Front-End Web & Mobile

What happens when you type a URL into your browser?

by Kyle Lee | on 26 AUG 2021 | in Thought Leadership | Permalink | Comments | Share

This article was written by Jenna Pederson.

Every day you open up your browser and navigate to your favorite websites — whether it be social media, news, or e-commerce sites. You go to this page by typing in a url or clicking on a link to the page. Have you ever thought about what happens behind the scenes? How does the news get to you when you press enter after typing in the URL? How did the images on this post show up in your browser? How does your Twitter feed and the tweet data show up in your browser securely?

In this post, we'll look at what happens when you type a URL into your browser and press enter. End to end, the process involves the browser, your computer's operating system, your internet service provider, the server where you host the site, and services running on that server. It's important to understand where things can go wrong, where to look for performance issues, and ensure you're offering a secure experience for your users.

First, we'll take a look at the relationship between websites, servers, and IP addresses. Then, we'll go through the steps your browser takes to:

- look up the location of the server hosting the website
- make the connection to the server
- send a request to get the specific page
- handle the response from the server and
- how it renders the page so you, the viewer, can interact with the website

Websites, servers, IP addresses, oh my!

Websites are collections of files, often HTML, CSS, Javascript, and images, that tell your browser how to display the site, images, and data. They need to be accessible to anyone from anywhere at anytime, so hosting them on your computer at home isn't be scalable or reliable. A powerful external computer connected to the Internet, called a server, stores these files.

When you point your browser at a URL like https://jennapederson.dev/hello-world, your browser has to figure out which server on the Internet is hosting the site. It does this by looking up the domain, jennapederson.dev, to find the address.

Each device on the Internet — servers, cell phones, your smart refrigerator — all have a unique address called an IP address. An IP address contains four numbered parts:

203.0.113.0

But numbers like this are hard to remember! That's where domain names come in. jennapederson.dev is much easier to remember than 203.0.113.0, right? Imagine having to remember all the phone numbers of your

contacts without having the Contacts app on your phone. Your Contacts app lets you look up phone numbers by name.

We do the same on the Internet. The domain name system, or DNS, is like the Contacts app on our phone. DNS helps our browser (and us) find servers on the Internet. We can do a DNS lookup to find the IP address of the server based on the domain name, jennapederson.dev. You can read more about DNS here.

Now that you know about the different parts and how they relate to one another, let's look at each step of the process and how the browser communicates with the server when you type in a URL. Whether you typed in a URL or clicked on a linked URL from the current page, the process is the same.

The process

To show how all these steps fit together, I'm going to use an Amazon Lightsail instance and a Lightsail DNS zone. I'm using Lightsail because it's one of the simplest services to manage virtual private servers (VPS) and DNS in one place, but these concepts work for any VPS and DNS service.

1. You type https://jennapederson.dev/hello-world in your browser and press Enter

Let's break down the parts of this URL you typed in to get here.

