# HTTP

The web’s protocol

# HTTP Connects the World

**H**yper**T**ext

**T**ransfer

**P**rotocol

The protocol was originally described in 1991, with the origins of hypertext dating back to as early as 1930!

# Hypertext

- The HTTP protocol was originally designed as a protocol for transmitting hypertext.
- The modern equivalent of hypertext is HTML
- Today, HTML has grown to support much more than just linking from one page to another, but it was the concept of linked pages that uniquely defined the early web.
- HTTP has also found new use cases, while transmitting HTML is a common use, HTTP can transfer any type of document or file.
- Today HTML isn’t even the most common type of document transmitted over HTTP. Images, CSS, Javascript, API responses, etc are far more common.

# HTTP Conversations

HTTP describes a conversation between two parties:

1. Client
2. Server

The client sends the HTTP request and the server sends the HTTP response.

Both HTTP requests & responses are plaintext. They are easily read by a human and each has a very simple structure.

```
GET /bio.html HTTP/1.1

Host: example.com

User-Agent: Google Chrome 52.34
```

Request

```
200 OK HTTP/1.1

Content-Length: 424

Date: Sat, 31 Dec 2016 20:20:14 GMT

Content-Type: text/html


<html>

  <head>

    <title>Superman Biography</title>

  </head>

...
```

Response

```
GET /bio.html HTTP/1.1

Host: example.com

User-Agent: Google Chrome 52.34
```

Request

```
200 OK HTTP/1.1

Content-Length: 424

Date: Sat, 31 Dec 2016 20:20:14 GMT

Content-Type: text/html


<html>

  <head>

    <title>Superman Biography</title>

  </head>

...
```

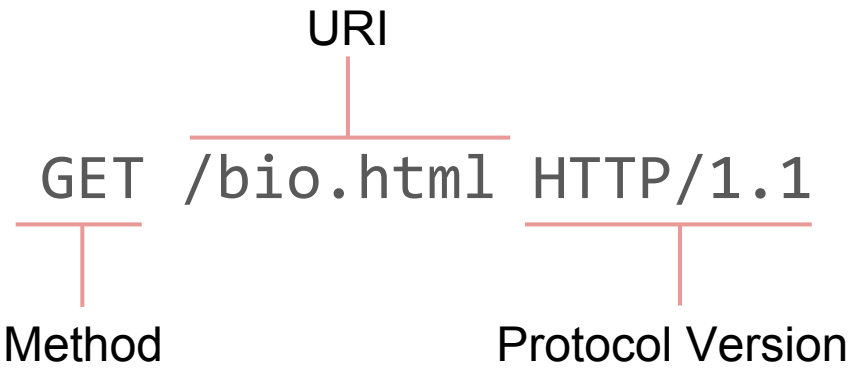
Response

# Request line

```
GET /bio.html HTTP/1.1

Host: example.com

User-Agent: Google Chrome 52.34
```



# Request Line

Common HTTP verbs include:

| | |
|---|---|
| `GET` | Retrieve a resource |
| `POST` | Send a resource |
| `DELETE` | Delete a resource |
| `HEAD` | Retrieve information about a resource, but not the resource itself (e.g. when does this resource expire?) |

