# Web infrastructures - Course material

L. Delafontaine and H. Louis, with the help of GitHub Copilot.

Based on the original course by O. Liechti and J. Ehrensberger.

# Table of contents

# Objectives

In this chapter, you will learn how to build a web infrastructure.

You will learn how to use a reverse proxy and a load balancer to scale your system.

Thanks to HTTP features, the reverse proxy will be able to serve multiple domains on the same IP address.

You will also learn how to use caching to improve the performance of your system.

Finally, you will learn how to monitor your system and how to calculate the number of servers needed to handle a certain amount of requests.

# Prepare and setup your environment

## Access your `hosts` file

In this section, you will access your `hosts` file.

The host file is a computer file used by an operating system to map hostnames to IP addresses. The hosts file is a plain text file and is conventionally named `hosts`. The hosts file can be used as an alternative to (or in conjunction with) DNS.

On Unix-like operating systems (Linux and macOS), the hosts file is located at `/etc/hosts`.

On Windows, the hosts file is located at `%WinDir%\System32\Drivers\Etc\Hosts`.

Ensure you can access your `hosts` file and edit it for the next steps.

## Traefik

In this section, you will install and configure Traefik using its official Docker image available on Docker Hub: https://hub.docker.com/_/traefik.

Traefik is an open-source Edge Router that makes exposing/publishing your services on the Internet a fun and easy experience. It receives requests on behalf of your system and finds out which components are responsible for handling them.

Traefik is full of features and can be used as a reverse proxy, a load balancer, a Kubernetes ingress controller, and much more.

One of the main features of Traefik is to issue and renew Let's Encrypt (HTTPS) certificates automatically in conjuncture with Docker Compose labels.

We will go into more details about Traefik in the next sections.

Run the `traefik-insecure` example from the heig-vd-dai-course/heig-vd-dai-course-code-examples repository. Read the README carefully. Take some time to explore the code, it should contain comments to help you understand what is going on.

You should be able to access the Traefik dashboard at http://localhost:8080 and http://traefik.localhost.

### Alternatives

*Alternatives are here for general knowledge. No need to learn them.*

- Caddy
- Nginx
- Apache
- HAProxy
- certbot

*Missing item in the list? Feel free to open a pull request to add it!*

## whoami

whoami is a tiny Go webserver that prints os information and HTTP request to output.

whoami is used to test various features of Traefik/HTTP.

In the next sections, you will use whoami to test Traefik/HTTP features using its official Docker image available on Docker Hub: https://hub.docker.com/r/traefik/whoami.

### Alternatives

*Alternatives are here for general knowledge. No need to learn them.*

- *None yet*

*Missing item in the list? Feel free to open a pull request to add it!*

# Functional and non-functional requirements

Functional and non-functional requirements are used to define the scope of a system.

It is an abstract representation of the system that will be implemented and that can help to define the architecture of the system.

Functional requirements are the features that a system must have to satisfy the needs of its users. It is the "what" of a system.

Some examples of functional requirements:

- **User management**: Users must be able to register, login, logout, etc.
- **Product management**: Users must be able to create, read, update, delete products, etc.
- **Order management**: Users must be able to create, read, update, delete orders, etc.
- **Payment management**: Users must be able to pay for their orders, etc.

In order to provide the features that a system must have, the system must respect some constraints. These constraints are called non-functional requirements.

Some examples of non-functional requirements:

- **Response time**: Time required between a request and the presentation of the result (most important for the end user)
- **Throughput**: Number of requests handled per time interval (most important for the service provider)
- **Scalability**: Property of a system to handle a varying amount of work - ideally we want linear scalability: 2x more server for 2x more users
- **Availability**: Percentage of time that the system provides a satisfactory service
- **Maintainability**: Ease with which the system can be managed and evolved

- **Security**: Confidentiality, integrity, availability, authentication, authorization, etc.
- ...and many, many more: https://en.wikipedia.org/wiki/Non-functional_requirement

Functional and non-functional requirements are used to define the scope of a system and they strongly depend on/influence the architecture of a system.

It is important to define the functional and non-functional requirements of a system before starting to design it.

Implementing all requirements is not always possible. Some requirements can be in conflict with each other. For example, increasing the security of a system can decrease its performance.

It is always a trade-off between the requirements.

# Web infrastructure definition

Web infrastructures are the software and hardware resources that are used to support the deployment of web applications, web services, and websites on the Internet that respect the functional and non-functional requirements of the system.

Web infrastructures are composed of several components:

- **Web server**: a computer that runs web server software and responds to requests from a web client (e.g. a web browser, a mobile application, a command line tool, etc.)
- **Reverse proxy**: a proxy server that retrieves resources on behalf of a client from one or more servers. These resources are then returned to the client as though they originated from the proxy server itself.
- **Load balancer**: a device that acts as a reverse proxy and distributes network or application traffic across a number of servers.
- **Cache**: a component that stores data so future requests for that data can be served faster.
- **Content delivery network (CDN)**: a geographically distributed network of proxy servers and their data centers.

In this chapter, we will explore a few of these components and how they can be used to build a web infrastructure.

