How Browsers Work

howbrowserswork.com

An interactive guide to how browsers work.

Why?

The guide is for engineers and curious people who use the web every day, but never built a mental model of how browsers work.

I find most guides too technical, too detailed, or too shallow, so I have decided to take a different approach.

I built the guide with many tiny interactive examples you can play with to help you go get through the technical details and **build an intuition of how browsers work**.

To keep it short and straight to the point, many critical details are omitted like different versions of the HTTP protocol, SSL, TLS, nuances of the DNS, and many more.

I made the guide open source. Feel free to suggest improvements by creating an issue or a pull request.

Browsers work with URLs

You can type literally anything in the address bar. But under the hood, browsers work with URLs:

- A random text like `pizza` will be transformed into a "search" URL like `https://google.com/search?q=pizza` (or `https://duckduckgo.com/?q=pizza` depending on your preferences).
- A domain name like `example.com` will be normalized as a full URL: `https://example.com`

To see how this works in practice, type something in the address bar and press `Enter` (or click the "Go" button):

Try “pizza” or “example.com”.

Turning a URL into an HTTP request

Once we know the exact URL we want to visit, we can send a request to the server to fetch the resource and display it in the browser. Browsers communicate with servers using the HTTP protocol.

To see how a URL is translated into an HTTP request format, enter a full URL like https://example.com and press `Enter` (or click the "Go" button):

Enter a full URL like https://example.com.

HTTP requests have headers in the format like:

```
Host: example.com
Accept: text/html
```

One of the headers is the host header. It is used to identify the server to which the request is sent: example.com .

Resolving the server address

Browsers can't send requests to names like example.com .

Computers talk to IP addresses, so the browser first asks the DNS system to resolve the domain name into an IP address before it can connect to the server and send the HTTP request.

Type a domain name in the input and press `Enter` to resolve it into an IP address:

Terminal

Establishing the TCP connection

After DNS gives the browser an IP address, it still needs a reliable connection to the server. TCP is the protocol that sets up this connection before any HTTP data is sent.

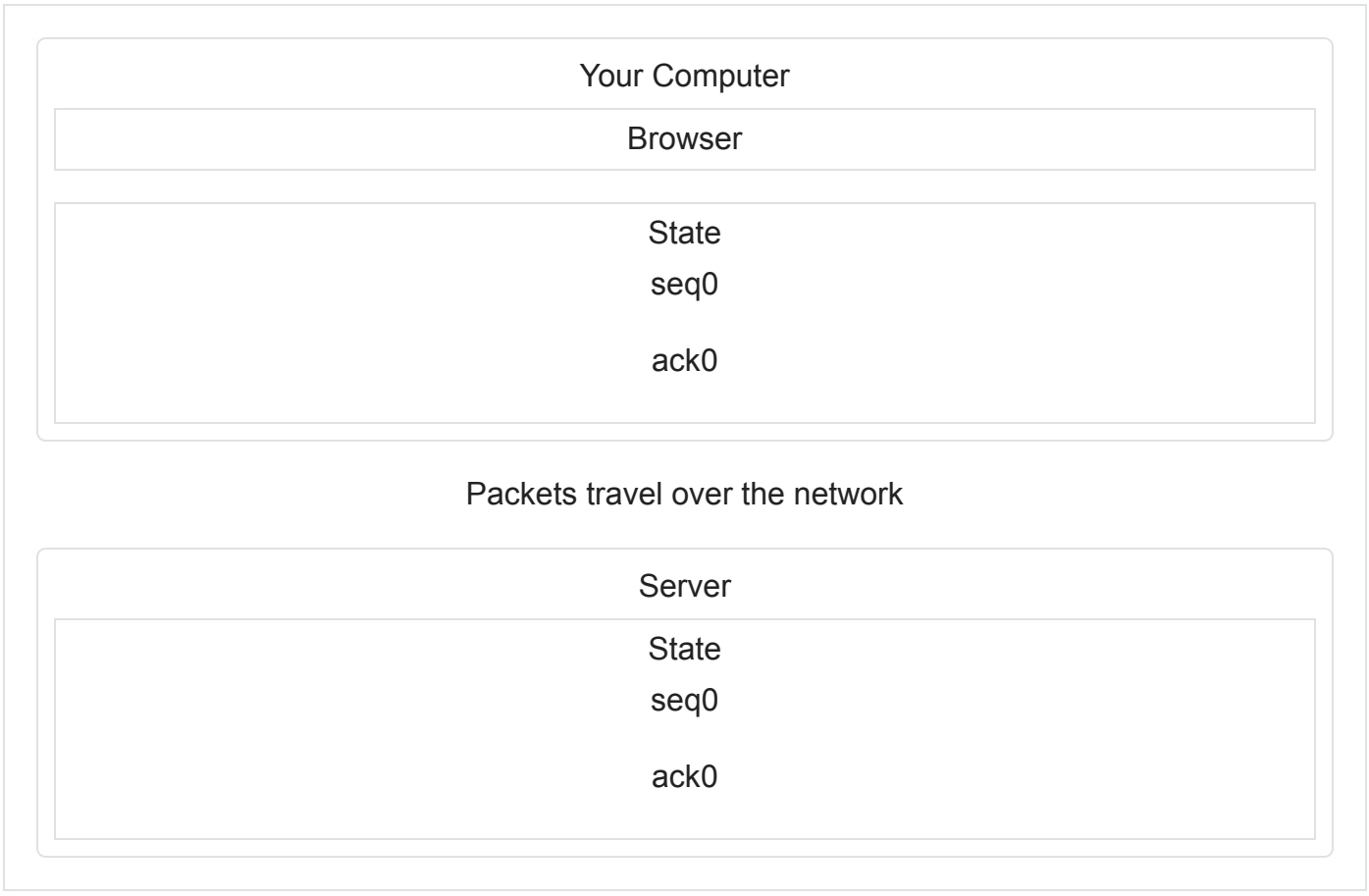
TCP establishes the connection using a three-step handshake that confirms both sides are ready to send and receive data.

