Chapter 6

Client-Server Communication

What is in This Chapter ?

This chapter explains the structure of the underlying communication that takes place between the client and the server. It explains how Web Apps differ from conventional applications that we have been working on in other courses. We will introduce important terms such as **IP Addresses**, **URL**s, **HTTP**, **DNS** and **DNS Lookup**. The chapter digs into how **HTTP Requests** and **HTTP Responses** are used to facilitate client/server communication. The chapter ends with a short discussion of **URL shortening** and **Caching**.

6.1 Client-Server Architecture

Recall that *web apps* differ from *websites* in that they are software built for interaction in a browser, as opposed to just information to be "browsed through". Unlike traditional desktop applications, web apps do not need to be installed on a user's device. They are hosted on remote (perhaps multiple) servers and delivered through standard web technologies such as **HTML**, **CSS**, and **JavaScript**. They may connect to backend services or databases via **API**s.

Web apps are platform-independent, meaning they can run on any device with a browser (i.e., PC, Mac, smartphone, tablet). They can range from simple tools (like a calculator) to complex systems (e.g., online banking platforms, multi-player games or collaborative tools like Google Docs). They are accessible via a **URL** using a web browser (e.g., Chrome, Firefox).

Web apps have advantages over installed (native) apps:

- ✓ **Cross-platform compatibility**
  - works on any device with a browser (Windows, macOS, Android, iOS, etc.)
  - no need to develop separate versions for each operating system

- ✓ **No installation required**
  - users don't need to download or install anything … just open a **URL**
  - easier for users to access and try
- ✓ **Easy updates and maintenance**
  - changes are made on server, so all users get the latest version instantly
  - no need for users to manually update the app
- ✓ **Lower development and deployment costs**
  - one codebase can serve all platforms
  - hosting and deployment are often simpler than native apps
- ✓ **Scalability**
  - easier to scale for more users or expanded features through backend infrastructure

Of course, there are disadvantages of web apps as well …

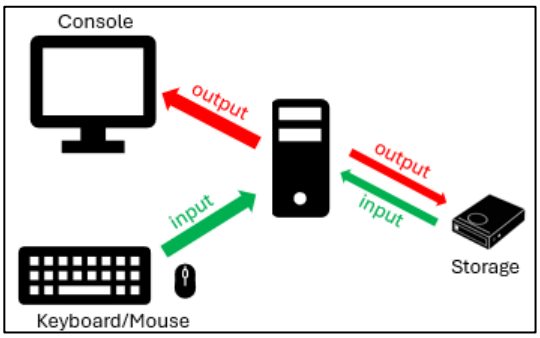
- × **Internet dependency**
  - requires an internet connection to function (or to function fully), although some can work offline with technologies like service workers
- × **Limited access to device hardware**
  - restricted access to hardware features like GPS, camera, Bluetooth, etc., compared to native apps

- **Performance limitations**
  - generally slower than native apps, especially for graphics-heavy or real-time applications (e.g., games, video editing)
- **Security risks**
  - being online exposes them to web-based threats
  - more effort is needed to secure web apps properly
- **Privacy concerns**
  - user data is stored on remote servers and may be exposed to third-party tracking, unauthorized access, or insufficient data protection practices
- **User experience**
  - may not feel as smooth or integrated as native apps, especially on mobile
  - limited offline functionality compared to native apps
- **Development**
  - can be more difficult to develop and debug

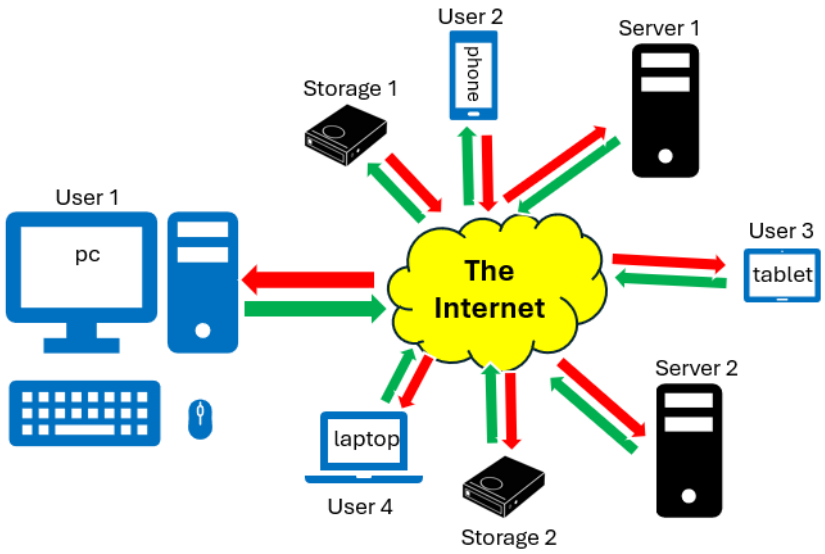
How does the development of web apps differ from what we have been doing?

We are used to writing desktop applications which were mostly console-based, although most will have a GUI component. Our applications involved a single user doing one thing at a time, often based on prompts, but sometimes based on events from the GUI. There were no interacting resources.

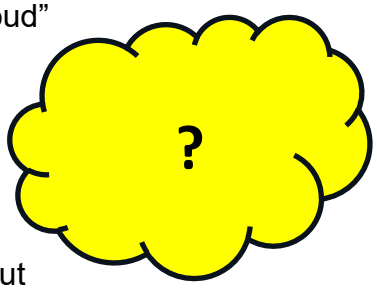
This is the architecture that we are familiar with →



In the case of web applications, we are working with a different kind of architecture … one that is distributed and decoupled. The **client** (the user's browser or app) is separate from the **server** (where the data and application logic live). Multiple clients can interact with the server simultaneously. Additionally, many resources and systems may interact behind the scenes, often across different computers or services. Web app behavior is typically event-driven, meaning the application responds to user actions or other triggers in real time.



Is the internet really a "cloud"?

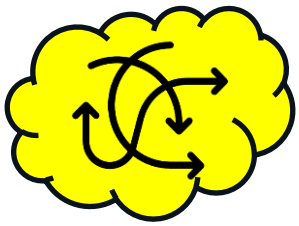


We sometimes think of the internet as "***the cloud***". However, the term "cloud" in computer science, refers to ***remote computing resources*** **that are accessible over the internet** (e.g., servers, storage apps). When we use services like Google Drive or Dropbox, we are storing data "in the cloud" (i.e., on internet-connected servers). From the user's perspective, the location and infrastructure are abstracted away (i.e., just as we can see a cloud in the sky floating above us, but we don't see all the air, water, and wind that hold it up … cloud computing gives us services without showing the servers, networks, and infrastructure that make them work).

In contrast, the internet is the **global network** (i.e., the infrastructure) that connects computers. The cloud refers more specifically to services and resources hosted on that network. So, not all the internet is "the cloud", and not all cloud services are public (some are private or hybrid). So …

> ***The internet is the global network that connects devices.***
>
> ***The cloud is the services and storage delivered via the internet.***

How does the internet know how to connect one device to another?

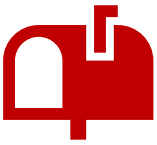


As it turns out, things are somewhat organized. Just as houses are organized by unique addresses, the devices on the internet also have somewhat unique addresses.

> ***An IP address is a unique identifier for a device on a network.***

IP is short for ***Internet Protocol***. An IP address is a lot like a home address. It uniquely identifies a device on a network and provides its location so data can be sent to and from it correctly.

If we were to use the postal service as an analogy …



- IP address = mailing address
- data being sent = package being delivered
- internet = postal system.

And just as a package won't arrive at its destination without a proper address, data won't reach our device without its correct IP.

Here are two different IP protocols:

- **IPv4** - most common, uses 4 numbers (each from **0** to **255**) separated by dots.

  `192.168.0.1` ← limited at ~4.3 billion address combinations

- **IPv6** - newer, more complex, supports more devices, uses 8 groups of 4 hex digits with colons.

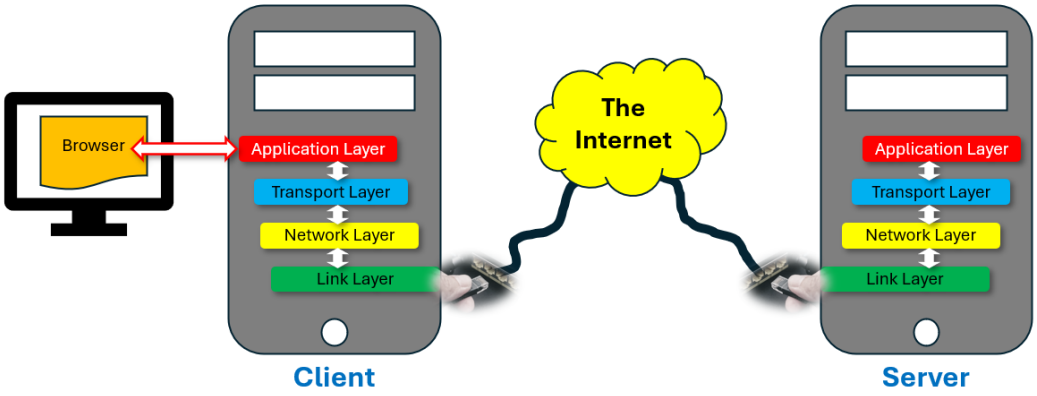
  `2001:0db8:85a3:0000:0000:8a2e:0370:7334`

  essentially unlimited at 6×10^{28} times more addresses than IPv4 😲

The IP is just the address, but a lot happens behind the scenes to make communication work. In fact, both the client and server go through several steps to send and receive data successfully. All these steps happen according to a framework called the ***TCP/IP model*** (short for **Transmission Control Protocol / Internet Protocol**), which provides the foundational set of communication rules (protocols) that makes the Internet and most modern networks work.

The **TCP/IP** model is made up of four layers … each handling part of the communication process:

- **Application Layer** - where the apps we use (e.g., our browser or email) talk to the internet and send or receive information.
- **Transport Layer** - makes sure our messages get sent safely and in the right order (like a reliable mail carrier).
- **Network Layer** - figures out the best route to send our data across the internet (like using a GPS for our messages).
- **Link Layer** - handles the actual physical connection (like our Wi-Fi or Ethernet cable, sending data between nearby devices).