# The Host header

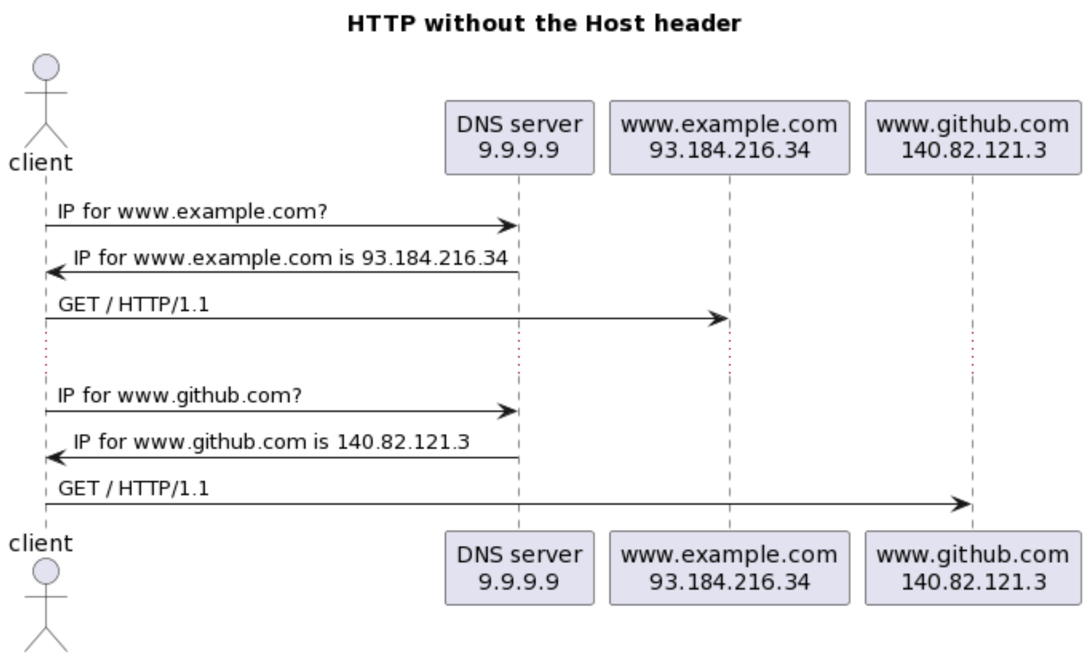
Thanks to the HTTP protocol, web infrastructures can be built easily. HTTP offers many features that can be used to build a web infrastructure.

One of the feature HTTP offers is the Host header.

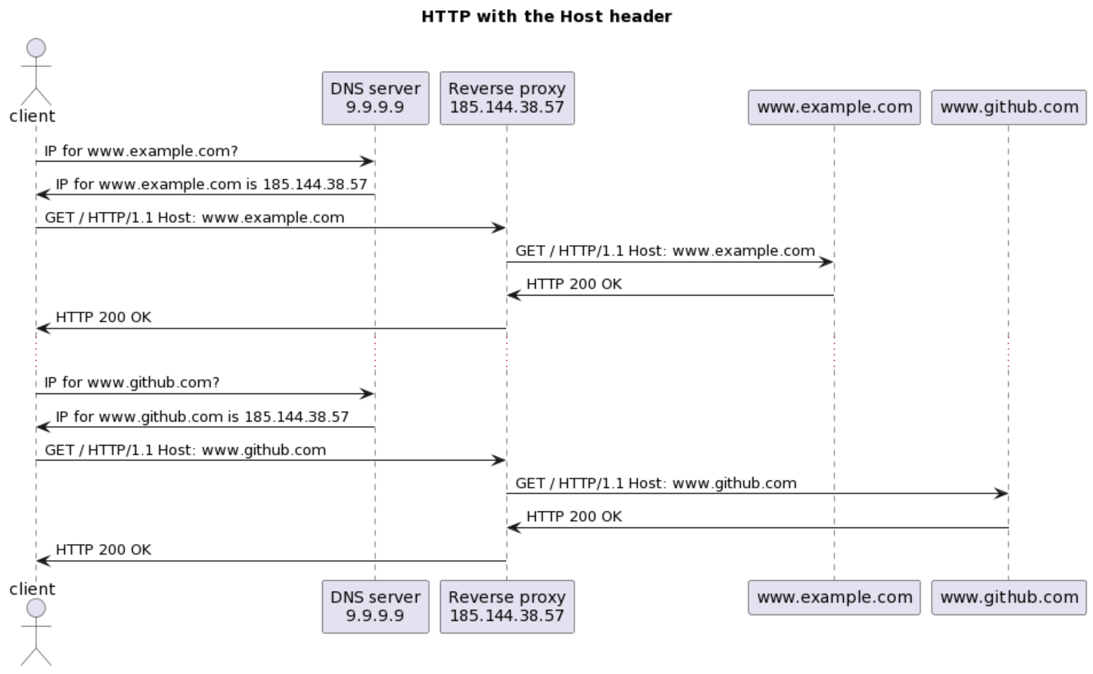
The Host header is a header which specifies the domain name of the server. It is a header sent by the client.

Using this header, a server can handle multiple domains on the same IP address with the help of a reverse proxy.

Before the Host header, a server could only handle one domain per IP address. The following diagram shows how it worked:


HTTP without the Host header
client
DNS server 9.9.9.9
www.example.com 93.184.216.34
www.github.com 140.82.121.3
IP for www.example.com?
IP for www.example.com is 93.184.216.34
GET / HTTP/1.1
IP for www.github.com?
IP for www.github.com is 140.82.121.3
GET / HTTP/1.1

The following PlantUML diagram shows how the Host header works:


HTTP with the Host header
client
DNS server 9.9.9.9
Reverse proxy 185.144.38.57
www.example.com
www.github.com
IP for www.example.com?
IP for www.example.com is 185.144.38.57
GET / HTTP/1.1 Host: www.example.com
GET / HTTP/1.1 Host: www.example.com
HTTP 200 OK
HTTP 200 OK
IP for www.github.com?
IP for www.github.com is 185.144.38.57
GET / HTTP/1.1 Host: www.github.com
GET / HTTP/1.1 Host: www.github.com
HTTP 200 OK
HTTP 200 OK

The reverse proxy receives the request from the client and forwards it to the web server. The web server receives the request and sends a response to the reverse proxy. The reverse proxy receives the response and forwards it to the client.

# Forward proxy and reverse proxy

Forward and reverse proxies are two different types of proxies.

A proxy is a component that acts as an intermediary for requests from clients seeking resources from other servers.

