Post 02: Client Server Basics

Abhinav Patel

11/27/2017

Introduction

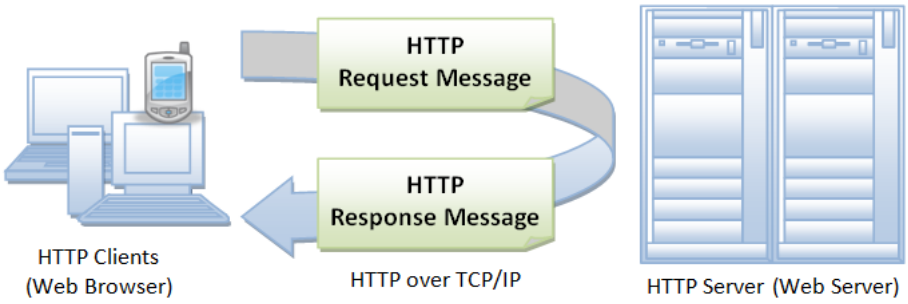
This post was inspired by the lesson on Shiny apps. It immediately became clear that all Shiny apps run on an instance of a server, and I wanted to see how exactly this model with a client, a server and its web traffic functions. This model is the foundation for the entirety of the internet, and this post will break down the basics of what a web server consists of, what an HTTP client is, and how data is communicated between a server and a client. Each computer in the client-server architecture model is either a client or a server. This differs from another common system architecture model, known as peer-to-peer (P2P), because in client-server, not all machines have equal responsibility.

The Client

The client is the simplest part of the client-server model. It is simply a machine that depends on another machine in order to perform some operations. For example, the most common instance of the client-server model is when a user connects to a website (let's say Amazon) via a web browser. In this case, the web browser running on the user's computer is the client. It wants to perform some operations (like display a list of items to buy), but it cannot do that without knowing which items are offered, which items fit that specific user's interests, which items are in stock, etc. The most common action undertaken by a client after recieved a response from a server is loading and displaying a web page.

HTTP

But how does a client tell the server what it needs, and how does the server give it this data? In order words, how does data actually flow in the request-response system? The answer is HTTP. HTTP is the foundation of the internet, and stands for HyperText Transfer Protocol. Nowadays, HTTP has evolved into HTTPS, which is a more secure version of HTTP, but the basics and uses of each are nearly identical. HTTP messages have three main parts: the message header, the message body, and the message type. The header for both requests and responses are usually fairly similar, and include information such as authentication (to make sure the request is going to and coming from valid sources), and the length of the message.



The Request

The request contains two main parts besides the header: the request type and the request body. There are fixed types of HTTP requests: *GET, POST, HEAD, PUT, DELETE, OPTIONS, CONNECT, PATCH*. Of these types, by far the most commonly used are *GET* and *POST*. The request type provides information to the server pertaining to how the client wants the request to be processed. The message body of an HTTP request is also called the request parameters, and there are two parameter types: URL parameters and JSON parameters.

A URL parameter is a pair that is visible in the URL of some loaded web page. In any URL with parameters, the parameters are the names and values present after the **?**. Multiple parameters are stated sequentially, each seperated by **&**. An example is `https://www.example.com/folder/file.php?name1=value1&name2=value2`.

The JSON parameters are not visible in the URL parameter, and are often encrypted. It contains key value pairs, just like a URL parameter, but are more secure and much more often used. Also, much more data can be sent through JSON than through a URL. The following UNIX command sends a *PATCH* request to a test server set up by PayPal. It contains no URL parameters, and JSON body is seen after the `-d`.

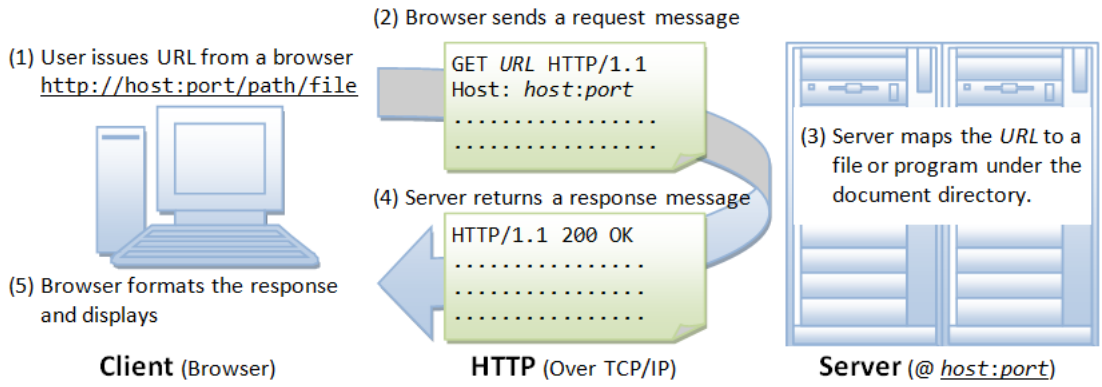
```
curl -v -X PATCH https://api.sandbox.paypal.com/v1/customer/partners/merchant-accounts/F9E99K66P3G77 \
-H "Content-Type:application/json" \
-H "Authorization: Bearer Access-Token" \
-d '[
  {
  "op": "add",
  "path": "/financial_info/bank_accounts",
  "value": [
    {
    "transfer_type": "NORMAL",
    "account_number": "11111113",
    "account_type": "CHECKING",
    "currency_code": "AUD",
    "identifiers": [
      {
      "type": "ROUTING_NUMBER_1",
      "value": "645"
      },
      {
      "type": "ROUTING_NUMBER_2",
      "value": "000"
      }
    ],
    "bank_name": "Bank of Australia",
    "branch_location": {
      "city": "Sydney",
      "country_code": "AU"
    }
    }
  ]
  }
]'
```

The Response

Once a server recieved a request, it performs some computations to process it, and returns an HTTP response to the sender. The response body is very similar to a request body in terms of containing relevant JSON for the response reciever to perform computations. The response type is also called the response code, and just like the request, takes on one from a set of predefined values.

These codes are seperated into five tiers: the 100s, the 200s, the 300s, the 400s, and the 500s. The 100s contain response codes pertaining to how the request was received and understood. The 200s state the action requested by the client was received, understood, accepted, and processed successfully. The 300s pertain to redirection, and that the client must take additional steps to complete the request. The 400s indicate that an error caused by the client occurred (you most likely have seen `404`, which is returned a page a client is requesting does not exist). The 500s indicate any type server error that have occurred.

Here is a more complete graphic describing the flow.



The Server

We are now at the final part of discussing the client-server architecture. We have discussed what defines a client, we have outlined how clients and servers send data to one another and how they communicate, but we have not discussed anything about what a server is or how a server works. By definition, a server is a machine that contains resources requested by a client. Each server is passive as is constantly listening and waiting for requests to handle on its own IP address. For example, the IP address for *google.com* is `172.217.5.110` and *amazon.com* is running on `205.251.242.103`. Once a server recieves a request from a client, it is responsible for extracting all relevant data from that request, processing that data, and sending a response. Until a client recieves a response from the server, it will continually send the same request (this is actually the main concept behind *TCP*). A single server must often be able to efficiently handle requests of many different clients at the same time, a characteristic that is known as *scalability*.

Sources

https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview
https://www.webopedia.com/TERM/C/client_server_architecture.html
https://www.w3schools.com/tags/ref_httpmethods.asp
https://msdn.microsoft.com/en-us/library/system.web.httprequest(v=vs.110).aspx

https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html
https://www.npmjs.com/package/request