

chapter one

From a URL to a Page

The running example begins on a laptop connected to a home network. Its browser receives this URL:

```
https://www.hackerbrewery.com/blog
```

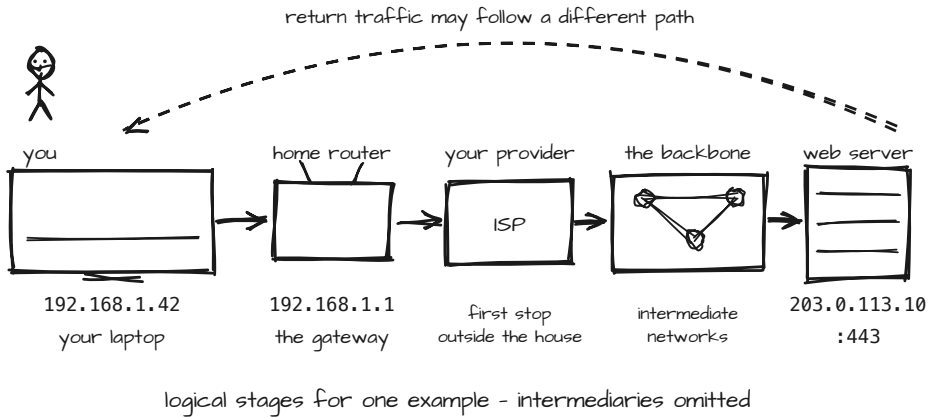
The page that eventually appears may combine a cached document, new responses, images, stylesheets, scripts, and fonts. This chapter locates the main systems in that process before later chapters examine them separately.

The example uses fixed documentation addresses:

Role	Address
Laptop	192.168.1.42
Laptop MAC	a4:5e:60:...
Default gateway	192.168.1.1
Recursive DNS resolver	1.1.1.1
Example web endpoint	203.0.113.10
HTTPS port	443

The example endpoint is a network destination, not necessarily one physical server.

The endpoint address stands for whichever service accepts the request. In a production deployment that might be a CDN edge, proxy, or load balancer rather than the application process that creates the response.



The fixed cast for the running example. The logical path omits many possible intermediaries and does not require the return path to match.

One possible trace

The outline below is a dependency map, not a strict schedule. Cached DNS data can remove a lookup. Browsers can reuse or pre-establish connections. A proxy can change the peer. Routing carries the DNS, transport, TLS, and HTTP traffic; it is not one isolated step after the request is written.

Dependencies constrain the trace, but caches, reuse, and parallel work change its schedule.

