Browser Basics

# **1 What is Browser? - Browser**

I want to open this course with a chapter aimed at understanding what browsers do and a brief explanation on how they work. Since most of your customers will interact with your web application through a browser, it’s imperative to understand the basics of these wonderful programs.

The browser is a rendering engine. Its job is to download a web page and render it in a way that’s understandable to a human being.

Although this is an oversimplification, it’s all we need to know for now: the user enters an address in the browser bar and the browser downloads the “document” at that URL and renders it.

## **Different Browsers**

You might be used to working with some of the most popular browsers such as Chrome, Firefox, Edge or Safari, but that doesn’t mean that there aren’t other browsers out there. Lynx, for example, is a lightweight, text-based browser that works from your command line.

At the heart of lynx lies the same principles that you would find in any other mainstream browser. A user enters a web address (URL), the browser fetches the document and renders it; the only difference is that lynx does not use a visual rendering engine but rather a text-based interface.

# **2) What Does a Browser Do? - Browser**

A browser’s job consists of:

1. DNS resolution
2. HTTP exchange
3. Rendering
4. Rinse and repeat

## **DNS resolution**

DNS resolution makes sure that once the user enters a URL, the browser knows which server it should connect to. The browser contacts a DNS server to find that google.ae translates to 216.58.207.110 which is an IP address the browser can connect to.

## **HTTP exchange**

Once the browser has identified which server is going to serve our request, it will initiate a TCP connection and begin the HTTP exchange. This is nothing but a way for the browser to communicate to the server what it wants, and for the server to reply back.

HTTP is simply the name of the most popular protocol for communication on the web. Browsers mostly talk via HTTP when communicating with servers. An HTTP exchange involves the client, our browser, sending a request and the server replying back with a response.

## **Parsing a sample request**

For example, after the browser has successfully connected to the server behind google.com, it will send a request that looks like the following:

GET / HTTP/1.1  
Host: google.com  
Accept: \*/\*

Let’s break the request down, line by line:

* GET / HTTP/1.1: with the start line, the browser asks the server to retrieve the document at the location /, adding that the rest of the request will follow the HTTP/1.1 protocol. It could also have used 1.0 or 2.
* Host: google.com: this is the only HTTP header mandatory in HTTP/1.1. Since the server might serve multiple domains (google.com, google.co.uk, etc.) the client here mentions that the request was for that specific host.
* Accept: \*/\*: an optional header, where the browser is telling the server that it will accept any kind of response back. The server could have a resource that is available in JSON, XML, or HTML formats, so it can pick whichever format it prefers.

## **Parsing a sample response**

In this example, the browser, acting as a **client**, is done with its request; now it’s the server’s turn to reply:

HTTP/1.1 200 OK  
Cache-Control: private, max-age=0  
Content-Type: text/html; charset=ISO-8859-1  
Server: gws  
X-XSS-Protection: 1; mode=block  
X-Frame-Options: SAMEORIGIN  
Set-Cookie: NID=1234; expires=Fri, 18-Jan-2019 18:25:04 GMT; path=/; domain=.google.com; HttpOnly  
  
<!doctype html><html">  
...  
...  
</html>

Whoa, that’s a lot of information to digest! Let’s break it down.

The server lets us know that the request was successful (200 OK) and adds a few headers to the response;

* For example, it advertises which server processed our request (Server: gws), what the X-XSS-Protection policy of this response is, and so forth.

You do not need to understand each and every piece of information right now, as we will go over the HTTP protocol, its headers, and so on in their dedicated chapters. For now, all you need to understand is that the client and the server are exchanging information and that they do so via HTTP.

## **How a browser renders an HTTP response**

Last but not the least, let’s discuss the rendering process. How good would a browser be if the only thing it showed to the user was a list of funny characters?

<!doctype html><html>  
...  
...  
</html>

In the **body** of the response, the server includes the representation of the response according to the Content-Type header. In our case, the content type was set to text/html, so we are expecting HTML markup in the response, which is exactly what we find in the body. This is where a browser truly shines: it parses the HTML, loads additional resources included in the markup (for example, there could be JavaScript files or CSS documents to fetch), and presents them to the user as quickly as possible.

Once more, the end result is something the average Joe can understand:

For a more detailed version of what really happens when we hit enter in the address bar of a browser, I would suggest reading “[What happens when…](https://github.com/alex/what-happens-when)”, an elaborate explanation of the mechanics behind the process.

Since this course is focused on security, I am going to drop a hint about what we’ve just learned, attackers easily make a living out of vulnerabilities in the HTTP exchange and rendering process. Vulnerabilities and malicious users lurk in other places as well, but a better security approach on those levels already allows you to make strides in improving your security posture.

# **3) Vendors - Browser**

The four most popular browsers belong to different vendors:

* Chrome by Google
* Firefox by Mozilla
* Safari by Apple
* Edge by Microsoft

Besides battling each other to increase their market penetration, vendors engage with each other in order to improve the web standards, a sort of minimum requirements for browsers.

## **Web standards**

The W3C is the body behind the development of web standards, but it’s not unusual for browsers to develop their own features that eventually make it as web standards, security is no exception to that.

In 2016, for example, Chrome 51 introduced SameSite cookies, a feature that would allow web applications to get rid of a particular type of vulnerability known as CSRF (more on this later). Other vendors decided this was a good idea and followed suit, leading to SameSite being a web standard. Today, all major browsers support SameSite cookies, with Safari being the last to jump on the ship in late 2018.

This tells us two things:

* Safari does not seem to care enough about their users’ security (just kidding: SameSite cookies are available since Safari 12).
* Patching a vulnerability on one browser does not mean that all your users are safe.

The first point is a shot at Safari (as I mentioned, just kidding!), while the second information is really important. When developing web applications, we need to make sure that they look the same across various browsers, but also that they ensure our users are protected in the same way across platforms.

## **Cater to each browser’s vulnerabilities**

Your strategy towards web security should vary according to what a browser’s vendor allows us to do. Nowadays, most browsers support the same set of features and rarely deviate from their common roadmap. Instances like the one above still happen, and it’s something we need to take into account when defining our security strategy.

In our case, if in 2017 we decided that we were going to mitigate CSRF attacks only through SameSite cookies, we should have been aware that we were putting our Safari users at risk. Our users should have known that too.

## **Support some browsers**

Last but not least, you should remember that you can decide whether to support a browser version or not. Supporting every browser version would be impractical (think of Internet Explorer 6); making sure that the last few versions of the major browser are supported, though, it’s generally a good decision. If you don’t plan to offer protection on a particular platform, it’s generally advisable to let your users know.

