# HTTP Request Lifestyle

.NET

*HTTP sessions consist of three phases.*
*1. The client establishes a (usually) TCP connection.*
*2. The client sends its request and waits for the answer.*
*3. The server processes the request, sending back its answer, with a status code and data.*

# DNS (Domain Name System)

Domain Name System (DNS) is an industry-standard protocol that makes up TCP/IP.

Together the DNS Client and DNS Server provide computer name-to-IP address mapping name resolution services to computers and users.

The DNS translates domain names (www.revature.com) to numerical IP addresses (255.255.255) for locating and identifying computer services and devices over the web.

The DNS also associates identifying data with the unique domain names assigned to each connected entity.



Complete DNS Lookup and Webpage Query
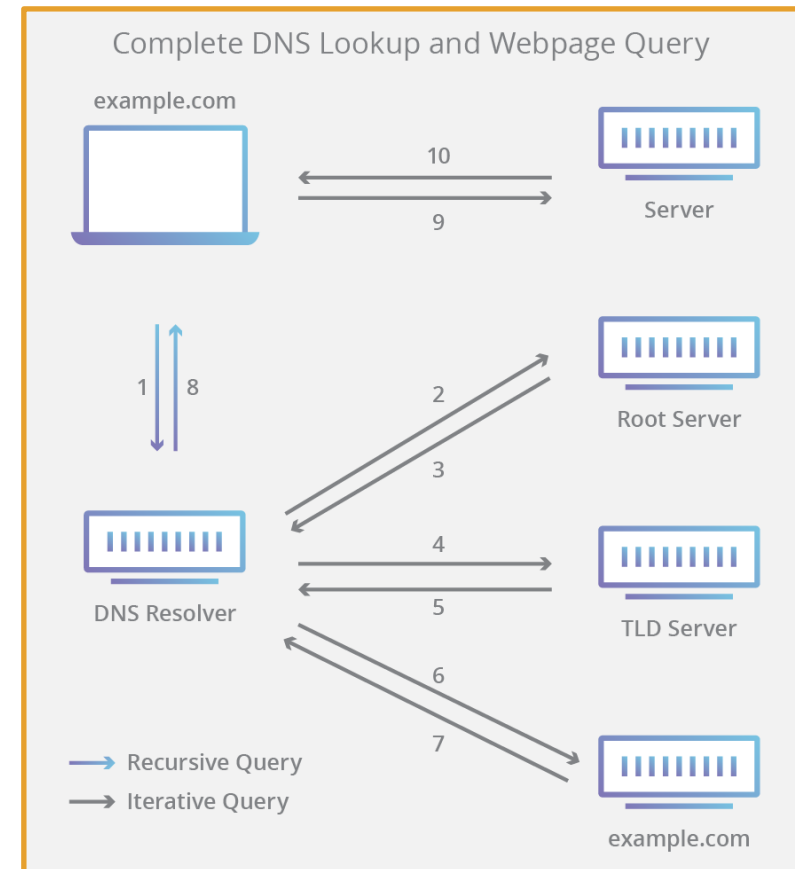
# Domain Name System

Domain Name System (DNS) is one of the industry-standard suite of protocols that comprise TCP/IP, and together the DNS Client and DNS Server provide computer name-to-IP address mapping name resolution services to computers and users.