What happens when a client talks to a web server?

Here is what happens when we (the client) open a browser and type in a web address:

1. Our device:
   - **puts the message in a box** (transport layer)
     … like wrapping a package so it won't get damaged or lost
   - **writes the address on the box** (network layer)
     … like writing the website's IP address so it knows where to go

2. The box travels through the internet and arrives at the server.
3. The server:
   - **unpacks the box** (transport layer, again).
   - **reads the message** … like saying "Hey, show me this web page!".

4. The server sends a reply back in the same way:
    - puts the web page in a box (transport layer).
    - writes our address on it (network layer).
    - sends it back to our device.
5. Our device opens the “box” and shows us the web page!

Of course, just as delivering physical mail can have delays and issues … there can be latency issues involved with this whole process. There is always a bit of uncertainty as to whether data will get lost or be unavailable (i.e., website down). And of course, when dealing with such a large network of devices, there are scalability issues as well (i.e., server might crash or slow down if thousands of users try to access a website at the same time).

When it comes to each layer, there are specific **protocols** (i.e., *tools* or *rules*) that each layer uses to do its job. Here is a table of the most common ones:

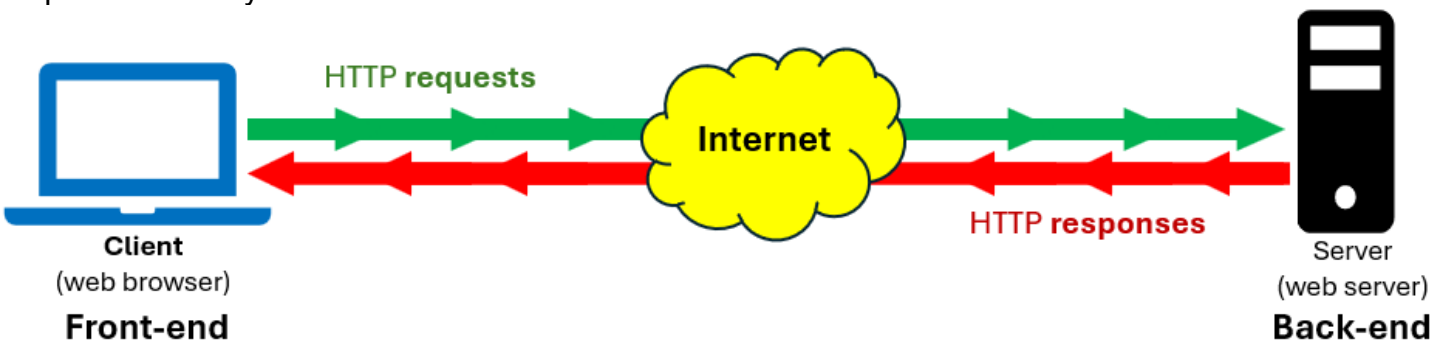
TCP/IP Layer	Common Protocols Used	Intuitive Description
Application Layer	**HTTP**, **HTTPS**, **FTP**, **DNS**, SMTP, IMAP, SSH	the kind of message it is
Transport Layer	**TCP**, **UDP**	how safely or quickly we send it
Network Layer	**IPv4**, IPv6, ICMP, ARP, IGMP	putting the right destination on it
Link Layer	**Ethernet**, **Wi-Fi** (IEEE 802.11), PPP, DSL	how it travels (car, bike, plane)

Here are definitions of some of these protocols:

- **HTTP (Hypertext Transfer Protocol)** - to request/display web pages
- **HTTPS (HTTP Secure)** - to request/display web pages securely (i.e., encrypts data)
- **FTP (File Transfer Protocol)** - to upload/download files between computers
- **DNS (Domain Name System)** - to translate human-readable names into their IP addresses
- **SMTP (Simple Mail Transfer Protocol)** - to send mail
- **IMAP (Internet Message Access Protocol)** - to receive mail
- **SSH (Secure Shell)** - to securely access and control a remote computer
- **TCP (Transmission Control Protocol)** - a protocol that ensures data is delivered accurately and in the correct order (it’s like making phone call)
- **UDP (User Datagram Protocol)** - a faster but less reliable protocol that sends data without checking if it arrives correctly (it’s like sending physical mail)

6.2 Communication Basics

In this course, we will be doing both client-side and server-side programming. All communication between a client and a server happens through **HTTP** requests sent by the client and **HTTP** responses sent by the server.



To make this work, we need software on both the client side as well as the server side:

> ***Client-side software*** **runs in the user's browser … handling the visual layout and user interaction using HTML, CSS, and JavaScript, and requesting pages, scripts, or data from the server.**
>
> ***Server-side software*** **runs on the web server … processes requests (e.g., by accessing a database or performing logic), and then sends back the appropriate resources (e.g., web pages, files, or data).**

In this course, we will learn what is involved in being …

- A ***front-end developer*** - proficient in client-side technologies (i.e., the parts of a web application that users interact with directly, such as **HTML**, **CSS**, and **JavaScript**).
- A ***back-end developer*** - proficient in server-side technologies (i.e., handles things like database interactions, server logic, **API**s, and authentication).
- A ***full-stack developer*** - has expertise across all major parts of web development, including both front-end and back-end technologies.

The **"stack"** in "full-stack" refers to the different layers or components of a web application … from the user interface down to the database. As tech stacks have become more complex, even full-stack developers often specialize in certain tools or layers while understanding the full picture.

You may not realize, but a simple request (e.g., to look at a webpage) actually requires a lot of interaction between the client and the server.

For example, assume that we want to watch some **YouTube** videos on our browser (or app). Let's walk through what happens, where the **USER** is "us", the **CLIENT** is our browser (or app) and the **SERVER** is **YouTube**'s web server:

1. **USER**: Enters www.youtube.com into the browser's address bar.
2. **CLIENT**: Contacts the Domain Name System (**DNS**) to find the IP address of **YouTube.com**.
3. **CLIENT**: Starts a connection to the server using the **HTTPS** protocol, using the IP address.
4. **CLIENT**: Sends an **HTTP** request to the **SERVER** saying: "Please send me the homepage"
5. **SERVER**: Receives the request, checks details like our location, account status, preferences, etc., and gathers the needed web page content.
6. **SERVER**: Sends back the response (usually includes **HTML**, **CSS**, **JavaScript**, Thumbnails, logos, images, etc.)
7. **CLIENT**: Builds ("renders") the web page using all the pieces it received, and displays it in the browser.
8. **LOOP:**

   CLIENT: Sends more requests for dynamic content (e.g., video thumbnails, ads, autoplay previews, personalized suggestions).

   SERVER: Receives each request and sends back more data as needed.

   ENDLOOP

Requests for information are usually initiated by user actions or client-side **JavaScript**. In the above example, it was the action of pressing the ENTER key (after typing in the **URL** into the browser's address bar) that initiated the communication with the server.

Of course, for client and server communication to work successfully, both sides must agree on how to communicate and how to interpret each other's requests and responses. This is why we use standard communication protocols (like **HTTP**), consistent naming and **URL** structures, and clearly defined web **API**s (***Application Programming Interfaces***) to ensure both sides understand what to send and expect.

Example

Let's look at an example of what is involved in understanding communication protocols. Consider going to a bank to perform some transactions. There are some things that need to be decided:

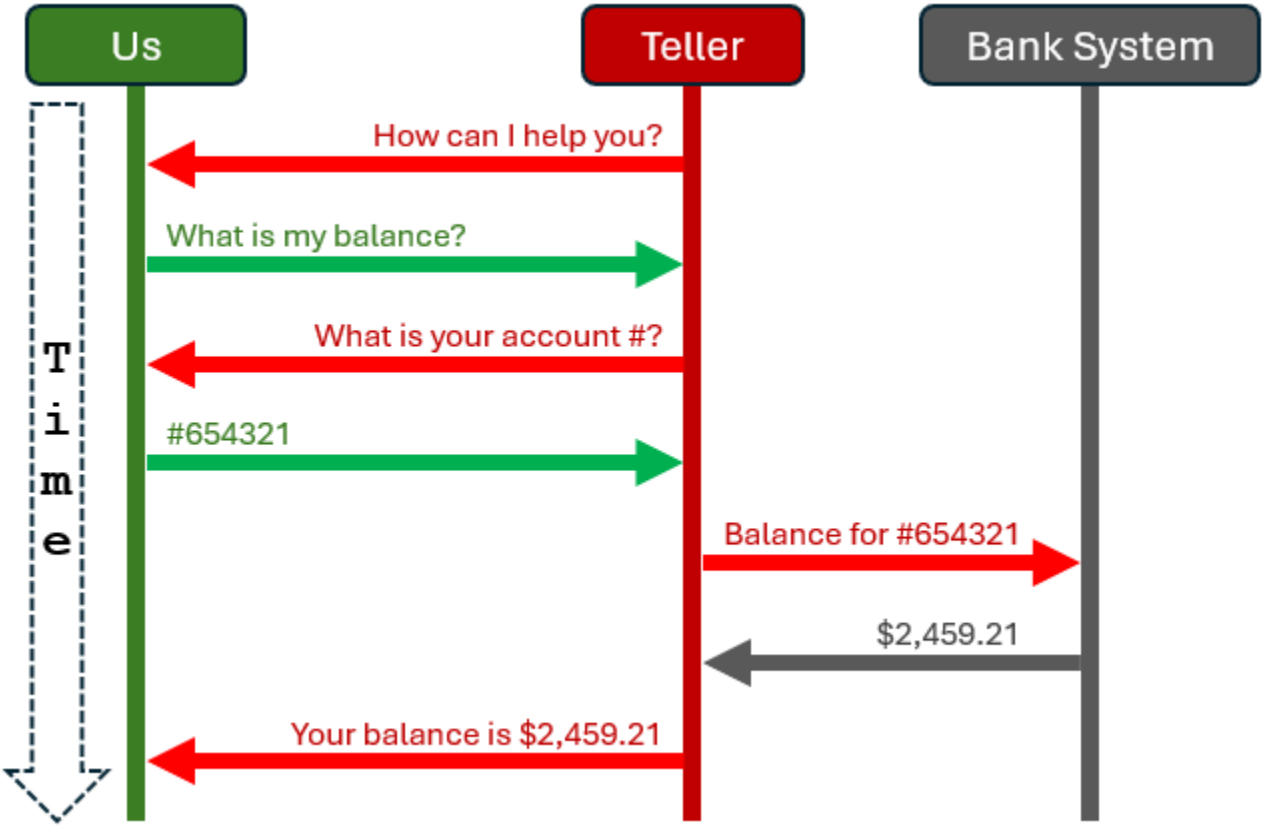
What kinds of transactions will we perform?