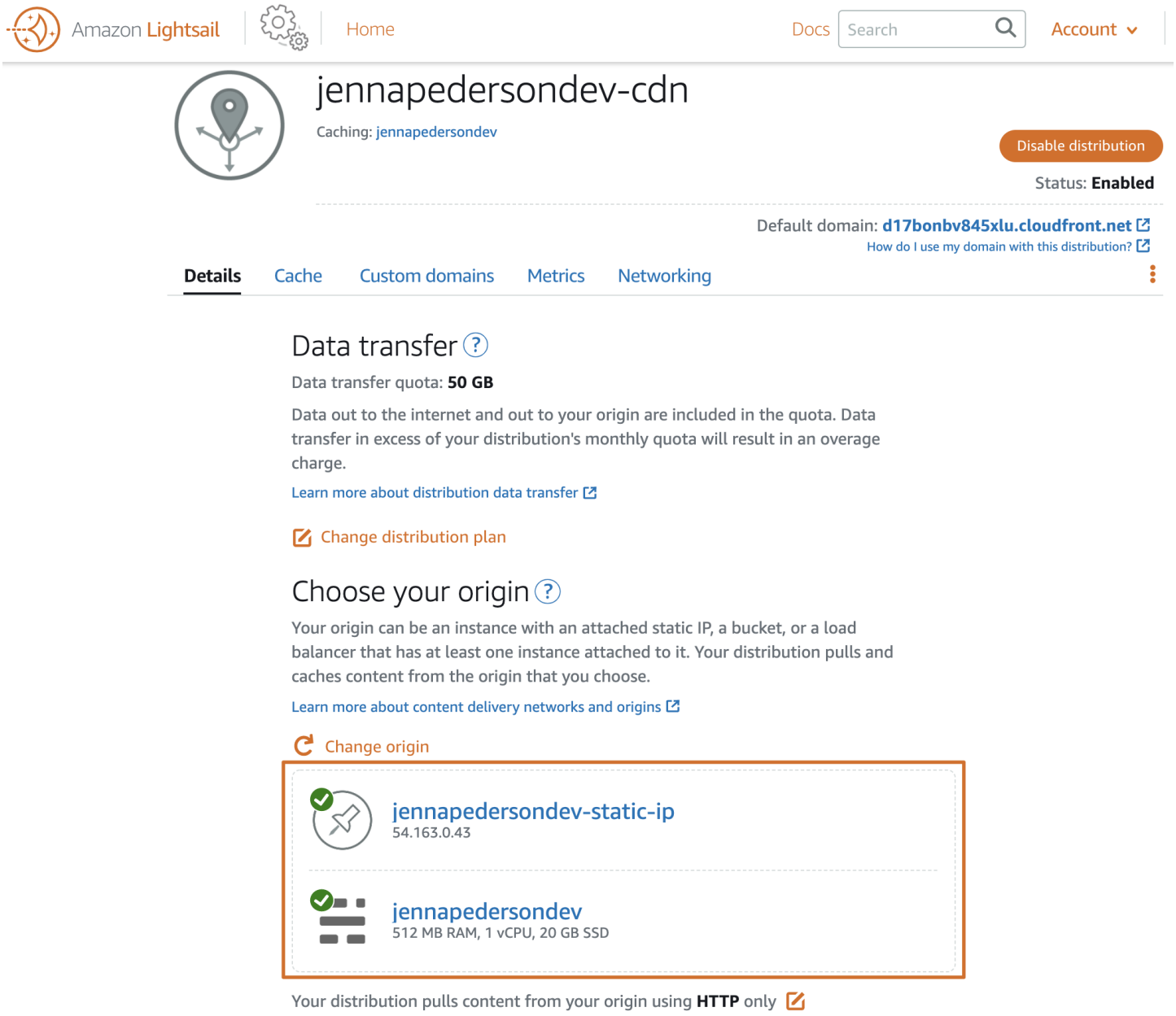
`https://jennapederson.dev/hello-world`

Scheme

`https://` is the scheme. HTTPS stands for Hypertext Transfer Protocol Secure. This scheme tells the browser to make a connection to the server using Transport Layer Security, or TLS. TLS is an encryption protocol to secure communications over the Internet. With HTTPS, the data exchanged between your browser and the server, like passwords or credit card info, is encrypted. You may have also seen `ftp://`, `mailto://`, or `file://`. These are other protocols that browsers know how to handle.

Domain

`jennapederson.dev` is the domain name of the site. It is the memorable address and points to a specific server's IP address. If you look at the Lightsail DNS zone below, you can see a DNS A record pointing to the Lightsail instance, `jennapedersondev-static-ip`, which represents the static IP address of the Lightsail instance.



Path

Sometimes there is an additional path to the resource in the URL. For example, for this URL, `https://jennapederson.dev/the-path-to/hello-world`, `the-path-to` is the path on the server to the requested resource, `hello-world`. You can think of this like the directory structure of files and other directories on your computer. It's a way to organize your resources, whether they are static HTML, CSS, Javascript, or image files, or dynamically generated content. Common examples of paths are `blog`, `posts`, `tags`, or `images`, each grouping different resources.