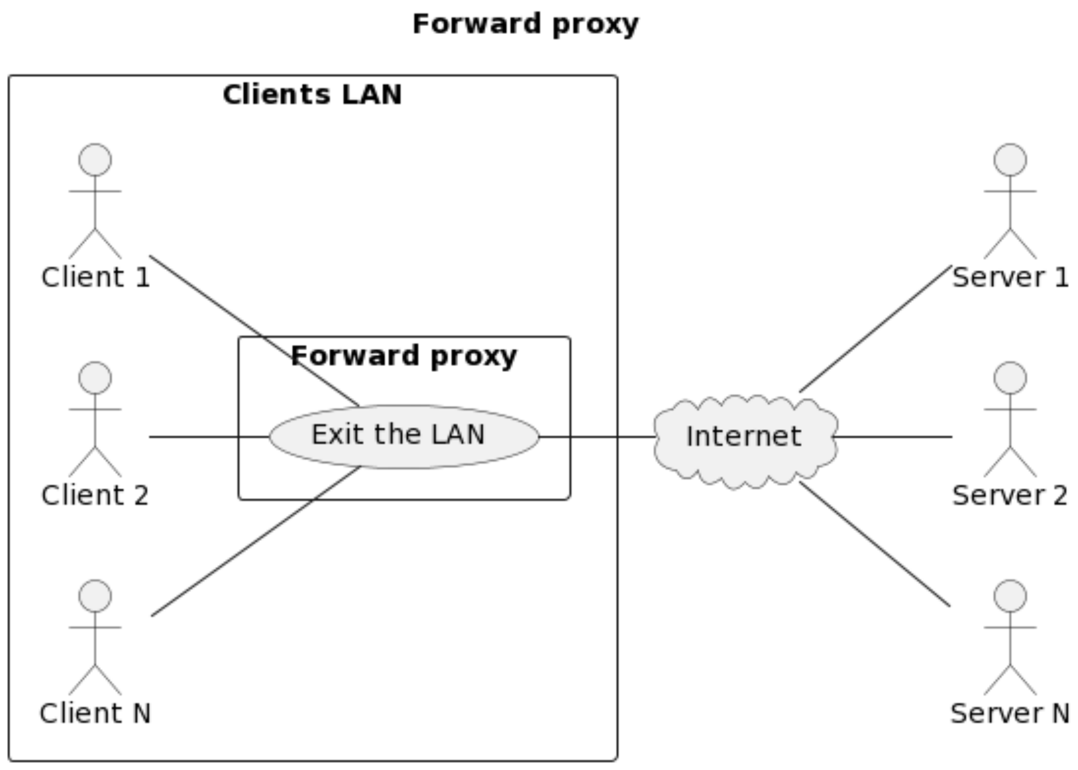
## Forward proxy

As described in Forward Proxy vs. Reverse Proxy: The Difference Explained article, a forward proxy…

> …also known as a proxy server, operates between clients and external systems, regulating traffic, masking client IP addresses, and enforcing security policies.

It acts as an intermediary for requests from clients seeking resources from other servers. A forward proxy is a proxy configured to handle requests from a group of clients to any other server.

Forward proxies are often used in corporate environments to enforce security policies (e.g. block access to certain websites, etc.) and resides in the LAN network.



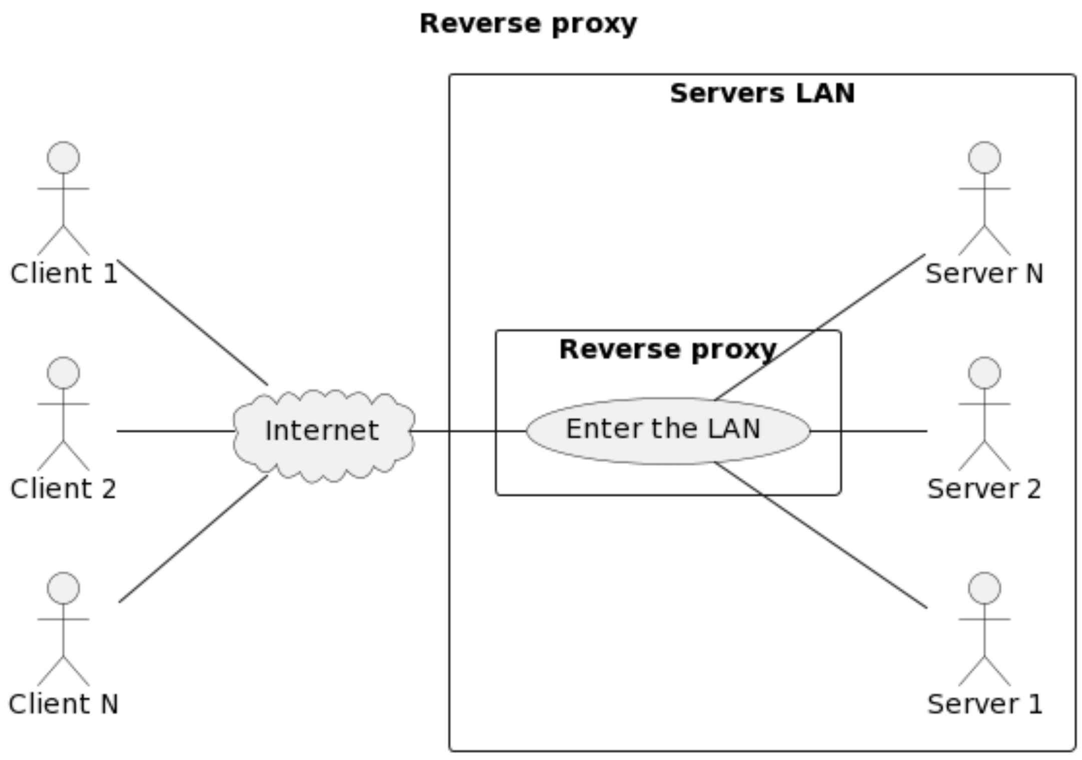
## Reverse proxy

As described in Forward Proxy vs. Reverse Proxy: The Difference Explained article, a reverse proxy…

> …is positioned between clients and servers, acting as a protective barrier for servers by accepting client requests, forwarding them to the appropriate server, and returning the results to the client.

It acts as an intermediary for requests from clients seeking resources from other servers. A reverse proxy is a proxy configured to handle requests from any client to a group of servers.

Reverse proxies are often used in web infrastructures to serve multiple domains on the same IP address and to scale.



Reverse proxies are used in web infrastructures to serve multiple domains on the same IP address and to scale.

