TASK

Network Protocols and System Architecture

Visit our website

Introduction

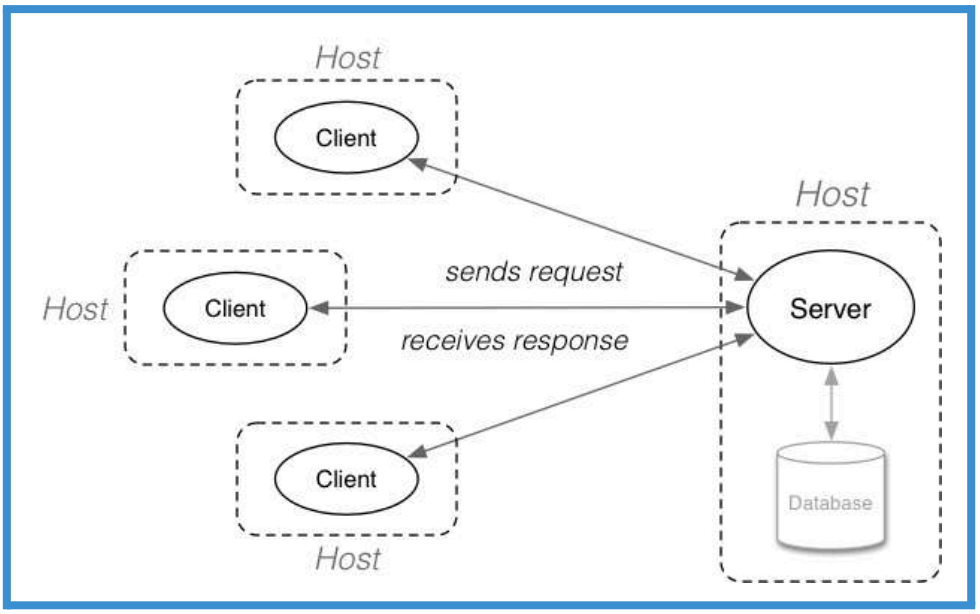
Every time you browse the web, stream a video, or send a message, there's an intricate system behind it making everything work. At its core is the client-server model, where devices (clients) communicate with servers to request and exchange information. This interaction relies on network protocols like HTTP, which set the rules for how data travels across the web. Understanding these systems gives insight into how the Internet connects us, enabling everything from loading a webpage to watching your favourite series.

WHAT IS CLIENT-SERVER ARCHITECTURE?

Client-server architecture, also known as a client-server model, is a network architecture that breaks down tasks and workloads between clients and servers that reside on the same system, or are linked by a computer network such as an intranet or the Internet.

Client-server architecture typically features multiple workstations, PCs, or other devices belonging to users, connected to a central server via an Internet connection, or another network connection. The client sends a request for data, and the server accepts and processes the request, sending the requested data back to the user who needs it on the client's side.

For example, when you open a website in your browser, your browser is the **client**. It sends an HTTP request to the **server** where the website is hosted. The server processes the request and sends back the necessary HTML, CSS, and JavaScript files, allowing your browser to render the page. For the purposes of this bootcamp, we will be focusing on how this model relates to web development in particular.



Client–server model

The above illustration shows how clients interact with a server and how that server interacts with the database so that data is processed by a server and sent back to the clients.

It is not entirely accurate to say that a server is a computer that clients make requests to. A computer must have appropriate server software running on it in order to make it a server. Examples of server software are **Apache**, **Tomcat**, and **NGINX**. In this course, you will learn to use Node.js and Express.js for setting up a web server.

A client also does not just refer to a device that makes the request to a server, but as with a server, a client needs the appropriate software to make requests. The most common client that you will have come across in your everyday life is your web browser! However, there are other clients as well, such as your Netflix application when you are streaming videos. Here, the application is the client and there is a server that is serving the video data to the client.

Another kind of client is a Secure Shell (SSH) client, which connects to the shell of a remote computer using the SSH protocol. The connection is encrypted, so only the

client and server computers know what data is being transmitted. Here, the client is the computer you connect from, and the server is the computer you connect to.

A client can therefore be any device with the software necessary to communicate with the server.

WHAT IS HTTP?

HTTP is an underlying protocol used by the World Wide Web. A protocol is basically a set of rules that are decided on and adopted so that there is consistency in how to perform particular tasks. HTTP defines how messages are formed and transmitted between clients and servers, and what actions web servers and clients should take in response to various commands.

A simple example of how HTTP is implemented is when a URL is entered into the browser, and the browser sends an HTTP command to the web server, directing it to search for and transmit the requested web page. The response would, in this case, be an HTML file that the browser can interpret and display to the user.

The HTTP protocol is a **stateless** one. This means that every HTTP request the server receives is independent and does not relate to requests that came prior to it. For instance, imagine the following scenario: A request is made for the first 10 user records from a database, and then another request is made for the next 10 records. These requests would be unrelated to each other.

