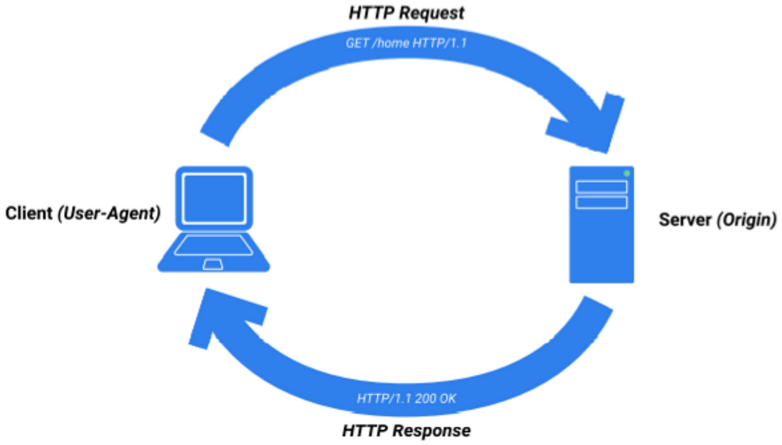
D1

Monday, September 7, 2020 2:42 PM

HTTP defines the process of exchanging hypertext between systems. Specifically, HTTP works between *clients*and *servers*. A *client*(sometimes called the *user agent*) is the data consumer. This is usually your web browser. A *server*(sometimes referred to as the *origin*) is the data provider, often where an application is running. In a typical HTTP exchange, the client sends a *request*to the server for a particular *resource*: a webpage, image, or application data. The server provides a *response*containing either the resource that the client requested or an explanation of why it can't provide the resource.

Here's a high-level overview of the exchange:



Stateless transfer

HTTP is considered a *stateless*protocol, meaning it doesn't store any information. Each request you send across an HTTP connection should contain

To help us with this, HTTP supports *cookies*, bits of data that a client sends in with their request. The server can examine this data and look up a *session*for your account, or it can act on the info in the cookie directly. Note that neither the cookie nor the session are part of HTTP. They're just workarounds we've created due to the protocol's stateless nature.

Intermediaries

The Web is a big place, and it's unlikely that your request will go directly to its destination! Instead, it will pass through a series of *intermediaries*: other servers or devices that pass your request along. These intermediaries come in three types:

- *proxies*, which may modify your request so it appears to come from a different source,
- *gateways*, which pretend to be the resource server you requested,
- and *tunnels*, which simply pass your request along.

Notice that these are interchangeable depending on the flow of data. When the response is sent back, "Their Router" is acting as a proxy and "Your Router" is acting as a gateway! This is an important part of HTTP: a single server may act

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

```
GET / HTTP/1.1
Host: appacademy.io
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_5) AppleWebKit/537.36 (K
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/ap
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
```

Let's break it down!

Request-line & HTTP verbs

The first line of an HTTP request is called the **request-line**, and it sets the stage for everything to come. It's made up of three parts, separated by spaces:

- the *method*, indicated by an *HTTP verb*,
- the *URI (Uniform Resource Indicator)*that identifies what we've requested,
- and the *HTTP version*we expect to use (usually `HTTP/1.1`or `HTTP/2`).

In our `appacademy.io`example, we can see that our version matches the most common HTTP version (`1.1`) and that our URI is `/`, or the *root*resource of our target. That first word, `GET` is the HTTP verb we're using for this request.

HTTP verbs are a simple way of declaring our intention to the server. We do the same thing with English verbs when asking for help: "Can you **get** me

- `GET` is used for direct requests. A `GET` request is generally how websites are retrieved, and they only require that the server return a resource. These types of requests will never have a body. Any data you need to send in a `GET` request must be shared via the URI.
- `POST` is typically used for creating new resources on the server. Most of the time, when you submit a form a `POST` request is generated. These types of requests can have a *body* containing any data the server might need to complete your request, like your username & password or the contents of your shopping cart.
- `PUT` requests are used to update a resource on the server. These will contain the whole resource you'd like to update. For example: when updating your name on a website, a `PUT` request will be generated containing not just your new name but also your user ID, email, etc.
- `PATCH` requests are very similar to `PUT` requests, but do not require the whole resource to perform the update. Keeping with our example of updating your name: a `PATCH` request would only require your new name, not the rest of your account details, to succeed.
- `DELETE` requests destroy resources on the server. These might be saved database records, like removing a product that's sold out, or more ephemeral resources, like logging a user out of their current session.