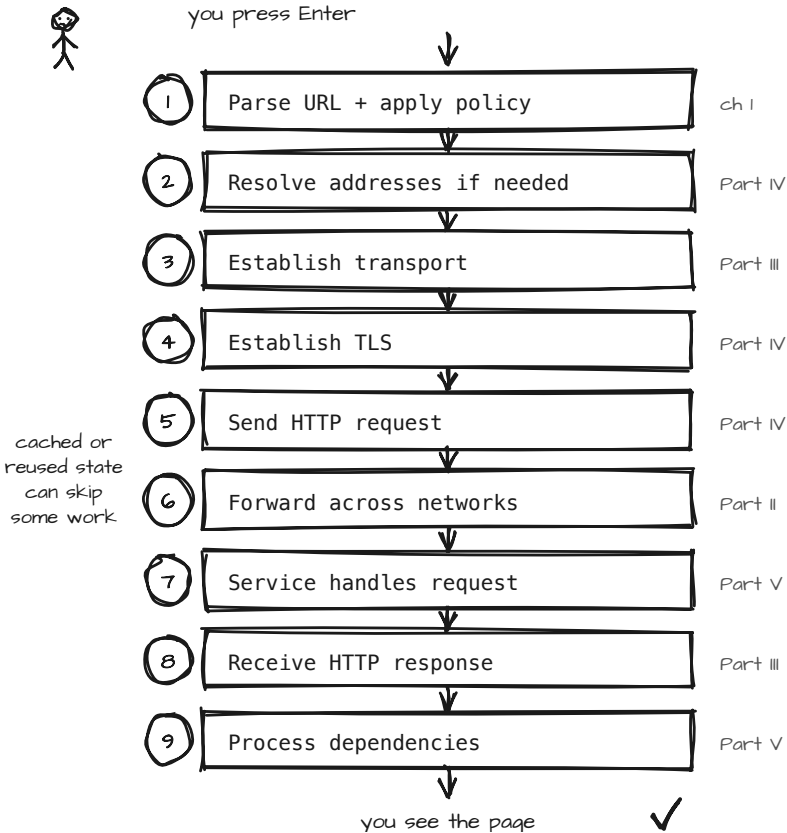
browser receives URL

|
v

1. Parse URL and apply browser policy
2. Resolve candidate addresses if needed
3. Establish transport to an endpoint
4. Establish TLS for this HTTPS origin
5. Send an HTTP request
6. Forward its packets across networks
7. Service handles the request
8. Receive the HTTP response
9. Process it and request dependencies

|
v

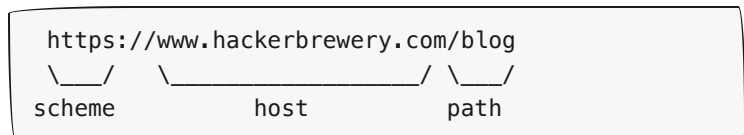
browser presents the page



A dependency-oriented overview of the running trace. Forwarding supports several stages, and cached or reused state can skip work.

1. Parse the URL

URL parsing begins locally. The browser identifies the scheme, host, and path:

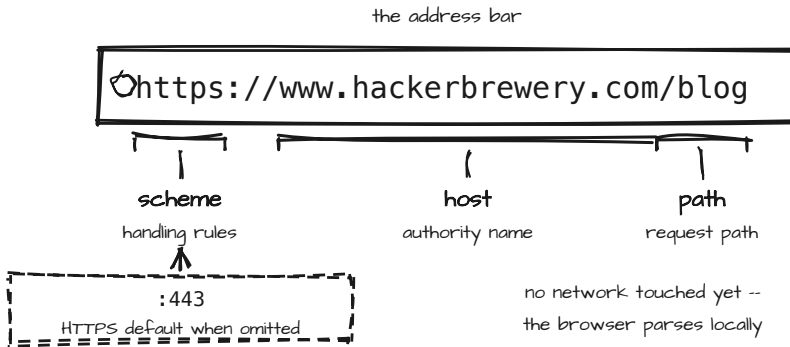


The `https` scheme selects HTTPS handling and, in the absence of an explicit port, its default port 443. The host participates in name resolution, certificate checks, HTTP

origin rules, and request metadata. It does not necessarily name one machine. The path is interpreted by the HTTP service; it need not correspond to a file on disk.

Browser policy can modify the next action. HSTS may upgrade an `http` URL, extensions or enterprise policy may use a proxy, and a service worker or cache may satisfy some requests locally.

A URL identifies an origin and request target; it is not a physical route.



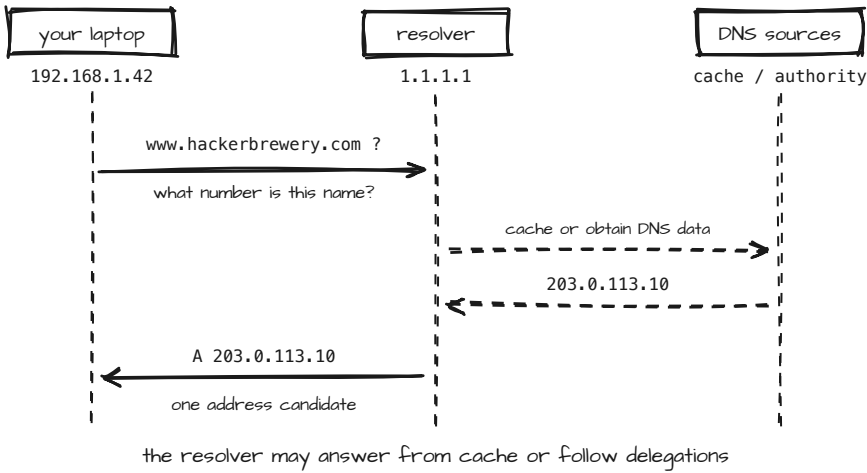
The browser parses scheme, host, and path locally; HTTPS supplies a default port when the URL omits one.

