IP is **80**, but other ports can also be used.



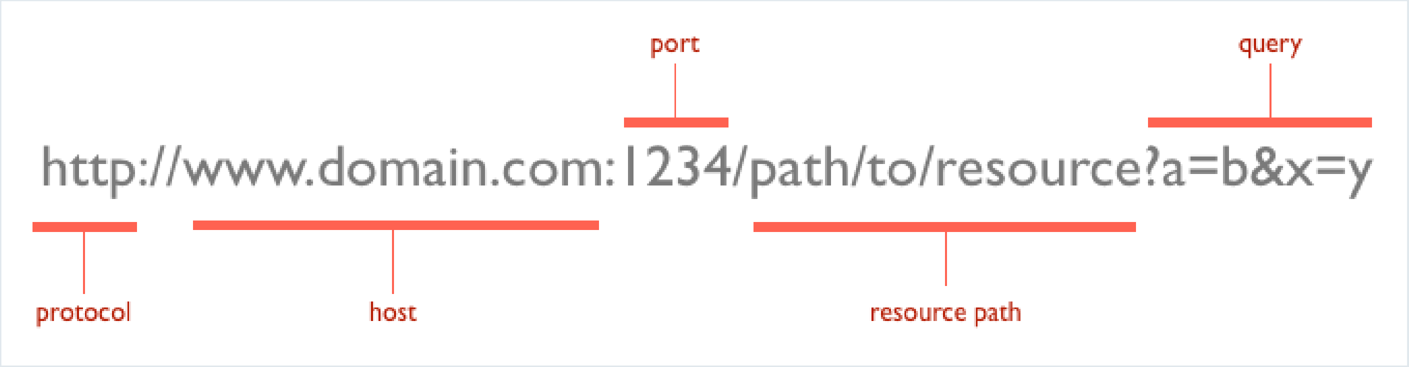
> *Custom headers can also be created and sent by the client.*

Communication between a host and a client occurs, via a **request/response pair**. The client initiates an HTTP request message, which is serviced through a HTTP response message in return. We will look at this fundamental message-pair in the next section.

The current version of the protocol is **HTTP/1.1**, which adds a few extra features to the previous 1.0 version. The most important of these, in my opinion, includes *persistent connections*, *chunked transfer-coding* and *fine-grained caching headers*. We'll briefly touch upon these features in this article; in-depth coverage will be provided in part two.

URLs

At the heart of web communications is the request message, which are sent via Uniform Resource Locators (URLs). I'm sure you are already familiar with URLs, but for completeness sake, I'll include it here. URLs have a simple structure that consists of the following components:



The protocol is typically `http`, but it can also be `https` for secure communications. The default port is `80`, but one can be set explicitly, as illustrated in the above image. The resource path is the *local path* to the resource on the server.

Verbs

> *There are also web debugging proxies, like* ***Fiddler*** *on Windows and* ***Charles Proxy*** *for OSX.*

URLs reveal the identity of the particular host with which we want to communicate, but the action that should be performed on the host is specified via HTTP verbs. Of course, there are several actions that a client would like the host to perform. HTTP has formalized on a few that capture the essentials that are universally applicable for all kinds of applications.

These request verbs are:

- **GET**: *fetch* an existing resource. The URL contains all the necessary information the server needs to locate and return the resource.
- **POST**: *create* a new resource. POST requests usually carry a payload that specifies the data for the new resource.
- **PUT**: *update* an existing resource. The payload may contain the updated data for the resource.
- **DELETE**: *delete* an existing resource.

The above four verbs are the most popular, and most tools and frameworks explicitly expose these request verbs. `PUT` and `DELETE` are sometimes considered specialized versions of the `POST` verb, and they may be packaged as `POST` requests with the payload containing the exact action: *create*, *update* or *delete*.

There are some lesser used verbs that HTTP also supports:

- **HEAD**: this is similar to GET, but without the message body. It's used to retrieve the server headers for a particular resource, generally to check if the resource has changed, via timestamps.
- **TRACE**: used to retrieve the hops that a request takes to round trip from the server. Each intermediate proxy or gateway would inject its IP or DNS name into the `Via` header field. This can be used for diagnostic purposes.