**NOTE**:

Don't support outdated browsers

You should never encourage your users to use outdated browsers, or actively support them. Even though you’ve taken all the necessary precautions, other web developers might not have. Encourage users to use the latest supported version of one of the major browsers.

## **Vendor or a standard bug?**

The fact that the average user accesses our application through a third party client (the browser) adds another level of indirection towards a clear, secure browsing experience The browser itself might present a security vulnerability.

Vendors generally provide rewards (aka bug bounties) to security researchers who can find a vulnerability on the browser itself. These bugs are not tied to your implementation, but rather to how the browser handles security on its own.

## **Bug bounties**

The Chrome reward program lets security engineers reach out to the Chrome security team to report vulnerabilities they have found. If these vulnerabilities are confirmed, a patch is issued, a security advisory notice is generally released to the public and the researcher receives a (usually financial) reward from the program.

Companies like Google invest a relatively large amount of capital into their Bug Bounty programs, as it allows them to attract researchers by promising a financial benefit should they find any problem with the application.

In a Bug Bounty program, everyone wins: the vendor manages to improve the security of its software, and researchers get paid for their findings. We will discuss these programs later on in the course, as I believe Bug Bounty initiatives deserve their own chapter in the security landscape.

## **Jake discovered a browser bug!**

Jake Archibald is a developer advocate at Google who recently discovered a vulnerability impacting more than one browser: He documented his efforts, how he approached different vendors, and their reactions in an interesting blog post that I highly recommend.

# **4) A Browser for Developers- Browser**

By now, we should understand the simple but important concept that browsers are simply HTTP clients built for the average web surfer.

They are more powerful than a platform’s bare HTTP client (think of NodeJS’s require('http'), for example), but at the end of the day, they’re just a natural evolution of simpler HTTP clients.

## **Introducing cURL**

As developers, our HTTP client of choice is usually cURL by Daniel Stenberg. It is one of the most popular software programs web developers use on a daily basis. It allows us to do an HTTP exchange on-the-fly by sending an HTTP request from our command line.

curl -I educative.io

In the example above, we have requested the document at educative.io, and Educative’s server replied successfully. Rather than dump the response’s body to the command line, we’ve used the -I flag to tell cURL we’re only interested in the response headers.

Taking it one step further, we can instruct cURL to dump some more information, including the actual request it performs, so that we can have a better look at the entire HTTP exchange. The option we need to use is -v (verbose).

curl -I -v educative.io

The same information is available in mainstream browsers through their DevTools. As we’ve seen, browsers are nothing more than elaborate HTTP clients. Sure, they add an enormous number of features like credential management, bookmarking, and history but the truth is that they were born as HTTP clients for humans. This is important because in most cases you don’t need a browser to test your web application’s security, as you can simply curl it and take a look at the response.

One final thing I’d like us to understand is that anything can be a browser. If you have a mobile application that consumes APIs through the HTTP protocol, then the app is your browser. It just happens to be a highly customized one that you built yourself, one that only understands a specific type of HTTP responses from your own API.

**Into the HTTP protocol**

As we mentioned, the HTTP exchange and rendering phases are the ones that we’re going to cover, as they provide the largest number of attack vectors for malicious users.

# **5) What is the Web? - Web**

* Network layers
* Physical layer
* Data link layer
* Network layer
* Transport layer
* Application layer

The web, simply put, is a network spread across the globe that connects a multitude of devices and allows them to communicate with one another. Websites on the Internet are hosted on devices referred to as servers, and when you’re interacting with a webpage on the Internet, what you’re essentially doing is exchanging data with the server that the website is hosted on. The device that you’re accessing the webpage using is referred to as the client in the context of the web. In short, the web enables the exchange of data between clients and servers through several elaborate mechanisms.

## **Network layers**

Since the web is an immensely intricate and widespread network, machines within the network are typically divided into abstract layers, each of which performs a specific task that aids in the overall communication process. The layers are enumerated below:

Layers Within End-systems

1. Application Layer
2. Transport Layer
3. Network Layer
4. Data Link Layer
5. Physical Layer

Each layer is built on top of the previous layer and has protocols that implement specific functionalities that are involved in the data exchange process.

### **Physical layer**

The physical layer of a machine refers to the physical wiring and circuits that go into making the machine available on the network.

### **Data link layer**

The data link layer is responsible for transmitting data from any given machine to a device or machine that is exactly one link away.

### **Network layer**

The network layer is responsible for connecting any two machines on the Internet. It provides global connectivity and allows for end-systems to communicate with one another on a large scale, beyond what the data link layer has to offer.

### **Transport layer**

The transport layer is responsible for connecting applications on the Internet. It demultiplexes data coming in from a single source and transmits it to the application it is intended for. The basic purpose of the transport layer in the context of the web is that it provides process-to-process communication; it allows two individual processes on either the same machine or separate machines to send messages to each other. To do so, it uses sockets, which are essentially just the gateway to a process. In other words, sockets are the means through which messages are received and sent out by a process.

### **Application layer**

The application layer is responsible for process-to-process communication across the Internet. It is the topmost layer in the hierarchy, and the application itself is built on top of this. The application layer provides a communication interface and end-user services to the application for its communication with single processes.

# **6) How Does It Work? – Web**

* Switches to connect devices
* Routers
* Data packets
* TCP
* HTTP & HTTPS
* Ports

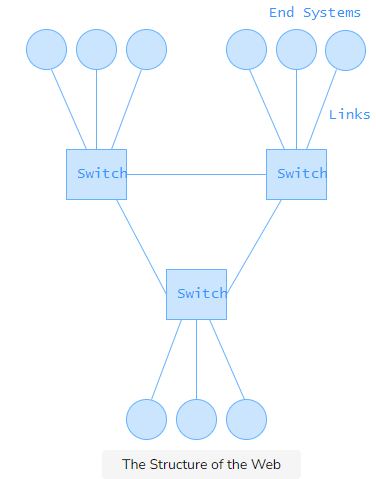
In this lesson, we will look into how the clients and servers communicate with each other.

## **Switches to connect devices**

In essence, the communication is pretty intuitive. Clients send messages to servers requesting data, and servers respond with the required data, but how is this data transferred? The answer requires defining the structure of the Internet first.

The Internet comprises of devices known as switches that facilitate the connection of each device to every other device on the network. The devices themselves are referred to as end-systems, and which is essentially just a fancy term for the computer you’re using to access this webpage right now!

