Semantic HTML cheat sheet

There are hundreds of semantic tags available to help describe the meaning of your HTML documents. Below is a cheat sheet with some of the most common ones you’ll use in this course and in your development career.

Sectioning tags

Use the following tags to organize your HTML document into structured sections.

<header>

The header of a content section or the web page. The web page header often contains the website branding or logo.

<nav>

The navigation links of a section or the web page.

<footer>

The footer of a content section or the web page. On a web page, it often contains secondary links, the copyright notice, privacy policy and cookie policy links.

<main>

Specifies the main content of a section or the web page.

<aside>

A secondary set of content that is not required to understand the main content.

<article>

An independent, self-contained block of content such as a blog post or product.

<section>

A standalone section of the document that is often used within the body and article elements.

<details>

A collapsed section of content that can be expanded if the user wishes to view it.

<summary>

Specifies the summary or caption of a <details> element.

<h1><h2><h3><h4><h5><h6>

Headings on the web page. <h1> indicates the most important heading whereas <h6> indicates the least important.

Content tags

<blockquote>

Used to describe a quotation.

<dd>

Used to define a description for the preceding <dt> element.

<dl>

Used to define a description list.

<dt>

Used to describe terms inside <dl> elements.

<figcaption>

Defines a caption for a photo image.

<figure>

Applies markup to a photo image.

<hr>

Adds a horizontal line to the parent element.

<li>

Used to define an item within a list.

<menu>

A semantic alternative to <ul> tag.

<ol>

Defines an ordered list.

<p>

Defines a paragraph.

<pre>

Used to represent preformatted text. Typically rendered in the web browser using a monospace font.

<ul>

Unordered list

Inline tags

<a>

An anchor link to another HTML document.

<abbr>

Specifies that the containing text is an abbreviation or acronym.

<b>

Bolds the containing text. When used to indicate importance use <strong> instead.

<br>

A line break. Moves the subsequent text to a new line.

<cite>

Defines the title of creative work (for example a book, poem, song, movie, painting or sculpture). The text in the <cite> element is usually rendered in italics.

<code>

Indicates that the containing text is a block of computer code.

<data>

Indicates machine-readable data.

<em>

Emphasizes the containing text.

<i>

The containing text is displayed in italics. Used to indicate idiomatic text or technical terms.

<mark>

The containing text should be marked or highlighted.

<q>

The containing text is a short quotation.

<s>

Displays the containing text with a strikethrough or line through it.

<samp>

The containing text represents a sample.

<small>

Used to represent small text, such as copyright and legal text.

<span>

A generic element for grouping content for CSS styling.

<strong>

Displays the containing text in bold. Used to indicate importance.

<sub>

The containing text is subscript text, displayed with a lowered baseline.

<sup>

The containing text is superscript text, displayed with a raised baseline.

<time>

A semantic tag used to display both dates and times.

<u>

Displays the containing text with a solid underline.

<var>

The containing text is a variable in a mathematical expression.

Embedded content and media tags

<audio>

Used to embed audio in web pages.

<canvas>

Used to render 2D and 3D graphics on web pages.

<embed>

Used as a containing element for external content provided by an external application such as a media player or plug-in application.

<iframe>

Used to embed a nested web page.

<img>

Embeds an image on a web page.

<object>

Similar to <embed> but the content is provided by a web browser plug-in.

<picture>

An element that contains one <img> element and one or more <source> elements to offer alternative images for different displays/devices.

<video>

Embeds a video on a web page.

<source>

Specifies media resources for <picture>, <audio> and<video> elements.

<svg>

Used to define Scalable Vector Graphics within a web page.

Table tags

<table>

Defines a table element to display table data within a web page.

<thead>

Represents the header content of a table. Typically contains one <tr> element.

<tbody>

Represents the main content of a table. Contains one or more <tr>elements.

<tfoot>

Represents the footer content of a table. Typically contains one <tr> element.