- **OPTIONS**: used to retrieve the server capabilities. On the client-side, it can be used to modify the request based on what the server can support.

Status Codes

With URLs and verbs, the client can initiate requests to the server. In return, the server responds with status codes and message payloads. The status code is important and tells the client how to interpret the server response. The HTTP spec defines certain number ranges for specific types of responses:

1xx: Informational Messages

> *All HTTP/1.1 clients are required to accept the `Transfer-Encoding` header.*

This class of codes was introduced in HTTP/1.1 and is purely provisional. The server can send a `Expect: 100-continue` message, telling the client to continue sending the remainder of the request, or ignore if it has already sent it. HTTP/1.0 clients are supposed to ignore this header.

2xx: Successful

This tells the client that the request was successfully processed. The most common code is **200 OK**. For a `GET` request, the server sends the resource in the message body. There are other less frequently used codes:

- **202** Accepted: the request was accepted but may not include the resource in the response. This is useful for async processing on the server side. The server may choose to send information for monitoring.
- **204** No Content: there is no message body in the response.
- **205** Reset Content: indicates to the client to reset its document view.
- **206** Partial Content: indicates that the response only contains partial content. Additional headers indicate the exact range and content expiration information.

3xx: Redirection

> *404 indicates that the resource is invalid and does not exist on the server.*

This requires the client to take additional action. The most common use-case is to jump to a different URL in order to fetch the resource.

- **301** Moved Permanently: the resource is now located at a new URL.
- **303** See Other: the resource is temporarily located at a new URL. The `Location`response header contains the temporary URL.
- **304** Not Modified: the server has determined that the resource has not changed and the client should use its cached copy. This relies on the fact that the client is sending `ETag` (Enttity Tag) information that is a hash of the content. The server compares this with its own computed `ETag` to check for modifications.

4xx: Client Error

These codes are used when the server thinks that the client is at fault, either by requesting an invalid resource or making a bad request. The most popular code in this class is **404 Not Found**, which I think everyone will identify with. 404 indicates that the resource is invalid and does not exist on the server. The other codes in this class include:

- **400** Bad Request: the request was malformed.
- **401** Unauthorized: request requires authentication. The client can repeat the request with the `Authorization` header. If the client already included the `Authorization` header, then the credentials were wrong.
- **403** Forbidden: server has denied access to the resource.
- **405** Method Not Allowed: invalid HTTP verb used in the request line, or the server does not support that verb.
- **409** Conflict: the server could not complete the request because the client is trying to modify a resource that is newer than the client's timestamp. Conflicts arise mostly for PUT requests during collaborative edits on a resource.

5xx: Server Error

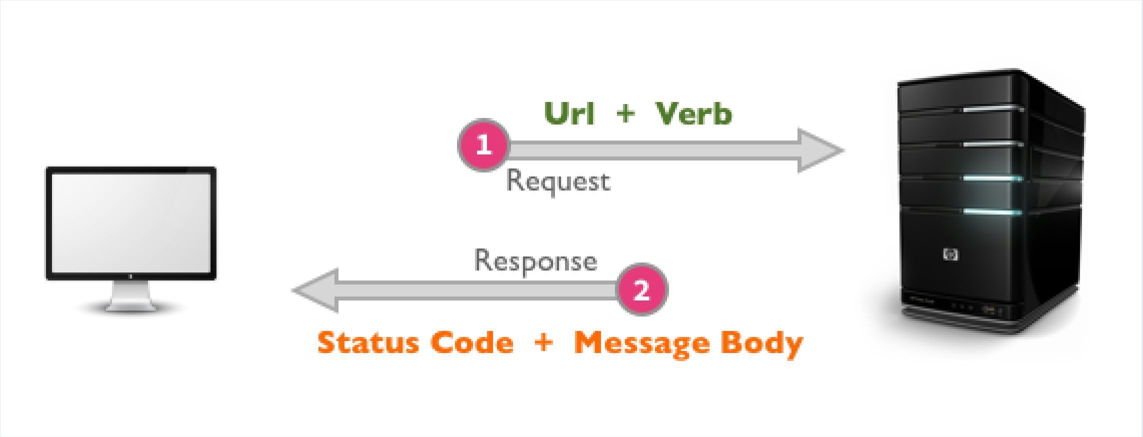
This class of codes are used to indicate a server failure while processing the request. The most commonly used error code is **500 Internal Server Error**. The others in this class are:

- **501 Not Implemented**: the server does not yet support the requested functionality.
- **503 Service Unavailable**: this could happen if an internal system on the server has failed or the server is overloaded. Typically, the server won't even respond and the

request will timeout.