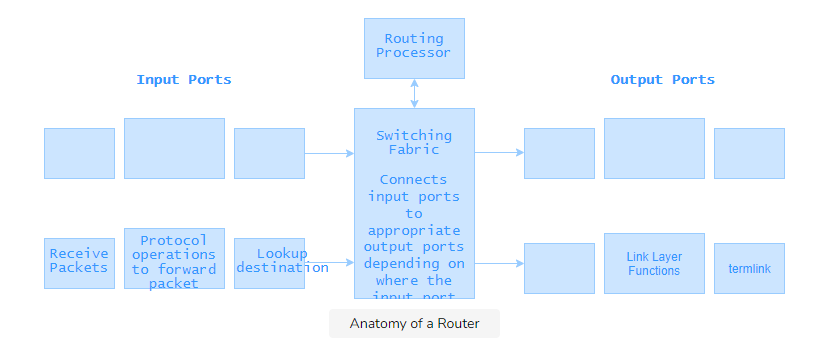
End-systems are connected to switches through links and all of the switches are, in turn, connected to each other, thus ensuring that every end-system on the Internet is implicitly connected to every other end-system.



In addition to connecting end-systems to one another, switches facilitate the communication between any two end-systems by forwarding packets along the path that they know exists between the packet source and destination. So basically, switches store pre-determined paths between end-systems and forwards packets among them.

## **Routers**

A commonly heard term when it comes to networks is routers. Routers have the same function as switches per se in that they also connect end systems to the rest of the web. However, routers are actually very different from switches since they have the additional capability of allowing lookups for destination addresses and determining the shortest or the least busy path from the source of a packet to its destination. Routers are, therefore, a more powerful version of switches and the Internet comprises of a mix of both routers and switches that facilitate the transfer of data between end-systems.



## **Data packets**

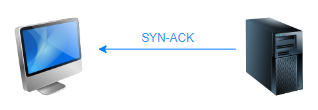
Now that we have established how the Internet is structured to ensure connectivity let’s talk about how data is actually transported across the network. This is done by dividing the data, which is just a set of bits, that needs to be transferred into several smaller chunks of bits known as packets and then sending each packet to its destination independently. The reason for this is that you cannot always send large amounts of data in a single packet because it is highly likely that data in large packets get corrupted on the path from its source to its destination due to a bit flip during transmissions. It is, therefore, more efficient and reliable to send multiple smaller packets.

## **TCP**

Since it is important for clients and servers across the network to be able to understand the same language, there are certain protocols that dictate the communication between devices on the Internet. The primarily used protocol for communication between a web application and a browser is referred to as the Transmission Control Protocol (TCP). TCP is a transport layer protocol that takes the responsibility of transmitting data and ensures reliable data transfer between clients and servers across the web. The way TCP does this is by adding additional information to data packets that allow for packet authentication and by allowing the exchange of acknowledgment messages between the client and server to confirm data transmissions.

The TCP protocol starts with a 3-way handshake. The handshake allows both ends (server and client) to initiate and maintain several TCP connections at once. Have a look at the simplified diagram of the TCP 3-way handshake below.







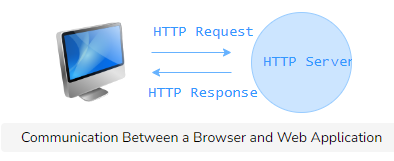
## **HTTP & HTTPS**

We have just learned that clients and servers communicate with each other by initiating TCP connections and then sending messages to one another. Now, we will look into exactly how these messages are structured.

HyperText Transfer Protocol, commonly referred to as HTTP, is an application layer protocol that dictates how the messages a client and server exchange on the web are structured as well as how they are exchanged. The client program and server program talk to each other by exchanging HTTP messages, and the benefit of HTTP is that it ensures messages are being delivered intact and in time.

This may seem ambiguous, but the high-level idea is that HTTP is built on top of TCP and creating an HTTP server for your web application basically just means that you are creating a server that clients create TCP connections with for reliable data transfer. In simple terms, HTTP is the means through which you can make sure your application is using TCP to transmit messages from clients to the server and vice versa. So basically, when you enter a URL in your browser, what actually happens is that an HTTP command gets sent to the server hosting the application to fetch and transmit the requested web page through TCP.

The more common structure of URLs is https://www.educative.io/. So, what does HTTPS stand for? HTTPS is an acronym for HyperText Transfer Protocol Secure, and it is basically just the secure version of HTTP. What this means is that communications between the browser and the hosting server are encrypted so that no third parties on the network can access information that is not intended to be shared.



## **Ports**

So far, we have discussed both the transport layer protocol as well as the application layer protocol that ensures efficient communication between end-systems on the web. But, where exactly do the messages these protocols allow end-systems to go? Ports are the endpoints of the communication between clients and servers. That is to say; ports are where messages from the network arrive on an end-system. We briefly discussed sockets earlier and talked about how they are the gateways to processes; sockets are opened on ports in order to allow processes to send and receive messages. Ports are designated by numbers, and all ports below 1024 are associated with a specific protocol each by default. The port number for HTTP, for instance, is 80, and what this means is that any messages you send or receive on the web come in to and leave your machine on a socket at port 80. Ports above 1024 are open ports available to programmers to use for any process they want to communicate with a network. They can build sockets on these ports and define the structure and type of messages this socket can cater to through socket programming. Socket programming is an aspect of Computer Networks that is beyond the scope of this discussion, but it is a highly useful skill to get under your belt and should definitely be looked into if you are interested in the field.

Now that we’ve given a high-level overview of the communication that occurs on the web starting from the protocols and devices that allow systems on the Internet to communicate all the way to the exact point at which the messages arrive, it is safe to assume that we understand how systems on the web are placed.

# **7) How Data Finds its Way? - Web**

A high-level overview of how packets determine and then traverse the path from their source to destination.

* Internet protocol & IP addresses
* DNS lookups

In this lesson, we will look into how clients and servers know where they want to send their data and what protocols exist to ensure each end-system on a network has a unique identity that allows it to be reached by other end-systems.

## **Internet protocol & IP addresses**

Internet Protocol, more commonly known as IP, is a network layer protocol that is responsible for assigning addresses to devices in order to give them unique identities that make them reachable and discoverable. Each device on the Internet has a unique IP address that other devices use to connect with it. You can find your own IP address by using the following command line​ instruction:

ipconfig

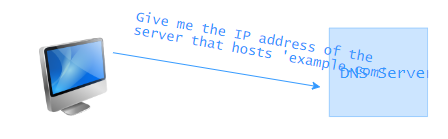
ifconfig is the command that works on Linux, which is what we’re using on our platform. The command ipconfig does the same thing on Windows so try that if you are running the command on a local Windows system.