<tr>

Represents a row in a table. Contains one or more <td> elements when used within <tbody> or <tfoot>. When used within <thead>, contains one or more <th> elements.

<td>

Represents a cell in a table. Contains the text content of the cell.

<th>

Defines a header cell of a table. Contains the text content of the header.

<caption>

Defines the caption of a table element.

<colgroup>

Defines a semantic group of one or more columns in a table for formatting.

<col>

Defines a semantic column in a table.

Querying APIs using JavaScript

Introduction

After the lesson on JavaScript you know you can do exciting things with it. You can use it for client-side programming, to make a web application interactive or to send and receive data from the server. JavaScript really is the default choice when it comes to front-end programming.

But it can also be used for back-end purposes and in this reading you will learn how to use JavaScript to fetch and send data to an API using the native fetch function.

Native solution versus third-party libraries

When it comes to fetching data from APIs you might ask yourself, should I use the native fetch function XMLHttpRequest object, or should I go with libraries like jQuery or Axios? Let’s contextualize the question with some background information.

Previously, fetching data from APIs with XMLHttpRequest object was difficult because you have to write quite a lot of code to make a simple request. But, at the time, there was no alternative option. Then came third-party libraries that work as a wrapper around XMLHttpRequest and which provide a simpler interface. These libraries quickly became popular. Later versions of JavaScript came with another native API called fetch, which is powerful, simple and easy to use.

Though using the fetch API is easy, you still have to manually adjust a few things and write extra code for error checking and header processing. This is why libraries like Axios are still popular because they provide a simpler interface and automatically take care of a lot of issues so that you don’t have to write so much code. Ultimately, you can use either the native fetch API or Axios library. Both are viable options since both solve the problem of fetching data from APIs in a straightforward manner.

Now let’s explore how you can make GET, POST, PUT, PATCH and DELETE calls using the fetch API and how to make authenticated calls using tokens.

Making a GET call

Placing a GET call using fetch is simple. All you have to do is make the call, convert the response to JSON or text and then process it any way you want. Here is the output from the menu-items endpoint of the Little Lemon restaurant app that was used in the course on APIs. Use the following code to make a GET call to this endpoint:

fetch('http://127.0.0.1:8000/api/menu-items')

.then(response => response.json())

.then(data => {

console.log(data)

})

POST, PUT and PATCH Calls

How do you make a POST call with data using the fetch API? You have to convert the JSON payload that contains all the data to a string using the JSON.stringify() function and pass it as body in the second argument to the fetch function. It’s also a good practice to add Accept and Content type headers while making the API calls.

Here is the sample code that creates a new menu item by making a POST call to the

http://127.0.0.1:8000/api/menu-items endpoint.

const payload = {

"title": "Ambrosia Ice cream",

"price": 5.00,

"inventory": 100

}

const endpoint = 'http://127.0.0.1:8000/api/menu-items'

fetch(endpoint,

{

method: 'POST',

headers: {

'Accept': 'application/json',

'Content-Type': 'application/json'

},

body: JSON.stringify(payload)

})

.then(response => response.json())

.then(data => {

console.log(data)

})

For PUT and PATCH calls, you just change the method from POST to PUT or PATCH. Everything else remains the same.

DELETE calls

For DELETE calls, change the method to DELETE and that’s all. In most cases, there is no body passed to a DELETE call. Here’s the code for a sample DELETE call to the menu-items endpoint:

const endpoint = 'http://127.0.0.1:8000/api/menu-items/17'

fetch(endpoint,

{

method: 'DELETE',

headers: {

'Accept': 'application/json',

'Content-Type': 'application/json'

}

})

.then(response => response.json())

.then(data => {

console.log(data)

})

Making authenticated calls with tokens

If you want to make authenticated API calls using bearer tokens, pass the Authentication header in the second argument in the fetch function. Here is an authenticated POST call. Note how the bearer token is passed in the header section.

const endpoint = 'http://127.0.0.1:8000/api/menu-items/17'

const token = “Some token”

fetch(endpoint,

{

method: 'POST',

headers: {

'Accept': 'application/json',

'Content-Type': 'application/json',

'Authentictation': 'Bearer ' + token

},

body: JSON.stringify(payload)

})

.then(response => response.json())

.then(data => {

console.log(data)

})

You can do the same for GET calls as well.