Resource

When you typed this URL into your browser, `hello-world` is the name of the resource on the website you want to view. Sometimes you'll see this with a file extension like `.html` which indicates this is a static file on the server with HTML content. Without an extension, like this URL, it usually indicates the server generated this content. For instance, a news site would show you customized, up to date, and local news content, which it can only do when it knows who you are or where the request came from.

2. Browser looks up IP address for the domain

After you've typed the URL into your browser and pressed enter, the browser needs to figure out which server on the Internet to connect to. To do that, it needs to look up the IP address of the server hosting the website using the domain you typed in. It does this using a DNS lookup. We'll go over the DNS lookup process at a high-level, but it is a complex topic beyond the scope of this post. You can read more about how DNS works here.

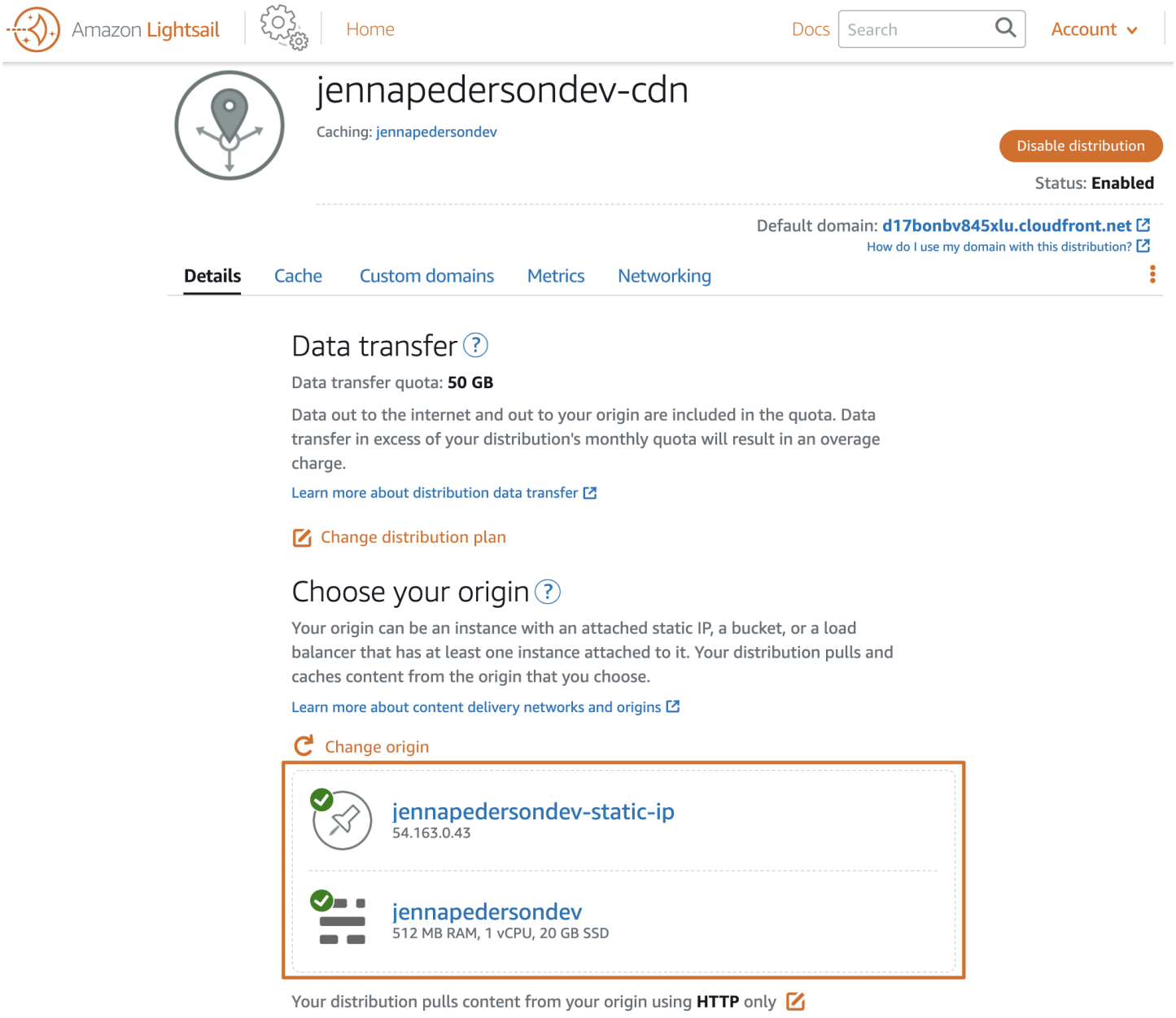
Because DNS is complex and has to be blazingly fast, DNS data is cached at different layers between your browser and at various places across the Internet. Your browser checks its own cache, the operating system cache, a local network cache at your router, and a DNS server cache on your corporate network or at your internet service provider (ISP). If the browser cannot find the IP address at any of those cache layers, the DNS server on your corporate network or at your ISP does a recursive DNS lookup. A recursive DNS lookup asks multiple DNS servers around the Internet, which in turn ask more DNS servers for the DNS record until it is found.