What this means is that once you have your website live on a server, users will be able to connect with it through the IP address of the server and receive data. You may be wondering now where the IP address comes in if you reach websites through URLs. Well, that’s where DNS comes in. Routers across the web do not store URLs for every single website hosted on the Internet; they only know IP addresses and forward packets towards their destinations based on them. Users, however, only know URLs, or domain names, for the websites they are visiting. To bridge this clear gap of information, the Internet makes use of DNS servers, which serve as translational intermediaries between end-systems and the routers that serve to forward data packets coming from them to their destinations. The process of making a lookup for a domain name on a DNS server and obtaining its corresponding IP address is referred to as a DNS resolution.

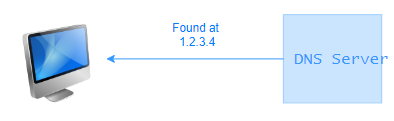
## **DNS lookups**

The last thing to be articulated is exactly how lookups in DNS servers work. Let’s take a look at the steps involved in the interaction between any end-system on the web, referred to as the client in this context, and a DNS server!









That covers everything you need to know about the communication between clients and servers on the Internet and exactly how your application would reach users.

# **8) Introduction to HTTP – HTTP**

Semantics vs. technical implementation

HTTP is a thing of beauty. It is a protocol that has survived for longer than 20 years with very few changes.

As we saw in the previous chapter, browsers interact with web applications through the HTTP protocol, and this is the main reason we’re drilling down on the subject.

If users enter their credit card details on a website and an attacker is able to intercept the data before it reaches the server, we would definitely be in trouble. Understanding how HTTP works, how we can secure the communication between clients and servers, and what security-related features the protocol offers is the first step towards improving our security posture.

## **Semantics vs. technical implementation**

When discussing HTTP we should always discern between the semantics and technical implementation, as they’re two very different aspects of how HTTP works.

The key difference between the two can be explained with a simple analogy; 20 years ago people cared about their relatives as much as they do now, even though the way they interact has substantially changed. 20 years ago, our parents would drive their car to their sister’s house in order to catch up and spend some quality time together. These days it’s more common to drop a message on WhatsApp, make a phone call, or use a Facebook group, things that weren’t always possible. This is not to say that people communicate or care more or less, but simply that the way they interact has changed.

HTTP is no different. The semantics behind the protocol haven’t changed much, but the technical implementation of how clients and servers talk to each other has been optimized over the years. If you look at an HTTP request from 1996 it will look very similar to the ones we saw in the previous chapter, even though the way those packets fly through the network is very different.

# **9) How HTTP Works**

* HTTP requests
* Headers
* Standard headers to the rescue of your servers
* Custom headers
* HTTP responses

As we’ve seen before, HTTP follows a request/response model where a client connected to the server issues a request and the server replies back to it.

An HTTP message (either a request or a response) contains multiple parts:

* start line
* headers
* body

## **HTTP requests**

In a request, the start line indicates the verb used by the client, the path of the resource it wants, and the version of the protocol it is going to use.

GET /players/lebron-james HTTP/1.1

In this case, the client is trying to GET the resource at /players/lebron-james through version 1.1 of the protocol. This shouldn’t be nothing hard to understand.

## **Headers**

After the start line, HTTP allows us to add metadata to the message through headers which take the form of key-value pairs separated by a colon.

GET /players/lebron-james HTTP/1.1  
Host: nba.com  
Accept: \*/\*  
Coolness: 9000

GET /players/lebron-james HTTP/1.1  
Host: nba.com  
Accept: \*/\*  
Coolness: 9000

In this request, for example, the client has attached three additional headers to the request, Host, Accept, and Coolness.

Wait, Coolness?!?!

Headers don’t have to use specific, reserved names, but it’s generally recommended to rely on the ones standardized by the HTTP specification. The more you deviate from the standards, the less the other party in the exchange will understand you.

Cache-Control is, for example, a header used to define whether a response is cacheable. Most proxies and reverse proxies understand this as they follow the HTTP specification to the letter. If you were to rename your Cache-Control header to Awesome-Cache-Control, proxies would have no idea how to cache the response as they’re not built to follow the specification you just came up with.

GET /players/lebron-james HTTP/1.1  
Host: nba.com  
Accept: \*/\*  
Coolness: 9000

In this request, for example, the client has attached three additional headers to the request, Host, Accept, and Coolness.

Wait, Coolness?!?!

Headers don’t have to use specific, reserved names, but it’s generally recommended to rely on the ones standardized by the HTTP specification. The more you deviate from the standards, the less the other party in the exchange will understand you.

Cache-Control is, for example, a header used to define whether a response is cacheable. Most proxies and reverse proxies understand this as they follow the HTTP specification to the letter. If you were to rename your Cache-Control header to Awesome-Cache-Control, proxies would have no idea how to cache the response as they’re not built to follow the specification you just came up with.

## **Standard headers to the rescue of your servers**

The HTTP specification considers multiple scenarios and has created headers to deal with a plethora of situations. Cache-Control helps you scale better through caching. Stale-If-Error makes your website available even if there’s a downtime, this is one of those headers that should be understood extremely well, as it can save a lot of troubles. Accept lets the client negotiate what kind of Content-Type is best suited for the response.

For a complete list of headers I would recommend taking a look at this exhaustive Wikipedia article

## **Custom headers**

Sometimes it might make sense to include a custom header in the message, as you might want to add metadata that is not a part of the HTTP spec. A server could choose to include technical information in its response so that the client can simultaneously execute requests and get important information regarding the status of the server that’s replying back.

..  
X-Cpu-Usage: 40%  
X-Memory-Available: 1%  
...

When using custom headers, it is always preferred to prefix them with a key so that they won’t conflict with other headers that might become standard in the future. Historically, this worked until everyone started to use “non-standard” X prefixes which, in turn, became the norm. The X-Forwarded-For and X-Forwarded-Proto headers are examples of custom headers that are [widely used and understood](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers#Proxies) by load balancers and proxies, even though they [weren’t part of the HTTP standard](https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html).

If you need to add your own custom header, it’s generally better to use a vendor prefix, like Acme-Custom-Header or A-Custom-Header.

After the headers, a request might contain a body, which is separated from the headers by a blank line.

POST /players/lebron-james/comments HTTP/1.1  
Host: nba.com  
Accept: \*/\*  
Coolness: 9000  
  
Best Player Ever

Our request is now complete with a start line (location and protocol information), headers, and a body. Note that the body is completely optional, and, in most cases, is only used when we want to send data to the server. That is why the example above used the verb POST.

## **HTTP responses**

A response is not very different:

HTTP/1.1 200 OK  
Content-Type: application/json  
Cache-Control: private, max-age=3600  
  
{"name": "Lebron James", "birthplace": "Akron, Ohio", ...}

The first information that the response advertises is the version of the protocol it uses, and the status of this response. Headers follow suit and, if required, a line break, followed by the body.

