How the Web Works, HTTP Request/Response Cycle

What is the Internet?

The internet, which for most people is the web…how does that work?

The basis of all web interactions is someone asking for information, and receiving information. In order to ask for and receive any information, we need two players - the asker and the producer. In basic web interactions, the 'asker' is a **client** and the 'producer' is a **server**. Clients send **Requests** to Servers asking for some kind of information. Upon receiving a Request, Servers send **Responses** back to the Client.

The **Internet** is the network between devices that allows clients and servers to exchange this information. **HTTP** is a set of rules for how this exchange of information happens. Clients and Servers adhere to these rules to ensure that they understand each other's Requests and Responses.

In the web development world, a client is a web browser, not an individual person. The person using the browser is referred to as a **user**.

Penpal Analogy

Okay, that was a lot of information. Let's break all of this down with a metaphor:

Imagine you're writing to a penpal. The process would look something like this:

1. Write a letter
2. Specify your penpal's address
3. Drop the letter in your mailbox
4. The letter goes through the postal system and arrives at your penpal's mailbox

Your penpal then goes through a very similar set of steps:

1. Read your letter and write a response
2. Specify your address
3. Drop their letter in their mailbox
4. The letter goes through the postal system and arrives at your mailbox

In this analogy:

- You are the **Client**
- Your penpal is the **Server**
- Your letter is the **Request**
- Your penpal's letter is the **Response**

- The postal system, the thing responsible for ensuring your letters are delivered, is **The Internet**

HTTP is the language you write in so that your penpal can understand you. You may write in English because you know you both understand English. If you wrote a letter in Spanish and your penpal only spoke English, they might write you a letter back saying “Please write to me in English”.

Metaphor aside, let’s run through the protocol as executed by computers:

1. You open your browser, the Client, and type in a web address like `http://zenrays.com` and hit enter.
2. The browser takes this address and builds an **HTTP Request**. It addresses it to the Server located at `http://zenrays.com`.
3. The Request is handed off to your Internet Service Provider (ISP) (like CenturyLink or Comcast) and they send it through the Internet, mostly a series of wires and fiber optic cables, to the Server
4. The Server reads the Request. It knows how to read it because it is formatted as an **HTTP Request**.
5. The Server generates an **HTTP Response** to that Request.
6. The server hands the Response off to their ISP and it goes through the internet to arrive at your computer.
7. Your browser reads the Response. It knows how to read it because it is formatted as an **HTTP Response**.
8. Your browser displays the data on your machine.

That’s the HTTP Request/Response cycle. At its core, it is a bunch of formatting rules that Clients and Servers use to talk to each other. You can read more on the wikipedia page or the IETF specification.

The Request and Response Cycle

As we start to build out web applications, it is important to be able to visualize the way information flows through the system; typically called the Request/Response Cycle.

First a user gives a client a URL, the client builds a **request** for information (or resources) to be generated by a 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 built, that response is sent back to the client in the requested format, to be rendered to the user.

It is our job as web developers to build out and maintain servers that can successfully build responses based on standardized requests that will be received. But, what does a standard request look like? We need to know that before we can start building servers that will respond successfully.

The standard, or protocol we use is **HTTP**.

HTTP Requests and Responses

The HyperText Transfer Protocol gives us rules about how messages should be sent around the Internet. The system that initiates a connection sends a “request”, and the system the answers sends a “response”.

HTTP Request

When a “client” (like a web browser) retrieves information, it sends a payload of data to a server as a “request”. This request has many parts, but for now we are going to focus on the **verb** and **path**.

Verb and Path

Every request needs to be able to tell a server what information is requested and how that information is being requested. The what is the **path** (also know as a URI), indicating what resource this request is referencing.