With a **stateful** protocol, the server remembers each client position inside the result set, and therefore the requests will be similar to the following:

- `Give me the first 10 user records.`
- `Give me the next 10 records.`

With a stateless protocol, however, the server does not hold the state of its client, and therefore the client's position in the result-set needs to be sent as part of the requests, like this:

- `Give me user records from user one to user 10.`
- `Give me user records from user 11 to user 20.`

Each request must independently specify what data it needs, because the server does not ‘remember’ prior interactions with the client.

HTTPS, a secure version of HTTP, adds authentication and encryption protocols. It ensures that data is transmitted securely over an otherwise insecure network using a method called Secure Socket Layer (SSL).

To test HTTP or HTTPS requests, developers often use tools like `curl`. `curl` is a command-line utility that allows you to send requests to a web server and view the server's responses. For example, here's how you can use `curl` to make a basic HTTP GET request:

```
# Using curl to make a basic HTTP request

curl -X GET http://example.com
```

```
# Using powershell to make a basic HTTP request

Invoke-WebRequest -Uri http://example.com -Method Get
```

It's important to note that PowerShell will default to "`GET`" if no "`-Method`" is specified. Therefore, the following will also work:

```
# Using powershell to make a basic HTTP request

Invoke-WebRequest -Uri http://example.com
```

The commands above send a GET request to a web server, which is a common method for requesting a webpage. The server's response will include the HTML content, which your browser or curl will display.

By testing requests in this way, you can see both the headers and the body of the server's response directly in the terminal. This can be useful for debugging or understanding how the server behaves when handling HTTP requests.

THE REQUEST/RESPONSE CYCLE

As we start to build out web applications, it's essential to be able to visualise the way information flows through the system, typically called the request/response cycle.

First, a user gives a client a URL, and the client builds a request for information (or resources) to be generated by a server. The request is sent from the client to the server. When the server receives that request, it uses the information included in the request to build a response that contains the requested information. Once created, that response is sent back to the client in the requested format to be rendered to the user.

It's a web developer's job to build out and maintain servers that can successfully build responses based on standardised requests that will be received from clients. But what does a standard request look like? We need to know that before we can build servers that will respond successfully.

THE HTTP MESSAGE

HTTP messages are used for the requests and responses. HTTP messages are composed of textual information encoded in ASCII, and span multiple lines.

Web developers, or webmasters, rarely craft these textual HTTP messages themselves (although this can be done): software, a web browser, proxy, or web server performs this action.

Compulsory reading

The Mozilla Developer Network documentation breaks down the anatomy of an **HTTP message** very well. **This is essential reading for this task.**

Please read the resource above before moving on.

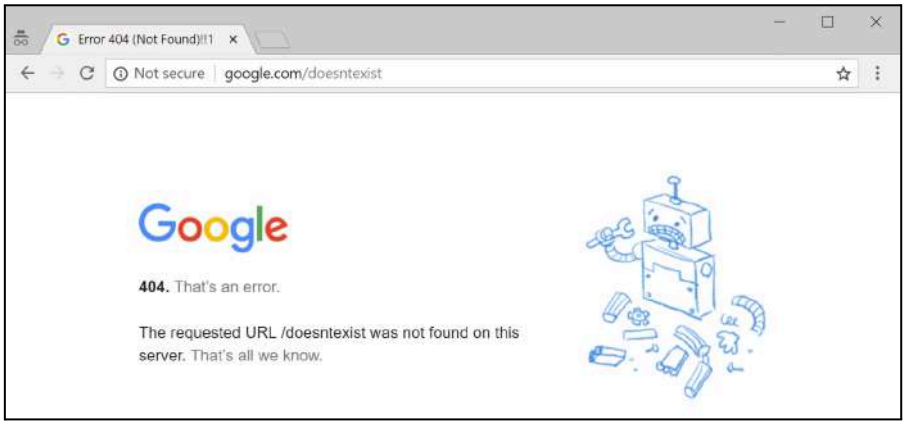
STATUS CODES

HTTP status codes are like short notes from a server that get tacked onto a web page. They're not actually part of the site's content. Instead, they're messages from the server letting you know how things went when it received the request to view a certain page.

These kinds of messages are returned every time your browser interacts with a server, even if you don't see them. If you're a website owner or developer, understanding HTTP status codes is critical. When they do show up, HTTP status codes are an invaluable tool for diagnosing and fixing website configuration errors.

HTTP status codes are delivered to your browser in the HTTP header. They are returned with an HTTP response message every time a request is made.

It's usually only when something goes wrong that you might see one displayed in your browser. A common status code that you may have discovered in your daily dealings in a browser is the **404 page not found** error:



HTTP status codes fall into five classes:

1. **100s**: Informational codes indicating that the request initiated by the browser is continuing.
2. **200s**: Success codes returned when the browser request was received, understood, and processed by the server.
3. **300s**: Redirection codes returned when a new resource has been substituted for the requested resource.

4. **400s**: Client error codes indicating that there was a problem with the request.
5. **500s**: Server error codes indicating that the request was accepted, but that an error on the server prevented the fulfilment of the request.