A reverse proxy is a component that sits in front of one or more servers. It receives requests from clients and forwards them to the servers. It also receives responses from the servers and forwards them back to the clients.

A reverse proxy can be used to:

- **Load balance**: Reverse proxy can receive all traffic and distribute the requests on a cluster of several identical Web server instances
- **Cache**: Reverse proxy can keep in cache responses from servers for a certain amount of time - if the same request is received again, the reverse proxy can return the cached response instead of forwarding the request to the server
- **Encrypt** and **decrypt** traffic: Reverse proxy manages secure HTTPS connections with clients and unsecure HTTP connections with servers
- **Protect** servers from attacks (e.g. DDoS, SQL injection, etc.): By filtering requests, reverse proxy can protect servers from attacks
- **Serve static content** (e.g. images, videos, etc.): Reverse proxy can serve static content from a cache
- **Serve multiple domains** on the same IP address: Reverse proxy can use the `Host` header to forward requests to the correct server

A reverse proxy is a very powerful component that can be used to build a web infrastructure.

Run the `whoami-with-traefik-pathprefix-rule` example from the heig-vd-dai-course/heig-vd-dai-course-code-examples repository. Read the README carefully. Take some time to explore the code, it should contain comments to help you understand what is going on.

You should be able to access whoami at http://localhost/whoami.

The output should be similar to the following:

```
Hostname: 629ffa2f25bd
IP: 127.0.0.1
IP: 172.26.0.3
RemoteAddr: 172.26.0.2:40742
GET /whoami HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:120.0) Gecko/20100101 Firefox/
120.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/
*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: en-GB,en;q=0.5
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
Upgrade-Insecure-Requests: 1
X-Forwarded-For: 192.168.65.1
X-Forwarded-Host: localhost
X-Forwarded-Port: 80
X-Forwarded-Proto: http
X-Forwarded-Server: 880dafe26d2a
X-Real-Ip: 192.168.65.1
```

Note the following headers:

- `Hostname`: the domain name of the whoami container
- `IP`: the IP addresses of the whoami container
- `GET /whoami HTTP/1.1`: the HTTP request from the client
- `Host: localhost`: the domain name requested by the client

- X-Forwarded-For: the IP address of the client
- X-Forwarded-Host: the domain name of the client
- X-Forwarded-Port: the port of the client
- X-Forwarded-Proto: the protocol of the client
- X-Forwarded-Server: the hostname of the reverse proxy container

Using both the Host header, the HTTP request and the X-Forwarded-* headers, the reverse proxy can forward the request to the correct server (whoami container).

The request from the client has been forwarded to the reverse proxy. The reverse proxy has forwarded the request to the whoami container. The whoami container has sent a response to the reverse proxy. The reverse proxy has forwarded the response to the client.

Now, run the whoami-with-traefik-host-rule example from the heig-vd-dai-course/heig-vd-dai-course-code-examples repository. Read the README carefully. Take some time to explore the code, it should contain comments to help you understand what is going on.

You should be able to access whoami at http://whoami.localhost.

Same as before, but this time, the reverse proxy has mapped the whoami.localhost domain to the whoami container.

Now, run the whoami-with-traefik-host-and-pathprefix-rules example from the heig-vd-dai-course/heig-vd-dai-course-code-examples repository. Read the README carefully. Take some time to explore the code, it should contain comments to help you understand what is going on.

You should be able to access whoami at http://whoami.localhost/whoami.

Same as before, but this time, the reverse proxy has mapped the whoami.localhost domain and the /whoami path to the whoami container.

Now, run the whoami-with-traefik-pathprefix-rule-and-stripper-middleware example from the heig-vd-dai-course/heig-vd-dai-course-code-examples repository. Read the README carefully. Take some time to explore the code, it should contain comments to help you understand what is going on.

You should be able to access whoami at http://localhost/whoami-without-stripprefix and http://localhost/whoami-with-stripprefix.

The output of http://localhost/whoami-without-stripprefix should be similar to the following:

```
Hostname: 25eeca6ff2bb
IP: 127.0.0.1
IP: 172.26.0.5
RemoteAddr: 172.26.0.2:55338
GET /whoami-without-stripprefix HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:120.0) Gecko/20100101 Firefox/
120.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/
*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: en-GB,en;q=0.5
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
Upgrade-Insecure-Requests: 1
X-Forwarded-For: 192.168.65.1
X-Forwarded-Host: localhost
X-Forwarded-Port: 80
X-Forwarded-Proto: http
X-Forwarded-Server: 880dafe26d2a
X-Real-Ip: 192.168.65.1
```

The output of http://localhost/whoami-with-stripprefix should be similar to the following:

```
Hostname: 27cf1df3b435
IP: 127.0.0.1
IP: 172.26.0.4
RemoteAddr: 172.26.0.2:33656
GET / HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:120.0) Gecko/20100101 Firefox/
120.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/
*;q=0.8
Accept-Encoding: gzip, deflate, br
```

```
Accept-Language: en-GB,en;q=0.5
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
Upgrade-Insecure-Requests: 1
X-Forwarded-For: 192.168.65.1
X-Forwarded-Host: localhost
X-Forwarded-Port: 80
X-Forwarded-Prefix: /whoami-with-stripprefix
X-Forwarded-Proto: http
X-Forwarded-Server: 880dafe26d2a
X-Real-Ip: 192.168.65.1
```

You should notice the following differences:

- `GET /whoami-without-stripprefix HTTP/1.1`: the HTTP request from the client
- `GET / HTTP/1.1`: the HTTP request from the client

The reverse proxy has stripped the `/whoami-with-stripprefix` path from the request before forwarding it to the whoami container. The whoami container has received the request without the `/whoami-with-stripprefix` path as if it was the root path, thanks to the `stripprefix` middleware.

This configuration can be useful in some contexts (e.g. when you want to serve static content from a subdirectory).

Thanks to these capabilities, a reverse proxy can be used to build a web infrastructure that can serve multiple domains on the same IP address and that can scale.

# System scalability

System scalability is the capability of a system to handle a growing amount of work, or its potential to be enlarged to accommodate that growth.