Request and Response Message Formats

So far, we've seen that *URLs, verbs and status codes* make up the fundamental pieces of an HTTP request/response pair.



Let's now look at the content of these messages. The HTTP specification states that a request or response message has the following generic structure:

```
message = <start-line>
          *(<message-header>)
          CRLF
          [<message-body>]

<start-line> = Request-Line | Status-Line
<message-header> = Field-Name ':' Field-Value
```

It's mandatory to place a new line between the message headers and body. The message can contain one or more headers, of which are broadly classified into:

- *general headers*: that are applicable for both request and response messages.
- *request specific headers*.
- *response specific headers*.
- *entity headers*.

The message body may contain the complete entity data, or it may be piecemeal if the chunked encoding (`Transfer-Encoding: chunked`) is used. All HTTP/1.1 clients are required to accept the `Transfer-Encoding` header.

General Headers

There are a few headers (general headers) that are shared by both request and response messages:

```
general-header = Cache-Control
               | Connection
               | Date
               | Pragma
               | Trailer
               | Transfer-Encoding
               | Upgrade
               | Via
               | Warning
```

We have already seen some of these headers, specifically `Via` and `Transfer-Encoding`. We will cover `Cache-Control` and `Connection` in part two.

> *The status code is important and tells the client how to interpret the server response.*

- `Via` header is used in a TRACE message and updated by all intermittent proxies and gateways
- `Pragma` is considered a custom header and may be used to include implementation-specific headers. The most commonly used pragma-directive

is `Pragma: no-cache`, which really is `Cache-Control: no-cache` under HTTP/1.1. This will be covered in Part 2 of the article.

- The `Date` header field is used to timestamp the request/response message
- `Upgrade` is used to switch protocols and allow a smooth transition to a newer protocol.
- `Transfer-Encoding` is generally used to break the response into smaller parts with the `Transfer-Encoding: chunked` value. This is a new header in HTTP/1.1 and allows for streaming of response to the client instead of one big payload.

Entity headers

Request and Response messages may also include entity headers to provide meta-information about the the content (aka Message Body or Entity). These headers include:

```
entity-header  = Allow
               | Content-Encoding
               | Content-Language
               | Content-Length
               | Content-Location
               | Content-MD5
               | Content-Range
               | Content-Type
               | Expires
               | Last-Modified
```

All of the `Content-` prefixed headers provide information about the structure, encoding and size of the message body. Some of these headers need to be present if the entity is part of the message.

The `Expires` header indicates a timestamp of whent he entity expires. Interestingly, a *"never expires"* entity is sent with a timestamp of one year into the future. The `Last-Modified` header indicates the last modification timestamp for the entity.

> *Custom headers can also be created and sent by the client; they will be treated as entity headers by the HTTP protocol.*

This is really an extension mechanism, and some client-server implementations may choose to communicate specifically over these extension headers. Although HTTP supports custom headers, what it really looks for are the request and response headers, which we cover next.

Request Format

The request message has the same generic structure as above, except for the request line which looks like:

```
Request-Line = Method SP URI SP HTTP-Version CRLF
Method = "OPTIONS"
       | "HEAD"
       | "GET"
       | "POST"
       | "PUT"
       | "DELETE"
       | "TRACE"
```

`SP` is the space separator between the tokens. `HTTP-Version` is specified as *"HTTP/1.1"* and then followed by a new line. Thus, a typical request message might look like:

```
GET /articles/http-basics HTTP/1.1
Host: www.articles.com
Connection: keep-alive
Cache-Control: no-cache
Pragma: no-cache
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*
```

Note the request line followed by many request headers. The **Host** header is mandatory for HTTP/1.1 clients. **GET** requests do not have a message body, but **POST** requests can contain the post data in the body.