- deposit, withdraw, get a bank card, take out a loan, update personal info, etc..

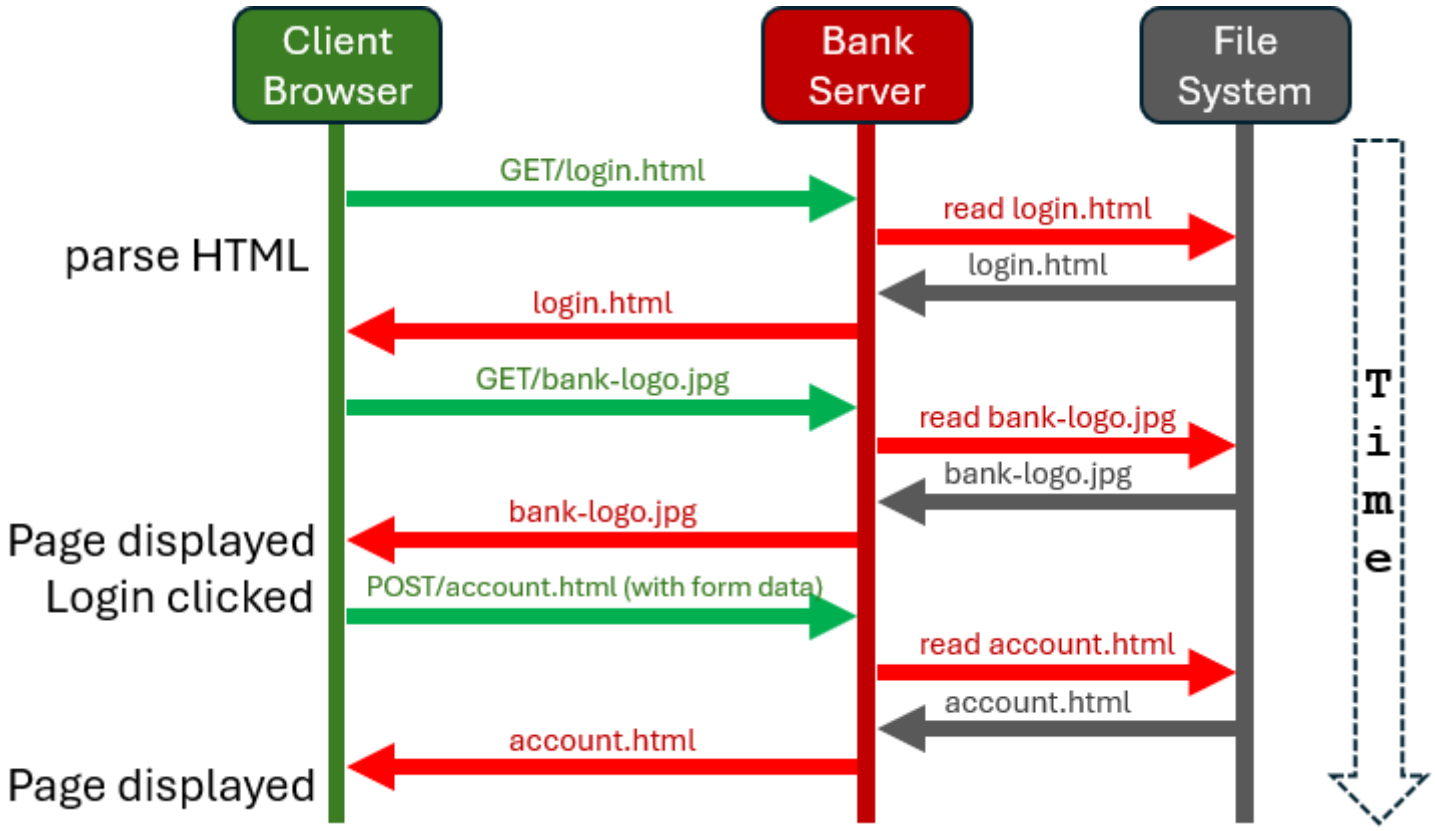
What kind of information will the bank employee want from us?

- proof of identity, account numbers, amounts, etc..

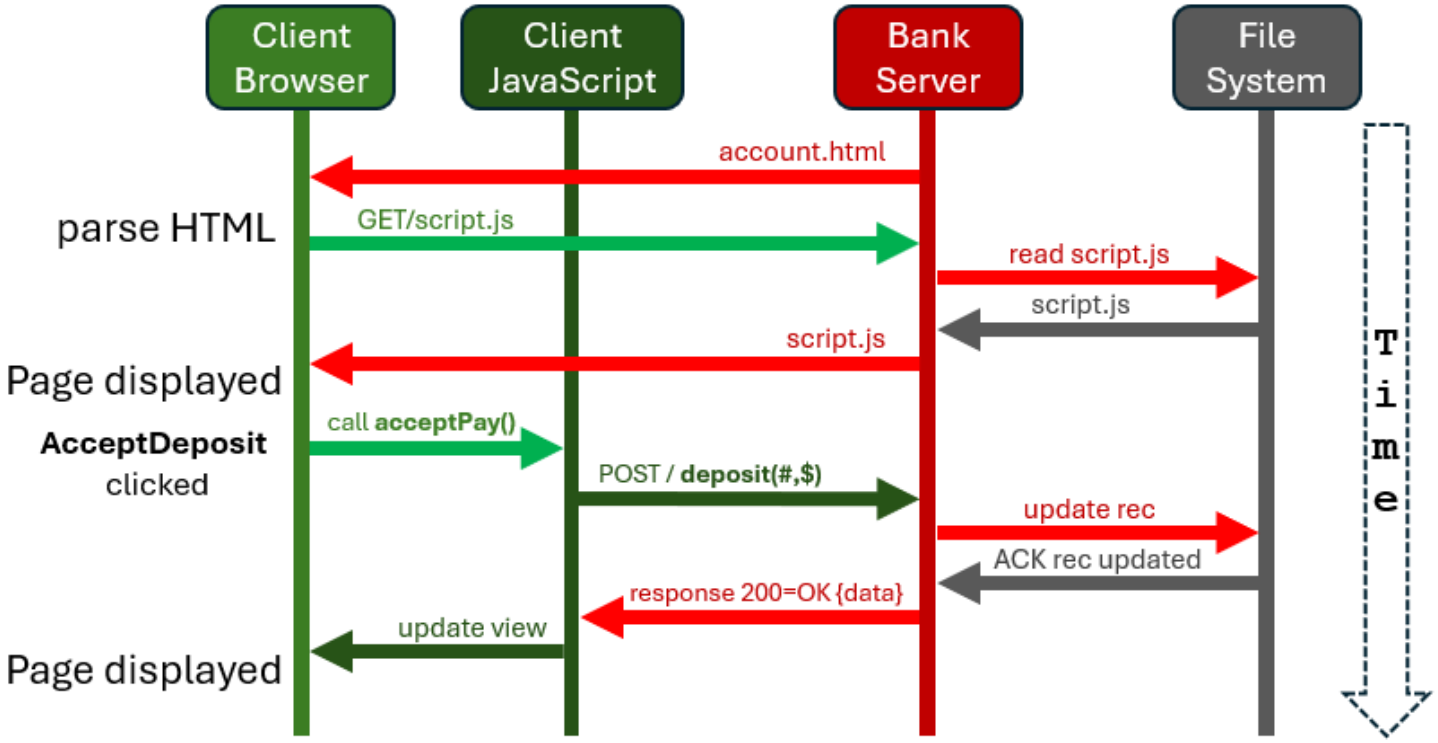
Our interaction with the bank employee can be captured by the following diagram:



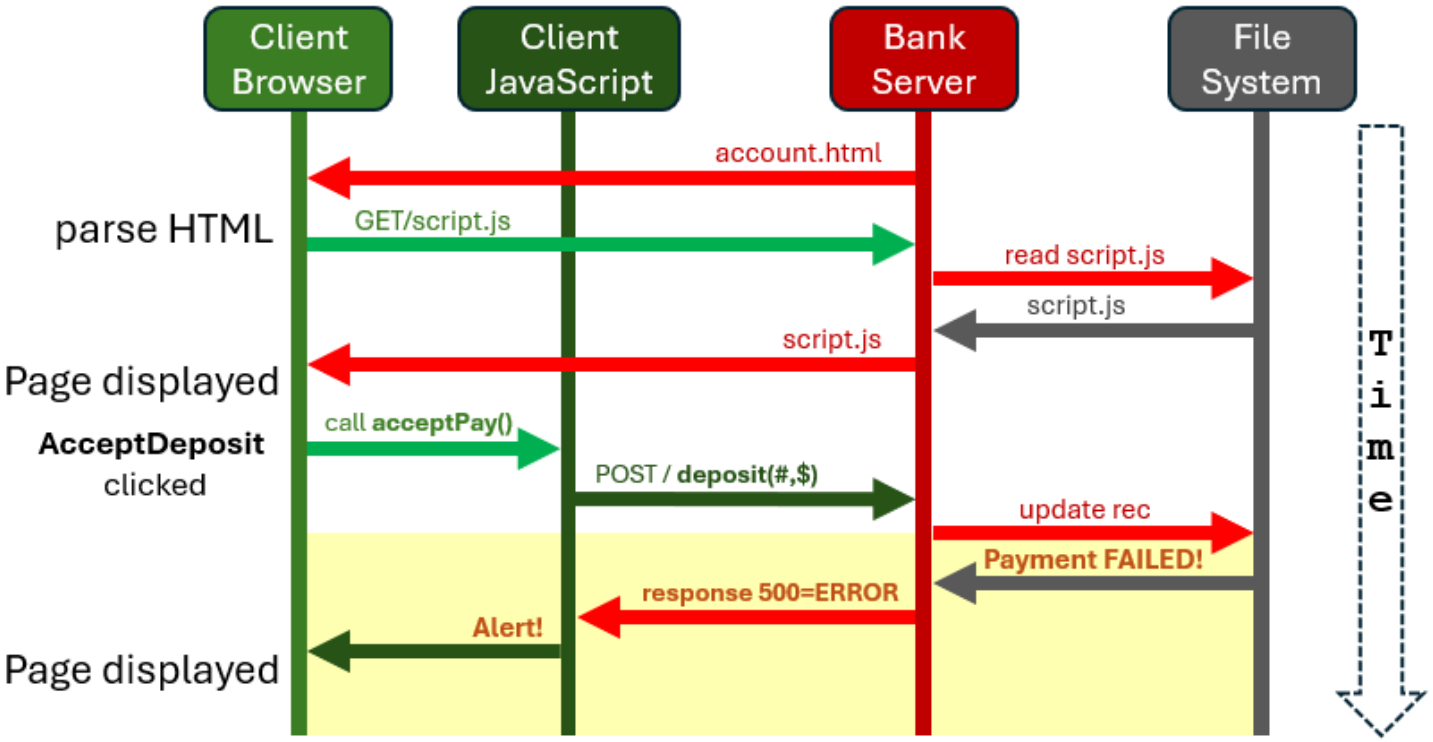
Through the browser, we end up with the same back-and-forth kinds of communication. When logging in, for example, we would see this kind of interaction:



Sometimes a request is triggered by user interaction (e.g., clicking an **Accept Payment** button):



Of course, things could fail …



The point is, transactions are asynchronous and must deal with unexpected results.