Try receiving some live HTTP responses below! Just replace the URL with any HTTP URL you’d like.

curl -I -s neverssl.com

As mentioned, the protocol has undergone numerous revisions and has added features over time like new headers, status codes, etc., but the underlying structure hasn’t changed much, i.e., start line, headers, and body.

What has changed is how the clients and servers are exchanging those messages; let’s take a closer look at that in the next lesson.

# **10) Mechanics: HTTP vs HTTPS vs H2 - HTTP**

* The evolution of HTTP
* HTTPS
* Public key certificates
* Certificate authority
* Chrome VS Symantec

## **The evolution of HTTP**

HTTP has seen two considerable semantic changes: HTTP/1.0 and HTTP/1.1.

“Where are HTTPS and HTTP2,” you ask?

HTTPS and HTTP2 (abbr. H2) are technical changes, as they introduced new ways to deliver messages over the internet without heavily affecting the semantics of the protocol.

HTTPS is a secure extension to HTTP. It involves establishing a common secret between a client and a server, making sure we’re communicating with the correct party, and encrypting messages that are exchanged with the common secret. We will go over more on this later.

While HTTPS was aimed at improving the security of the HTTP protocol, H2 was geared towards speeding the process up. H2 uses binary rather than plaintext messages, supports multiplexing, and uses the HPACK algorithm to compress headers. To make a long story short, H2 was a performance boost to HTTP/1.1.

Websites owners were reluctant to switch to HTTPS since it involved additional round-trips between client and server (as mentioned, a common secret needs to be established between the two parties), slowing the user experience down. With H2, which is encrypted by default, there are no more excuses as features like multiplexing and server push make it perform better than plain HTTP/1.1.

## **HTTPS**

HTTPS (HTTP Secure) aims to let clients and servers talk securely through TLS (Transport Layer Security), the successor to SSL (Secure Socket Layer).

The problem that TLS solves is fairly simple, and can be illustrated with a simple metaphor: Your significant other calls you in the middle of the day, while you’re in a meeting, and asks you to tell them the password of your online banking account as they need to transfer money for your child’s tuition. It is critical that you communicate it right now, otherwise you face the prospect of your child being turned away from school the following morning.

You are now faced with two challenges

* **authentication**: ensuring you’re actually talking to your significant other and not someone pretending to be them.
* **encryption**: communicating the password without your coworkers being able to understand it and write it down.

What do you do? This is the problem that HTTPS tries to solve.

## **Public key certificates**

In order to verify who you’re talking to, HTTPS uses Public Key Certificates that state the identity behind a particular server. When you connect to an IP address via HTTPS, the server behind that address will present you its certificate for you to verify their identity. Going back to our analogy, this could simply be you asking your significant other to tell you their social security number. Once you verify that the number is correct, you gain an additional level of trust.

This does not prevent attackers from learning the victim’s social security number, stealing your partner’s smartphone and calling you. How do we verify the identity of the caller?

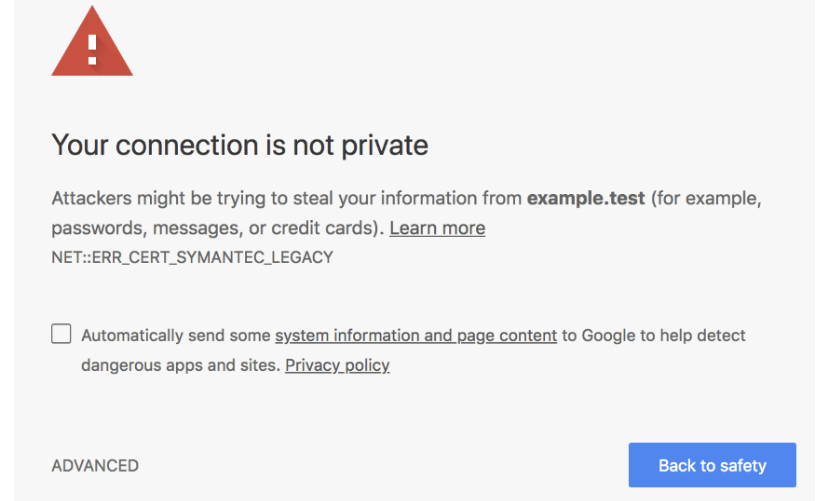
## **Certificate authority#**

Rather than directly asking your partner to say their social security number, you make a phone call to your mom who happens to live next door and you ask her to go to your apartment to make sure your partner is saying their social security number. This adds an additional level of trust, as you do not consider your mom a threat and rely on her to verify the identity of the caller.

In HTTPS terms your mom is called a CA, or Certificate Authority. A CA’s job is to verify the identity behind a particular server and issue a certificate with its own digital signature. This means that, when I connect to a particular domain, I will be presented with a certificate generated by the CA and not the domain’s owner, called a self-signed certificate.

An authority’s job is to make sure they verify the identity behind a domain and issue a certificate accordingly. When you order a certificate, commonly called an SSL certificate, the authority might give you a phone call or ask you to change a DNS setting in order to verify you’re in control of the domain in question. Once the verification process is complete, it will issue the certificate that you can then install on your webservers.

Clients will then connect to your servers and be presented with this certificate so that they can verify it is genuine. Browsers have relationship with CAs, in the sense that they keep track of a list of trusted CAs in order to verify that the certificate is in fact trustworthy. If a certificate is not signed by a trusted authority, the browser will display an informative warning to the users.



## **Chrome VS Symantec**

As you might expect, the relationship between browser vendors and CAs is extremely important. If a vendor distrusts a particular authority, all their certificates are going to be flagged to the average user.

An interesting accident happened with Symantec and its subsidiaries in late 2017 when Google Chrome decided to distrust and phase out support for certificates the authority issued before a certain date. They did this by citing “questionable website authentication certificates issued by Symantec Corporation’s PKI. […] During the subsequent investigation, it was revealed that Symantec had entrusted several organizations with the ability to issue certificates without the appropriate or necessary oversight, and had been aware of security deficiencies at these organizations for some time”.

In other words, this meant Google decided Symantec was issuing certificates without the right background checks, resulting in Chrome displaying a security warning when browsing websites that used old Symantec certificates.

In the next lesson, we’ll study encryption.