2. Resolve candidate addresses

If the browser and operating system do not already have usable address data, name resolution looks for records associated with `www.hackerbrewery.com`. In this example the result includes the IPv4 address `203.0.113.10`.

The laptop normally sends a query to a configured recursive resolver. That resolver may answer from cache or obtain referrals and authoritative data. A real answer can contain several IPv4 and IPv6 addresses, aliases, service parameters, and lifetimes. The client still has to choose what to try.

DNS can return several candidates; connection logic chooses among them.



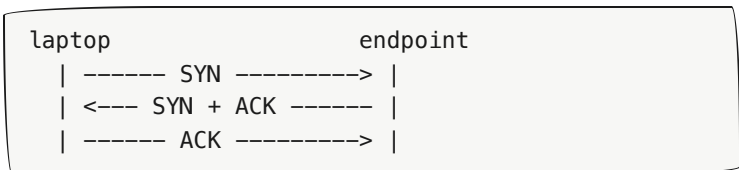
The laptop asks a recursive resolver. A cache hit can answer immediately; otherwise the resolver obtains the necessary DNS data.

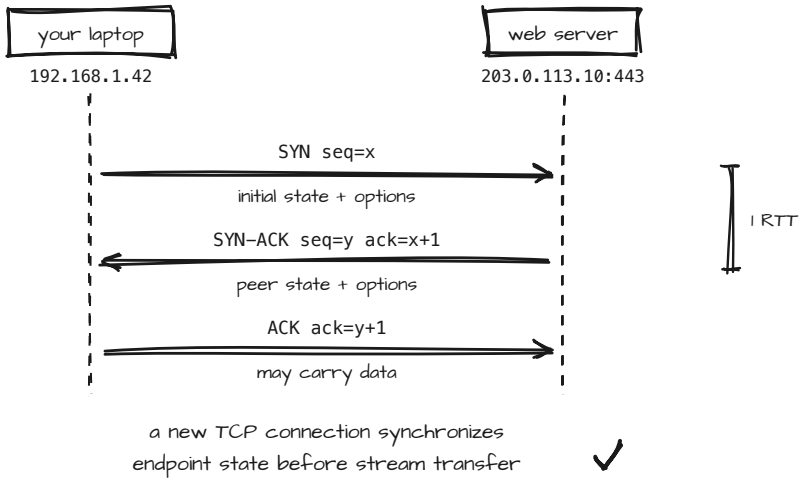
3. Establish transport

The book's main trace uses TCP to connect to `203.0.113.10:443`. TCP exposes an ordered byte stream and implements reliability, flow control, and congestion control between the endpoints.

A new TCP connection begins with SYN, SYN-ACK, and ACK segments. Their sequence numbers are chosen by the endpoints rather than starting at a fixed zero. The final ACK can carry application data, and TCP options negotiated in the handshake affect later transfer behavior.

The diagram shows a new TCP connection; a browser may instead reuse one already open.





A new TCP connection synchronizes endpoint state with SYN, SYN-ACK, and ACK. Sequence values and options vary.

An HTTP/3 connection follows a different path: QUIC combines transport and TLS work over UDP. That alternative appears later; it is one reason the overview should not be read as a universal stack.

4. Establish TLS

Because the URL uses HTTPS, the client establishes TLS before sending HTTP application data on this new connection. Modern TLS negotiates protocol parameters, authenticates the server name through the certificate system, and derives traffic keys. The browser checks such details as the certificate chain, validity period, requested hostname, and relevant policy.

TLS protects application contents against passive observers and detects tampering between its endpoints. It does not conceal every item of connection metadata, prove that a site is honest, or guarantee that the endpoint software is secure. The TLS endpoint may also be a CDN or reverse proxy acting for the origin.

A valid HTTPS connection authenticates a name under browser policy; it does not endorse the site's content.

5. Send an HTTP request

For an HTTP/1.1 example, the browser's request can be displayed as text:

```
GET /blog HTTP/1.1
Host: www.hackerbrewery.com
```

GET states the method, /blog is the request target, and Host identifies the requested authority for HTTP/1.1 routing. Real requests include more headers. HTTP/2 and HTTP/3 preserve related semantics but encode them with binary framing rather than these literal lines.