Once the browser gets the DNS record with the IP address, it's time for it to find the server on the Internet and establish a connection.

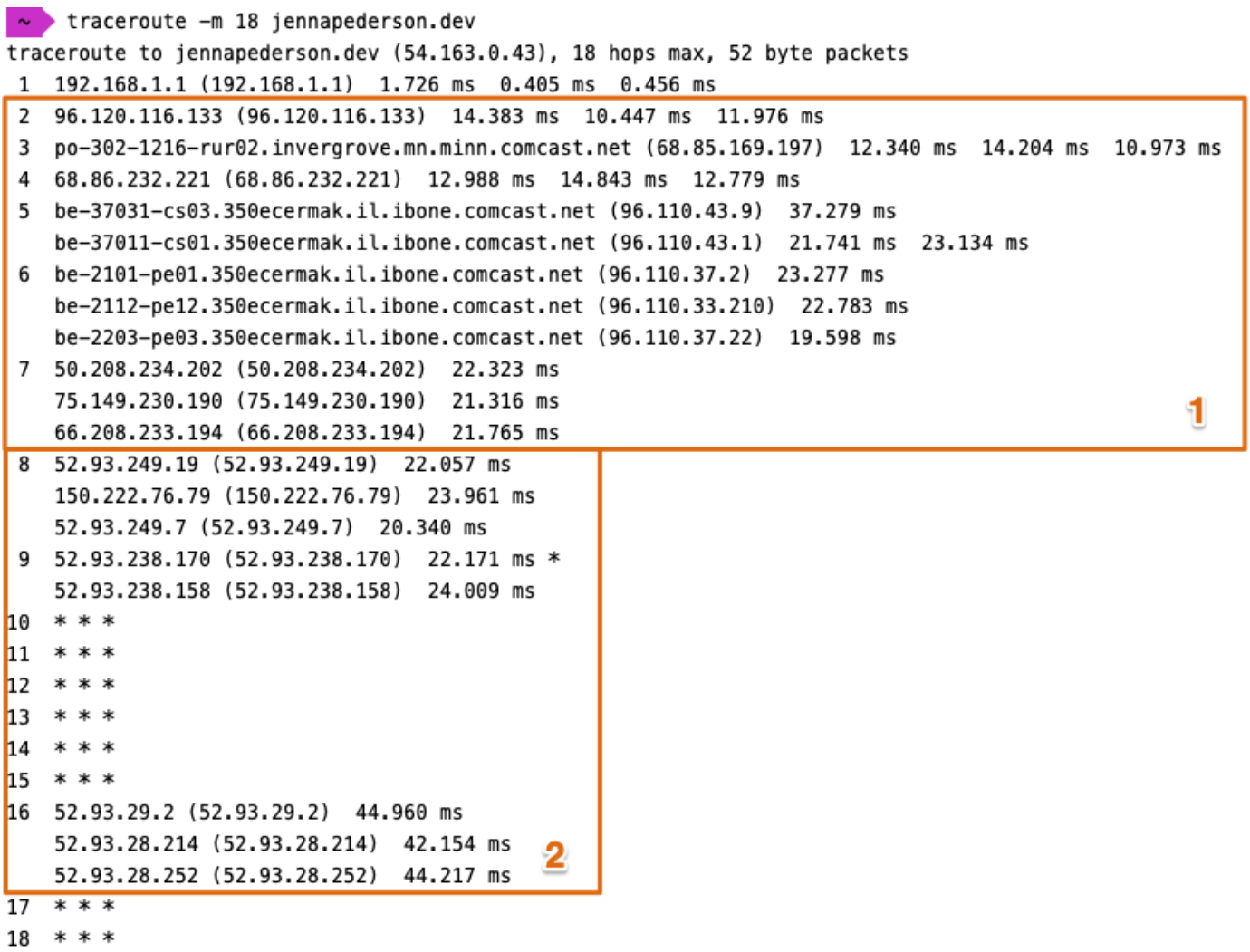
3. Browser initiates TCP connection with the server