There is a lot to think about when it comes to the design of client-side & server-side software:

What Resources Are Necessary?

- define which parts of the app the client will request (e.g., account details, transaction history, statements, or payment features).
- also includes static resources (like **HTML**, **CSS**, **JavaScript**) and dynamic data (like balances or alerts).

What Should Be the Naming Scheme?

- The **URL**s or **API** endpoints need to follow a clear and consistent naming pattern so that the system is kept organized, predictable, and easy to maintain (e.g., `/api/accounts`, `/api/transactions`, `/api/users/login`)

What Are the Data Requirements?

- define what data the client needs to send or receive so that the client and server understand each other clearly and securely
  - input formats (e.g., `amount: number`, `accountID: string`)
  - required fields
  - validation rules
  - **JSON** structures for requests/responses

How do we Handle the Requests and Responses?

- Need to determine:
  - What **HTTP methods** to use (e.g., **GET**, **POST**, etc.)
  - How to structure **responses** (e.g., with status codes, success/error messages)
  - How to handle **errors, timeouts, or failed transactions**
  - How to **authenticate and authorize** users (e.g., using tokens or sessions)

We should also think of the resources the bank will provide. For example, we can use the naming scheme shown on the left for transactions described on the right:

`/`	- **Home** page
`/login`	- **Login** page (UI)
`/api/login`	- **API** endpoint to handle login
`/api/accounts/{id}`	- **Account info** for a specific user
`/api/accounts/{id}/deposit`	- **Deposit** money into an account
`/api/accounts/{id}/withdraw`	- **Withdraw** money from an account
`/api/transactions`	- **View** or create **transaction** records

We will talk more about this later.

6.3 The Hypertext Transfer Protocol

For the average internet user, **HTTP** appears at the start of a web address (e.g., `http://`) in the browser's address bar, indicating how the browser should communicate with the website's server. Every time we click a link, submit a form, or visit a site, **HTTP** works behind the scenes to handle that connection. It is the foundational protocol of the web, enabling clients to make requests and servers to respond, facilitating all communication and data exchange online.

HTTP uses a request/response model:

> ***An HTTP request is a message sent from a web browser to the web server, usually asking for a resource such as an HTML file, image, CSS stylesheet, JavaScript file, or video.***
>
> ***The HTTP response is the message the server sends back, often containing the resource we requested.***

HTTP is ***stateless***, meaning the server doesn't remember anything about us or our previous requests each time we interact with it. Every time we send a request, all the information the server needs to understand and respond must be included within that request … nothing is carried over from before.

Like most things in life, there are advantages and disadvantages of statelessness:

- ✓ **Simplicity** - Since the server doesn't have to keep track of past interactions, the protocol stays simple and easier to implement.
- ✓ **Scalability** - Servers can handle many more requests since they don't need to store session data, making it easier to distribute requests across multiple servers.
- ✓ **Resilience to Failure:** - The server isn't concerned with the client once a request is completed, so failures in one request don't impact others, improving overall stability.
- ✓ **Reliability** - With no stored state, there's less risk of errors related to outdated or lost session information, making each request independent and predictable.

- × **Extra Data in Each Request:** Since the server doesn't remember previous interactions, clients have to send ALL necessary information with EVERY request, which can add overhead.
- × **More Complex Client-Side Handling:** To maintain things like user login sessions or shopping carts, extra mechanisms (e.g., cookies, tokens) need to be added on top of **HTTP**.
- × **Less Efficient for Some Applications:** Statelessness can make certain tasks less efficient because the server can't rely on stored context and must process each request fully on its own.

HTTP transmits data in plain text, meaning that any information sent between the browser and the web server (e.g., login credentials, personal details, or financial data) can be intercepted and read by third parties. This makes **HTTP** inherently insecure, especially on public or untrusted networks (e.g., Wi-Fi in a coffee shop or airport).

HTTPS (Hypertext Transfer Protocol Secure) adds a critical layer of security by using **SSL/TLS** encryption to protect data during transmission. This encryption ensures that:

- The data is **confidential** - third parties can't read it.
- The data is **authentic** - it hasn't been tampered with.
- The communication is **verified** - the server is who it claims to be.

By using **HTTPS**, sensitive information such as passwords, credit card numbers, account details, and personal messages is securely encrypted. This protects users from eavesdropping, man-in-the-middle (MITM) attacks, and data breaches. Modern browsers also warn users when they're visiting a site that uses **HTTP** instead of **HTTPS**, particularly if the page contains login forms or other sensitive inputs.

For now, we will just discuss **HTTP**.

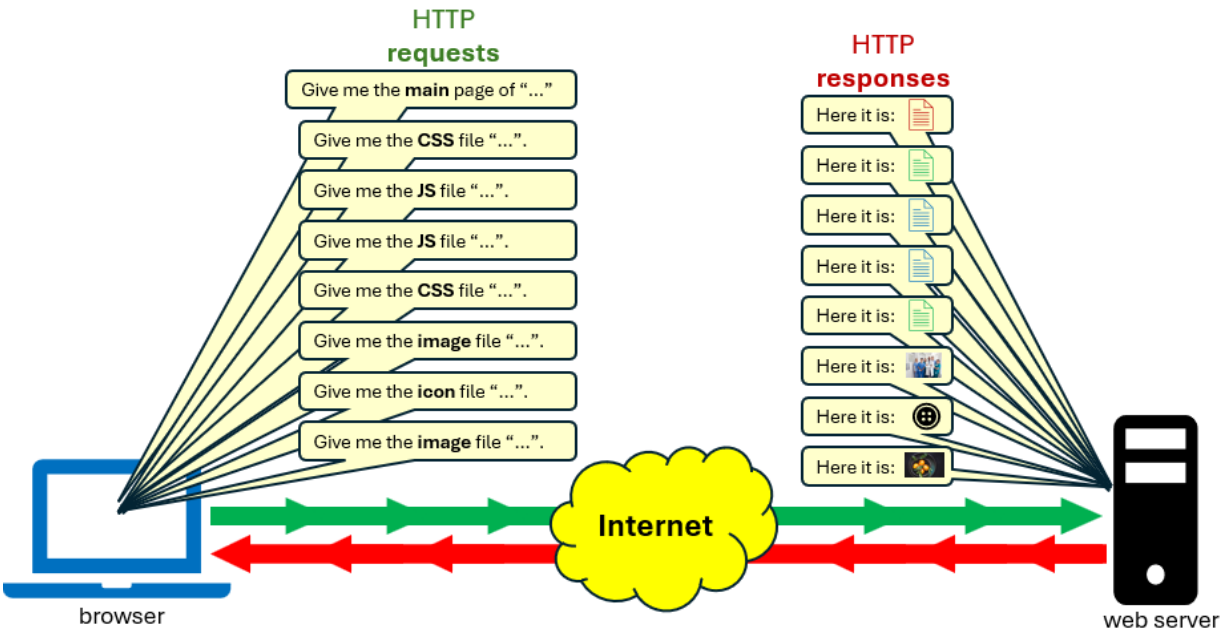
An **HTTP request** or **response** consists of two main parts: a ***header*** and an optional ***body***. While the headers are always plain text, the body can contain either plain text or binary data depending on the type of content being transmitted. Plain text is easy for humans to read, which makes debugging web communications straightforward. However, this also means that anyone who intercepts the data can read it, including people with bad intentions. **HTTPS** can be used to make transmission safe.

The ***header*** provides key information that helps both the client and server understand how to process the message.

The ***request's body*** is optional (e.g., not used for a `GET` request), but it is typically included when the client (i.e., browser or app) needs to send data to the server. This may be submitted form data from the user, **JSON** data sent to an **API**, or file uploads.

The ***response's body*** is also optional, but it's typically present when the server sends back content to the client. This may be in the form of a webpage (e.g., **HTML**), data (e.g., **JSON**, **XML**), or files (e.g., images, videos, documents).

