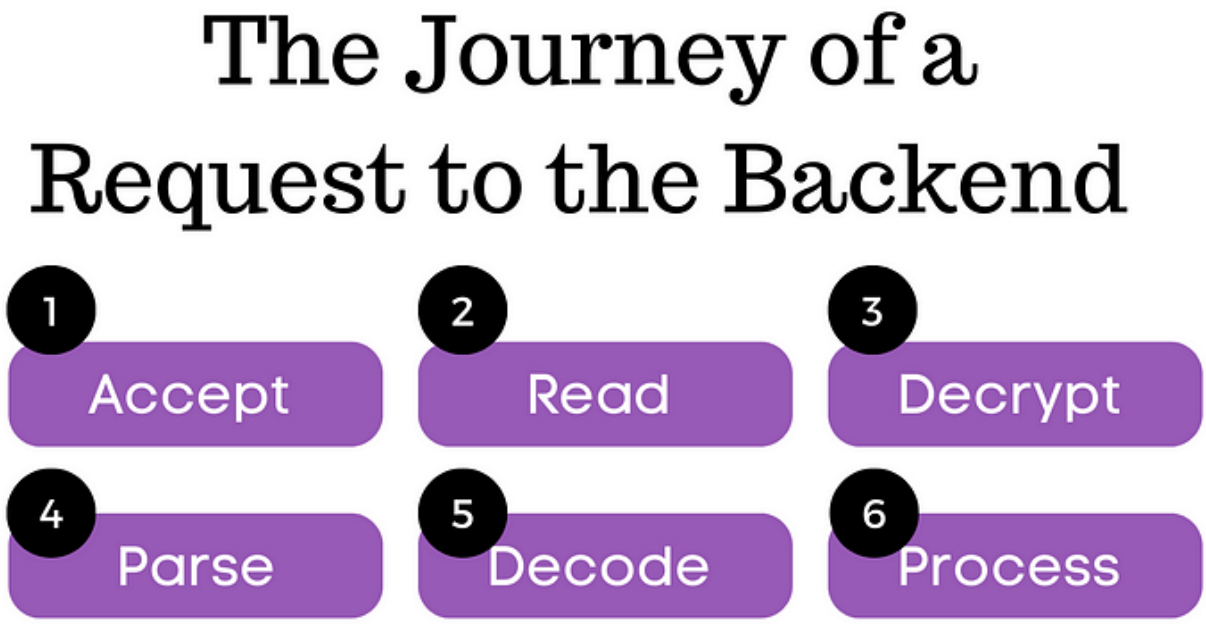
The Journey of a Request to the Backend

From the frontend through the kernel to the backend process

When we send a request to a backend most of us focus on the processing aspect of the request which is really just the last step.

There is so much more happening before a request is ready to be processed. I break this into 6 steps, each step can theoretically be executed by a dedicated thread or process. Pretty much all backends, web servers, proxies, frameworks and even databases have to do all these steps and they all do choose to do it differently.

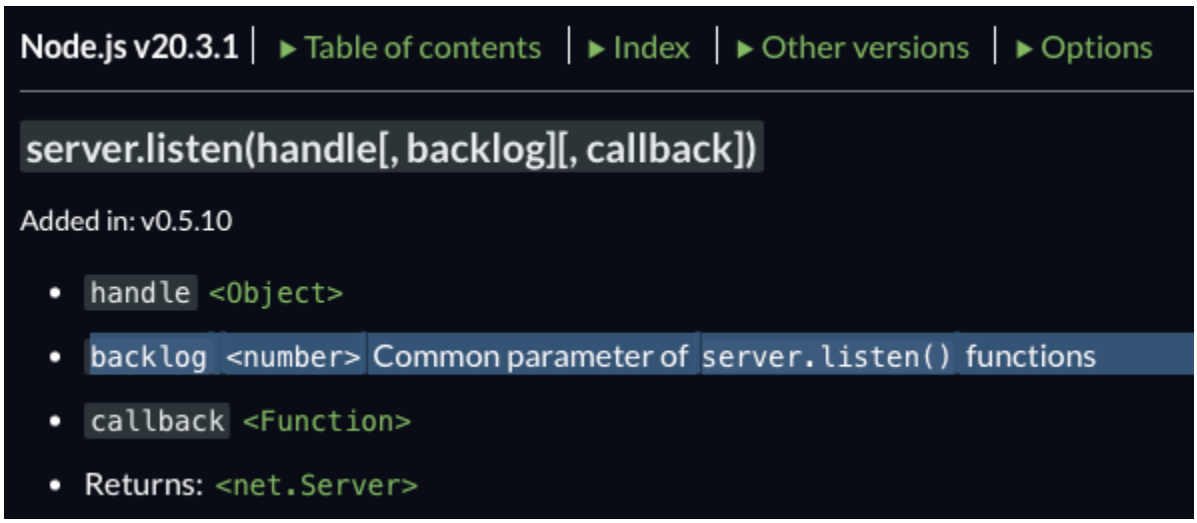
In another medium post I explored step 1 and step 2 in details, let us explore the rest in this post.