Disconnected

1. SYN: Client sends its sequence number (seq=1000) to open a connection.

2. SYN-ACK: Server acknowledges the packet by adding its own sequence number (seq=5000) and acknowledging the client sequence number by incrementing it by 1 (ack=1001).

3. ACK: Client confirms the server number by incrementing it by 1 (ack=5001) and the connection is ready.



These numbers are how the client and the server keep track of the conversation. They count bytes, so both sides agree on where the data stream starts and what should come next. If some data doesn't arrive, the sender can see the gap and retransmit the missing bytes. This is how TCP keeps data ordered and reliable once the connection is established.

Start sending packets and try to disrupt the network to see what happens.

Connected

Your Computer

Browser

State

seq1001

ack5001

Packets travel over the network

Server

State

seq5000

ack1001

HTTP requests and responses

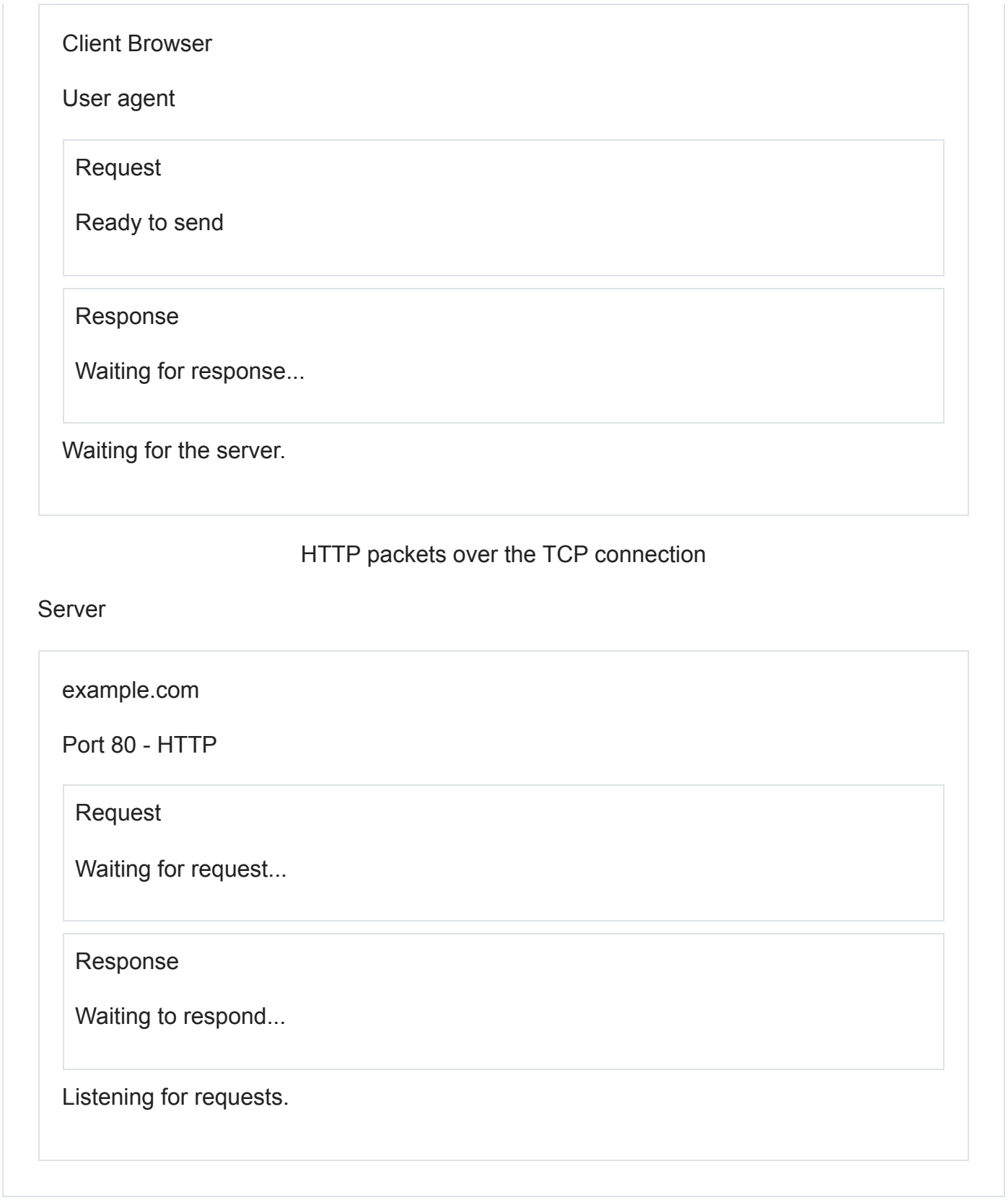
Once the TCP connection is established, the browser can send an HTTP request to the server.

Click the "Go" button to watch the HTTP request travel to the server and the HTTP response return to the browser:

Ready to send

Watch the packets move between the browser and the server.

Browser (Client)



When the HTTP response arrives, the browser reads the raw HTTP response and starts rendering the HTML content.

Parsing HTML to build the DOM tree

After the HTTP response arrives, the browser separates the headers from the body and feeds the HTML bytes into the parser. The parser turns tags like <h1> into tokens and builds a DOM tree.

Click the "Parse" button to watch the HTML stream being parsed into the DOM tree:

The HTML stream

```
 <!doctype html>  <html>    <head>      <title>Example Domain</title>
</head>    <body>      <main>        <h1 style="color: red;">Example
Domain</h1>        <p>An example paragraph.</p>        <p>