The request headers act as modifiers of the request message. The complete list of known request headers is not too long, and is provided below. Unknown headers are treated as entity-header fields.

```
request-header = Accept
               | Accept-Charset
               | Accept-Encoding
               | Accept-Language
               | Authorization
               | Expect
               | From
               | Host
               | If-Match
               | If-Modified-Since
               | If-None-Match
               | If-Range
               | If-Unmodified-Since
               | Max-Forwards
               | Proxy-Authorization
               | Range
               | Referer
               | TE
               | User-Agent
```

The `Accept` prefixed headers indicate the acceptable media-types, languages and character sets on the client. `From`, `Host`, `Referer` and `User-Agent` identify details about the client that initiated the request. The `If-` prefixed headers are used to make a request more conditional, and the server returns the resource only if the condition matches. Otherwise, it returns a `304 Not Modified`. The condition can be based on a timestamp or an ETag (a hash of the entity).

Response Format

The response format is similar to the request message, except for the status line and headers. The status line has the following structure:

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

- HTTP-Version is sent as `HTTP/1.1`
- The Status-Code is one of the many statuses discussed earlier.
- The Reason-Phrase is a human-readable version of the status code.

A typical status line for a successful response might look like so:

```
HTTP/1.1 200 OK
```

The response headers are also fairly limited, and the full set is given below:

```
 response-header = Accept-Ranges
                 | Age
                 | ETag
                 | Location
                 | Proxy-Authenticate
                 | Retry-After
                 | Server
                 | Vary
                 | WWW-Authenticate
```

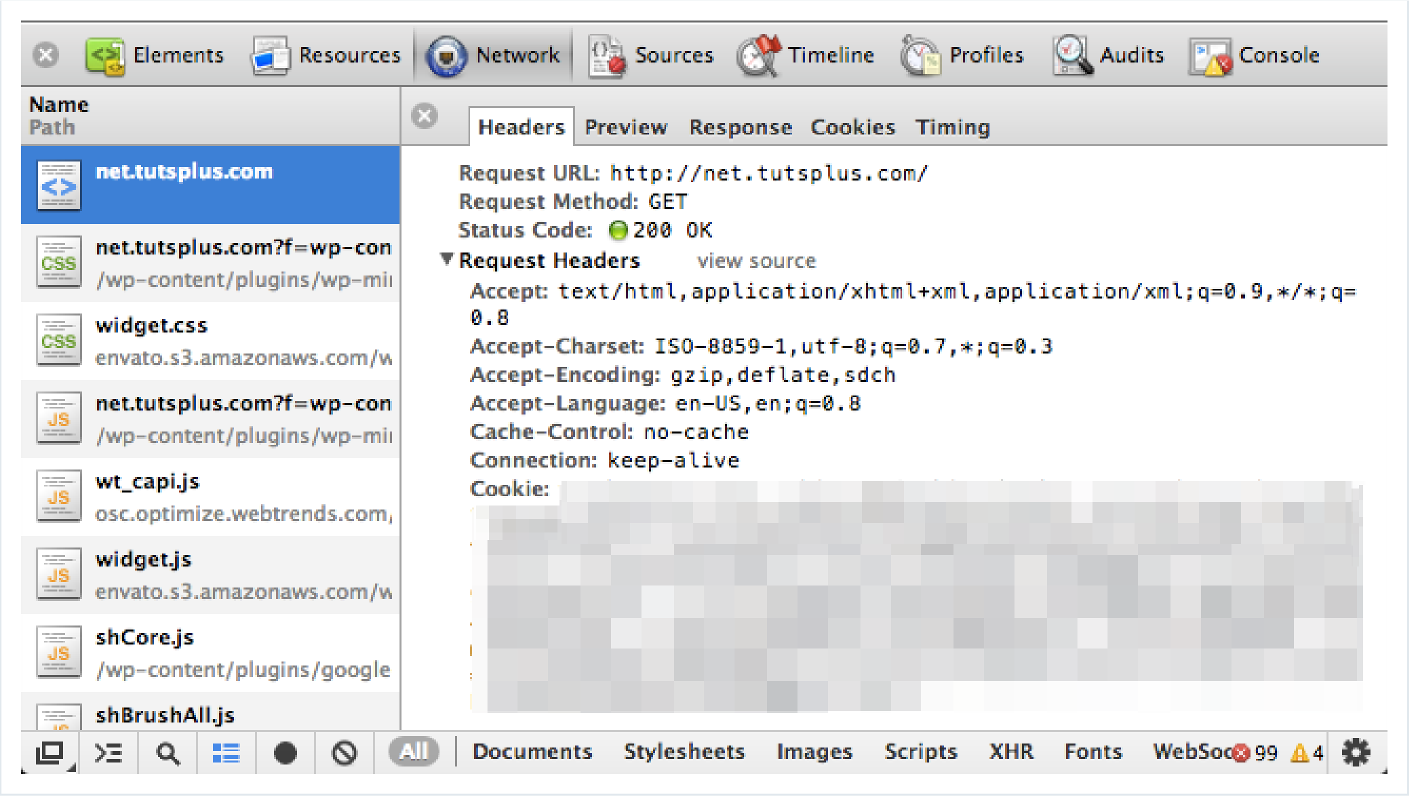
- `Age` is the time in seconds since the message was generated on the server.
- `ETag` is the MD5 hash of the entity and used to check for modifications.
- `Location` is used when sending a redirection and contains the new URL.
- `Server` identifies the server generating the message.