Conclusion

In this reading, you learned about making API calls in JavaScript using the native fetch function and how to process the response. You also learned how to send a JSON payload to make POST, PUT and PATCH calls.

**Key elements of cloud computing**

In volatile storage data is only removed when the machine reboots. To deal with this issue, you must add a persistent storage solution to your server.

The cloud computing units can be resized, which means you can upgrade or downgrade the number of

processing cores or memory as needed. For complex programs like data analysis or machine-learning, most Cloud providers offer a variety of processor cores and memories. Some even offer use of GPUs. Since you're charged only for what you use, you can deploy a powerful computing unit when you need it. Let it run as long as required, and then delete it. What's more, you can install your preferred operating system and applications in these Cloud computing units. Some of these computing units come with volatile storage, which means everything you store in it will be gone when the machine reboots. In such cases, you must always equipped you're computing unit with a permanent storage solution.

**Storage**

As significant benefit of using Cloud computing. You can purchase storage, keep your data as long as you want, and then delete it whenever necessary. Cloud computing providers offer storage solutions in a few ways. You can purchase gigabytes or terabytes to store your application files or personal data. It works just like a regular hard disk, but the risk of data loss due to hardware fault is nearly zero. This is because Cloud providers automatically backup your data in redundant storage. Another storage solution offered by public Cloud providers is called object storage. In object storage you get an API to upload and download your files, and that's all. You can store small or massive files and object storage. What's more? You can also make these files available to anyone for a certain amount of time with a timestamp signature. Cloud providers give you a unique URL to access your file, and that link automatically expires after the duration you specified in the signature. Now let's move on to databases. Most public cloud providers offer SQl, NoSQL or Time-series databases to store your application data. Examples of SQL or MySQL, MariaDB, and PostgreSQL, and examples of NoSQL are MongoDB, Cassandra and DynamoDB. Whereas examples of Time-series are Influx dB and Prometheu. Tuning a database can be tricky and scaling it can be tedious and requires a lot of effort and careful supervision. With cloud computing, these issues disappear. Cloud providers offer fully managed database solutions compliant with popular database engines like MySQL, MariaDB, or PostgreSQL, so no change in the application code is necessary. Since these are managed solutions, you can allow these databases to scale automatically as they grow and they'll be able to withstand substantial traffic loads without extra effort. Some Cloud providers also offer in-memory database solutions, which are great for caching or super-fast data operations. Cloud providers can also offer specialized solutions to deal with huge amounts of data or big data. With Cloud computing, you can deploy a single database node or multiple ones and just a few seconds and link them to your applications. Finally, there's machine-learning which requires powerful computing units and GPU for data modeling and training. But this can be very exciting, expensive if you purchase physical hardware. However, with cloud computing and its on-demand pricing model, you can start using machine learning at a cheaper cost. Cloud providers offer the latest hardware and software solutions that are efficient for machine learning, natural language processing, voice processing, and so on. For machine-learning, Cloud Computing is a perfect and affordable solution. Cloud Computing is an excellent solution that allows you to spend more time on application development instead of managing hardware and infrastructure.

# **Networking in the cloud**

**Introduction**

In this reading, you will learn about some exciting networking concepts used in cloud computing. You can use this knowledge to understand how servers communicate privately and publicly.

Public versus private network

In a public network, the cloud computing units are publicly accessible using IP addresses or URLs. On the other hand, computing units in a private network are not publicly accessible. These units are still accessible from the management console but don’t provide any public interface. To communicate publicly, the computing units can be on totally different networks. However, the units must be on the same network for private communication.

