Write Your Own HTTP Server (Part 1)

Learn the basics of building your own HTTP server in Python

Vaibhav Sinha Follow
Jan 3, 2020 · 6 min read ★

Photo by Ilya Pavlov on Unsplash

This is the first piece in the series "Write Your Own HTTP Server." You can read Part 2 and Part 3 here.

Back-end developers today have access to a lot of awesome open source web servers and frameworks. This has let people work on solving business problems through code without having to worry about networking, concurrency, and protocols. That is true for most of the typical web applications we have.

But then there are applications where you have to deal with scalability, concurrency, and performance issues. And it's not possible to deal with these by just optimizing our application logic. This is when we need to look at the nitty-gritty details of how our web server is configured: what concurrency model it's using, whether our application is IO-bound or compute-bound, whether the server is configured to optimize that kind of workload, etc.

To be able to make these choices in an informed manner, we need to understand how these web servers actually work. In my opinion, the best way to do this is to write a web server yourself. We don't need to write a production-grade server. Neither do we need to incorporate all the features these servers have or deal with all the use cases. We just need to develop a bare-bones working server to understand the different parameters and design choices that we have at our disposal.

In this series of pieces, we will develop an HTTP server from scratch. While we'll develop the server in Python, the learnings here will be generic, and one can use any programming language of choice for implementation. Some of the development might rely on specific features of the Python programming language, but similar features are mostly available in other programming languages as well, either as part of the language specification or through a library.

I will also link to external resources wherever necessary so that we can learn about the technologies used without cluttering the pieces with too many divergences.

With the introduction out of the way, let's get started.

. . .

A Common Deployment Scenario

In the Python world, Django is a popular choice of framework for developing web applications. While Django is used to develop applications, we use a WSGI server to serve these applications. Popular WSGI servers are Gunicorn, uWSGI, Apache HTTP

Server with `mod_wsgi`, etc. And generally Nginx is used in front of these servers as a reverse proxy.

Flow of requests from client to application

We want to understand how an HTTP request that originates from a client actually reaches our application. To that end, we want to understand the role of each of the pieces in the above diagram.

This series will unfold as follows:

1. In this piece, we'll discuss the very basics of what an HTTP server is.
2. In the second piece, we'll implement a basic thread-based HTTP server.
3. In the third piece, we'll understand what WSGI is and make our server WSGI compliant (well, almost). We will also build a bare-bones web framework that can actually talk to our server using WSGI.

. . .

The Basics

Before we dive into implementing our own server, let's take some time and understand what an HTTP server actually is.

When we talk about client-server architecture, the server is an application that has access to a resource. The resource may be a file, a database, some compute logic, etc. The client wants access to that resource. Hence the client needs to first connect with the server and then make a request for that resource. During this process, it might also need to authenticate itself, and the server might also check whether this client has the

authorization to access the requested resource or not. Also, the server needs to check whether the requested resource is even available. If not, it needs to inform the client.

Hence the client and server need to agree on two things:

- How the client will connect to the server
- What the format of the requests and responses will be

As long as they agree on these and they have a way to reach each other (like the internet), they can talk. Let's deal with both of these, one by one.

Transport

For two applications to communicate, they must be able to send bytes of data to one another. The data may be anything and is application dependant. But there must be a way to take data from one application process to another. That's the job of the transport. There are a lot of different ways of transporting data, and which ones can be used depends on where the server and client processes reside relative to each other.

For example, if both the client and server processes are running on the same machine, then they can use a file as the transport. The client might write requests to one file and the server might write responses to another. We can even use files when the processes are not running on the same machine, but then both the processes need to have access to the same file somehow. This is possible if there is a network filesystem like NFS. While using a file might not be the best way to orchestrate this kind of communication, it's important to realize that it is entirely doable.

Another way for two processes on the same machine to communicate is through Pipes or Unix sockets. Another option might be shared memory. Basically any IPC mechanism works.

But when we want processes on different machines to communicate, we need to involve the network somehow. And hence we need to use a network-based transport. There are a lot of network-based transport protocols, but the most popular, reliable transport for IP-based networks is the Transmission Control Protocol or TCP. TCP is used by all the browsers to connect to the servers of various websites. Hence if you want to talk to a client that is a browser, your choices are pretty much limited to TCP. Even when a browser is not involved, TCP is the protocol of choice for reliable communication.

Message format

Now that we know how to reach the other process, we need to decide on the message format that we'll use for communication. Let's see what data needs to be sent from the client to the server, and vice versa, to be able to request and provide a resource.

One, the client needs to somehow recognise the resource it wants access to. It might also need to provide some more metadata, like authentication details. Also, it might need to pass other information to the server, such as whether it can accept compressed data.

The server, on the other hand, needs to be able to send the actual resource in a way that the client knows its beginning and end. The server should also communicate other metadata, such as whether the content is compressed, and the client should uncompress it before consumption. It also needs to communicate if there was an error because either the client is not permitted to access the resource, the client asked for a resource that doesn't exist, or there was a fault on the server.

This is where HTTP comes in. HTTP is an application-layer protocol. That means it is the format that applications use to interpret the message from their remote peer. While HTTP is not the only application-layer protocol, it's the one that browsers use to access a website. Hence any web application that needs to be reachable by a browser needs to talk HTTP. If the application will only be accessed by other applications, they can choose to talk via whatever protocol they wish.

HTTP has URLs that are used to identify resources. It has methods/verbs that can be used to represent the action one wants to take on a resource. It has headers for all kinds of metadata. There is status code to represent the success/error scenarios. And finally, there is the body that can hold the actual data represented by the resource.

. . .

Up Next

When we say that we want to build an HTTP server, we mean that we want to create an application that can be connected to, using TCP as the transport layer protocol, and that can interpret the HTTP messages sent to it, parse them, and hand them over to the web framework. When we start building an HTTP server, these are the bare minimums that we'll need to support. Later we can start to worry about other aspects, such as

whether the server can handle multiple requests in parallel or whether it supports features like Keep Alive, etc.

In the next installment of this series, we will build a server that supports both of the above requirements and hence would qualify as an HTTP server.

Web Development | Software Development | Python | Programming | API