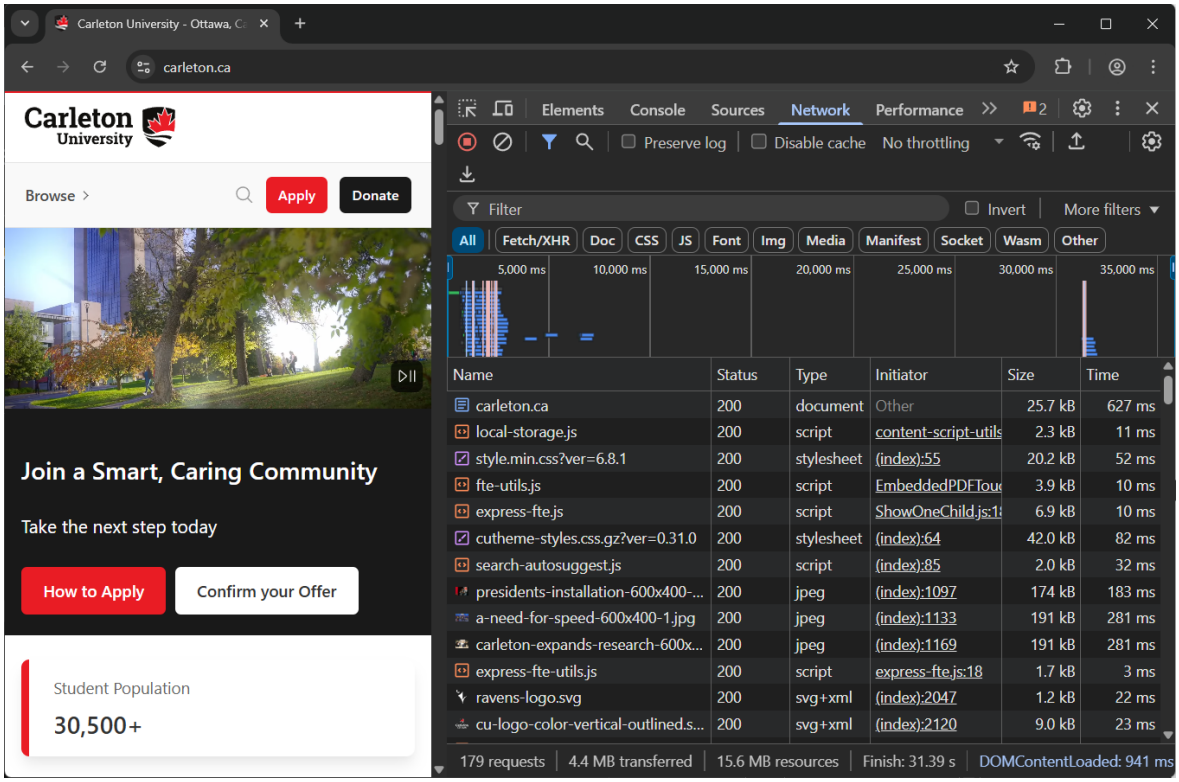
The most common example of interaction is when we visit a website by typing a web into the address bar and pressing ENTER. The browser sends an **HTTP request** message to the web server, typically asking for the main **HTML** of that page (e.g., **index.html**). This initial request usually has an empty body. The server responds with an **HTTP response** where the body contains the **HTML** content of the webpage and the header indicates the content (i.e., `Content-Type: text/html`). After receiving the **HTML**, the browser parses it and discovers additional resources it needs, such as **CSS** stylesheets, **JavaScript** files, images, and videos. It then sends separate **HTTP requests** for EACH of these resources. This creates a flurry of back-and-forth communication between the browser and the server, all happening in the background, just from the simple action of visiting a website:



Let's look at an example. Follow these instructions:

1. Using the google chrome browser, lets go to the **Carleton.ca** website
2. Right click to get the context menu … the select **Inspect** to open the Web Dev tools.
3. Select the **Network** tab on the top row of tabs.
4. Press the **Shift** key while clicking the **reload** button to the left of the address bar.

We will see a lot of things happening quickly, but it will settle after a few seconds. We should see something like this screen snapshot if we scroll the vertical scroll bar to the top:



It shows a list of all the requests that were sent to the server, the first being the request for the **carleton.ca** webpage. Then we will see requests for various **JavaScript** files, style sheets and images … **155** requests in total at the time these notes were created.

The **Status** column tells us how the request turned out (e.g., **200** = the request was successful, **204** = there was no content, **302** = the resource is temporarily located at a different **URL**, etc..).

The **Type** column indicates the type of "thing' returned. Other information includes the **Size** of the "thing" returned, how much **Time** it took to load etc.. Regarding loading, there is a timeline graphic above the list that gives us an idea as to how long it takes to load everything.

We can click on a request to examine the header. Try this:

1. Scroll down to the **favicon-32x32.png** file (hopefully the page has not changed since these notes were written).
2. Scroll down to the **Request Headers** section.
3. Select the **Raw** checkbox (you may have to 1st check off the **Disable cache** under the top menu bar).

We will see the raw request data that will look something like this:

```
GET /rds/assets/favicons/favicon-32x32.png HTTP/1.1
Accept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9
Cache-Control: no-cache
Connection: keep-alive
Cookie: _tt_enable_cookie=1; _ttp=pzoF7Jc82dR9h1G73JfnFUVV7at.tt.1;
_ga_QZQG9H9WS9=GS1.2.1736453813.3.1.1736454206.60.0.0;
_ga_JNCSQWNYYS=GS1.1.1741365158.4.0.1741365158.0.0.0; _gcl_au=1.1.465567102.1751743898;
_scid=B3VAnXyYCT1ZWve80dMu2RcKDBnFi3KhnMs7Hg; _ga=GA1.1.698271906.1712855254; _ScCbts=%5B%5D;
_fbp=fb.1.1751743898858.683354044531652541; _sctr=1%7C1751688000000; cookieConsent=true;
TS012103f9=0169a005069a5ddbd0fbd8cb53593f6584ed5e888a23d42adb4e9c562b2672b1fbcc3714b079479d00a3cbcc
f79999626dfc913090; _scid_r=AnVAnXyYCT1ZWve80dMu2RcKDBnFi3KhnMs7KA;
ttcsid=1751743898723::67y9Gj0cOYgbQ22CtFD1.1.1751744881344;
_ga_GPRE4N72YN=GS2.1.s1751743898$o5$g1$t1751744881$j42$l0$h0;
_ga_BDW6WNFPJQ=GS2.1.s1751743898$o4$g1$t1751744881$j42$l0$h0;
_ga_755K5XM3HX=GS2.1.s1751743898$o1$g1$t1751744881$j42$l0$h0;
_ga_9NTBQTVYKG=GS2.1.s1751743898$o4$g1$t1751744881$j42$l0$h0;
ttcsid_CJNP7KJC77U5TJETM050=1751743898722::xG8sQVePVXHVlRvdzqe_.1.1751744881752
Host: cdn.carleton.ca
Pragma: no-cache
Referer: https://carleton.ca/
Sec-Fetch-Dest: image
Sec-Fetch-Mode: no-cors
Sec-Fetch-Site: same-site
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/138.0.0.0 Safari/537.36
sec-ch-ua: "Not)A;Brand";v="8", "Chromium";v="138", "Google Chrome";v="138"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
```

Notice the **GET** request at the start, along with the requested favicon image. The **Accept:** line indicates the types of files the browser would like for this image. ***/*** indicates I'll accept anything and try to make it work … but the preferred image types are listed first. The **Accept-Encoding:** line lets the browser tell the server the types of encodings it can understand and handle.

Now, scroll up a bit and select the **Response Headers** and check the **Raw** checkbox. Notice … `200 OK` was returned, indicating that all went well. Another common result could be `404 Not Found` … if we, for example, spelled the file name wrong. The `Last-Modified:` line tells the client when the requested resource (in this case the favicon image) was last changed on the server. The browser can use this information for caching or conditional requests.

If we hit the browser's **reload** button again (without the **Shift** pressed), then click on the **Status** heading at the top of the list, it will sort by status. Uncheck the **Disable cache** under the top menu bar if it is checked off. Scroll to the bottom and we will see some **304** status values … indicating that the resource hasn't changed (i.e., **Not Modified**) since the last request (i.e., it was locally cached), and so it is telling us that it did not need to be reloaded (saves time). If we reload again with the **Shift** key pressed, it will go back to **200** again, because the **Shift+reload** indicates a forced reload of everything, regardless of whether it was cached.

Over time, we will get to know a little more about the attributes shown here. Browse around a little and see if we can make sense of some of the attributes and interactions going on.

6.4 Domain Name System (DNS) and URLs

DNS stands for ***Domain Name System***. It's like the phonebook of the internet. When we type a website address (e.g., *www.google.com*) into our browser, our computer doesn't know where that is. The **DNS** helps translate that human-friendly name into an IP address (e.g., **142.250.73.142**) that computers use to find each other on the internet.

As a developer, when we put a website online, it is hosted on a server with an IP address. But users don't want to type numbers to visit our site. **DNS** lets people visit our site using a domain name (e.g., my-amazing-site.com) which is not some kind of confusing IP address. It is nice to have as well because if we ever move our site to a new hosting provider, we don't have to change our domain. We just update the **DNS** records to point to the new server. The domain stays the same for users.

When a client (usually a web browser) sends an **HTTP** request to a server, it needs to know the server's IP address. Here is where the **DNS** "fits in" in the context of an **HTTP request**:

1. The user enters a **URL** (e.g., https://www.google.com) into the browser.
2. **DNS lookup** occurs to resolve www.google.com to an IP address (e.g., **142.250.73.142**).
3. The browser opens a **TCP connection** (or possibly **TLS** for **HTTPS**) to that IP address.
4. An **HTTP request** is sent over that connection:

```
GET / HTTP/1.1
Host: www.google.com
```

5. The server responds with an **HTTP response** (**HTML**, **JSON**, etc.).

But how does this ***DNS lookup*** (in step 2 above) actually work? Here is what goes on behind the scenes:

1. The user types in a website (e.g., google.com) into their browser.
2. The browser checks its local cache. If the IP address is already known (from a recent visit), it uses that right away.
3. Otherwise, it asks the ***DNS resolver*** (usually set by our Wi-Fi, ISP, or a public **DNS** like Google's 8.8.8.8)
4. The resolver checks the ***DNS hierarchy***:
   - **Root server** → points to the right Top-Level Domain (**TLD**) server (for **.com**, **.org**, etc.)
   - **TLD server** → points to the domain's authoritative name server
   - **Authoritative name server** → holds the actual IP address for the domain
5. The IP address is returned to the browser.
6. The browser uses the IP to connect to the correct web server and load the website.

Intuition: This is all a bit like asking a series of people where someone's house is, starting with a general address book (root), then a neighborhood (TLD), then a specific person (authoritative server) who tells us the exact house number (IP address).

The good thing is that our computer already knows which **DNS** server to query because this information is specified in the network settings, so no manual action is required by the user.

There are millions of **DNS** servers (globally), and they are managed by different organizations. The whole system is fault-tolerant … with redundant servers, caching, anycast routing (i.e., traffic directed to nearest server), and multiple layers of fallback. So, if one server goes down, others can take over.