A load balancer sits in front of a typical infrastructure and is connected to multiple web servers. This load balancer will be connected to the public, but the web servers can connect to the load balancer both publicly and privately if they are on the same network.

**IP Address**

An IP address is a communication protocol used in networks. This is a unique identifier to locate a machine or a device on the internet or on a private network. You can use one of the two popular IP addressing systems for this. The first one is the IP version 4 or IPv4 or simply IP4. The other one is IPv6 or IP6.

In the IPv4 system, the address is divided into four different parts, and each part can be numbers ranging from 0 to 255, separated by periods. So, the IPv4 address range is anything from 0.0.0.0 to 255.255.255.255. The maximum length of an IPv4 address can be 15 characters. Here is a sample IPv4 address that belongs to Meta:

69.63.176.13

In the IPv6 system, there are eight groups separated by a colon (:), and in each group, there are four hexadecimal digits. Facebook has adopted an IPv6 address:

2a03:2880:2130:cf05:face:b00c:0:1

Did you notice the “face” and “b00c” in the IP address? They're both valid hexadecimal digits.

The IPv4 system can have a maximum of 4,294,967,296 unique IP addresses. There needs to be more in this growing modern world, and it can run out soon, which is why IPv6 was invented. You can have 3.4 x 1038 or 340 trillion, trillion, trillion unique addresses.

IPv6 is a new technology, and many countries worldwide are still adapting to IPv6.

When using IPv4, these ranges are used only in the private network, 10.0.0.0 — 10.255.255.255; 172.16.0.0 — 172.31.255.255 and 192.168.0.0 — 192.168.255.255 so they are called private IP ranges.

One device or machine can have more than one IP address and both IPv4 and IPv6 addresses together.

**DNS system**

Remembering IP addresses is tough. Imagine you host your application in a server that has a public IP address, but you access it using a domain or hostname. This is where DNS servers come in. DNS servers are public servers that store these domain names against their IP addresses. Whenever you request a URL starting with a domain name on your browser or a request, it goes to the DNS server, receives the IP address, and then connects to that machine.

**Bandwidth**

Data that comes in or goes out from your server is called bandwidth. There are two types of bandwidths – incoming and outgoing. The data that comes to the server, for example, when you submit a form or upload files, is called incoming bandwidth or ingress bandwidth. And the data that leaves, for example, the webpage or API response or downloaded files, are counted as outgoing bandwidth or egress bandwidth.

**Uplink and downlink**

In a server, uplink refers to the network or communication protocol used to send data outside the server. Similarly, the downlink is the network used to accept incoming data.

**Conclusion**

In this reading, you have learned the basic networking terms and how servers communicate with the public or other servers using IPv4 or IPv6 protocol. You also learned about egress and ingress bandwidth, uplink and downlink.

**What is scaling?**

Imagine your application is featured on a famous blog, and as a result, many more visitors are coming to your site, or you've received many sign-ups after a successful social media campaign and your database is growing fast. If your application is not ready to handle such loads, or sudden traffic spikes, the web server, or the database server will fail to handle requests. They will either be extremely slow to respond, or go down. That is why your production server must be scalable to sustain such loads or be responsive as your users grow.