# Request Line


**U**niform **R**esource **I**dentifier

# URI

```
GET /bio.html HTTP/1.1

Host: example.com

User-Agent: Google Chrome 52.34
```

Request

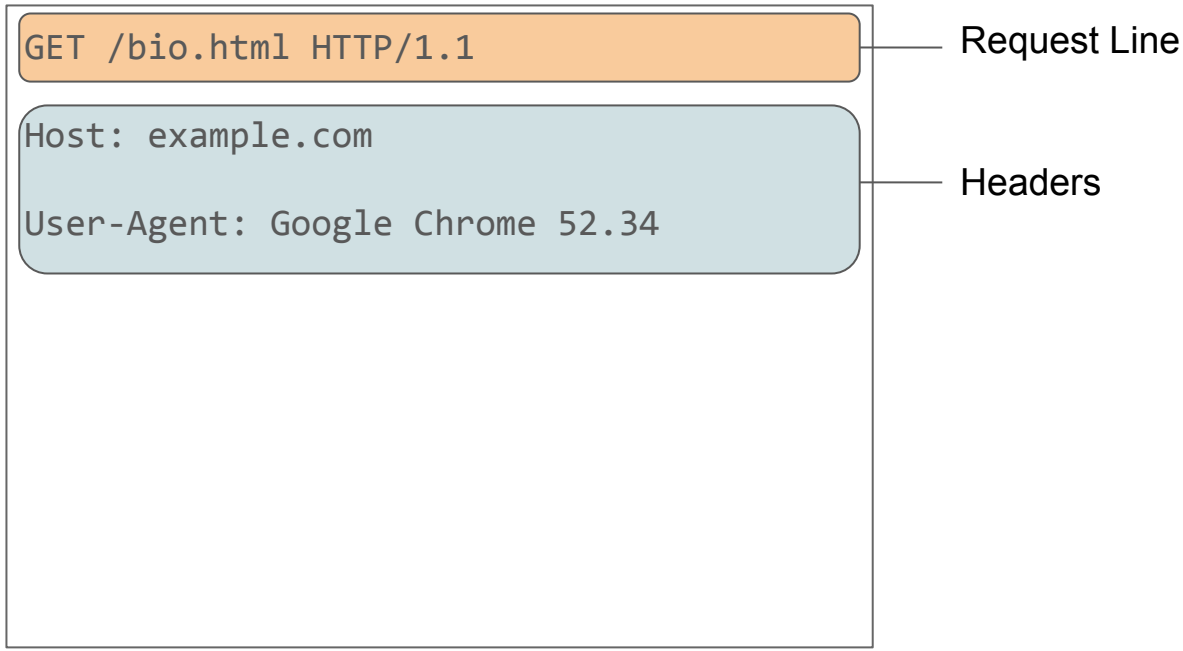
## URN

host + URI
example.com/bio.html

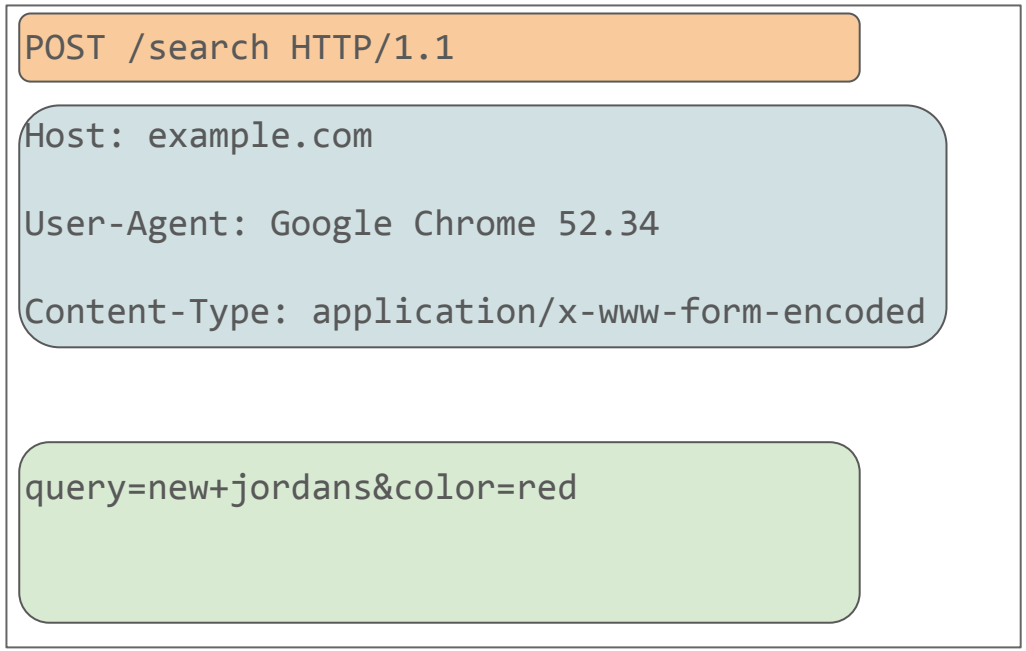
## URL

protocol + host + URI
http://example.com/bio.html

# HTTP Request



# HTTP Request (with body)




Request

Request Line

Headers

Blank Line

Body (optional)

Note: Request bodies are most commonly included with POST, PUT, PATCH & DELETE requests and can include all types of content (e.g. image uploads, form content, etc)

# HTTP Request with Query Params

```
GET /search?query=air%20jordans&color=red HTTP/1.1
```

```
Host: example.com

User-Agent: Google Chrome 52.34
```

Additional request data can be included as part of the URI instead of being included in the request body:

`/search?query=air%20jordans&color=red`

Query params are separated from the URI with a ? and individual query params are separated by an &. Query params can be combined the request bodies.