It's been a lot of theory upto this point, so I won't blame you for drowsy eyes. In the next sections, we will get more practical and take a survey of the tools, frameworks and libraries.

Tools to View HTTP Traffic

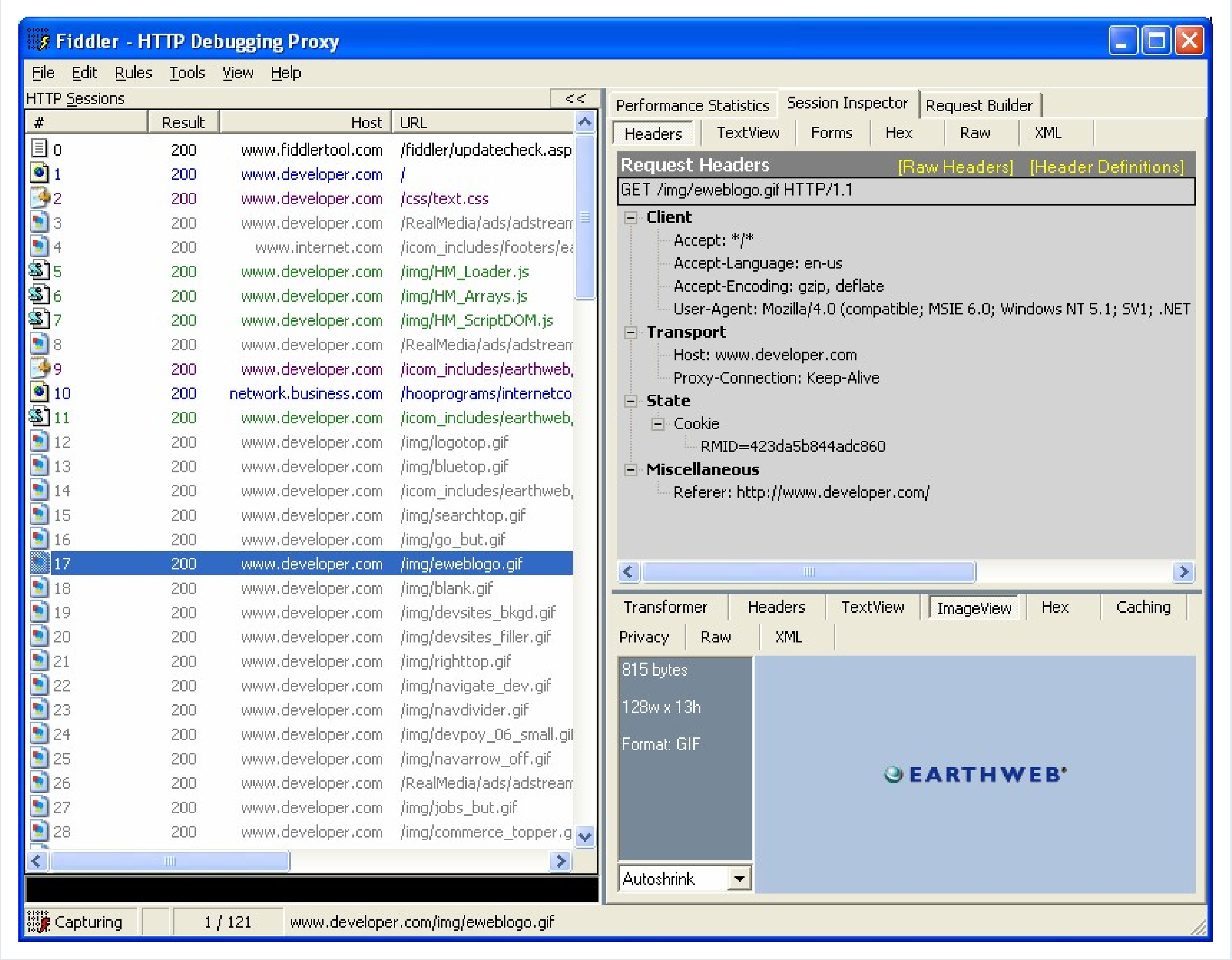
There are a number of tools available to monitor HTTP communication. Here, we list some of the more popular tools.