Using the public Internet routing infrastructure, packets from a client browser request get routed through the router, the ISP, through an internet exchange to switch ISPs or networks, all using transmission control protocol, more commonly known as TCP, to find the server with the IP address to connect to. But this is a very roundabout way to get there and it's not efficient.

Instead, many sites use a content delivery network, or CDN, to cache static and dynamic content closer to the browser. In our example, I've set the Lightsail instance, `jennapedersondev`, as an origin for a CDN distribution.



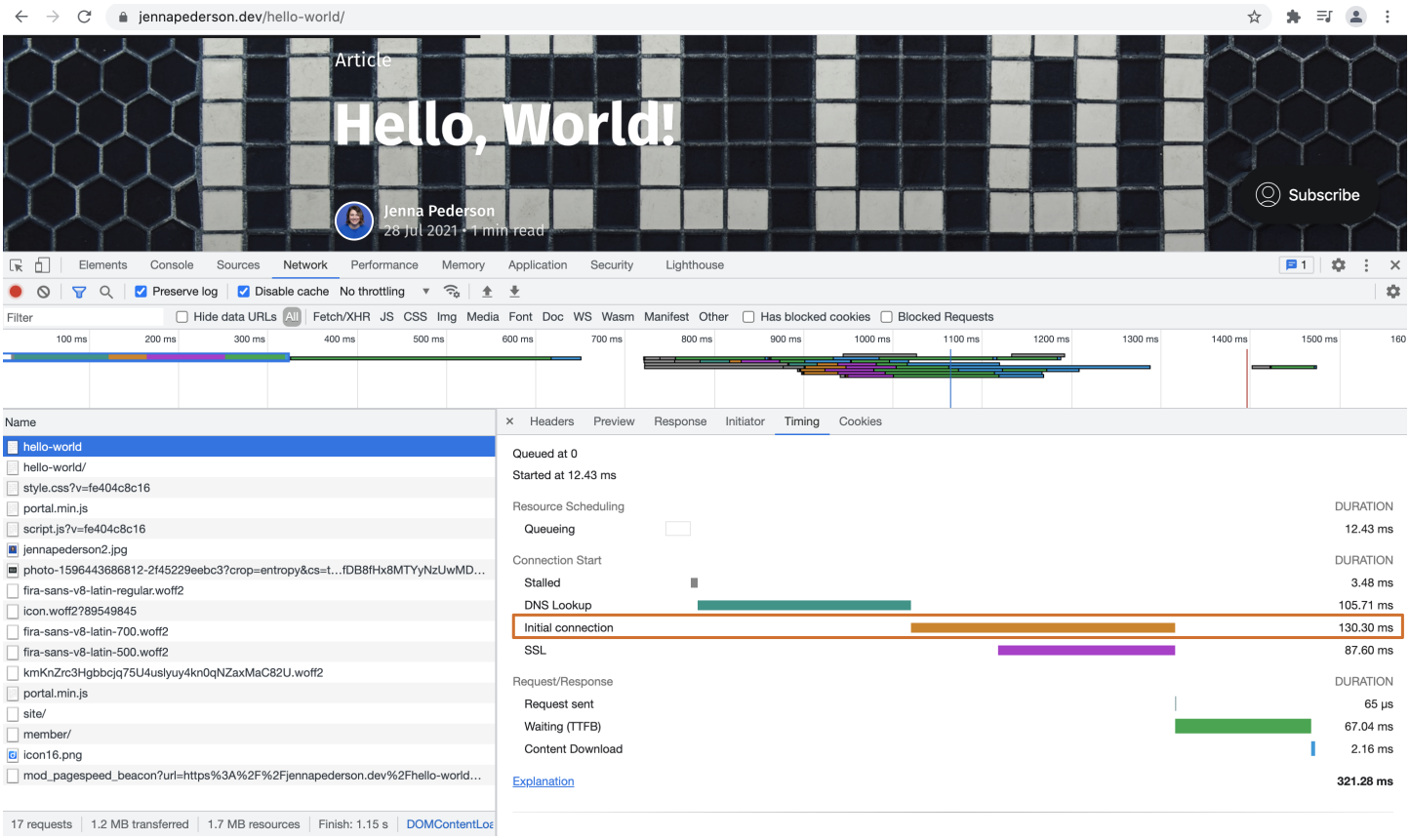
A CDN is a globally distributed network of caching servers that improve the performance of your site or app (the origin) by bringing the content closer to your users. The Lightsail CDN uses CloudFront, which is part of the AWS global network. Requests from the client browser get to take advantage of this private network that has ultra-low latency and high availability. To track the hops the request takes from my computer to jennapederson.dev, we can use `traceroute`. In the image below, the first hop (the first row) is to my router. Hops in box one are on my ISP's network and hops in box two are on the AWS global network.