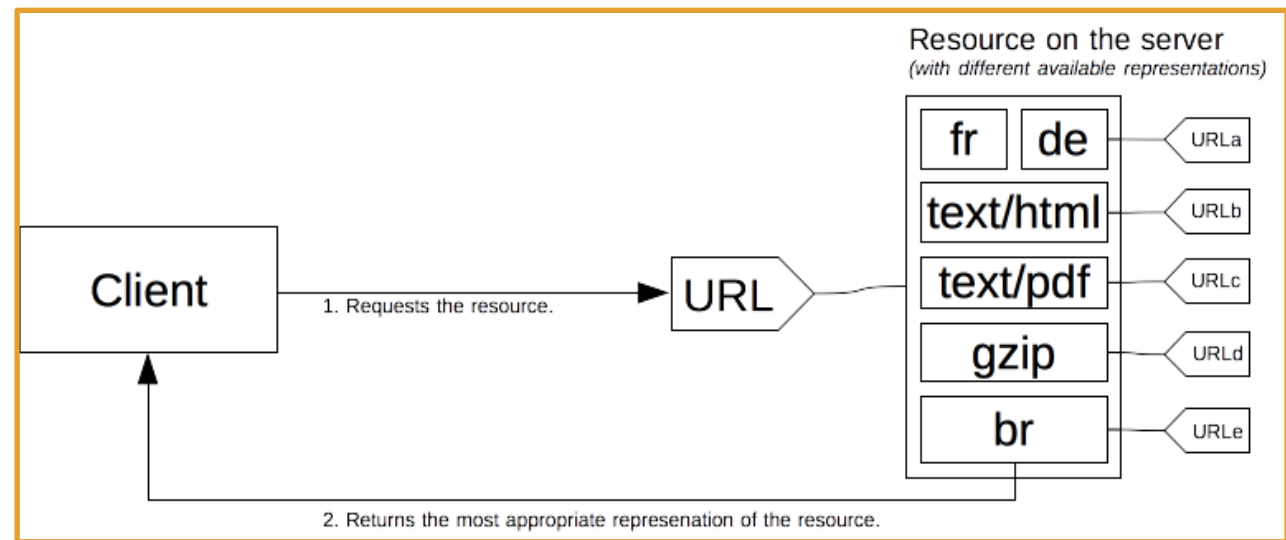There are 4 DNS servers used to look up a domain name.

| Name | Purpose | Analogy |
|---|---|---|
| DNS recursor | A server that receives queries from client machines through a web browser. It then makes additional requests to satisfy the client's DNS query. | A librarian that shows you the card catalog of books. |
| Root Nameserver | The first step in translating human-readable hostnames into IP addresses. It serves as a reference to other more specific locations. | The card catalog. |
| TLD Nameserver | The Top-Level Domain server hosts the last portion of a hostname. For example, in Revature.com, the TLD server is "com"). | A specific shelf of books in the library. |
| Authoritative Nameserver | The last stop in the nameserver query. If it has access to the requested record, it will return the IP address for the requested hostname back to the DNS Recursor (the librarian) that made the initial request. | The number on the book spine. |

# Content negotiation

In HTTP, *content negotiation* is the mechanism that is used for serving different representations of a resource at the same URI so that the user agent can specify which is best suited for the user (which language, which image format, or which content encoding).

The client requests a resource using its URL. The server uses this URL to choose one of the variants (representations) it provides to return. The overall resource and each representation have a specific URL. How a specific representation is chosen when the resource is called is determined by *content negotiation*.

# Types of Content Negotiation

HTTP headers provide "Proactive" content negotiation. This is the standard method, is server-driven, and has many types.

The browser sends several HTTP headers along with the URL with every request. Headers describe the preferences of the user. The server chooses the best content to serve to the client. If it cannot provide a suitable resource, as a fallback it might respond with 406 (Not Acceptable) or 415 (Unsupported Media Type) and set headers for the types of media that it supports.

| Header | Purpose |
| --- | --- |
| Accept | Negotiation by format. A comma-separated list of MIME types the client can process. |
| Accept-Charset | Negotiation by character encoding. Tells what char characters are supported by the browser. |
| Accept-Language | Negotiation by natural language. Indicates users' language preference. |
| Accept-Encoding | Negotiation by compression. Defines the acceptable content-encoding compressions. This is for optimization. Ex. br, gzip;q=0.8 |

# Drawbacks of Content Negotiation

https://wiki.whatwg.org/wiki/Why_not_conneg
https://developer.mozilla.org/en-US/docs/Web/HTTP/Content_negotiation

---

Server-driven content negotiation is the most common way to agree on a specific representation of a resource, but it has several drawbacks:

- It doesn't scale well. With more specificity more and more headers must be sent with every request.

- The server doesn't have total knowledge of the browser. Even with the *Client Hints* extension, it has an incomplete knowledge of the capabilities of the browser. Unlike *Reactive Content Negotiation* where the client makes the choice, the server choice is always somewhat arbitrary.

- The information by the client is quite verbose (HTTP/2 header compression mitigates this problem) and a privacy risk (HTTP fingerprinting)

- As several representations of a given resource are sent, shared caches are less efficient and server implementations are more complex.

HTTP-based negotiation in general is worse than letting the browser choose from alternative URLs. HTTP-based codec negotiation solution is often worse than the actual browser-side codec negotiation solution.