Undoubtedly, the **Chrome/Webkit inspector** is a favorite amongst web developers:



```
Request URL: http://net.tutsplus.com/
Request Method: GET
Status Code: 200 OK
▼Request Headers      view source
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=
  0.8
  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
  Accept-Encoding: gzip,deflate,sdch
  Accept-Language: en-US,en;q=0.8
  Cache-Control: no-cache
  Connection: keep-alive
  Cookie: [redacted]
```

There are also web debugging proxies, like **Fiddler** on Windows and **Charles Proxy** for OSX. My colleague, Rey Bango wrote an excellent article on this topic.

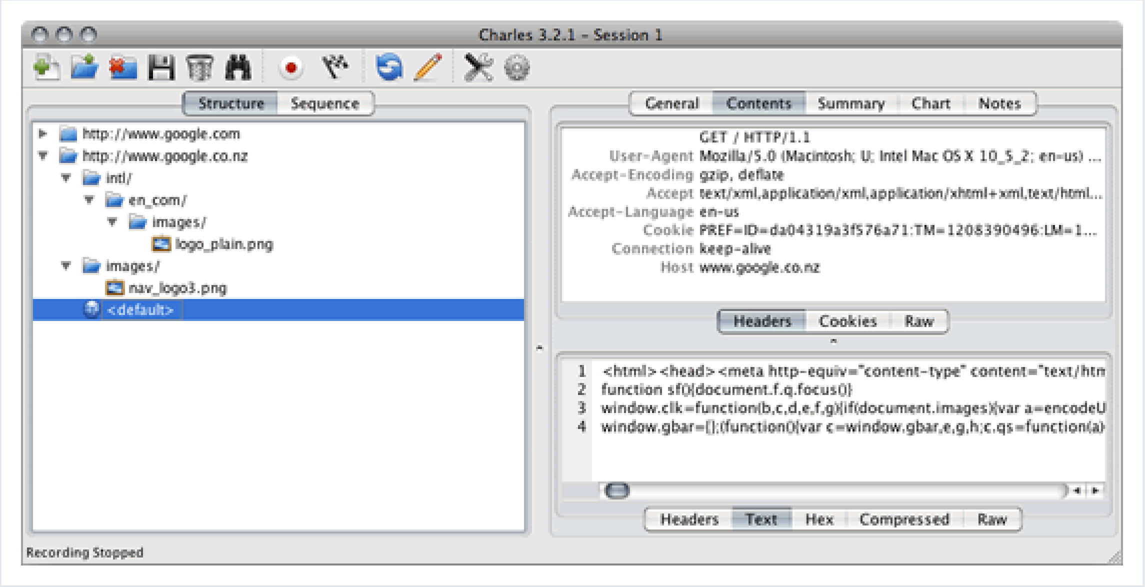

Fiddler - HTTP Debugging Proxy
File Edit Rules Tools View Help
HTTP Sessions
Performance Statistics
Session Inspector
Request Builder
Headers TextView Forms Hex Raw XML
Transformer Headers TextView ImageView Hex Caching
Privacy Raw XML
EARTHWEB
Autoshrink
Capturing

#	Result	Host	URL
0	200	www.fiddlertool.com	/fiddler/updatecheck.asp
1	200	www.developer.com	/
2	200	www.developer.com	/css/text.css
3	200	www.developer.com	/RealMedia/ads/adstrean
4	200	www.internet.com	/icom_includes/footers/ei
5	200	www.developer.com	/img/HM_Loader.js
6	200	www.developer.com	/img/HM_Arrays.js
7	200	www.developer.com	/img/HM_ScriptDOM.js
8	200	www.developer.com	/RealMedia/ads/adstrean
9	200	www.developer.com	/icom_includes/earthweb,
10	200	network.business.com	/hooprograms/internetco
11	200	www.developer.com	/icom_includes/earthweb,
12	200	www.developer.com	/img/logotop.gif
13	200	www.developer.com	/img/bluetop.gif
14	200	www.developer.com	/icom_includes/earthweb,
15	200	www.developer.com	/img/searchtop.gif
16	200	www.developer.com	/img/go_but.gif
17	200	www.developer.com	/img/eweblogo.gif
18	200	www.developer.com	/img/blank.gif
19	200	www.developer.com	/img/devsites_bkgd.gif
20	200	www.developer.com	/img/devsites_filler.gif
21	200	www.developer.com	/img/righttop.gif