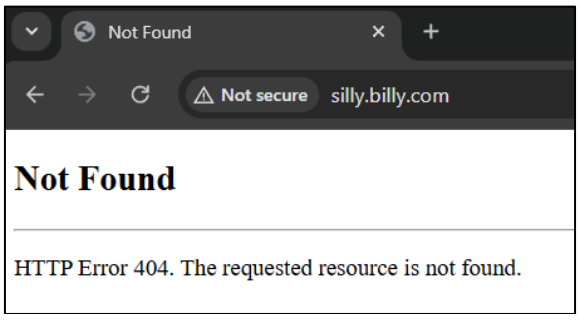
Here is a website that lets us find the IP address of a domain, server or website:

https://www.site24x7.com/tools/find-ip-address-of-web-site.html

We can go to this site to find out what our IP address is:

https://whatismyipaddress.com/

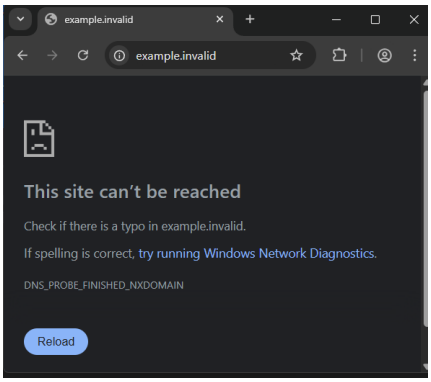
What happens if I type in a **URL** that does not have an IP address (e.g., spelling the website name wrong) …

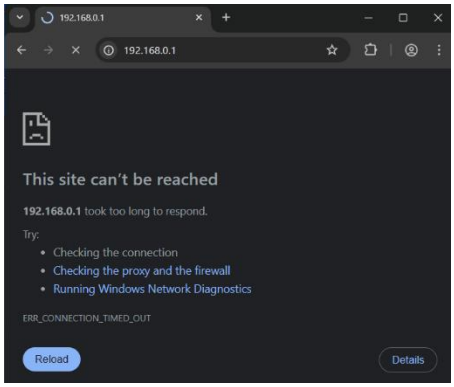


We would see the above message, for example, if we click on a link that is no longer valid. Overtime, sites go offline and links become invalid. This is called ***link rot***.

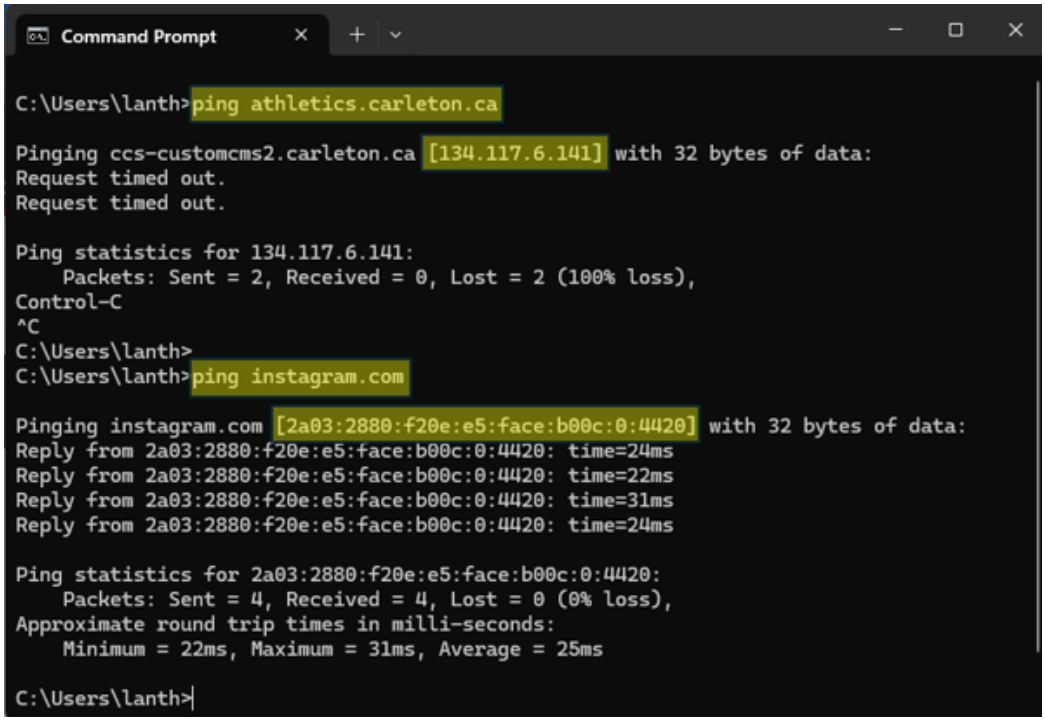
The internet Archive (https://archive.org) is one of many organizations around the world that fights link rot by archiving the web for posterity. Also, the **Wayback Machine** on that site is interesting … it can often show users what a particular **URL** looked like at different times in history.

If the **DNS** is resolved to an IP address, but the server is off line at the moment, the chrome browser will give an error message as shown here on the left. And if it is a website that is not responding quick enough, we will get something like what is shown here on the right.





Alternatively, we can open a shell window and type **`ping`** followed by the domain name:



Domain Name Structure:

Understanding how domain names are structured helps when setting up websites, managing **DNS** records, or troubleshooting domain-related issues.

Domain names are hierarchical, organized in levels separated by dots. Each dot indicates a more specific part within the hierarchy. For example, the following is a three-level domain name:

athletics.carleton.ca

- **.ca** = the top-level domain (**TLD**). It is always the rightmost part and indicates the category or location of the domain e.g., **.com**, **.org**, **.net**, **.edu**, **.gov** or country codes such as **.ca**, **.uk**

- **carleton** = the second-level domain just to the left of the **TLD**. It is usually the name of a company, organization, or project.

- **athletics** = a subdomain or third-level domain. This can be used to organize or separate different sections of a website (e.g., **athletics** a subdomain of **carleton.ca** and points to the athletics department).

URL Structure:

Let's look more into the structure of the **URL**s that we type into an address bar. Consider this:

```
https://www.example.com:443/blog/article?id=123&col=255#comments
```

This can be broken up into its components as follows:

```
https://www.example.com:443/blog/article?id=123&col=255#comments
```

Here is the explanation:

`https://` is the protocol, which determines how the request will be transmitted. (e.g., `http` or `https` for websites, `ftp` for transferring files, `file` for retrieving files from the local computer, `mailto` to create an email link that opens the user's default email app)

`www.example.com` is the domain (or host) name (may also be an IP address). (e.g., `google.ca`, `localhost`, `en.wikipedia.org`, `www.w3schools.com`)

`:443` is the port number, which allows info to be directed to a specific program within the server. If using our own, choose port numbers > **1023**. A default port is used if we do not specify one: port **80** (for **HTTP**), **443** (for **HTTPS**).

`/blog/article` is the path, which is the location of a specific page or file on the site. It is generally the first part of the **URL** our server application cares about.

`?id=123&col=255` is the query string, which is an optional collection of `key`=`value` pairs. It always starts with a `?` and pairs are separated with an `&`. This string should be "**URL** Encoded" (i.e., spaces and special characters are replaced). **JavaScript** has **encodeURIComponent(str)** to do this (note: **URI** stands for (***Uniform Resource Identifier***)).

`#comments` - is a fragment, which is an anchor (i.e., bookmark) to a specific section on the page. It is only used in the browser and is not passed to the server).

When working with **URL**s in **JavaScript**, we can create a **URL object** and access each of its parts. Here is a coding example that we can run in node.js:

```
let url = require("url");
let urlString = 'http://localhost:3000/index.html?year=2022&month=october#content';
let q = url.parse(urlString, true);     // true indicates to parse the query into an object

console.log("href: ",     q.href);      //
http://localhost:3000/index.html?year=2022&month=october#content
console.log("host: ",     q.host);      // localhost:3000
console.log("hostname: ", q.hostname);  // localhost
console.log("port: ",     q.port);      // 3000
console.log("path: ",     q.path);      // /index.html?year=2022&month=october
console.log("pathname: ", q.pathname);  // /index.html
console.log("search: ",   q.search);    // ?year=2022&month=october
console.log("hash: ",     q.hash);      // #content

let qdata = q.query;                    // returns the query portion of the URL
console.log(qdata);                     // { year: 2022, month: 'october' }
console.log(qdata.month);               // october

for (x in qdata) {
    console.log(x + ": " + qdata[x]);   // year: 2022 and month: october
}
```

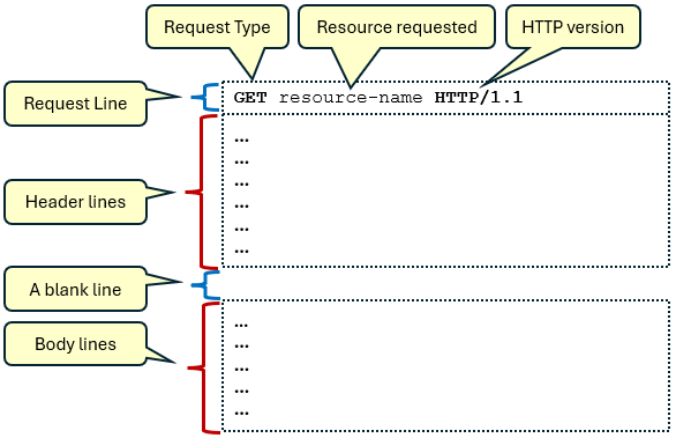
The `require()` function, at the top, will return a **URL** module object, which is built into **node.js** and contains many methods for parsing, formatting and resolving **URL** strings and objects. Two common ones are:

- `url.parse()` - parse **URL** strings (older **API**)
- `url.format()` - format **URL** objects back to strings

Notice how we can access the various attributes/components of the **URL** by simply using the dot operator followed by the attribute name. The `query` attribute returns the parse query string as a **JavaScript** object that we can iterate through.

6.5 Structure of HTTP Requests and Responses

As we discussed, all client/server communication occurs through **HTTP Requests** and **HTTP Responses**. We will dig a bit more here to see what these are made of. Each request is simply a bunch of lines of text with this format →



The first line indicates the type of request, the resource (e.g., webpage) being requested, and the **HTTP** version.

The **HTTP** request consists of a header, which contains multiple lines of information, followed by a blank line. After that, there may be a body containing textual data (usually absent in **GET** requests).