# HTTP Request Life Cycle

HTTP sessions consist of three basic phases:

1. The client establishes a TCP connection.
2. The client sends its *Request* and waits for the *Response*.
3. The server processes the request and sends back its *Response* with a *status code* and appropriate data.



**HTTP Request**

World Wide Web or Internet

**HTTP Response**

200 OK

# Establish a connection
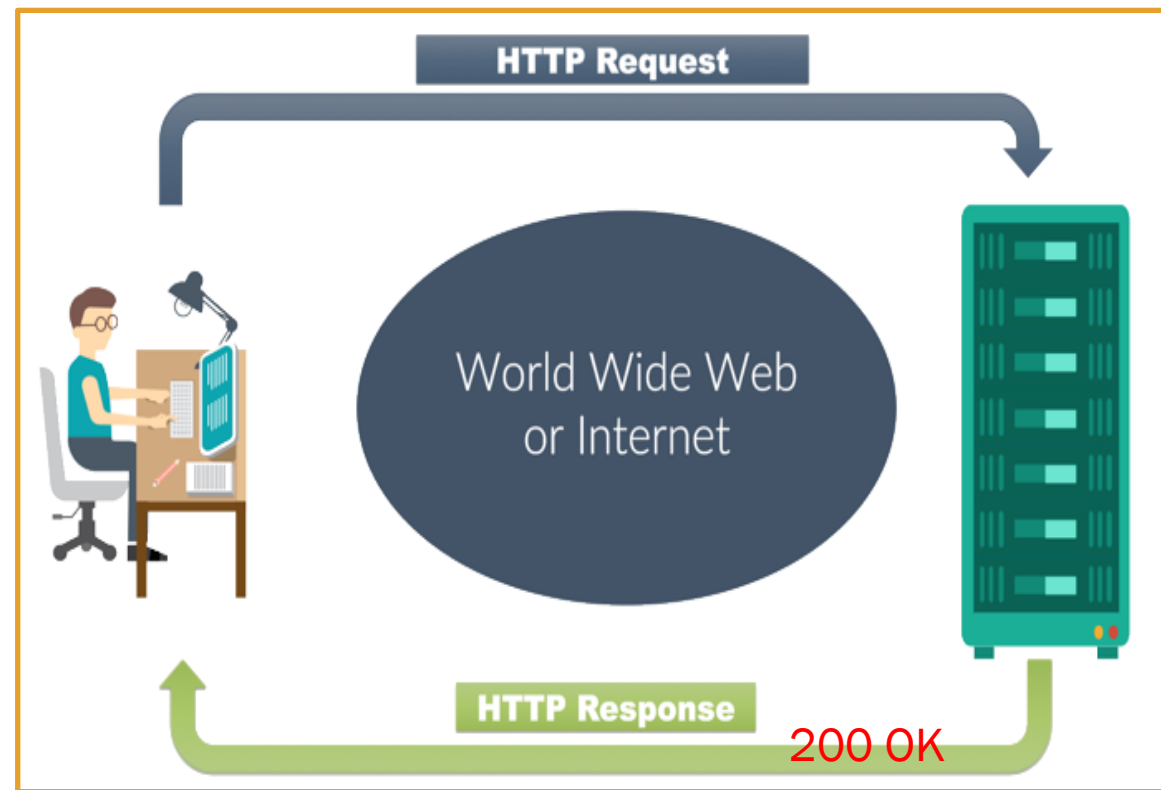
https://developer.mozilla.org/en-US/docs/Web/HTTP/Session

1. The client establishes the connection.
   - The HTTP default port is :80.
2. The client sends its request and waits for the answer.
   - The *URL* of a page to fetch contains both the domain name, and the port number.
3. The server processes the request, sending back its answer, providing a status code and appropriate data.

The client-server model requires an explicit *Request*.