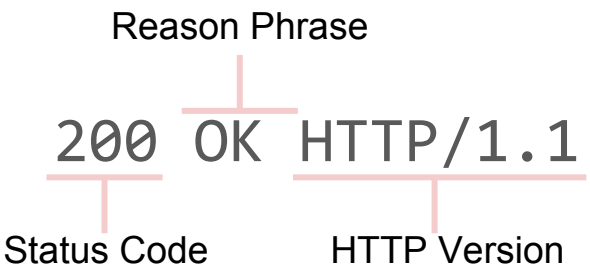
# Request Headers

Request headers provide additional context about the request. Some examples:

| | |
|---|---|
| `Host: example.com` | Tells the web server which domain the client is requesting content from. Often a single web server will serve content for multiple domains (sometimes called virtual hosts). The server users this header to determine which content to serve. |
| `Accept-Language: en` | The client let's the server know which language(s) it prefers the response be sent back in. |
| `User-Agent: Google Chrome 53.2` | Which type of client sent this request? Sometimes the server will respond differently based on the UA field. |
| `Accept: text/html, text/plain` | What format response is the client interested in receiving? HTML? An image? JSON? |

# HTTP Response

# Response Status Line



# Status Codes

The response status code tells the client the result of the request. Common status codes include:

| |
|---|
| `200 OK` |
| `302 See Other` (aka redirect) |
| `400 Bad Request` |
| `401 Unauthorized` |
| `404 Not Found` |
| `500 Internal Server Error` |

# Response Headers

These headers serve a similar purpose as the request headers. They provide additional context that the client uses to interpret the response. Common response headers include:

| | |
|---|---|
| `Content-Type: text/html` | What format is the response body? The client uses this header to properly parse the response. |
| `Content-Length: 492` | How many bytes long is the response body? This header tells the client how many bytes to read from the network socket. |
| `Set-Cookie: currency=usd` | This header requests that the client save a cookie with particular name & value. |
| `Date: Fri, 22 Jan 2010 04:00:00 GMT` | When was the response sent? |

# Many HTTP Conversations

Imagine a user browsing a few pages on a typical website. Dozens or hundreds of HTTP conversations will take place.

- Loading the homepage HTML: One HTTP request & response
- Loading CSS: One HTTP request & response per CSS file
- Loading Javascript: One HTTP request & response per JS file
- Loading Images: One HTTP request & response per image

And that’s just for a single page!

# Stateless

A very important characteristic of HTTP is that every conversation shares no state with any previous conversation.

Every HTTP conversation begins with a new connection from the client to the server. It’s as if the two parties are meeting for the first time. Even requests made just moments before have no bearing on the current request.

**Server State**

Most servers clear all state (variables) with each request. This means all state when processing a request come from the request itself.

# Stateless?

HTTP cookies help retain state like “who is logged in” or “what was their last search” across multiple requests.

This might sound like a total violation of the stateless nature of HTTP, but even with HTTP cookies, HTTP as a protocol remains stateless. It all has to do with how HTTP cookies are implemented.

# Cookies in Short

You'll learn more about how HTTP cookies work, but for now, here's a summary:

1. The server sets cookies by sending them to the client as part of the HTTP response.

2. The client stores the cookies.

3. The server does not store any cookies.

3. Any cookies for the current domain are sent to the server as part of the HTTP request.

# Summary

HTTP is a proven protocol. It's extensible and has has found uses far beyond what was imagined in 1991.

It's relatively simple, human readable and easy to parse with code.

HTTP first changed the world by connecting users and servers to form "the web", but today it's the foundation of so much more than the web. You'll soon realize that HTTP is *everywhere*.