Instead of relying on the public internet routing infrastructure and being subject to extra hops, redeliveries, and packet loss, the client browser request gets to take a ride on the AWS global network. The request is intelligently routed through the most performant location to deliver content to your browser.

Once the browser finds the server on the Internet, it establishes a TCP connection with the server and if HTTPS is being used, a TLS handshake takes place to secure the communication. TCP and and TLS are extremely important topics, but we'll cover them in another post.

In image the below, this connection (Initial connection) took 130.30ms.



Once the browser has established a connection with the server, the next step is to send the HTTP request to get the resource, or the page.

4. Browser sends the HTTP request to the server

Now that the browser has a connection to the server, it follows the rules of communication for the HTTP(s) protocol. It starts with the browser sending an HTTP request to the server to request the contents of the page. The HTTP request contains a request line, headers (or metadata about the request), and a body. The request line contains information that the server can use to determine what the client (in this case, your browser) wants to do.

The request line contains the following:

- a request method, which is one of GET, POST, PUT, PATCH, DELETE, or a handful of other HTTP verbs
- the path, pointing to the requested resources
- the HTTP version to communicate with

The request line for the URL request looks like this:

```
GET /hello-world HTTP/1.1
```

The request line tells the server that you want to GET resource at `/hello-world` and to communicate with `HTTP/1.1`.

Remember that HTTP verbs express the semantic intent of your request. You could also use the POST, PUT, or PATCH methods to send data to the server for storage, either to create new data or update existing data at the

request path. The DELETE method would delete the resource at the given path. However, it's important to know that servers don't have to support all verbs. A server could respond with a `200 OK` status to a DELETE method and not do anything with the resource.

The next part of the request is the request headers. Headers pass extra information along from the client that help route the request, indicate what type of client is making the request (the user agent), and can be used for supporting A/B testing, blue/green deployments, and canary deployments.

Headers are key-value pairs like this:

```
Host: jennapederson.dev
User-Agent: curl/7.64.1
Accept: */*
```

The last part of the request is the body. The body is (usually) empty for a GET request like ours. For a request that manipulates resources, like POST, PUT, or PATCH, the body will contain the data from the client to create or update.