Scalability is the process of resizing your production infrastructure to withstand the load as it goes up, or down. Sometimes the load could be sudden, otherwise it will continue to grow as your application gets bigger and starts to have more users. Usually in a production environment the main bottlenecks are the web server and the database server which needs to be scalable. There are two types of scaling, vertical and horizontal. Let's examine them in detail. Under load, the web server and database is mostly fail for two reasons, lack of resources like CPU and RAM, or incorrect configuration which prevents them from using available resources. If you are failing due to the lack of resources, it's a comparatively easy fix. The solution is to add more storage, RAM, or processing cores to the Virtual Machine, or upgrade the physical processor. This process of adding RAM, or CPU, or storage is called vertical scaling. But there is a problem with vertical scaling, you cannot add an infinite amount of processor cores, RAM, or storage in a physical machine. How many resources you can add depends on the hardware configurations. Once the limit is reached, you cannot scale anymore. Moreover, adding physical hardware like this can be very expensive and unrealistic over time. Vertical scaling is quicker than horizontal scaling and has easier configuration. Therefore, it looks appealing as an easy fix, but beware of its limitations. It can be costly over time, or when it needs repairs. Always keep in mind that you cannot scale infinitely with vertical scaling. However, there is another type of scaling that overcomes these issues and that is horizontal scaling. In fact, it's considered more efficient and cost-effective. In this process you add nodes, or virtual servers as the load goes up and remove the nodes when they go down. For example if one web server can handle 2,000 requests per minute, to withstand the load of 10,000 requests per minute, you need to add four extra nodes to the web server. Five nodes will be capable of handling 10,000 requests per minute without any issues. The same thing goes for the database, you can use database replication, or database clustering to create multiple database nodes as the load grows and when the load returns to normal, you can remove those nodes again. An application with a proper horizontal scaling setup can be scaled to handle a tremendous amount of load. It's infinitely scalable with horizontal scaling. However, it's not a plug and play process. Configuring your application infrastructure to be horizontally scalable takes time and effort. Finally, let's discuss auto-scaling. Some Cloud providers offer an excellent auto-scaling solution with no manual intervention, or extra configuration. When you turn on Auto Scaling, the Cloud provider automatically adds, or removes computing nodes, memory, or other resources to keep the applications sustainable and healthy under load. With this feature you can save a lot of time and ensure better uptime for your application.

**Load balancing**

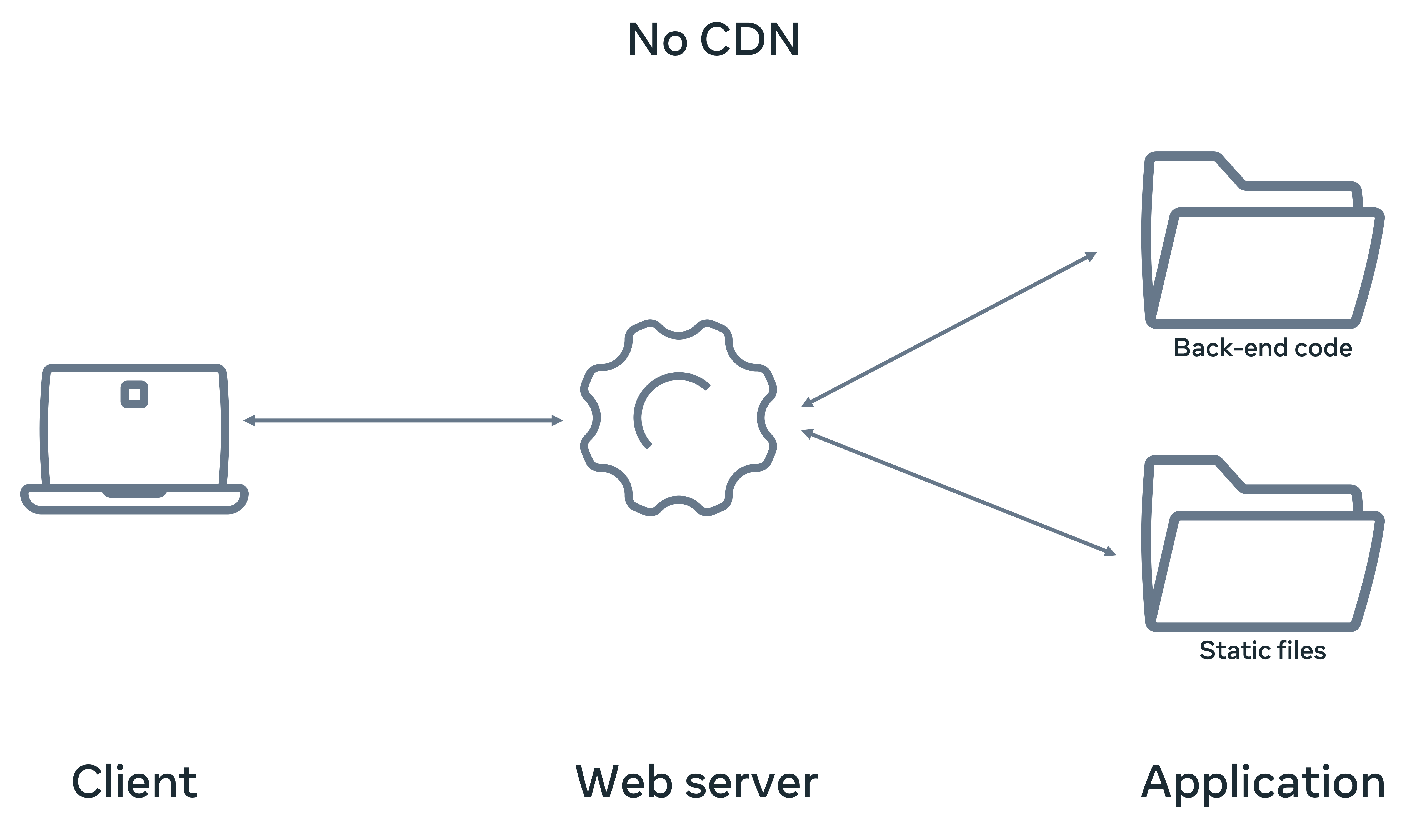
Load balancers help scale your application by evenly distributing loads on multiple nodes in a round-robin fashion, or based on node health. You can use various load-balancing techniques to reduce the pressure on your application infrastructure.