Examples of a path:

- `/tasks`
- `/tasks/4`
- `/items/6/reviews`

The how is the **verb**, indicating what actions the server should take regarding the requested resource. While the path can vary greatly based on the application, the verbs follow common patterns. There are 5 common HTTP verbs:

- GET - retrieve some information to be READ by the client/user
- POST - CREATE a new resource with information contained in the request
- PUT - UPDATE an entire resource with information contained in the request
- PATCH - UPDATE a part of a resource with information contained in the request
- DELETE - DESTROY a resource, typically indicating that it is removed from the database

With these 5 verbs, we send requests that allow us to perform all CRUD functions (create, read, update, destroy) for resources in a database!

HTTP Status Code

The Server issues an HTTP Status Code in response to a request of the client made to the server. Status code is a 3-digit integer. The first digit of status code is used to specify one of five standard classes of responses. The last two digits of status code do not have any categorization role.

The status codes are divided into 5 parts, as follows:

S.N.	Code and Description
1	**1xx: Informational Response** It is used to show that the request was received, and the process is continuing.
2	**2xx: Successful** It is used to show that the request was successfully received, understood, and accepted.
3	**3xx: Redirection** It is used to show that further action needs to be taken to complete the request.
4	**4xx: Client Error** It is used to show that the request contains bad syntax or cannot be fulfilled.
5	**5xx: Server Error** It is used to show that the server is failed to fulfill an apparently valid request.

Node (or more formally Node.js) is an open-source, cross-platform runtime environment that allows developers to create all kinds of server-side tools and applications in JavaScript. The runtime is intended for use outside of a browser context (i.e. running directly on a computer or server OS). As such, the environment omits browser-specific JavaScript APIs and adds support for more traditional OS APIs including HTTP and file system libraries.

From a web server development perspective Node has a number of benefits:

- Great performance! Node was designed to optimize throughput and scalability in web applications and is a good solution for many common web-development problems (e.g. real-time web applications).
- Code is written in "plain old JavaScript", which means that less time is spent dealing with "context shift" between languages when you're writing both client-side and server-side code.
- JavaScript is a relatively new programming language and benefits from improvements in language design when compared to other traditional web-server languages (e.g. Python, PHP, etc.) Many other new and popular languages compile/convert into JavaScript so you can also use TypeScript, CoffeeScript, ClojureScript, Scala, LiveScript, etc.
- The node package manager (NPM) provides access to hundreds of thousands of reusable packages. It also has best-in-class dependency resolution and can also be used to automate most of the build toolchain.
- Node.js is portable. It is available on Microsoft Windows, macOS, Linux, Solaris, FreeBSD, OpenBSD, WebOS, and NonStop OS. Furthermore, it is well-supported by many web hosting providers, that often provide specific infrastructure and documentation for hosting Node sites.
- It has a very active third party ecosystem and developer community, with lots of people who are willing to help.

You can use Node.js to create a simple web server using the Node HTTP package.

Hello Node.js

The following example creates a web server that listens for any kind of HTTP request on the URL `http://127.0.0.1:8000/` — when a request is received, the script will respond with the string: "Hello World". If you have already installed node, you can follow these steps to try out the example:

1. Open Terminal (on Windows, open the command line utility)

2. Create the folder where you want to save the program, for example, `test-node` and then enter it by entering the following command into your terminal:

```
cd test-node
```

3. Using your favorite text editor, create a file called `hello.js` and paste the following code into it:

```
// Load HTTP module
const http = require("http");
const hostname = "127.0.0.1";
const port = 8000;
// Create HTTP server
const server = http.createServer(function(req, res) {
  // Set the response HTTP header with HTTP status and Content type
    res.writeHead(200, {'Content-Type': 'text/plain'});
  // Send the response body "Hello World"
  res.end('Hello World\n');
});
// Prints a log once the server starts listening
server.listen(port, hostname, function() {
  console.log(`Server running at http://${hostname}:${port}/`);
})
```

4. Save the file in the folder you created above.
5. Go back to the terminal and type the following command:

```
node hello.js
```

Finally, navigate to `http://localhost:8000` in your web browser; you should see the text "Hello World" in the upper left of an otherwise empty web page.