The request body can be in different formats and the server understands the format based on a request header, `Content-Type`.

Here's an example of the full request for the URL, with the request line and headers (no body since this is a GET):

```
GET /hello-world/ HTTP/1.1
Host: jennapederson.dev
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="90", "Google Chrome";v="90"
sec-ch-ua-mobile: ?0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTM
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: <https://jennapederson.dev/>
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
dnt: 1
sec-gpc: 1
```

Once the server has received the request from the client, the server processes it and sends back a response.

5. Server processes request and sends back a response

The server takes the request and based on the info in the request line, headers, and body, decides how to process the request. For the request, `GET /hello-world/ HTTP/1.1`, the server gets the content at this path, constructs the response and sends it back to the client. The response contains the following:

- a status line, telling the client the status of the request
- response headers, telling the browser how to handle the response
- the requested resource at that path, either content like HTML, CSS, Javascript, or image files, or data

The status line contains both the HTTP version and a numeric and text representation of the status of the request. The response looks like this:

```
HTTP/1.1 200 OK
Date: Tue, 25 May 2021 19:40:59 GMT
Server: Apache
X-Frame-Options: SAMEORIGIN
X-Powered-By: Express
Cache-Control: max-age=0, no-cache
Content-Type: text/html; charset=utf-8
Vary: Accept-Encoding
X-Mod-Pagespeed: 1.13.35.2-0
Content-Encoding: br
Keep-Alive: timeout=1, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked

<!DOCTYPE html>
<html lang="en">
<head>
    THE REST OF THE HTML
```

The browser considers a status code in the 200s to be successful. The response was 200, so the browser knows to render the response.
Resources can be static files, either text (i.e `index.html`) or non-text data (i.e. `logo.img`). Browsers aren't always requesting static files, though. Often, these are dynamic resources generated at the time of the request and there is no file associated with the resource. In fact, in this request, that's exactly what is happening. There is no file `hello-world`, but the server still knows how to process the request. The server generates a dynamic resource, building the HTML from fragments or templates and combining it with dynamic data to send back in the response, as text, so the browser can render the page.
Now that you know how the server builds the response to send back to the browser, let's take a look at how the browser handles the response.