Let's start with load balancing the web server. Your web server plays a vital role in accepting all HTTP requests, sending them to the backend scripts, getting the response, and delivering it to the client. The web server also receives requests to serve static HTML, CSS, and JavaScript files. Many Cloud providers offer load-balancing services. Using these, you can create multiple web servers with the same application code and link them to the load balancer. The load balancer then accepts all HTTP requests and sends them to the web servers. This load distribution can happen in two styles, round-robin and health based. In the round-robin style, the load balancer distributes the load equally on all web servers. But there is a problem. Not every request is the same. Some requests take a fraction of a second to finish and some take longer. In a round-robin load distribution model, if a web server is already working on a request, it may receive subsequent requests, and then take a long time to process those requests. There's a better way to deal with this problem. Some load balancers are configured in a way that can monitor the load on these web servers and then distribute the load only on those nodes which are available and not already loaded. This is called health-based load balancing and works better than round-robin style. You can use a reverse proxy to perform load balancing. The web server serves everything including application URLs, API calls, and static files. A reverse proxy can be configured in such a way as to serve the static files by itself, and forward the applications specific URLs and API calls to the web server. Static files includes HTML, CSS, JS, text files, and images. This technique is widely used and improves the performance of your web server by reducing the number of requests the web server has to handle. Similar to serving the static assets using a reverse proxy, you can completely offload them to an external content delivery network or CDN. This can speed up your applications page rendering time, because CDN services create multiple copies of these static files and serve them from the client's nearest data center. The database is one of the most crucial parts of every web application. A well-tuned database server can handle a huge load. Scaling a database is more complex than scaling a web server. Sometimes vertical scaling helps, but horizontal scaling has better sustainability. There are two popular techniques for scaling databases horizontally. The first one is sharding. This is when a large database is split into multiple chunks and one database server holds one chunk. One extra database is needed to keep track of these chunks for better lookup. With the master-slave replication approach, the whole database is replicated on multiple servers. One works as a master server where the data modifications happen. As soon as something changes, the master server writes those changes on other servers, which are called the slave servers. All the read requests are evenly distributed to these slave servers. This way, multiple slave servers can handle more SQL queries and still remain healthy. Most Cloud providers offer managed database servers where they internally balanced the loads and process a massive amount of data without creating trouble or going down. Using this managed database service, the developer remains worry-free and focuses on development and business growth instead of tedious infrastructure management and DevOps processes. The load balancing techniques discussed are considered public load balancing, because they are directly dealing with the external client requests. Private load balancing is sometimes necessary where heavily loaded internal non-public applications, or microservices are communicating with each other over private IP addresses.