# **11) Mechanics: Encryption – HTTP**

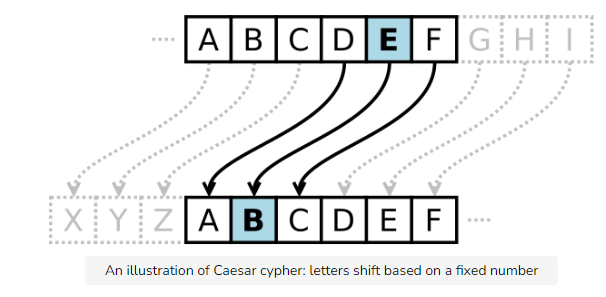
* Why encryption?
* Encryption via shared secret
* Try out encryption!
* Secure exchange of the secret
* More on Public Key exchange algorithms

## **Why encryption?**

We’re halfway to securing the communication between you and your partner. Now that we’ve solved authentication (verifying the identity of the caller) we need to make sure we can communicate safely without others eavesdropping in the process. As I mentioned, you’re right in the middle of a meeting and need to spell your online banking password. You need to find a way to encrypt your communication so that only you and your partner will be able to understand your conversation.

## **Encryption via shared secret**

You can do this by establishing a shared secret between the two of you, and encrypt messages through that secret. You could, for example, decide to use a variation of the Caesar cipher based on the date of your wedding.



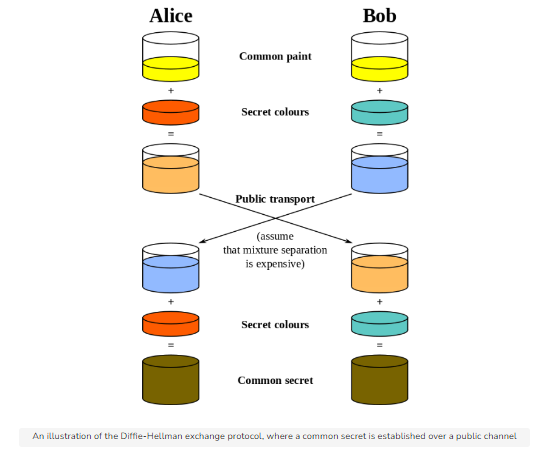
## **Try out encryption!**

Try encrypting your first name using the Caesar Cipher app below!

This would work well if both parties have an established relationship, like you and your partner, as they can create a secret based on a shared memory no one else has knowledge of. Browsers and servers cannot use the same kind of mechanism as they have no prior knowledge of each other.

## **Secure exchange of the secret**

Variations of the Diffie-Hellman key exchange protocol are used instead. This ensures parties without prior knowledge can establish a shared secret without anyone else being able to sniff it. This involves using a bit of math, an exercise left to the reader.



Once the secret is established, a client and a server can communicate without having to fear that someone might intercept their messages. Even if attackers do so, they will not have the common secret that’s necessary to decrypt the messages.

## **More on Public Key exchange algorithms**

For more information on HTTPS and Diffie-Hellman, I would recommend reading “How HTTPS secures connections” by Hartley Brody and “How does HTTPS actually work?” by Robert Heaton. In addition, “Nine Algorithms That Changed The Future” has an amazing chapter that explains Public-key encryption, and I highly recommend it to Computer Science geeks interested in ingenious algorithms.

In the next lesson, we’ll study HTTPS more closely.

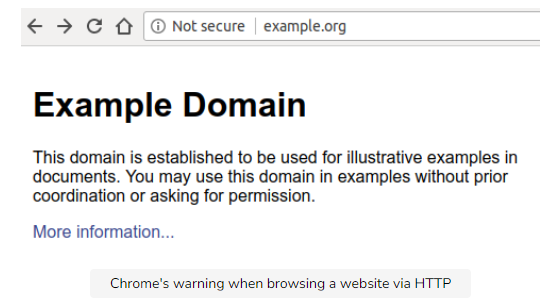
12) HTTPS Everywhere – HTTP

* Discontinued acceptance of HTTP
* Use HTTPS
* CloudFlare: understanding what security really means

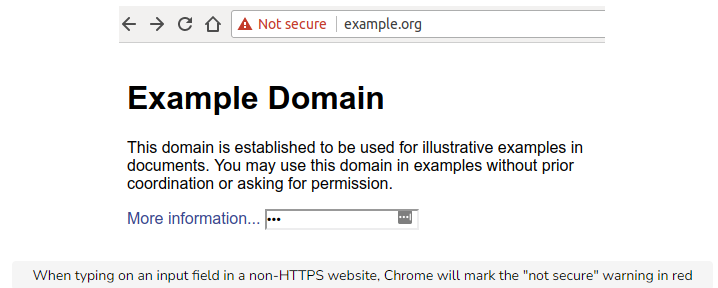
## **Discontinued acceptance of HTTP**

Still debating whether you should support HTTPS on your website? I don’t have good news for you; browsers have started to push users away from websites that don’t support HTTPS in order to force web developers towards providing a fully encrypted browsing experience.

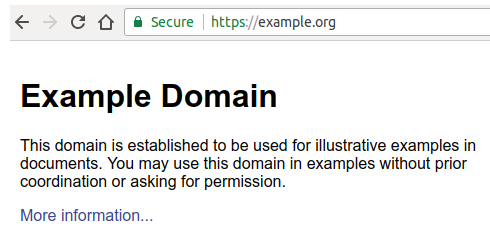
Behind the motto “HTTPS everywhere”, browsers started to take a stand against unencrypted connections. Google was the first browser vendor who gave web developers a deadline, announcing that starting with Chrome 68 (July 2018) it would mark HTTP websites as “not secure”.



Even more worrying for websites not taking advantage of HTTPS is the fact that, as soon as the user inputs anything on the webpage, the “Not secure” label turns red, a move that should encourage users to think twice before exchanging data with websites that don’t support HTTPS.



Compare this to what a website equipped with a valid certificate and running on HTTPS looks like.



In theory, a website does not have to be secure; in practice, this scares users away. Rightfully so! Back in the day, when H2 was not a reality, it could have made sense to stick to unencrypted, plain HTTP traffic. Nowadays, the downsides of doing so are very dangerous, as unwanted ads or crypto-mining software can be injected by attackers or even ISPs when users browse unencrypted websites. That is impossible to do when a website uses HTTPS. Join the HTTPS everywhere movement and help us make the web a safer place for surfers. I recommend you read “Here’s why your static website needs HTTPS” by Troy Hunt, a post that highlights the dangers of not using encrypted connections when communicating with any website.

## **Use HTTPS**

Securing our customers’ experience is a high priority when developing web apps. By using HTTPS, you ensure that the information the user exchanges with your application is transferred securely over the network.