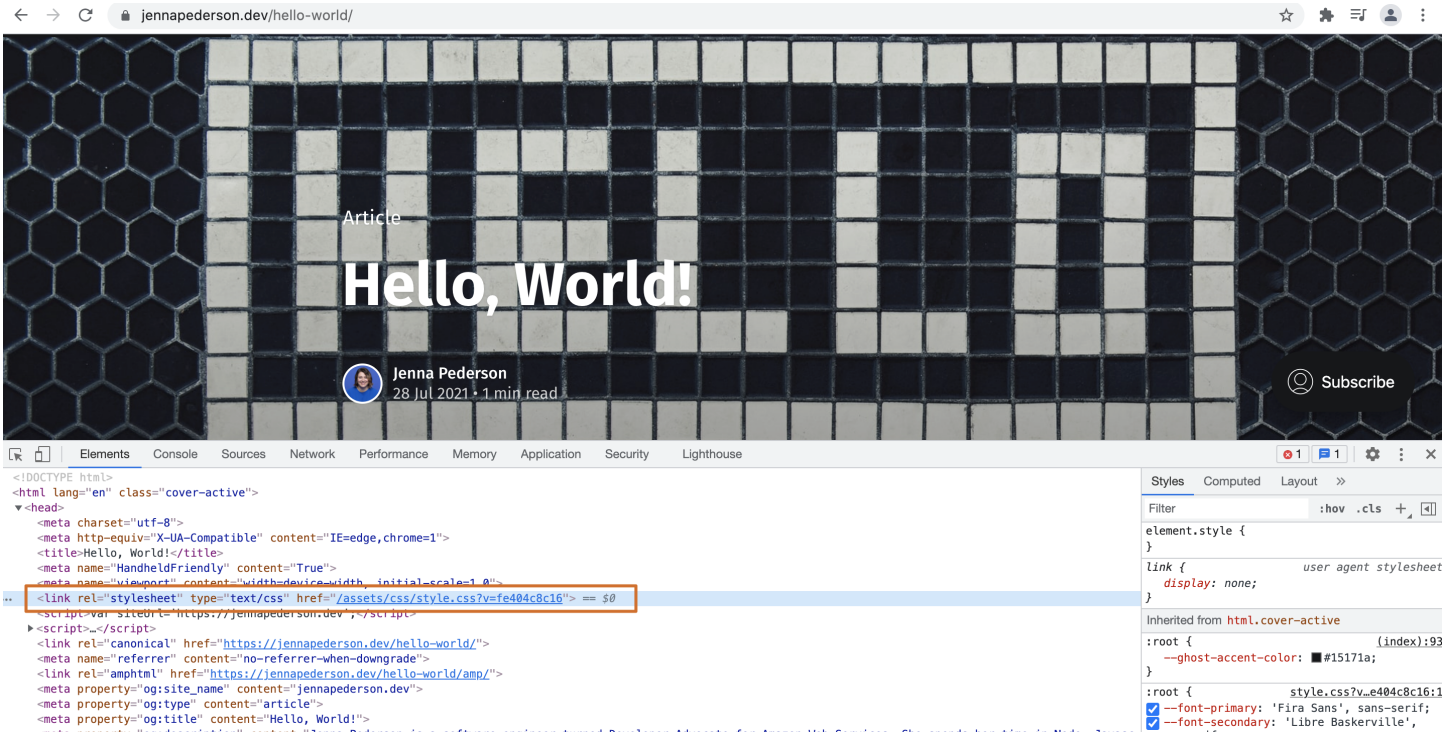
6. Browser renders the content

Once the browser has received the response from the server, it inspects the response headers for information on how to render the resource. The `Content-Type` header above tells the browser it received an HTML resource in

the response body. Lucky for us, the browser knows what to do with HTML!

The first GET request returns HTML, the structure of the page. But if you inspect the HTML of the page (or any web page) with your browser's dev tools, you'll see it references other Javascript, CSS, image resources and requests additional data in order to render the web page as designed.

As the browser is parsing and rendering the HTML, it is making additional requests to get Javascript, CSS, images, and data. It can do much of this in parallel, but not always and that's a story for a different post.



In the image above, you can see the HTML references a CSS resource. The browser makes a subsequent request to the server to get this CSS resource to style the page. The `Content-Type` header of the request for the CSS resource tells the browser to render CSS. If the browser requests an image resource, the `Content-Type` header tells the browser it is non-text data and to render it accordingly.

Wrapping Up

You did it! You traced a URL request from the browser all the way to the server hosting it and it's response back to the browser to be rendered. We covered the relationship between websites, servers, IP addresses and stepped through each of the steps that your browser goes through when you type a URL into your browser and press enter. For review, here are those six steps:

1. You type a URL in your browser and press Enter
2. Browser looks up IP address for the domain
3. Browser initiates TCP connection with the server
4. Browser sends the HTTP request to the server
5. Server processes request and sends back a response
6. Browser renders the content

Knowing what happens when you type a URL into your browser can help you figure out where things go wrong, where to look for performance issues with your website, and to offer a secure experience for your users.

If you'd like to try this out for yourself, you can build your own virtual private server, setup a CDN, and manage domains with Amazon Lightsail. Get started now with this tutorial and check out the latest pricing promotion so you can get started even quicker.
TAGS: Amazon Lightsail, CDN

Comments

ALSO ON AWS MOBILE BLOG

Introducing Merged APIs on AWS ...

8 months ago • 10 comments

AWS AppSync is a serverless GraphQL service that makes it easy to create, manage, ...

Client-side Caching Strategies for a ...

7 months ago • 3 comments

This post builds on the initial posts in this series, Build a Product Roadmap ...

Benchmarking your Mobile App with ...

9 months ago • 2 comments

Until recently, the primary reasons for rooting an Android device were to ...

Automate with ...

8 months ago

Note: This p and extensio blog post “R