**How a CDN improves scaling**

Content Delivery Networks or CDNs are widely used to store static content files like HTML, CSS, JS, and images of web applications. CDNs deliver content from servers worldwide but always from the nearest server to the client, which has many advantages.

## **What is CDN**

A CDN has multiple servers which are called PoP or point of service and they run in different countries across the world. When you connect your web application to a CDN, each PoP can store a copy of the static files of your web application. For instance, when a visitor browses your application from Asia, the PoP closest to that visitor's ISP or internet service provider serves the application’s content. Below is a diagram of a typical web application architecture without a CDN.

There are two significant benefits to this process of serving content via CDNs. First, your web application visitors receive the content faster than those web applications that do not use CDNs. And second, your web application is not receiving direct traffic requesting those static files. Thus, the load on the web server is reduced to a great extent.

## **Push versus pull CDNs**

There are two types of CDN, push and pull CDNs. When your web application is connected to a push CDN you need to upload or push static files to the CDN every time you make changes to them. This can be done manually or automatically. But it requires extra effort from the application developer, and is therefore less popular.

On the other hand, a pull CDN updates files automatically. When visitors request a file, pull CDNs will check the origin server automatically to detect if the original static files have changed. An origin server is a server you use to host your web application, and this is where the static files are stored. If any changes are detected, the CDN server pulls those changed files and serves them. The new version of the files then also replaces the previous versions on the CDN. Since this is entirely automated, pull CDNs areused most often. Below is a diagram of how a pull CDN works.

## **Advantages**

The most significant benefit of using a CDN is the reduced delivery time for static files. When a web page renders, the static files take the longest to render. So, the faster they are served, the quicker the page will display.

Another benefit is the reduced workload on the web server. A web page can have hundreds of static CSS, JS, and image files linked to the page. For example, if there are 10 JavaScript files, 2 CSS files, and 60 image files, a single page visit will make 1+10+2+60=63 requests to the web server. If your webserver can serve 2000 requests per minute, with this calculation, it will be capable of serving only 2000/63 = 31 visitors per minute. Now, if you offload all the static files to a CDN, that means for every page visit, your webserver is getting only one request because other files are served directly from the CDN servers. The result is that your server can instantly serve 2000 visitors per minute! That's a massive improvement for little extra cost. With less load, the web server also works more efficiently.

Another benefit is that CDNs reduce the bandwidth from the webserver. CDNs come with huge uplink capacity and can serve a tremendous amount of data every second, which may not be possible to serve from a single web server with your current provider.

And lastly, it is beneficial to use images on the CDN. Many CDN automatically reduces image size or even convert them as they are requested to more modern formats like webp. Webp is a lossless format that preserves the quality of the image.

## **Disadvantages**

Sometimes, viewing the changed files delivered from the CDN may take time. This is because some CDNs take time to update the stored files with the newer versions. So, changes to any static file may not be instantly visible to your visitors. It may take 10 to 30 minutes or more due to different cache settings. But this rate varies significantly from provider to provider.

Another disadvantage is the bandwidth cost. When you deliver data from CDNs, the bandwidth cost will be higher than serving the files from your web server. And as your applications begin to grow, it can be costly.

Still, compared to the advantages CDNs offer, the disadvantages are far fewer and you should try to use a CDN whenever possible.

## Conclusion

In this reading, you learned more about content delivery networks or CDNs, the different types and how they enable web applications to scale up. You also learned about the pros and cons of using CDNs.