Workarounds to this limitation are:
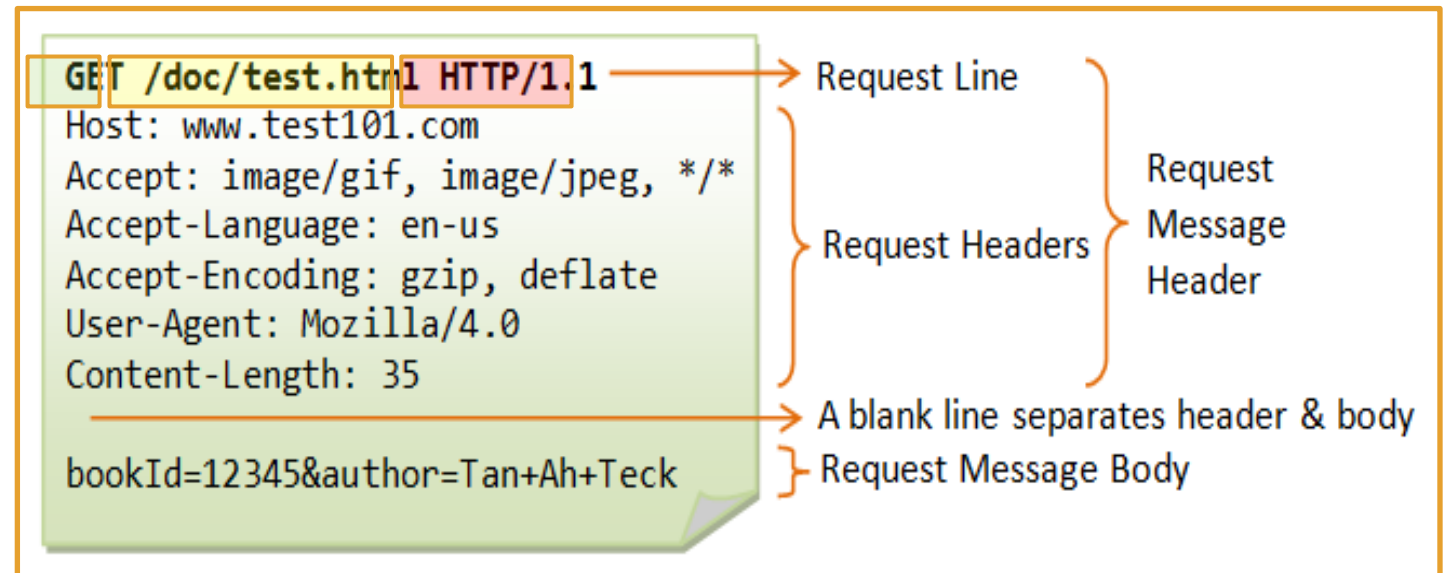
- ping the server periodically via the XMLHTTPRequest,
- Fetch APIs,
- using the WebSockets API
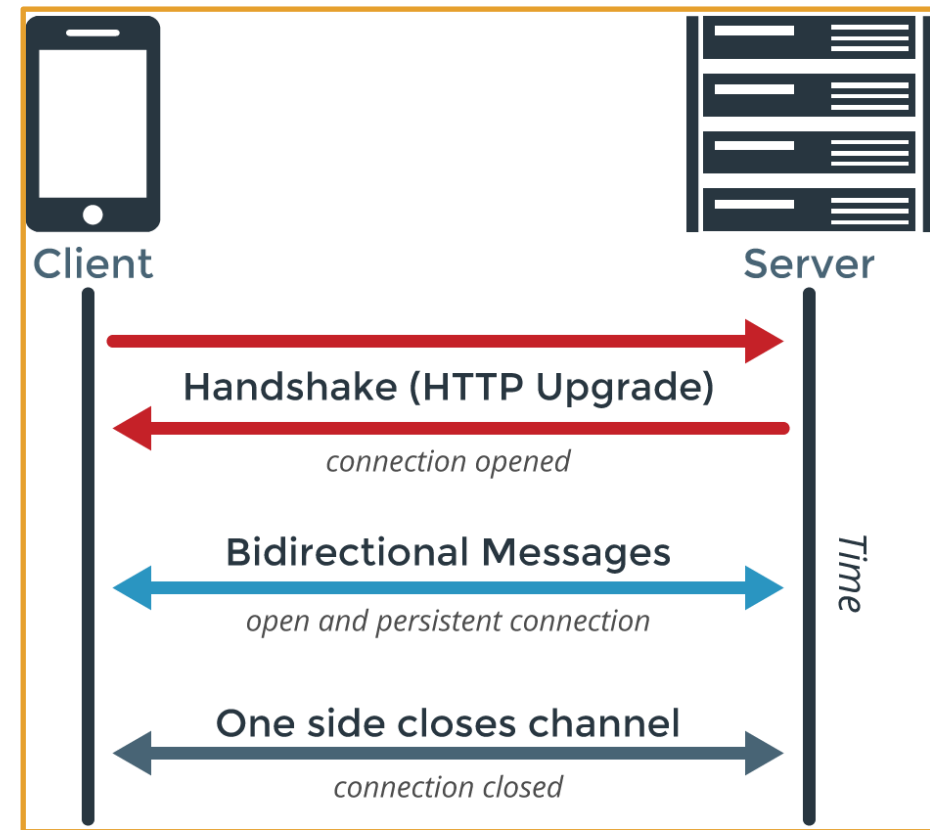
# Connecting and Sending A Request(1/2)