22	200	www.developer.com	/img/navigate_dev.gif
23	200	www.developer.com	/img/navdivider.gif
24	200	www.developer.com	/img/devpoy_06_small.gil
25	200	www.developer.com	/img/navarrow_off.gif
26	200	www.developer.com	/RealMedia/ads/adstrean
27	200	www.developer.com	/img/jobs_but.gif
28	200	www.developer.com	/img/commerce_topper.g

```
Request Headers                    [Raw Headers]  [Header Definitions]
GET /img/eweblogo.gif HTTP/1.1
Client
    Accept: */*
    Accept-Language: en-us
    Accept-Encoding: gzip, deflate
    User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET
Transport
    Host: www.developer.com
    Proxy-Connection: Keep-Alive
State
    Cookie
        RMID=423da5b844adc860
Miscellaneous
    Referer: http://www.developer.com/
```

815 bytes
128w x 13h
Format: GIF

1 / 121 www.developer.com/img/eweblogo.gif


Charles 3.2.1 - Session 1
Structure Sequence
General Contents Summary Chart Notes
Headers Cookies Raw
Headers Text Hex Compressed Raw
Recording Stopped

```
▸ http://www.google.com
▾ http://www.google.co.nz
    ▾ intl/
        ▾ en_com/
            ▾ images/
                logo_plain.png
    ▾ images/
        nav_logo3.png
    <default>
```

```
                  GET / HTTP/1.1
       User-Agent Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_2; en-us) ...
  Accept-Encoding gzip, deflate
           Accept text/xml,application/xml,application/xhtml+xml,text/html...
  Accept-Language en-us
           Cookie PREF=ID=da04319a3f576a71:TM=1208390496:LM=1...
       Connection keep-alive
             Host www.google.co.nz
```

```
<html><head><meta http-equiv="content-type" content="text/htm
function sf(){document.f.q.focus()}
window.clk=function(b,c,d,e,f,g){if(document.images){var a=encodeU
window.gbar={};(function(){var c=window.gbar,e,g,h;c.qs=function(a)
```

For the command line, we have utilities like curl, tcpdump and tshark for monitoring HTTP traffic.