The most common request types are:

- **GET** - request data from the server (e.g., loading a webpage, searching with parameters).
- **POST** - submit data to the server (e.g., login credentials, creating a new user, uploading a file).
- **HEAD** - like **GET**, but retrieve just the headers (e.g., check if a webpage has been updated).
- **PUT** - store a new resource or replace one (e.g., updating an email address or profile pic).
- **DELETE** - remove a specific resource (e.g., deleting a post or comment, removing a file).

Let's look at some specific examples. Here is a webpage that connects to my path planning research overview page on my webpage:

https://people.scs.carleton.ca/~lanthier/research/PathPlanning/researchPath.html

When we enter this in the browser address bar and press ENTER, the following **HTTP Request** is sent to the server (we can go to Dev Tools in Chrome to see it):

```
GET /~lanthier/research/PathPlanning/researchPath.html HTTP/1.1
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp
,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9
Cache-Control: max-age=0
Connection: keep-alive
Cookie: _tt_enable_cookie=1; _ttp=pzoF7Jc82dR9h1G73JfnFUVV7at.tt.1;
_ga_QZQG9H9WS9=GS1.2.1736453813.3.1.1736454206.60.0.0;
_ga_JNCSQWNYYS=GS1.1.1741365158.4.0.1741365158.0.0.0;
_gcl_au=1.1.465567102.1751743898;
_scid=B3VAnXyYCT1ZWve80dMu2RcKDBnFi3KhnMs7Hg;
_ga=GA1.1.698271906.1712855254; _ScCbts=%5B%5D;
_fbp=fb.1.1751743898858.683354044531652541; _sctr=1%7C1751688000000;
cookieConsent=true; _scid_r=BnVAnXyYCT1ZWve80dMu2RcKDBnFi3KhnMs7LA;
ttcsid=1751743898723::67y9Gj0cOYgbQ22CtFD1.1.1751746213119;
ttcsid_CJNP7KJC77U5TJETM050=1751743898722::xG8sQVePVXHVlRvdzqe_.1.175174621
3382; _ga_BDW6WNFPJQ=GS2.1.s1751743898$o4$g1$t1751746242$j21$l0$h0;
_ga_GPRE4N72YN=GS2.1.s1751743898$o5$g1$t1751746317$j60$l0$h0;
_ga_755K5XM3HX=GS2.1.s1751743898$o1$g1$t1751746317$j60$l0$h0;
_ga_9NTBQTVYKG=GS2.1.s1751743898$o4$g1$t1751746317$j60$l0$h0
Host: people.scs.carleton.ca
If-Modified-Since: Wed, 05 Nov 2014 19:10:38 GMT
If-None-Match: "3cc5-5072155c76780"
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/138.0.0.0 Safari/537.36
sec-ch-ua: "Not)A;Brand";v="8", "Chromium";v="138", "Google Chrome";v="138"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
```

Notice the request line. The browser uses the **GET** request when accessing a webpage.

Notice next, that there are many components in the header, which consists of **key: value** pairs. We will not discus all header options, but here is an explanation of a few of the common ones:

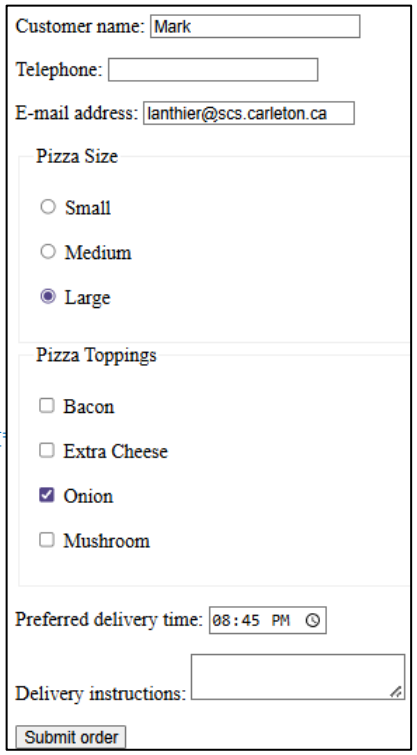
Key	Value
`Accept`	Specifies media types the client is willing to receive
`Accept-Encoding`	Informs the server which content-encoding methods are supported
`Accept-Language`	Preferred languages
`Cache-Control`	Tells caches whether to revalidate before serving
`Content-Type`	Specifies the media type of the data being sent in the **body**
`Content-Length`	Indicates the size (in bytes) of the request **body**
`Host`	Specifies the domain name of the server being requested
`User-Agent`	Identifies the browser (or client) and OS making the request

Since this was a **GET** request, there is no body. Therefore, we do not see **key: value** pairs for **`Content-Type`** nor **`Content-Length`**.

Go here https://www.iana.org/assignments/http-fields/http-fields.xhtml for a more thorough list of header fields.

Now let's try a **POST** request. We will go to this test website and fill out a few values in a form and then submit the form:

https://httpbin.org/forms/post



Here is what was sent to the server (check Request Headers in Dev Tools):

```
:authority                  httpbin.org
:method                     POST
:path                       /post
:scheme                     https
Accept                      text/html,application/xhtml+xml,application/
                            xml;q=0.9,image/avif,image/webp,image/apng,
                            */*;q=0.8,application/signed-exchange;v=b3;q
accept-encoding             gzip, deflate, br, zstd
accept-language             en-US,en;q=0.9
cache-control               max-age=0
content-length              111
content-type                application/x-www-form-urlencoded
origin                      https://httpbin.org
priority                    u=0, i
referrer                    https://httpbin.org/forms/post
sec-ch-ua                   "Chromium";v="140", "Not=A?Brand";
                            v="24", "Google Chrome";v="140"
...
upgrade-insecure-requests   1
user-agent                  Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
                            (KHTML, like Gecko) Chrome/140.0.0.0 Safari/537.36
```

The **Content-Type:** is now set to **application/x-www-form-urlencoded**, which indicates it is **URI**-encoded text. The **Content-Length:** is also set to the total encoded characters of **111**.

And here is the **body** of the request now (check Payload in Dev Tools):

```
custname=Mark&custtel=&custemail=lanthier%40scs.carleton.ca&size=large&topping=onion&
delivery=20%3A45&comments=
```

Notice the encoding puts the **key**, an **=** character and the **value** for each form field and that the data for each field is separated by a **&** character.

The **Content-Type** helps the receiving end know how to interpret and handle the data properly. Here are some common content types:

text/plain - plain text
text/html - **HTML** documents (webpages)
text/css - **CSS** stylesheets
text/javascript - **JavaScript** code

image/png - PNG images
image/jpeg - JPEG images
image/gif - GIF images
image/svg+xml - SVG vector images

application/javascript - **JavaScript** code
application/json - JSON data
application/xml - XML data
application/pdf - PDF documents
application/octet-stream - generic binary data
application/x-www-form-urlencoded - form data

audio/mpeg - MP3 audio
video/mp4 - MP4 video

It is also possible to send data along with a **GET** request. We would just need to append the **URI**-encoded text after the resource with a **?** in between:

```
GET /post?custname=Mark&custtel=&custemail=lanthier%40scs.carleton.ca&
size=large&topping=onion&delivery=20%3A45&comments= HTTP/1.1
Host: httpbin.org
Accept:text/html
User-Agent: Mozilla/5.0
etc..
```

However, some old browsers do not support **URL**s with more than **~2000** characters. So, even though we can use **GET** to send data (if we are writing the client code), it is generally not a good idea to do that. Instead, follow these guidelines:

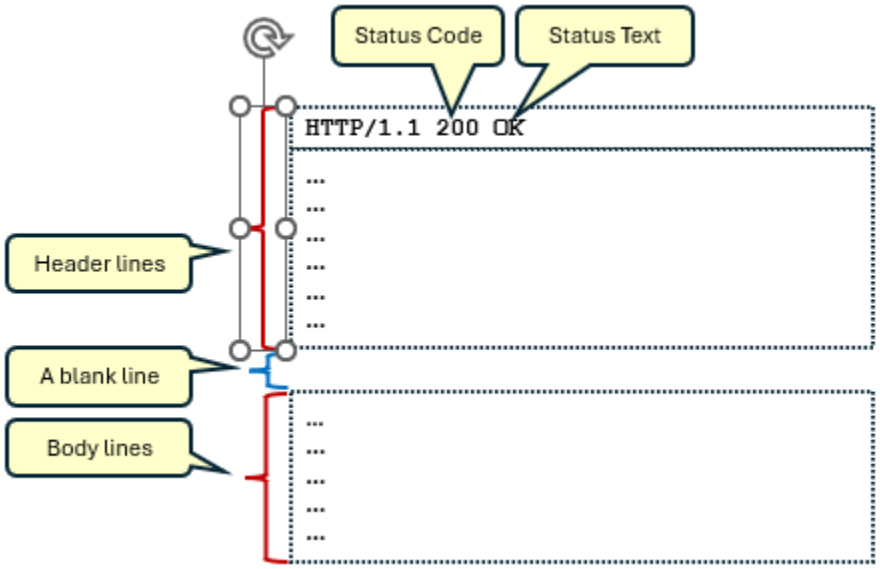
Use **GET** to:

- Fetch or read data. We can include small amounts of non-sensitive data (be aware that the **URL** line is visible to everyone).
- Make requests that can be bookmarked or cached and remain in the browser history. **POST** requests are never cached, cannot be bookmarked, and do not remain in the browser history.

Use **POST** to:

- Send sensitive data or large amounts of data (e.g., login credentials, file uploads).
- Perform operations that modify server state (create/update/delete).

Now, what does an **HTTP Response** look like? It has a similar format to a request being sent, except that there is no request line. Instead, the first line of the header indicates a **Status Code** and a **Status Text** that explains what happened.



For our **GET** request, we were requesting an **HTML** page that represents my path planning research overview page on my webpage. So, the response will look like this:

```
HTTP/1.1 200 OK
Date: Tue, 30 Sep 2025 00:24:22 GMT
Server: Apache
Last-Modified: Wed, 05 Nov 2014 19:10:38 GMT
ETag: "3cc5-5072155c76780"
Accept-Ranges: bytes
Content-Length: 15557
Connection: close
Content-Type: text/html; charset=UTF-8

<!DOCTYPE doctype PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html> … </html>
```

Notice that the body is the entire **HTML** page requested (I cut out everything between the **<html>** tags to save space). Also notice that the **Content-Type:** is set to text/html and that the **Content-Length:** is the size of the **HTML** file.