1- Accept

Requests are often sent on connections (either TCP/QUIC) and connections need to be accepted by the backend. When a client connects to a server on port 443, a 3 way handshake is completed by the server OS kernel and the connection is placed on the listener queue, we call this queue the accept queue. The backend application is responsible to invoke syscall accept() on the listener socket to create a file descriptor which represents the connection.

This step can become a bottleneck if the backend is slow in accepting connections. Leading to a large backlog of connections which can eventually fill up the queue resulting in new connections failures.



When you listen on a port you can essentially specify the size of the accept queue, this parameter is called the backlog. Here is example in NodeJS.

To speed up connection acceptance, most backend dedicate one thread just to accept connections. If a single thread cannot keep up multiple thread can start accepting connections, however this create a bottleneck as threads block each other when accepting on the same socket.

In that case the SO_REUSEPORT option can be used to create multiple listener sockets (and thus multiple accept queues) on the same port with each thread/process owning a socket queue. This is now the default options in most backends like NGINX, HAPROXY.

2- Read

Once the connection is established, the client can send requests to the backend. The request is really nothing but a series of bytes with a clear start and end defined by the protocol that is used. This is where the client and the backend must agree on the protocol HTTP being the most common.

Note even HTTP protocol has versions that has completely different on-wire representation which adds additional cost to parsing.

The client encrypts the request (if TLS is used on the connection), compresses body (if request compression is supported) and serialize the data type (JSON/protobuff etc) to an on-wire representation. Then finally writes the raw bytes in network byte order to the connection.

What we are interested in is really the backend side of it, those raw bytes reach the OS kernel from the NIC and go into the connection receive queue managed by the kernel.

Packets set there until the backend application invoke read() or rcv() syscall which then moves the data from the receive queue to the backend process user space memory.

We have to remembers that those are raw bytes that are encrypted and encoded, there is no request here just bytes, for all we know those bytes we read could be 10 requests or could be half of a request. We don’t know.

Reading can be done in its own thread or done in the same thread as the acceptor.

3- Decrypt

Now that we have raw bytes in the backend process memory and we know those are encrypted, we invoke the SSL library that our code is linked to (whether OpenSSL or other) and let it decrypts the content for us so we can make sense of it.

Remember we can't see any requests or know the boundary of the protocol until we decrypt the content to see the headers and other metadata. This could be HTTP/1.1 or HTTP/2 or even SSH.

Decryption is CPU bound operations, it can be done in its own thread or in same thread as read and accept.

4- Parse

Now that we have plaintext readable bytes we can use our knowledge of the agreed upon protocol to parse requests, the chunk of bytes we read might have a full request or it might not. For all we know it might be simply no requests but protocol system headers (like SETTINGS frame in h2).

This is where our library of choice kicks in to do the parsing based on the protocol, if it is HTTP/1.1 the library you used will read plaintext and look for the start and end of request based on the definition of HTTP spec. For instance with content-length or transfer encoding.

If it is an HTTP/2 or HTTP/3 library the same thing apply although much more work is required to parse those as there are much more metadata associated with the binary protocol.

Keep in mind that parsing cost CPU cycles and can tax your backend especially for h2 and h3. Something lucid chart found out the hardway.

But regardless once we parse the bytes and find the requests we are almost ready.

Protocol parsing can be done in its own thread or in the same thread as the other.

5- Decode

This step is where further work required on the request. Request using JSON or protobuf can be deserialized in this step to objects based on the language of choice. We turn the raw bytes into language structures which has its own cost and memory footprint.

Remember we can't use JSON string even in JavaScript we have to call JSON.parse on it, even if this is automatically done for us by libraries such as express doesn't make it free.

This also applies to bytes representing text encoded in UTF8. The raw bytes must be decoded to UTF8 if we know the content is of that

format, otherwise we get a jumbled mess because as you know UTF8 uses up to 4 bytes to represent some characters. 20 bytes in ASCII might look different than 20 bytes in UTF8.

Another step in the decoding is request decompression while rare it is possible that large request body sent with POST are compressed. Before processing the request the body need to be decompressed to find out what is in it.

6- Process

Finally once we understand the request we actually process it, whether this requires a query to the database, a read from disk, an expensive compute. This step can be done in the same thread although it is recommended to have a dedicated worker for processing, this is where worker pool pattern works nicely.

Summary

The request goes through a long journey before it is processed, knowing this allows us backend engineers to architect the most appropriate design so that each step doesn’t become a bottleneck.

One might put all these steps on one thread while another might dedicate a thread for each step, another might combine steps together. No wrong or right.