https://developer.mozilla.org/en-US/docs/Web/HTTP/Session

- The client always establishes the connection.
- The client-server model does not allow the server to send data to the client without an explicit Request.
- The HTTP default port is port:80.
- The URL of a page to fetch contains both the domain name, and the port number.

```
GET /doc/test.html HTTP/1.1          → Request Line
Host: www.test101.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate       Request Headers
User-Agent: Mozilla/4.0
Content-Length: 35

                                      → A blank line separates header & body
bookId=12345&author=Tan+Ah+Teck      Request Message Body
```

Request Message Header

On successful connection, the web browser sends the request. A client request consists of text directives, separated by CRLF (Carriage Return, Line Feed), divided into three blocks: Line 1 - request method followed by the absolute URL doc path without the protocol or domain name and the HTTP protocol version.
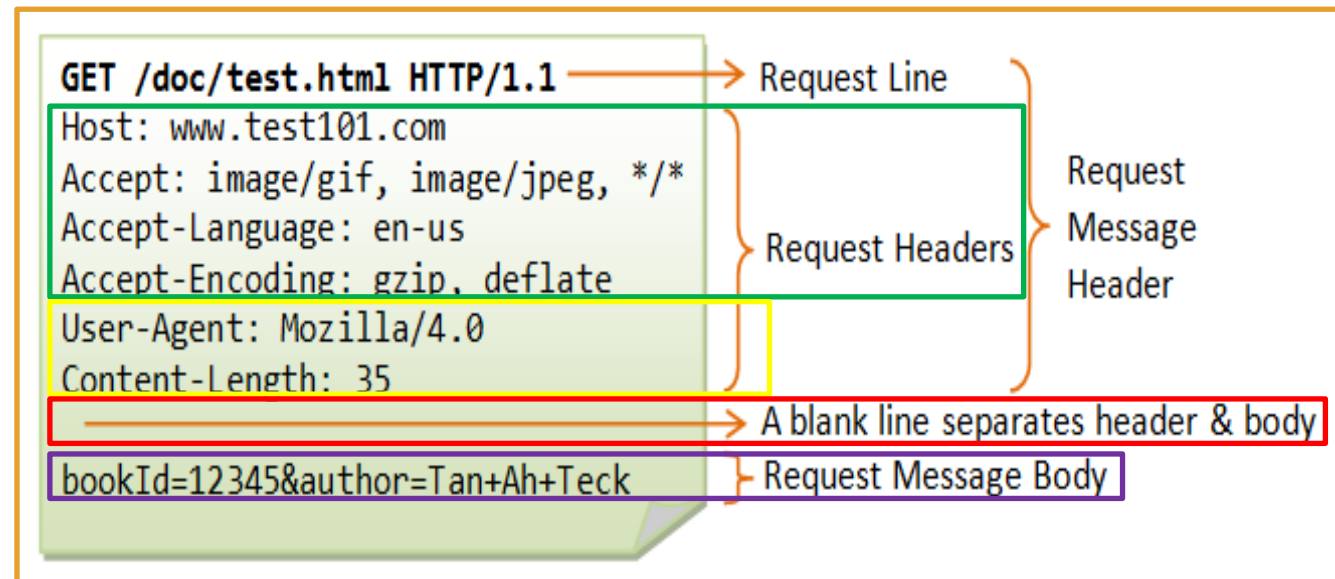
# Sending A Request (2/2)

HTTP headers provide the server with:

- information about what type/formats of data is appropriate in the response (e.g., what language, what MIME types, etc),

- other headers which inform and may alter its behavior such as not sending an answer if it is already cached,

The Header section is followed by:

- An empty line.