In web infrastructures, scalability is the capability of a system to handle a growing amount of requests that are sent by clients to one or more servers.

There are two types of scalability:

- **Vertical scaling**: adding more resources (CPU/RAM/Disk) to an existing server. This is also called **scaling up**.
- **Horizontal scaling**: adding more servers to an existing system. Then, use load-balancing to distribute the work between the servers. This is also called **scaling out**.

Some variations of these two main types of scalability exist, such as elastic scaling (adding or removing resources (CPU/RAM/Disk) on the fly). However, we will not cover them in this course.

Both vertical and horizontal scaling are used to improve the performance of a system to handle more requests (e.g. a lot of users on your website).

## Vertical scaling

Vertical scaling is limited by the hardware: at a certain point, you cannot add more/better resources to a server.

## Horizontal scaling

Horizontal scaling is often limited by the software.

Instead of adding more resources to your server, you add more (smaller) servers to your system.

Your software must be able to scale horizontally as well (multiple backends/API accessing to one or more databases at the same time, multiple frontends accessing the same backends, etc.), and it is not always possible.

## When to use scale up or scale out?

Scaling must be determined by the non-functional requirements (= needs) of of your system. In order to determine the best scaling strategy, you must know the bottlenecks of your system. Bottlenecks can only be identified by load testing your system and monitoring it to get metrics.

There are a few metrics that can help you to determine the bottlenecks of your system:

- CPU usage
- Memory usage
- Disk usage
- Network usage
- Number of requests per second
- Response time
- Availability
- etc.

**Only from metrics, you can determine the bottlenecks of your system.**

Once you know the bottlenecks of your system, you can determine the best scaling strategy.

I (Ludovic) recommend to use vertical scaling as much as possible before switching to horizontal scaling. Adding new resources to a server is (usually) easy and fast. Horizontal scaling is (much) more complex to setup and maintain and can introduce new issues (e.g. network latency, data consistency, etc.). It is also more expensive (usually).

## How to monitor your system?

There are a lot of tools to monitor your system. Here are a few examples:

- Prometheus
- Grafana
- Sentry

- LibreNMS

Monitoring your system is a complex task and is out of the scope of this course. You will learn more about monitoring in future courses.

# Load balancing

Load balancing is the process of distributing a set of tasks over a set of resources (computing units), with the aim of making their overall processing more efficient.

Thanks to the reverse proxy and the `Host` header, a load balancer can be used to distribute requests between multiple servers to achieve horizontal scaling. This is only possible thanks to the fact that HTTP is a stateless protocol.

In order to distribute requests between multiple servers, a load balancer must know the servers it can forward requests to. This is called a **pool** of servers.

A load balancer can distribute requests between multiple servers using different strategies:

- **Round-robin**: the load balancer forwards requests to each server in the pool in turn.
- **Sticky sessions**: the load balancer forwards requests from the same client to the same server with the help of a cookie.
- **Least connections**: the load balancer forwards requests to the server with the least number of active connections.
- **Least response time**: the load balancer forwards requests to the server with the least response time.
- **Hashing**: the load balancer forwards requests to the server based on a hash of the request (e.g. the IP address of the client, the URL of the request, etc.).

Run the `whoami-with-traefik-host-rule-and-sticky-sessions` example from the heig-vd-dai-course/heig-vd-dai-course-code-examples repository. Read the README carefully. Take some time to explore the code, it should contain comments to help you understand what is going on.

You should be able to access the whoami instances at http://whoami1.localhost and http://whoami2.localhost.

Notice that the `Hostname` and `IP` values are different for each instance.

In the first example, the load balancer uses the round-robin strategy to distribute requests between the three whoami instances.

In the second example, the load balancer uses the sticky-session strategy to always forward requests from the same client to the same whoami instance with the help of a cookie.

The sticky-session strategy is useful when you want to keep the state of a client on the same server: if a client is making an order on your website, you want to keep the state of the order on the same server to avoid issues.

# Caching

Caching is the process of storing data in a cache. A cache is a temporary storage component area where data is stored so that future requests for that data can be served faster.

Caching can be used to improve the performance of a system by serving cached data instead of processing a request again. Caching significantly improves the performance of a system because it avoids processing the same request multiple times.

This has several advantages:

- The client will receive the response faster, especially when the client itself (browser) has cached the response.
- The server does not have to process the request (parse the request, query the database, compose the response, etc).
- The network does not have to carry the messages along the entire path between client and server.

It however introduces some complexity because it is difficult to know when to invalidate a cache. If a cache is not invalidated, it can serve stale data.

Caching can be done on the client-side or on the server-side:

- **Client-side caching**: once a client has received a response from a server, it can store the response in a cache. The next time the client needs the same resource, it can use the cached response instead of sending a new request to the server.
- **Server-side caching**: the server stores data in a cache with the help of a reverse proxy. The next time the server needs the same resource, it can use the cached response instead of processing the request again.

## Managing cache with HTTP

Managing cache is challenging because it is difficult to know when to invalidate a cache. If a cache is not invalidated, it can serve stale data.

There are two main caching models:

- **Expiration model**: the cache is valid for a certain amount of time.
- **Validation model**: the cache is valid until the data is modified.

Expiration and validation are two mechanisms that can be used to control caching.

Expiration is the process of specifying how long a response can be cached.

Validation is the process of checking if a cached response is still valid.

Both can be used at the same time to improve the performance of the system.

Much more details about caching with HTTP can be found on MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching.