## **CloudFlare: understanding what security really means**

You might have heard of CloudFlare’s free SSL certificate offering, as it encouraged hundreds of thousands of webmasters to serve their websites through HTTPS. Similar to their free DDoS protection service, CloudFlare’s service was a breakthrough for small site owners as SSL certificates used to be quite expensive and many webmasters decided it wasn’t worth offering HTTPS to their customers.

You might think CloudFlare makes the internet a safe place, but I argue that even though it improves the overall security of sites, problems can still happen. Webmasters might be oblivious to them due to the idea that their site is served through HTTPS.

In a traditional network setup, the connection between the customer and your servers is secured until you reach your own network. Your load balancer typically does the SSL termination and then forwards the plain, unsecured request to one of your servers. This was an acceptable tradeoff as you would not expose those servers publicly. Intercepting the unencrypted messages would mean breaking into your secured network, a fairly hard task.

With CloudFlare, things change; the connection is secured between the client and CloudFlare’s edge server, but then flies unencrypted over the internet towards your HTTP-only server. This means that if someone can intercept traffic between you and CloudFlare, they can sniff unencrypted traffic.

Does this mean CloudFlare offers a bad service? Not at all, it simply means we need to understand it before thinking that using it offers the greatest level of protection out there.

Scott Helme perfectly summarized the issue in a blog post titled “My TLS conundrum and why I decided to leave CloudFlare”, which I’d recommend you read.

In the next lesson, we’ll compare and contrast GET and POST requests.

# **13) GET vs POST – HTTP**

* Sending data in the body is safer than in URL

--Avoid storing durable and sensitive information in URLs

* In HTTP headers we trust

As we’ve seen earlier, an HTTP request starts with a peculiar start line:

GET / HTTP/1.1

First and foremost, a client tells the server what verbs it is using to perform the request. Common HTTP verbs include GET, POST, PUT, and DELETE, but the list could go on with less common verbs like TRACE, OPTIONS, or HEAD.

In theory, no method is safer than others; in practice, it’s not that simple.

GET requests usually don’t carry a body, so parameters are included in the URL (i.e., www.example.com/articles?article\_id=1) whereas POST requests are generally used to send (“post”) data which is included in the body. Another difference is in the side effects that these verbs carry with them. GET is an idempotent verb, meaning no matter how many requests you will send, you will not change the state of the webserver. POST, instead, is not idempotent; for every request you send you might be changing the state of the server. Think of, for example, POSTing a new payment, now you probably understand why sites ask you not to refresh the page when executing a transaction.

To illustrate an important difference between these methods we need to have a look at webservers’ logs, which you might already be familiar with:

## **Sending data in the body is safer than in URL**

192.168.99.1 - [192.168.99.1] - - [29/Jul/2018:00:39:47 +0000] "GET /?token=1234 HTTP/1.1" 200 525 "-" "Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.181 Safari/537.36" 404 0.002 [example-local] 172.17.0.8:9090 525 0.002 200  
192.168.99.1 - [192.168.99.1] - - [29/Jul/2018:00:40:47 +0000] "GET / HTTP/1.1" 200 525 "-" "Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.181 Safari/537.36" 393 0.004 [example-local] 172.17.0.8:9090 525 0.004 200  
192.168.99.1 - [192.168.99.1] - - [29/Jul/2018:00:41:34 +0000] "PUT /users HTTP/1.1" 201 23 "http://example.local/" "Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.181 Safari/537.36" 4878 0.016 [example-local] 172.17.0.8:9090 23 0.016 201

As you can see, webservers log the request path. This means that if you include sensitive data in your URL, it will be leaked by the web server and saved somewhere in your logs. Your secrets are going to be somewhere in plaintext, something we need to avoid at all costs. [Imagine an attacker being able to gain access to one of your old log files](https://threatpost.com/leaky-backup-spills-157-gb-of-automaker-secrets/134293/), which could contain credit card information, access tokens for your private services, and so on. That would be a disaster.

Webservers do not log HTTP headers or bodies, as the data to be saved would be too large. This is why sending information through the request body, rather than through the URL, is generally safer.

From here we can derive that POST and other, non-idempotent methods are safer than GET, even though it’s more a matter of how data is sent when using a particular verb rather than a specific verb being intrinsically safer than others. If you were to include sensitive information in the body of a GET request, you’d face no more problems than when using a POST, even though the approach would be considered unusual.

## **Avoid storing durable and sensitive information in URLs**

URLs are usually logged by web servers, so any information stored there could potentially be leaked. If you need to, consider using one-time/expiring secrets that are of no use in the long run. Amazon S3 signed URLs are a brilliant example of using expiring secrets to grant temporary access to a resource.

## **In HTTP headers we trust**

In this chapter we looked at HTTP, its evolution, and how its secure extension integrates authentication and encryption to let clients and servers communicate through a safe(r) channel. This is not all HTTP has to offer in terms of security, as we will see shortly. HTTP security headers offer a way to improve our application’s security posture, and the next chapter is dedicated to understanding how to take advantage of them.

# **14) The Server-side and The Client-side**

Introduction

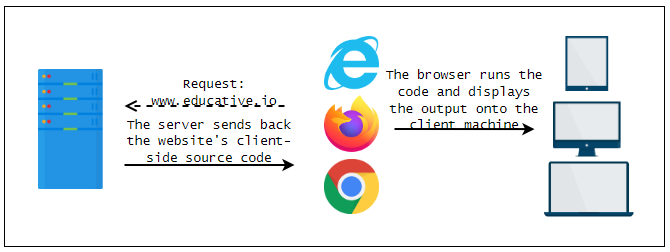
* Client-side
* Server-side

## **Introduction**

Let’s get into the basics of coding a website.

## **Client-side**

The source code of a website can be broken down into two parts: client-side and server-side. The code on the ‘client’ side runs on the client computer’s browser and handles what the website looks like, how it requests data from the server, and how it interacts with temporary and local storage on the client machine. This includes but is not limited to selecting and styling user interface components, creating layouts, navigation, form validation, and cache handling. Client-side languages include HTML, CSS, and JavaScript.



## **Server-side**

Server-side programming is used to deliver information that is queried from websites to clients.

Companies like Amazon use server-side programming to construct search results for products and make targeted product suggestions based on client preferences and previous buying habits. Banks use server-side programming to store account information and restrict access from unauthorized accounts. Social media sites such as Facebook, Twitter, and Instagram, use server-side programming to highlight, share, and control access to content that is relevant to each user. Server-side programming includes but is not limited to,

1. Delivering and storing information efficiently
2. Providing a custom experience for each user
3. Controlling access to content
4. Storing session/state information
5. Notifying and communicating
6. Analyzing data

Server-side languages include JavaScript, Python, PHP, Ruby, and C#.

# **15) What is the Back End? -BE**

The back end of a website loosely consists of a server, databases, and APIs.