- An (optional) **data** block, mainly used by the POST method which contains further data.

```
GET /doc/test.html HTTP/1.1              → Request Line
Host: www.test101.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us                    Request Headers
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Content-Length: 35
                                          → A blank line separates header & body
bookId=12345&author=Tan+Ah+Teck          ⊢ Request Message Body
```

Request Message Header

# Request Methods

*Request methods* (*HTTP verbs*) indicate the desired action to be performed on a resource. Request verbs are *GET*, *POST*, *PUT*, *DELETE*, *TRACE*, *HEAD*, *CONNECT*, and *OPTIONS*.

- The *POST* method sends data to a server. *POST* is used mainly for HTML *Forms*.
- The *GET* method requests/retrieves data.

```
1   GET / HTTP/1.1
2   Host: developer.mozilla.org
3   Accept-Language: fr
```

```
1   POST /contact_form.php HTTP/1.1
2   Host: developer.mozilla.org
3   Content-Length: 64
4   Content-Type: application/x-www-form-urlencoded
5
6   name=Joe%20User&request=Send%20me%20one%20of%20your%20catalogue
```

# Request Methods

https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

| HTTP Verb | Description |
|---:|:---|
| *GET | Requests the specified resource. GET should only retrieve data. |
| HEAD | Just like a GET request, but without the response body. |
| *POST | Used to submit an entity to the specified *resource*. |
| *PUT | Replaces <u>all</u> current representations of the target resource with the payload. |
| *DELETE | Deletes the specified resource. |
| CONNECT | Establishes a tunnel to the server identified by the target resource. |
| OPTIONS | Describes the communication options for the target resource. |
| TRACE | Performs a message loop-back test along the path to the target resource. |
| PATCH | Applies partial modifications to a resource. |

*most common

# Response

```
1  HTTP/1.1 200 OK
2  Content-Type: text/html; charset=utf-8
3  Content-Length: 55743
4  Connection: keep-alive
5  Cache-Control: s-maxage=300, public, max-age=0
6  Content-Language: en-US
7  Date: Thu, 06 Dec 2018 17:37:18 GMT
8  ETag: "2e77ad1dc6ab0b53a2996dfd4653c1c3"
9  Server: meinheld/0.6.1
0  Strict-Transport-Security: max-age=63072000
1  X-Content-Type-Options: nosniff
2  X-Frame-Options: DENY
3  X-XSS-Protection: 1; mode=block
4  Vary: Accept-Encoding,Cookie
5  Age: 7
6
              ...Here is the empty line...
7
8  <!DOCTYPE html>
9  <html lang="en">
0  <head>
1    <meta charset="utf-8">
2    <title>A simple webpage</title>
3  </head>
4  <body>
5    <h1>Simple HTML5 webpage</h1>
6    <p>Hello, world!</p>
7  </body>
8  </html>
```

The server processes the *Request* and returns a *Response*. A server *Response* is formed of text directives, separated by CRLF (Carriage Return Line Feed), divided into three blocks:

Line 1 (the status line) :an acknowledgment of the HTTP version used, and a request *status code* (and its meaning).

Subsequent lines represent specific HTTP headers, giving the client information like data type and size, compression algorithm used, hints about caching, etc. It ends with an empty line.