Now what about the **HTTP** Response from our **POST** to https://httpbin.org/forms/post? Here is what it looks like:

```
HTTP/1.1 200 OK
Access-Control-Allow-Credentials: true
Access-Control-Allow-Origin: https://httpbin.org
Content-Length: 1414
Content-Type: application/json
Date: Tue, 30 Sep 2025 00:08:40 GMT
Server: gunicorn/19.9.0

{
  "args": {},
  "data": "",
```

```
  "files": {},
  "form": {
    "comments": "",
    "custemail": "lanthier@scs.carleton.ca",
    "custname": "Mark",
    "custtel": "",
    "delivery": "20:45",
    "size": "large",
    "topping": "onion"
  },
  "headers": {
    "Accept":
"text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q
=0.8,application/signed-exchange;v=b3;q=0.7",
    "Accept-Encoding": "gzip, deflate, br, zstd",
    "Accept-Language": "en-US,en;q=0.9",
    "Cache-Control": "max-age=0",
    "Content-Length": "111",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "Origin": "https://httpbin.org",
    "Priority": "u=0, i",
    "Referer": "https://httpbin.org/forms/post",
    …
  },
  "json": null,
  "origin": "174.114.92.200",
  "url": "https://httpbin.org/post"
}
```


Notice that the response body is a **JSON** object that contains the form data within it. It also contains header information from the request that we sent as well as the **URL** used etc..

When it comes to getting responses, the status codes tell the browser (i.e., the client) what happened. Here is a table showing common **Status Codes**, **Status Text** and their **Descriptions**:

Informational:

100	Continue	Client may continue with request
101	Switching Protocols	Server is switching protocols

Success:

200	OK	Request succeeded
201	Created	Resource successfully created
202	Accepted	Request accepted for processing
204	No Content	Request succeeded, no content returned

Redirection:

301	Moved Permanently	Resource has been moved permanently
302	Found	Temporary redirect
303	See Other	Redirect for **GET** after **POST**
304	Not Modified	Resource hasn't changed (cached version valid)
307	Temporary Redirect	Temporary redirect (method must not change)
308	Permanent Redirect	Permanent redirect (method must not change)

Client Errors:

400	Bad Request	Request malformed or invalid
401	Unauthorized	Authentication required or failed
403	Forbidden	Authenticated, but access is not allowed
404	Not Found	Resource not found
405	Method Not Allowed	Method is not allowed on this resource
408	Request Timeout	Client took too long to send request
429	Too Many Requests	Client sent too many requests (rate limit)

Server Errors:

500	Internal Server Error	Generic server error
501	Not Implemented	Method not supported by server
502	Bad Gateway	Invalid response from upstream server
503	Service Unavailable	Server is overloaded or down
504	Gateway Timeout	Upstream server didn't respond in time

Look here https://www.restapitutorial.com/httpstatuscodes or here https://en.wikipedia.org/wiki/List_of_HTTP_status_codes for a more thorough list.

What will the response look like if I go to a page that does not exist:

www.carleton.ca/junk

Below is the response. Notice the **404** response. Also notice that there is a **body** returned … which is an **HTML** document that will be displayed on the client browser:

```
HTTP/1.1 404 Not Found
Date: Tue, 30 Sep 2025 00:44:13 GMT
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Cache-Control: no-cache, must-revalidate, max-age=0, no-store, private
Link: <https://carleton.ca/wp-json/>; rel="https://api.w.org/"
X-Content-Type-Options: nosniff
Strict-Transport-Security: max-age=31536000; includeSubDomains
Content-Type: text/html; charset=UTF-8
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked

<!doctype html>
<html class="no-js" lang="en-US">
    <head>
        <meta charset="UTF-8"/>
        <!-- Prefetching dns's -->
        <link rel="dns-prefetch" href="//ajax.googleapis.com"/>
        <link rel="dns-prefetch" href="//google-analytics.com"/>
        <link rel="dns-prefetch" href="//www.google-analytics.com"/>
        <!-- FavIcon -->
…
```

6.6 URL Shortening & Caching

URL shortening is the process of taking a long, complex **URL** and converting it into a much shorter and simpler one that redirects to the original **URL**. There are benefits to doing this:

- **Easier to share:** Short **URL**s are easier to copy, paste, tweet, or text, especially when space is limited (like on Twitter).
- **Cleaner links:** Long **URL**s with lots of parameters can look messy; short ones look neat and professional.
- **Tracking:** Many **URL** shorteners offer click analytics (how many times the link was clicked, where, when, etc.).

How does it work?

1. We take a long **URL** like this: https://www.example.com/articles/2025/07/07/how-to-use-url-shortening-effectively?utm_source=newsletter&utm_campaign=july

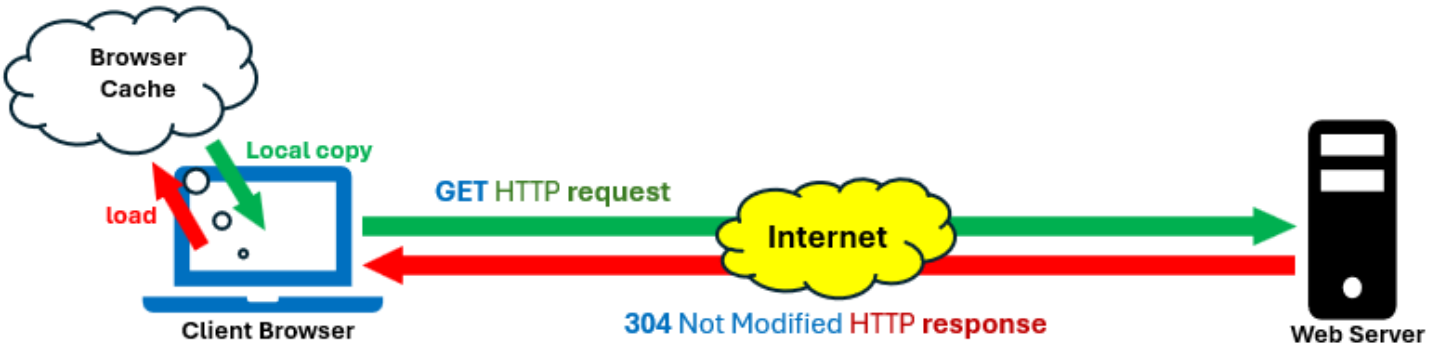
2. We submit it to a **URL** shortening service (e.g., **bit.ly** or **tinyurl.com**).

3. The service generates a unique short code (e.g., https://bit.ly/3aBcD9X)

4. When someone clicks on the short **URL**, they get redirected (via an **HTTP** Response with a `301 Moved Permanently` or `302 Found` redirection code) to the original long **URL**.

The shortening service maintains a database mapping short codes to original long **URL**s. The short **URL** typically has this format: `[domain]/[shortcode]`. So, when accessed, the service looks up the `shortcode` and sends the browser to the original **URL**.

Behind the scenes, browsers will use caching to cut down on the amount of information being sent to and from the server. For example, if the browser requests a webpage using **GET** and the server indicates that the page has not been modified since the last time it was accessed by that browser, then the browser can simply load the local copy from its cache. This reduces the need for the server to send the exact same webpage back as the local cache version.



Caching can significantly reduce the amount of data transmitted between the server and client, leading to faster load times. Static resources (e.g., images, **CSS** stylesheets, and **JavaScript** files) typically do not change often.

Once these assets have been downloaded, it's inefficient to resend them with every subsequent request if they haven't changed. Additionally, if the same image is used across multiple pages of a website, caching allows the browser to reuse the image without re-downloading it, improving overall performance.

HTTP headers such as **Cache-Control**, **Last-Modified**, **Expires** and **ETag** control how long and under what conditions resources are cached. Here are a few options for the **Cache-Control** key:

Directive	Description
`public`	content can be cached by any cache (browser, ContentDeliveryNetwork, etc.)
`private`	content is intended for a single user & must not be stored by shared caches
`no-cache`	cache must check with the server before using the cached copy
`no-store`	do not store any part of the response or request (used for sensitive data like banking or login pages)
`max-age=N`	specifies how long (in seconds) content can be reused before it must be revalidated (e.g., max-age=3600 … which is 1 hour)
`must-revalidate`	cache must revalidate the content with the origin server after it becomes stale
`no-transform`	intermediate caches or proxies should not modify the content (e.g., image compression)

The **Last-Modified** header specifies the date and time that the content was last changed, while the **Expires** header indicates when the content should be considered no longer fresh and must be revalidated or re-fetched. (Note: **Expires** is now mostly replaced by the more flexible **Cache-Control: max-age** in modern **HTTP**/.)

Conditional headers are **HTTP** headers that let clients make requests based on the state of a resource. They are used for caching efficiency, optimistic concurrency control, and bandwidth savings. Here are some:

Header	Method	Description
`If-Modified-Since`	**GET**	Only returns the resource if it was modified after the specified date (works with **Last-Modified**)
`If-Unmodified-Since`	**PUT, DELETE**	Performs the action only if the resource has not been modified since the given date
`If-None-Match`	**GET, HEAD**	Returns the resource only if the **ETag** does not match (used to validate cache)

If-Match	**PUT, DELETE**	Proceeds only if the **ETag** matches (prevents conflicting updates)
If-Range	**GET (with Range)**	Sends a partial response only if the resource is unchanged since the provided ETag or date

The **ETag:** (entity tag) mentioned in the above table, is a unique identifier (i.e., an ID). When our browser caches a file, it stores the **ETag**. Later, instead of re-downloading the whole resource, the browser can ask **If-None-Match:** `"3cc5-5072155c76780"` in its request. If the content has not changed (i.e., the modified date has not changed and the **ETag** is the same), then the browser is informed with a **304 Not Modified** status code and text to inform it that it can used the cached version. This saves bandwidth by avoiding unnecessary downloads and improves speed by leveraging the browser cache.

As an example, assume that we sent this **HTTP** Request for a **logo.png** file:

```
GET /logo.png HTTP/1.1
If-Modified-Since: Tue, 01 Jul 2025 10:00:00 GMT
```

If the image has not changed since that date, the server responds with:

```
HTTP/1.1 304 Not Modified
```

If it has changed, the new image is sent with a status of **200 OK**.

We will talk much more about **HTTP** requests and responses after we discuss **Node.js** in a later chapter.