Using HTTP in Web Frameworks and Libraries

Now that we have looked at the request/response messages, it's time that we learn how libraries and frameworks expose it in the form of an API. We'll use *ExpressJS for Node*, *Ruby on Rails*, and *jQuery Ajax* as our examples.

ExpressJS

If you are building web servers in NodeJS, chances are high that you've considered ExpressJS. ExpressJS was originally inspired by a Ruby Web framework, called Sinatra. As expected, the API is also equally influenced.

Because we are dealing with a server-side framework, there are two primary tasks when dealing with HTTP messages:

- Read URL fragments and request headers.
- Write response headers and body

> *Understanding HTTP is crucial for having a clean, simple and RESTful interface between two endpoints.*

ExpressJS provides a simple API for doing just that. We won't cover the details of the API. Instead, we will provide links to the detailed documentation on ExpressJS guides. The methods in the API are self-explanatory in most cases. A sampling of the request-related API is below:

- req.body: get the request body.
- req.query: get the query fragment of the URL.
- req.originalUrl
- req.host: reads the `Host` header field.
- req.accepts: reads the acceptable MIME-types on the client side.
- req.get OR req.header: read any header field passed as argument.

On the way out to the client, ExpressJS provides the following response API:

- res.status: set an explicit status code.
- res.set: set a specific response header.
- res.send: send HTML, JSON or an octet-stream.
- res.sendFile: transfer a file to the client.
- res.render: render an express view template.

- res.redirect: redirect to a different route. Express automatically adds the default redirection code of 302.

Ruby on Rails

> *The request and response messages are mostly the same, except for the first line and message headers.*

In Rails, the ActionController and ActionDispatch modules provide the API for handling request and response messages.

ActionController provides a high level API to read the request URL, render output and redirect to a different end-point. An end-point (aka route) is handled as an action method. Most of the necessary context information inside an action-method is provided via the `request`, `response` and `params` objects.

- params: gives access to the URL parameters and POST data.
- request: contains information about the client, headers and URL.
- response: used to set headers and status codes.
- render: render views by expanding templates.
- redirect_to: redirect to a different action-method or URL.

ActionDispatch provides fine-grained access to the request/response messages, via the `ActionDispatch::Request` and `ActionDispatch::Response` classes. It exposes a set of query methods to check the type of request (`get?()`, `post?()`, `head?()`, `local?()`). Request headers can be directly accessed via the `request.headers()` method.

On the response side, it provides methods to set `cookies()`, `location=()` and `status=()`. If you feel adventurous, you can also set the `body=()` and bypass the Rails rendering system.

jQuery Ajax

Because jQuery is primarily a client-side library, its Ajax API provides the opposite of a server-side framework. In other words, it allows you to *read* response messages and *modify* request messages. jQuery exposes a simple API via jQuery.ajax(settings):

By passing a `settings` object with the `beforeSend` callback, we can modify the request headers. The callback receives the jqXHR (jQuery XMLHttpRequest) object that exposes a method, called `setRequestHeader()` to set headers.

```
$.ajax({
    url: 'http://www.articles.com/latest',
    type: 'GET',
    beforeSend: function (jqXHR) {
      jqXHR.setRequestHeader('Accepts-Language', 'en-US,en');
    }
  });
```

- The jqXHR object can also be used to read the response headers with the `jqXHR.getResponseHeader()`.
- If you want to take specific actions for various status codes, you can use the `statusCode` callback:

```
$.ajax({
  statusCode: {
    404: function() {
      alert("page not found");
    }
  }
});
```

Summary

So that sums up our quick tour of the HTTP protocol.

> *We reviewed URL structure, verbs and status codes: the three pillars of HTTP communication.*

The request and response messages are mostly the same, except for the first line and message headers. Finally, we reviewed how you can modify the request and response headers in web frameworks and libraries.

Understanding HTTP is crucial for having a clean, simple, and RESTful interface between two endpoints. On a larger scale, it also helps when designing your network infrastructure and providing a great experience to your end users.

In part two, we'll review connection handling, authentication and caching! See you then.

References

- HTTP specification
- HTTP Definitive Guide