### Expiration model

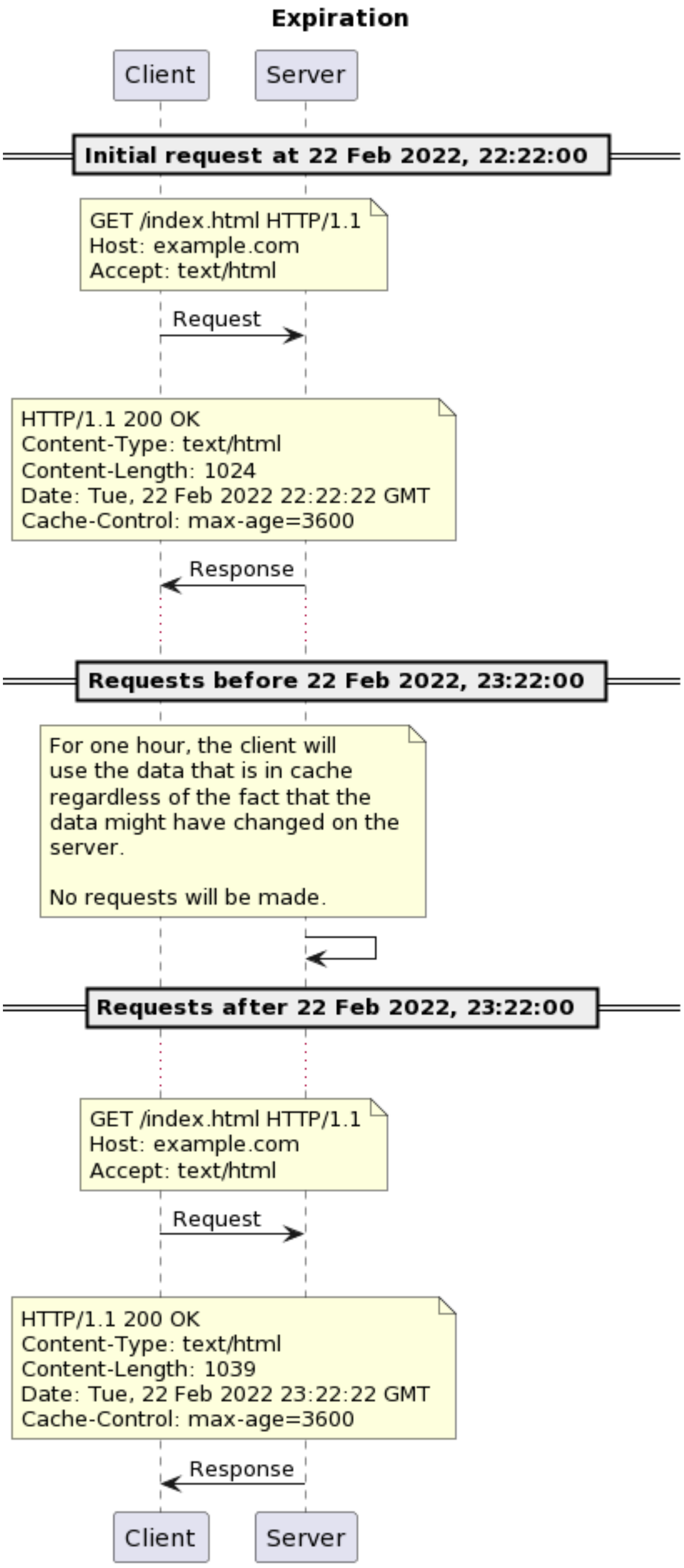
The expiration model is the simplest caching model. It is described in RFC 2616.

The cache is invalidated after a certain amount of time. The cache can be invalidated after a certain amount of time because the data is not expected to change.

The expiration model can be used to cache static content (e.g. images, videos, etc.) or to cache responses from servers to improve the performance of the system.

The expiration model can be implemented with the following header:

- `Cache-Control: max-age=<number of seconds>`: specifies the maximum amount of seconds a resource will be considered fresh. and responses.


Expiration
Client
Server
Initial request at 22 Feb 2022, 22:22:00
Request
Response
Requests before 22 Feb 2022, 23:22:00
Requests after 22 Feb 2022, 23:22:00
Request
Response
Client
Server

```
GET /index.html HTTP/1.1
Host: example.com
Accept: text/html
```

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1024
Date: Tue, 22 Feb 2022 22:22:22 GMT
Cache-Control: max-age=3600
```

For one hour, the client will use the data that is in cache regardless of the fact that the data might have changed on the server.

No requests will be made.

```
GET /index.html HTTP/1.1
Host: example.com
Accept: text/html
```

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1039
Date: Tue, 22 Feb 2022 23:22:22 GMT
Cache-Control: max-age=3600
```

## Validation model

The validation model is more complex than the expiration model. It is described in RFC 2616.

The cache is invalidated when the data is modified. The cache can be invalidated when the data is modified because the data is expected to change.

The validation model can be used to cache responses from servers to improve the performance of the system.

The main idea of the validation model is:

1. Send a request to the server to check if the data has changed.
2. If the data has not changed, the server can return a `304 Not Modified` response to the client.
3. If the data has changed, the server can return a `200 OK` response to the client with the new data.

The request to check if the data has changed is called a **conditional request**.

There are two types of conditional requests:

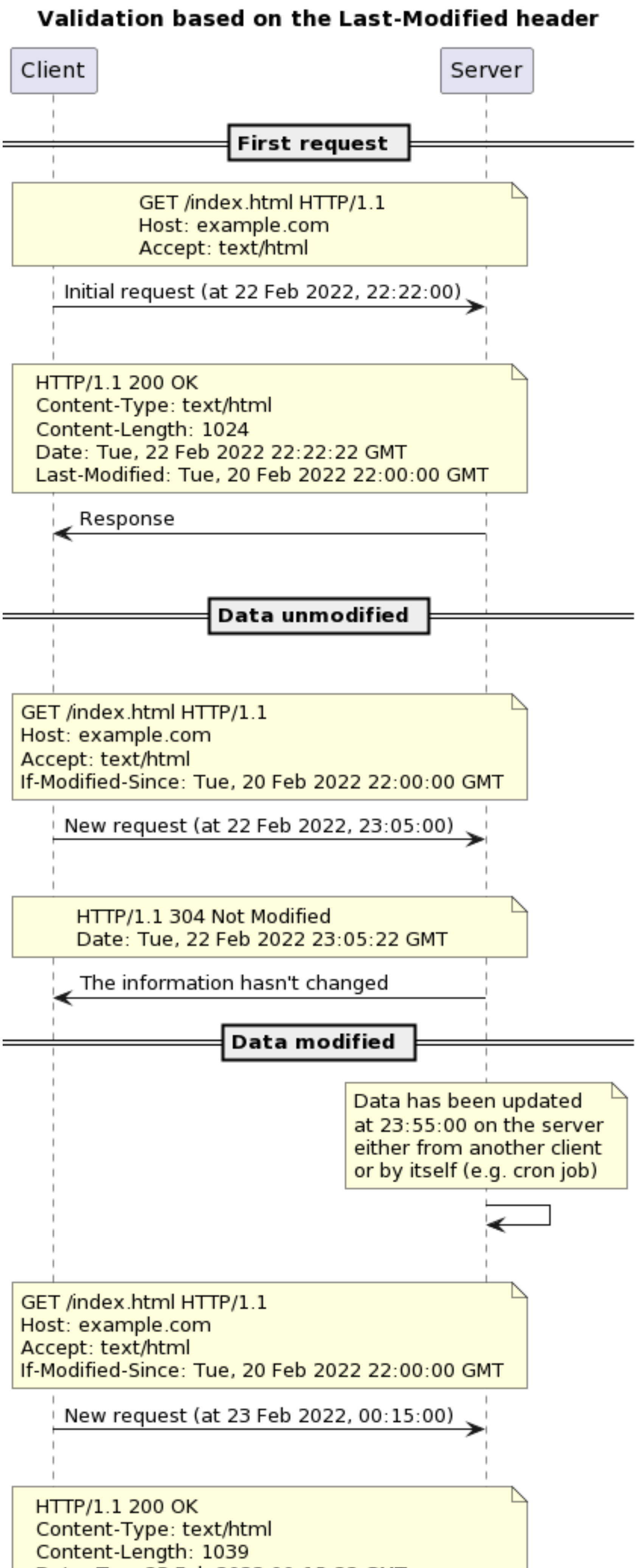
- **Based on the `Last-Modified` header**: allows a `304 Not Modified` to be returned if content is unchanged since the last time it was modified.
- **Based on the `ETag` header**: allows a `304 Not Modified` to be returned if content is unchanged for the version/hash of the given entity.

### *Based on the `Last-Modified` header*

With HTTP, the validation model can be implemented with the following headers:

- `Last-Modified`: indicates the date and time at which the origin server believes the selected representation was last modified.
- `If-Modified-Since`: allows a `304 Not Modified` to be returned if content is unchanged since the time specified in this field (= the value of the `Last-Modified` header).

The `Last-Modified` header is used to check if the data has changed since the last time it was modified.


Validation based on the Last-Modified header
Client
Server
First request
Initial request (at 22 Feb 2022, 22:22:00)
Response
Data unmodified
New request (at 22 Feb 2022, 23:05:00)
The information hasn't changed
Data modified
Data has been updated at 23:55:00 on the server either from another client or by itself (e.g. cron job)
New request (at 23 Feb 2022, 00:15:00)

```
GET /index.html HTTP/1.1
Host: example.com
Accept: text/html
```

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1024
Date: Tue, 22 Feb 2022 22:22:22 GMT
Last-Modified: Tue, 20 Feb 2022 22:00:00 GMT
```

```
GET /index.html HTTP/1.1
Host: example.com
Accept: text/html
If-Modified-Since: Tue, 20 Feb 2022 22:00:00 GMT
```

```
HTTP/1.1 304 Not Modified
Date: Tue, 22 Feb 2022 23:05:22 GMT
```

```
GET /index.html HTTP/1.1
Host: example.com
Accept: text/html
If-Modified-Since: Tue, 20 Feb 2022 22:00:00 GMT
```

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1039
```