This command shows the HTTP headers returned by the server, including the status code. Notice that there are two separate sets of commands below (depending on your OS: PowerShell or mac and Linux). For instance, if the server is running fine, the status code would be 200 OK:

```
# Using curl to check the status code of a webpage

curl -I http://example.com
```

```
# Using powershell to check the status code of a webpage

$response = Invoke-WebRequest -Uri http://example.com

$response.StatusCode

$response.StatusCode

$response.Headers

$response.Content
```

Below, you can find an example of what the output might look like:

```
HTTP/1.1 200 OK

Date: Wed, 29 Nov 2024 12:34:56 GMT

Server: Apache/2.4.41 (Ubuntu)

Content-Type: text/html; charset=UTF-8

Content-Length: 1270

<!DOCTYPE html>

<html>

  <head>

      <title>Example Domain</title>

  </head>

  <body>

      <h1>This is an example page</h1>

      <p>You are viewing the response body from the server!</p>

  </body>

</html>
```

Breakdown of the output:

1. Headers:
    - `HTTP/1.1 200 OK`: The status code confirming a successful request.
    - `Date`: When the server responded.
    - `Server`: The server software (e.g., Apache).
    - `Content-Type`: Specifies the type of content (HTML in this case).
    - `Content-Length`: The size of the body in bytes.
2. Body:
    - The actual HTML content returned by the server.

Extra resource

Familiarise yourself with the types of **HTTP response status codes**.

MIME TYPES

A **multipurpose internet mail extension**, or MIME type, is an Internet standard that describes the contents of Internet files based on their natures and formats. Nowadays, they are more frequently called media types, because they are no longer only used in emails, but also in web applications.

This cataloguing helps the browser open the file with the appropriate extension or plugin, and also helps a server identify how to process a specific file if it is sent from client to server. Although the term includes the word "mail" for electronic mail, it's used for web pages, too. MIME types contain two parts: a type and a subtype. The type describes the categorisation of MIME types linked to each other. In contrast, a subtype is unique to a specific file type that is part of the larger type category.

The syntax of the structure is: `type/subtype`

A MIME type is insensitive to text case, but traditionally is written in lower case. Discrete types indicate the category of the document; this can be one of the following:

Type	Typical subtypes	Description
text	• `text/plain` • `text/html` • `text/css` • `text/javascript`	Represents any document that contains text and is theoretically human-readable.
image	• `image/gif` • `image/png` • `image/jpeg` • `image/bmp` • `image/webp`	Represents any kind of image. Videos are not included, though animated images (like an animated GIF) are described with an image type.
audio	• `audio/midi` • `audio/mpeg` • `audio/webm` • `audio/ogg` • `audio/wav`	Represents any kind of audio file.
video	• `video/webm` • `video/ogg`	Represents any kind of video file.
application	• `application/octet-stream` • `application/pkcs12` • `application/vnd.mspowerpoint` • `application/xhtml+xml` • `application/xml` • `application/pdf`	Represents any kind of binary data.

MIME types (Mozilla Developer Network, 2016)

Here is a **list of all MIME types**.

Practical task

Understand HTTP communication by performing hands-on tasks using `curl` and documenting your findings.

1. **Basic request analysis**

   Run the following command for `curl`:

   ```
   curl -v https://httpbin.org/get
   ```

   Run the following command for PowerShell:

   ```
   Invoke-WebRequest -Uri https://httpbin.org/get -Verbose
   ```

   What to do:

   - Observe the output in the terminal.
   - Identify and document:
     - Request headers (what the client sends).
     - Response headers (what the server sends back).
     - HTTP status code.
     - Connection workflow (how the request and response are exchanged).

2. **Method interaction**

   Run this command to test a POST request for `curl`:

   ```
   curl -X POST https://httpbin.org/post \
        -d "key=value" \
        -H "Content-Type: application/x-www-form-urlencoded"
   ```

Run this command to test a POST request for PowerShell:

```
Invoke-WebRequest -Uri "https://httpbin.org/post" `
  -Method Post `
  -ContentType "application/x-www-form-urlencoded" `
  -Body "key=value"
```

What to do:

- Compare how POST behaves differently from GET.
- Explain the purpose of the payload (`key=value`) and how it's transmitted.
- Document the response body and note how the server reflects your data.

3. Status code exploration

Run the following commands to explore specific HTTP status codes for `curl`:

```
curl -I https://httpbin.org/status/404
curl -I https://httpbin.org/status/403
```

Run the following commands to explore specific HTTP status codes for PowerShell:

```
Invoke-WebRequest -Uri https://httpbin.org/status/404 -Method Head
Invoke-WebRequest -Uri https://httpbin.org/status/403 -Method Head
```

What to do:

- Note the status codes and their meanings:
  - `404` (Not Found): Why might this occur?
  - `403` (Forbidden): What does this tell you about access restrictions?

- Analyse how these errors are communicated in the headers.

4. **Document commands and observations**

   Create a file named **http_investigation.txt** with the following:

   - Executed commands
     - List all the `curl` commands you ran.
   - Observations
     - For each section, include:
       - Results and outputs you observed.
       - Key takeaways or conclusions about how HTTP handles requests, methods, status codes, and errors.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.