## **What are web servers?**

A web server is a system that delivers content and services to users over the Internet. Web servers are exactly like regular computers except that they are more powerful and robust. Also, most servers don’t have a display like a monitor or a screen that most PCs do, but you can connect to them with SSH. To put it simply, SSH is a way to get another machine’s terminal on your own. A web server can ‘host’ a website, i.e., the webserver will have all the code for the website on it: front and back and anyone with an Internet connection can access it with their machine to view the website.

Technically, you can open up port 80 on your PC and host your website there. However, this is usually never done because a regular PC would not be able to handle the load of potentially thousands of client connections, it would put your personal data at risk, and you would have to keep it on with the server running 24/7. So web hosting services solve this problem.

## **Web hosting services**

Web hosting companies provide space on a server that is owned or leased for use by clients, as well as providing Internet connectivity, usually in a data center. They typically charge a monthly fee for this service. A data center is a building that houses multiple servers. Data centers usually require extensive cooling and maintenance to keep the temperature of their machines down, which reduces crash rates and so increases profits. Facebook, for example, has some of the largest data centers in the world!

Rack servers in a data center

Selecting the appropriate web hosting service plays a crucial role in web deployment, however, and is a critical decision.

## **Web server operating systems**

Just like your personal computer has an operating system, Windows, Mac OS, or Linux, servers require an operating system to function too. Although, most server operating systems are specialized for servers and so slightly different from desktop operating systems.

## **Linux**

Linux is not one operating system. It is a group of free and open-source operating systems. Typically, Linux is packaged in a form known as a Linux distribution (or distro for short) for both desktop and server use. There are many flavors or distros of Linux for server use. For example,

* CentOS
* Debian
* Ubuntu
* Windows Server 2019
* Mac OS X Server
* Solaris

## **CentOS**

The benefit of using CentOS is that it is stable and seldom crashes. It is also the most popular distro for servers as of now, so it also has the benefit of great support from the web, which means you won’t be stuck on any issue for long.

Furthermore, most Linux distros update regularly, but CentOS does not. The benefit of this is that once you get your server up and running, it will be in good shape for a long time without having to upgrade to a newer version regularly.

As for the cons, you will have difficulties running newer packages and making use of the latest technologies. If you are looking to use the latest technology, CentOS might not be for you.

## **Debian**

Like CentOS, Debian does not update as frequently as well. Debian is also a community distro, which means that new versions of Debian are released only when the community arrives at a consensus. So every Debian release is thoroughly tested and reliable. There is only one kind of Debian available for you to download and install, which works on both server and desktop.

## **Ubuntu**

Ubuntu is the most popular Linux distro for the desktop. You might want to go with Ubuntu on your server if you want the advantage of being familiar with it!

Here are some others that you might want to consider,

## **Windows Server 2019**

## **Mac OS X Server**

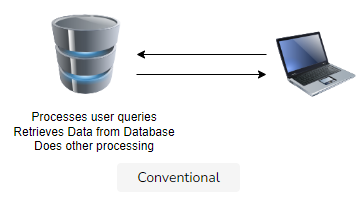
## **Solaris**

# **16) Microservice Architecture - BE**

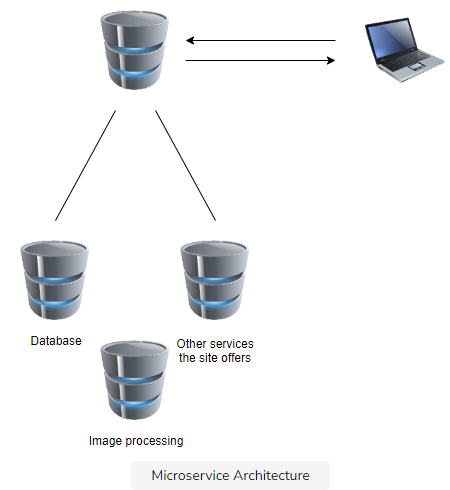
The back end consists of a server that receives user queries and a database from which the appropriate results are retrieved.

## **Microservices**

However, most modern websites don’t rely on this style of web hosting, i.e., the code of all of the functionalities of the entire website does not exist on one server. Instead, websites are hosted on virtual machines (just think of them as computers within computers). Virtual machines can allow us to have several ‘machines’ on one server, each with different operating systems. So, the website is hosted on a server within a server, and that server delegates tasks to other servers.



In the diagram above, a client connects to one server, and it handles everything and responds.



In the Microservices architecture, however, many virtual machines exist on one physical server, and each has a separate job. For example, one of them may be the ‘interface’ that the user interacts with, and it might connect to another for the database.

According to the Microservices website,

"Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are

* Highly maintainable and testable
* Loosely coupled
* Independently deployable
* Organized around business capabilities.

The microservice architecture enables the continuous delivery/deployment of large, complex applications. It also enables an organization to evolve its technology stack."

# **17) Back-end programming – BE**

What do backend engineers do?

* Efficient storage and delivery of information
* Customized user experience
* Controlled access to content
* Store session/state information
* Notifications
* Data analysis

## **What do backend engineers do?**

Backend engineers program servers to process user requests and respond with the requested resource correctly. They also write code to process and store user data.

## **Efficient storage and delivery of information**

Data that pertains to a website has to be stored in a database and delivered upon request. Consider Amazon’s database of items; if their database query system becomes inefficient at any point, they’ll lose customers.

## **Customized user experience**

Servers often store and use information about clients to provide a tailored user experience. For example, many sites store credit card details so that the information doesn’t have to be entered again.

## **Controlled access to content**

Backend programming entails restricting access to information appropriately. For example, in a ride-hailing application such as Uber, one user should not be able to view another’s car travel history.

## **Store session/state information**

Backend engineers also do user session handling, i.e., a string is associated with each user that visits the website, and data associated with the string such as their emails or order history are saved and displayed when the user visits again. Another example is saving the state of a simple game so that the user can go to the site again and carry on from where they left it.

## **Notifications**

Servers can be programmed to send general or user-specific notifications through the website itself or via email, SMS, instant messaging, video conversations, or other communications services.

A few examples include:

* Facebook and Twitter send emails and SMS messages to notify you of new communications.
* Amazon regularly sends product emails that suggest products similar to those already bought or viewed that you might be interested in.
* A web server might send warning messages to site administrators, alerting them to low memory on the server or suspicious user activity.

## **Data analysis**

A website may collect a lot of data about users: what they search for, what they buy, what they recommend, how long they stay on each page. Server-side programming can be used to refine responses based on an analysis of this data.

For example, Amazon and Google both advertise products based on previous searches (and purchases).