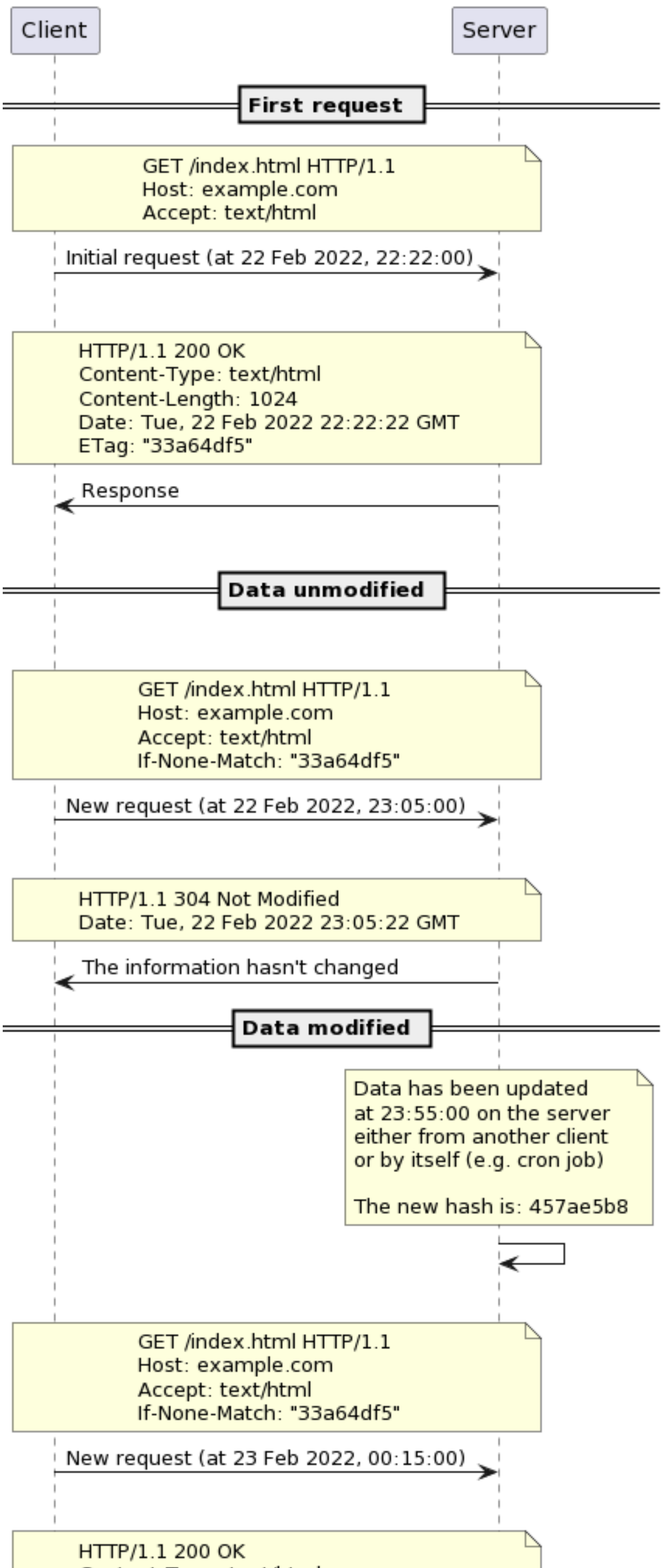
### *Based on the `ETag` header*

With HTTP, the validation model can be implemented with the following headers:

- `ETag`: provides the current entity tag for the selected representation. Think of it like a version number or a hash for the given resource.
- `If-None-Match`: allows a `304 Not Modified` to be returned if content is unchanged for the entity specified (`ETag`) by this field (= the value of the `ETag` header).

The `ETag` header is used to check if the data has changed since the last time it was modified.

**Validation based on the ETag header**


Client
Server
First request
Initial request (at 22 Feb 2022, 22:22:00)
Response
Data unmodified
New request (at 22 Feb 2022, 23:05:00)
The information hasn't changed
Data modified
New request (at 23 Feb 2022, 00:15:00)

```
GET /index.html HTTP/1.1
Host: example.com
Accept: text/html
```

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1024
Date: Tue, 22 Feb 2022 22:22:22 GMT
ETag: "33a64df5"
```

```
GET /index.html HTTP/1.1
Host: example.com
Accept: text/html
If-None-Match: "33a64df5"
```

```
HTTP/1.1 304 Not Modified
Date: Tue, 22 Feb 2022 23:05:22 GMT
```

Data has been updated
at 23:55:00 on the server
either from another client
or by itself (e.g. cron job)

The new hash is: 457ae5b8

```
GET /index.html HTTP/1.1
Host: example.com
Accept: text/html
If-None-Match: "33a64df5"
```

```
HTTP/1.1 200 OK
```

## CDN

Content delivery networks (CDNs) are a type of cache that can be used to serve static content (e.g. images, videos, etc.) to clients.

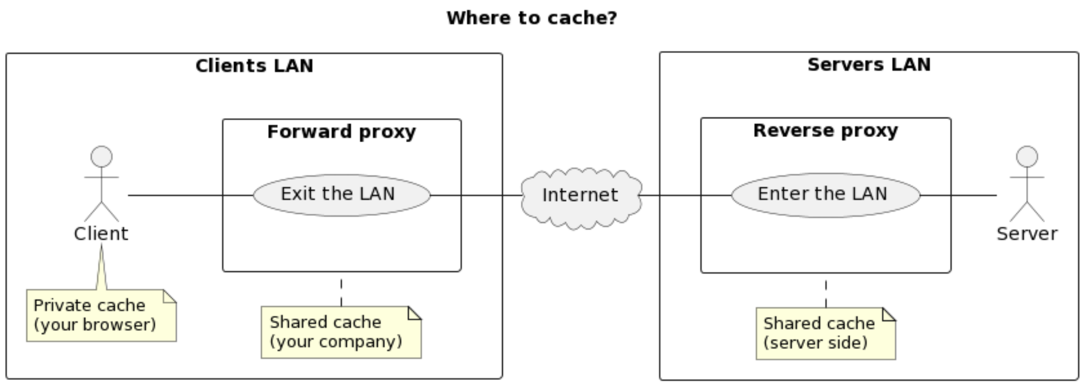
A CDN is a geographically distributed network of proxy servers and their data centers.

A CDN can be used to improve the performance of a system by serving static content to clients from the closest server.

## Where to cache?

Caching can be done on the client-side, on the server-side, or on a CDN.

Private caches are caches that are only used by one client. Public caches are caches that are used by multiple clients.



The best would be to cache at each level of the system to ensure the best performance but it is not always possible or faisable.

# Go further

This is an optional section. Feel free to skip it if you do not have time.

- Are you able to add a basic authentication to the Traefik dashboard using a middleware?

# Conclusion

## What did you do and learn?

In this chapter, you have learned about functional and non-functional requirements, what a web infrastructure is and what components it is composed of, what a reverse proxy and a load balancer are and how they can be used to build a web infrastructure.

Thanks to the `Host` header, you have learned how a reverse proxy can serve multiple domains on the same IP address.

Thanks to the following features of HTTP, you were able to make use of them to build a web infrastructure to serve multiple domains on the same IP address and to scale:

- **Statelessness**: HTTP servers don't have to store information about the state of a client: the client has to send all the information with each request (a cookie session for example) so the server can find the context to handle the request.
- **Scalability**: Several identical servers can handle requests without coordination: the client can send a request to any server, and the server can handle the request.
- **Reliability**: After a server failure, another server can easily take over the work (if the server is stateless).

## Test your knowledge

At this point, you should be able to answer the following questions:

- What is a reverse proxy? What is a load balancer? How do they differ?
- What is the `Host` header? How can it be used to serve multiple domains on the same IP address?
- What is the difference between vertical and horizontal scaling?
- What is the difference between a CDN and a reverse proxy cache?
- What is the difference between expiration and validation caching models?

# Finished? Was it easy? Was it hard?

Can you let us know what was easy and what was difficult for you during this chapter?

This will help us to improve the course and adapt the content to your needs. If we notice some difficulties, we will come back to you to help you.

➡ GitHub Discussions

You can use reactions to express your opinion on a comment!

## What will you do next?

You will start the practical work!

# Additional resources

*Resources are here to help you. They are not mandatory to read.*

- *None yet*

*Missing item in the list? Feel free to open a pull request to add it!*

# Sources

- Main illustration by Nicolas Picard on Unsplash