Headers

The *request-line* sets the table, but it's the headers that describe the menu! *Headers* are key/value pairs that come after the *request-line*. They each appear on separate lines and define metadata needed to process the request. Here are some common request headers you'll see:

- `Host`: The root path for our URI. This is typically the *domain* we'd like to request our resource from. As you can see above, our `Host` header for `appacademy.io` is, appropriately, `appacademy.io`.!
- `User-Agent`: This header displays information about which browser the request originated from. It's generally formatted as `name/version`. You can see in the `User-Agent` header above that we're using `Chrome/76.0`

  Our `User-Agent` has much more content, including references to Mozilla, makers of the popular Firefox browser, and Safari, Apple's default browser of choice. What gives?

  There is some interesting history behind those additional references, and you can use www.useragentstring.com for additional details about your current browser's `user-agent`.

- `Referer`: This defines the URL you're coming from. There's none in our example since we navigated directly to the App Academy website, but if we click any link on the page, the resulting HTTP request will have `Referer: https://appacademy.io/` in its headers. Also, you're not reading it wrong - this header is misspelled! It should be "referrer", but it was written incorrectly in the original specification and the typo stuck. Let this be a lesson: your poorly-written code might still be around in 20 years, too!
- `Accept`: "Accept-" headers indicate what the client can receive. When we go to most websites, our `Accept` header will be long to ensure we get all the various types of content that site might include. However, we can modify this header in our requests to only get back certain types of data. One common use is setting `Accept: application/json` to get a response in JSON format instead of HTML. You may see variations of this header like `Accept-Language` for internationalized websites or `Accept-Encoding` for sites that support alternative compression formats.
- `Content-*`: Content headers define details about the body of the request. The most common content header is `Content-Type`, which lets the server know what format we're sending our body data as. This might be `application/json` from a JavaScript app or `application/x-www-form-urlencoded` for info submitted from a web form. Content headers will only show up on requests that support content in the body, so `GET` requests should never have this!

Body

When we need to send data that doesn't fit in a header & is too complex for the URI, we can place it in the *body* of our HTTP request. The body comes right after the headers and can be formatted a few different ways.

The most common way form data is formatted is *URL encoding*. This is the default for data from web forms and looks a little like this:

```
name=claire&age=29&iceCream=vanilla
```

appropriate `Content-Type` header so the server knows how to interpret your body.

Structure of a Response

Responses are formatted similarly to requests: we'll have a *status-line*(instead of a request-line), headers that provide helpful metadata about the response,

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: close
X-Frame-Options: SAMEORIGIN
X-Xss-Protection: 1; mode=block
X-Content-Type-Options: nosniff
Cache-Control: max-age=0, private, must-revalidate
Set-Cookie: _rails-class-site_session=BAh7CEkiD3Nlc3Npb25faWQGOgZFVEkiJTM5NWM5YTV
X-Request-Id: cf5f30dd-99d0-46d7-86d7-6fe57753b20d
X-Runtime: 0.006894
Strict-Transport-Security: max-age=31536000
Vary: Origin
Via: 1.1 vegur
Expect-CT: max-age=604800, report-uri="https://report-uri.cloudflare.com/cdn-cgi/
Server: cloudflare
CF-RAY: 51d641d1ca7d2d45-TXL

<!DOCTYPE html>
<html>
...
...
</html>
```

Here's the status line from our `appacademy.io` response:

```
HTTP/1.1 200 OK
```

We open with the HTTP version the server is responding with. `1.1` is still the most commonly used, though you may occasionally see `2` or even `1.0`. We follow this with a `Status-Code` and `Reason-Phrase`. These give us a quick way of understanding if our request was successful or not.

HTTP status codes are a numeric way of representing a server's response. Each code is a three-digit number accompanied by a short description. They're grouped by the first digit (so, for example, all "Informational" codes begin with a `1`: `100` - `199`).

Let's take a look at the most common codes in each group.

Status codes 100 - 199: Informational

Informational codes let the client know that a request was received, and provide extra info from the server. There are very few informational codes defined by the HTTP specification and you're unlikely to see them, but it's good to know that they exist!

Status codes 200 - 299: Successful

Successful response codes indicate that the request has succeeded and the server is handling it. Here are a couple common examples:

- **200 OK**: Request received and fulfilled. These usually come with a `body` that contains the resource you requested.

- **201 Created**: Your request was received and a new record was created as a result. You'll often see this response to `POST` requests.

Status codes 300 - 399: Redirection

These responses let the client know that there has been a change. There are a few different ways for a server to note a redirect, but the two most common are also the most important:

- **301 Moved Permanently**: The resource you requested is in a totally new location. This might be used if a webpage has changed domains, or if resources were reorganized on the server. Most clients will automatically process this redirect and send you to the new location, so you may not notice this response at all.

- **302 Found**: Similarly, to *301 Moved Permanently*, this indicates that a resource has moved. However, this code is used to indicate a temporary move. It's not often that you see temporary moves online, but this code may be used to indicate a permanent move where the old domain should still be valid too. Clients will usually follow this redirect automatically as well, but you shouldn't necessarily update your links until the server returns a `301`.

301 Moved Permanentlyand 302 Foundoften get confused. When might we want to use a 302 Found `? The most common use case today is for the transition from HTTP to HTTPS. _HTTPSis secure HTTP messaging, where requests & responses are encryptedso they can't be read by prying eyes while en route to their destinations.

Status codes 400 - 499: Client Error

The status codes from 400 to 499, inclusive, indicate that there is a problem with the client's request. Maybe there was a typo, or maybe the resource we requested is no longer available. You'll see lots of these as you're learning to format HTTP requests. Here are the most common ones:

- **400 Bad Request**: Whoops! The server received your request, but couldn't understand it. You might see a *400 Bad Request*in response to a typo or accidentally truncated request. We often refer to these as *malformed*requests.
- **401 Unauthorized**: The resource you requested may exist, but you're not allowed to see it without authentication. These type of responses might mean one of two things: either you didn't log in yet, or you tried to log in but your credentials aren't being accepted.
- **403 Forbidden**: The resource you requested may exist, but you're not allowed to see it *at all*. This response code means this resource isn't accessible to you, even if you're logged in. You just don't have the correct permission to see it.
- **404 Not Found**: The resource you requested doesn't exist. You may see this response if you have a typo in your request (for example: going to`appaccccademy.io`), or if you're looking for something that has been removed.

Status codes 500 - 599: Server Error

This range of response codes are the Web's way of saying "It's not you, it's me." These indicate that your request was formatted correctly, but that the server couldn't do what you asked due to an internal problem.

There are two common codes in this range you'll see while getting started:

- **500 Internal Server Error**: Your request was received, and the server tried to process it, but something went awry! As you're learning to write your own servers, you'll often see a *500 Internal Server Error*as your code fails unexpectedly.
- **504 Gateway Timeout**: Your request was received but the server didn't respond in a reasonable amount of time. Timeout errors can be tricky: your first instinct may be that your own connection is bad, but this code means the problem is likely on the server's side. You'll often see these when a server is no longer reachable (maybe due to an unexpected outage or power failure).

Here are a few common response headers you'll see:

- **Location**: Used by the client for redirection responses. This contains the URL the client should redirect to.
- **Content-Type**: Lets the client know what format the body is in. Your client will display different types of response content in different ways, to setting the *Content-Type*is important! Notice that this header can be present on responses **and**requests. It's a generic header for any HTTP interaction involving content.
- **Expires**: When the response should be considered *stale*, or no longer valid. The *Expires*header lets your client *cache*responses (that is: save them locally to prevent having to repeatedly re-download them). The client may ignore requests to that same resource until after the date set in the*Expires*header.
- **Content-Disposition**: This header lets the client know how to display the response, and is specifically devoted to whether the response should be visible to the client or delivered as a download. Think about your own experience online: sometimes you click a button and get an immediate download, while in other cases you click a button and get to "preview" the content before you download it. This is controlled by the *Content-Disposition*header.
- **Set-Cookie**: This header sends data back to the client to set on the*cookie*, a set of key/value pairs associated with the server's domain. Remember how HTTP is *stateless*? Cookies are one way to get around that!*Set-Cookie*may send back information like a unique ID for the user you've logged in as or details about other resources you've requested on this domain.

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

Body

Assuming a successful request, the *body*of the response contains the resource you've requested. For a website, this means the HTML of the page you're accessing.

The format of the body is dictated by the *Content-Type*header. This is an important detail! If you accidentally configure your server to send "Content-Type: application/json" along with a body containing HTML, your HTML won't be rendered properly and your users will see plain text instead of beautifully-rendered elements. In the same way, API responses should be clearly marked so that other applications know how to manage them.

We can see in our `appacademy.io`response above that the body begins with `!<DOCTYPE html>`and ends with `</html>`. If you inspect the source of the page in your browser, you'll see that this is exactly what's being rendered. Headers may change **how** the browser handles the body, but they won't **modify**the body's content.