            <a href="https://example.com">An example link</a>
          </p>      </main>    </body>  </html>
```

The DOM tree

```
Document|- <!doctype html>`- html   |- head   |  `- title   |     `- "Example
Domain"   `- body      `- main         |- h1 (style: color: red)         |  `-
"Example Domain"         |- p         |  `- "An example paragraph."         `-
p            `- a (href="https://example.com")               `- "An example
link"
```

Parsing is streaming and error-tolerant: the browser starts building nodes before the full document is downloaded, and it inserts missing tags to keep the tree valid. When a <script> tag appears, parsing may pause so the script can run.

The DOM tree then combines with CSS to produce the render tree that layout and paint use to draw pixels.

On the importance of the DOM

The DOM is the browser's in-memory model of the document. It is the shared contract between the HTML parser, CSS selector engine, and JavaScript runtime, so changes to it immediately affect layout, styling, and what users can interact with.

The DOM powers everything from query selection to dynamic styling and event handling. Try editing the script and watch how the DOM changes on the right.

Layout, Paint, and Composite

Once the DOM and CSS are ready, the browser runs the rendering pipeline: Layout (reflow) to calculate sizes and positions, Paint to fill pixels, then Composite to stitch layers together on the GPU.

Not every change reruns every stage. Changing colors usually repaints, while changing sizes forces layout and paint to recompute.

Click a change to see which stages rerun.

Layout

Reflow sizes + positions

Paint

Fill pixels into layers

Composite

Stitch layers on the GPU

DOM preview

Hero card

Width: 200px

Composite always blends layers into the final frame.

This is why layout-heavy pages feel slower: more work needs to happen before the next frame can be shown.