The final block is a data block which contains the (optional) data
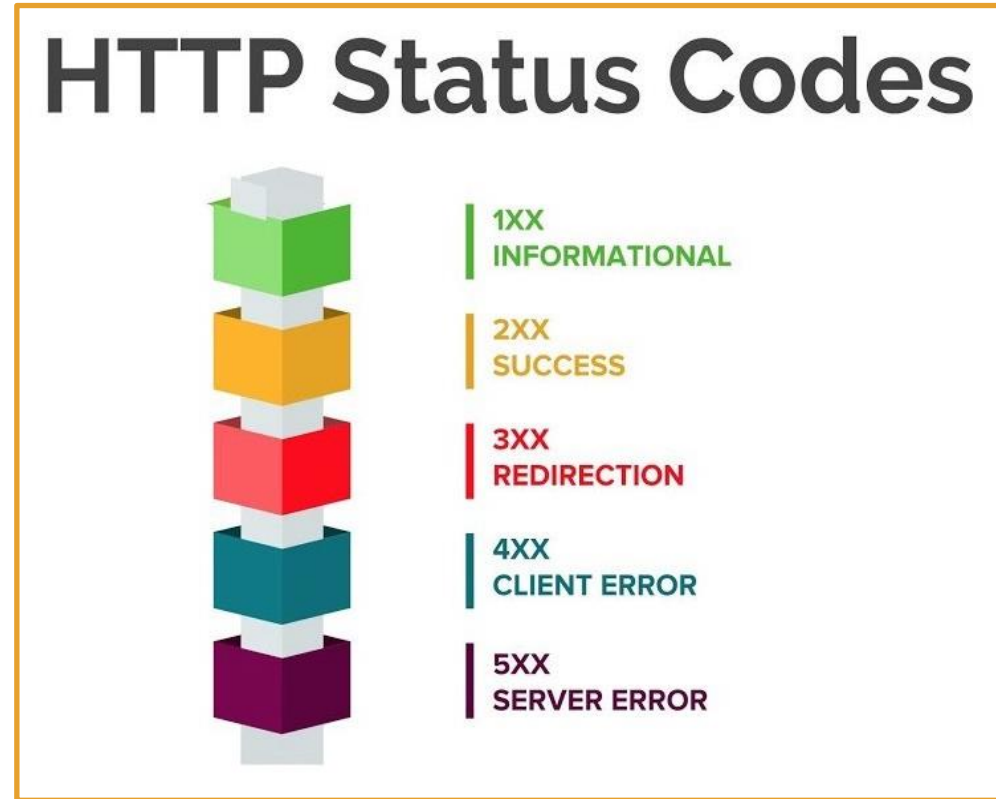
# Response Status Codes

HTTP response status codes give the result of an HTTP request. Responses are grouped in five classes:

- Informational responses (100–199),
- Successful responses (200–299),
- Redirects (300–399),
- Client errors (400–499),
- and Server errors (500–599).
- Cheat Sheet



HTTP Status Codes

- 1XX INFORMATIONAL
- 2XX SUCCESS
- 3XX REDIRECTION
- 4XX CLIENT ERROR
- 5XX SERVER ERROR

# Common Response Status Codes

| Code number | Code Meaning |
| --- | --- |
| 200 OK, 201 | The request has succeeded. Request has been fulfilled resulting in new resource(s) created. |
| 300 Multiple Choices | The requested resource has different choices and cannot be resolved into one. |
| 301 Moved Permanently | The requested resource has been assigned a new permanent URI |
| 304 Not Modified | Client performed a conditional GET request. Access is allowed. The document is unmodified |
| 307 Temporary Redirect | The requested resource resides temporarily under a different URI. |
| 400 Bad Request | The request could not be understood by the server due to malformed syntax. |
| 401 Unauthorized | The request requires user authentication. |
| 403 Forbidden | The server understood the request but is refusing to fulfill it. |
| 404 Not Found | The server has not found anything matching the Request-URI. |
| 410 Gone | The requested resource is no longer available at the server and no forwarding address is known. |
| 500 Internal Server Error | The server encountered an unexpected condition which prevented it from fulfilling the request. |
| 501 Not Implemented | The server does not support the functionality required to fulfill the request. |
| 503 Service Unavailable | Your web server is unable to handle your HTTP request at the time. |
| 550 Permission Denied | Your account does not have permission to perform the action you are attempting. |

# 'Safe' and 'Idempotent'

https://developer.mozilla.org/en-US/docs/Glossary/safe
https://developer.mozilla.org/en-US/docs/Glossary/Idempotent

An HTTP method is called *safe* if it doesn't alter the state of the resource, meaning it leads to a read-only operation. *GET*, *HEAD*, and *OPTIONS* are *safe.*

*idempotent* - an identical request can be made once or several times in a row with the same effect while leaving the server in the same state. Implemented correctly, the *GET*, *HEAD*, *PUT*, and *DELETE* method are *idempotent*, but not the *POST* method. All *safe* methods are *idempotent*.

All *safe* methods are also *idempotent*, but not all *idempotent* methods are *safe*. For example, *PUT* and *DELETE* are both *idempotent* but unsafe.

- Description of common idempotent methods: GET, HEAD, PUT, DELETE, OPTIONS, TRACE
- Description of common non-idempotent methods: POST, PATCH, CONNECT

# Additional Resources

https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#HTTP_Messages

https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design