Readable HTTP/1.1
syntax is a
representation;
HTTPS encrypts it on
the wire.

On the wire, TLS carries the encoded HTTP data. An observer without the traffic keys does not see this plaintext request.

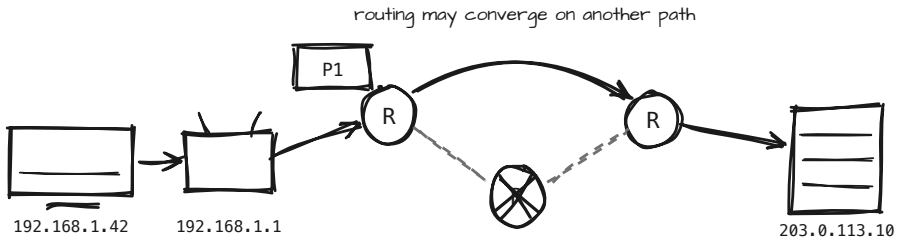
6. Forward packets across networks

DNS queries, connection setup, TLS records, and HTTP data all rely on network forwarding. For the HTTP traffic, the laptop first determines that 203.0.113.10 is outside its local IPv4 subnet and sends a link-layer frame to the default gateway at 192.168.1.1.

```
laptop -> gateway -> provider -> other networks
      -> service network -> endpoint
```

Routers make local
forwarding decisions
from distributed
routing state.

Each router consults forwarding state and selects a next hop. It need not store the complete future path of an individual packet. Routing policy, failures, load sharing, and tunnels can change the observed route; packets in a flow often follow a stable path, but that is not an end-to-end guarantee. Return traffic can use a different route.



each router applies forwarding state and policy
to select a next hop for the destination

failures can drop traffic until another route is usable

A forwarding table can select another next hop after a failure; the diagram does not imply every router knows only one neighbor.

7. Handle the request

Traffic arriving at `203.0.113.10:443` may first reach a firewall, CDN edge, TLS terminator, reverse proxy, or load balancer. Eventually a service handles the request. It might read stored content, run application code, query another service, or return a cached representation.

For the HTTP/1.1 illustration, the response might begin:

```
HTTP/1.1 200 OK
Content-Type: text/html

<!doctype html> ...
```

Status `200` reports successful handling under HTTP semantics. It does not by itself say whether the body came from a file, cache, generated page, or another upstream system.

The public endpoint and the component that generates a response may be different systems.

8. Receive the response

The response bytes travel in TLS records over the selected transport. With TCP, the sender can segment the byte stream according to current limits and network conditions. IP packets carry those segments, and link-layer frames change at each applicable hop.

Packets can be lost, duplicated, or reordered. TCP uses acknowledgements, sequence numbers, retransmission, and receive buffering to present an ordered byte stream to TLS. Packet boundaries are not preserved as application message boundaries.

9. Process resources and render

The browser parses the returned representation and determines what else it needs. HTML can refer to stylesheets, scripts, images, and fonts. Some are already cached; others create requests that may reuse existing DNS results and connections or establish new ones.

A page load is a dependency graph of network and rendering work, not one repeated serial trip.

The browser combines document structure, styling, layout, painting, and script execution to update the display. These names form a useful model, though browsers pipeline and parallelize the work rather than executing one fixed list from beginning to end.

Using the overview

The rest of the book expands each box and records where the simplified trace branches. When a later example differs, compare its retained state, address family, transport, proxy path, protocol version, and cache policy with this baseline.

Exercises

Exercise 1.1. Split the running URL into scheme, host, path, and effective port. Which field is absent from the text but supplied by the scheme's default?

Exercise 1.2. Identify three conditions that can remove or reorder work in the outline: one involving DNS, one involving a connection, and one involving cached content. Why is the outline a dependency map rather than a fixed script?

Exercise 1.3. Compare the HTTP/1.1 request shown as text with what a passive observer sees on an HTTPS connection. Which layer supplies confidentiality, and which connection metadata may remain observable?

Exercise 1.4. Explain why `203.0.113.10:443` need not identify the process that generates the HTML. Name two intermediary roles that may accept the connection before an application handles the request.