# HTTP and curl - Course material

https://github.com/heig-vd-dai-course

Markdown · PDF

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

So far in the course, you have built network applications using the TCP and UDP protocols.

You have mastered these protocols and you are now able to build network applications using them.

TCP and UDP are low-level protocols. They are used to transfer data between computers. They do not define how the data should be structured.

It is up to the developers to define how the data should be structured, using an application protocol, for example.

This is where the HTTP protocol comes into play.

In this final part, you will learn how to use the HTTP protocol to build network applications using all the features offered by this protocol.

This will allow you to build more complex network applications, built on top of a solid foundation: HTTP.

As HTTP offers many features and is a very complex protocol, this chapter will be a mixed between theory and practice to introduce you to the most important concepts.

# Disclaimer

In this chapter, you will learn and experiment with the HTTP protocol. We will focus on the version 1.1 of the protocol as it is the most used version today and is supported by all browsers. Other versions of the protocol will be mentioned but will not be covered in details.

You will also experiment with HTTP with the help of Javalin.

Javalin is a lightweight web framework for Java and Kotlin. It is built on top of Jetty.

Even though you will have a good understanding of HTTP at the end of this chapter, **this is not a web course**.

The web is a complex ecosystem with many different technologies. HTTP is only one of them. You will see other technologies in future courses.

Javalin is the perfect tool to learn and experiment with HTTP. However, it is not a production-ready library. It is only meant to be used for learning purposes and for prototyping.

If you want to develop a network application using HTTP that you want to use in production, you will have to use a third-party library such as Quarkus or Spring Boot.

As these libraries are out of the scope of this course (and mostly because you will see them details in future courses), we will not use them.

# Prepare and setup your environment

## curl

In this section, you will start curl using its official Docker image available on Docker Hub: https://github.com/curl/curl-container.

curl is a command line tool used to transfer data over the Web. It supports numerous protocols including HTTP, HTTPS, FTP, FTPS, SFTP, etc.

curl is a very powerful tool. It is used by developers to test their APIs.

### Start and configure curl

To start curl, run the following command:

```
# Pull the Docker image
docker pull curlimages/curl:latest

# Start the Docker image
docker run --rm curlimages/curl:latest
```

The output should be similar to the following:

```
Unable to find image 'curlimages/curl:latest' locally
latest: Pulling from curlimages/curl
96526aa774ef: Already exists
b3ed3d59459c: Pull complete
4f4fb700ef54: Pull complete
Digest: sha256:4a3396ae573c44932d06ba33f8696db4429c419da87cbdc82965ee96a37dd0af
Status: Downloaded newer image for curlimages/curl:latest
curl: try 'curl --help' or 'curl --manual' for more information
```

Now start the container overwriting the default entrypoint to access the container:

```
# Start the Docker image
docker run --rm -it --entrypoint /bin/sh curlimages/curl:latest
```

The output should be similar to the following:

```
~ $
```

You are now in the container. You should be able to use `curl` inside the container for the following sections. To exit the container, type `exit` and press `Enter`.

### Alternatives

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

- Bruno
- Insomnia
- Postman

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

### 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!*

## Javalin

In this section, you will create a new Maven project and add Javalin to the project.

Javalin is a lightweight web framework for Java and Kotlin. It is built on top of Jetty.

### Create and configure a new IntelliJ IDEA project

Create a new IntelliJ IDEA project as seen in the Java, IntelliJ IDEA and Maven chapter.

## Add Javalin to the project

Add the latest **stable** version of Javalin available in the Maven repository: https://mvnrepository.com/artifact/io.javalin/javalin to the pom.xml file as seen in previous chapters:

```
<!-- https://mvnrepository.com/artifact/io.javalin/javalin-bundle -->
<dependency>
  <groupId>io.javalin</groupId>
  <artifactId>javalin-bundle</artifactId>
  <version>5.6.3</version>
</dependency>
```

As stated in the official documentation, the pom.xml file must be slightly modified to correctly use the Maven shade plugin.

Update the following properties to the pom.xml file with the following content:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>3.5.0</version>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>shade</goal>
          </goals>
          <configuration>
            <transformers>
              <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                <mainClass>ch.heigvd.dai.Main</mainClass>
              </transformer>
              <transformer
implementation="org.apache.maven.plugins.shade.resource.DontIncludeResourceTransformer">
                <resource>MANIFEST.MF</resource>
              </transformer>
            </transformers>
```

```
                <filters>
                  <filter>
                    <artifact>*:*</artifact>
                    <excludes>
                      <exclude>META-INF/*.SF</exclude>
                      <exclude>META-INF/*.DSA</exclude>
                      <exclude>META-INF/*.RSA</exclude>
                    </excludes>
                  </filter>
                </filters>
              </configuration>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
```

The difference with the previous pom.xml file is the addition of the filters section. This section is required to correctly use the Maven shade with Javalin.

### Update the Main.java file

Update the Main.java file with the following code:

```
package ch.heigvd;

import io.javalin.Javalin;

public class Main {
  public static final int PORT = 8080;

  public static void main(String[] args) {
    Javalin app = Javalin.create();

    app.get("/", ctx -> ctx.result("Hello, world!"));

    app.start(PORT);
  }
}
```

Run the application and open your browser at http://localhost:8080. You should see the following:

```
Hello, world!
```

Using curl, you can also access the server:

```
# Send a GET request to the server
curl "http://host.docker.internal:8080"
```

The host `host.docker.internal` is a special host that allows you to access the host from inside the container. If you do not use curl inside a container, you can use `localhost` instead.

The output should be the same as in the browser.

This file will be our starting point for the next sections. In future sections, we will refer to this file as `Main.java`.

## Explore and understand the code

Let's take a look at the code.

```
Javalin app = Javalin.create();
```

This line creates a new Javalin instance.

```
app.get("/", ctx -> ctx.result("Hello, world!"));
```

This line creates a new context for the server. A context is a path on the server. In this case, the context is `/`. This means that the server will respond to requests to the path `/` using the `GET` method (more on this later).

The second parameter is a HTTP handler. It is a functional interface that defines a method to handle HTTP requests. In this case, the method is a lambda expression that sends a response to the client. A lambda expression is a way to define a method in a more concise way, sometimes called an anonymous method.

```
// Example of a lambda expression
() -> System.out.println("Hello, world!");

// Example of a lambda expression with parameters
(String name) -> System.out.println("Hello, " + name + "!");
```

```
app.start(PORT);
```

This line will start the server on the port 8080. You might have noticed that no concurrency is specified. This is because Javalin uses good defaults that they describe in their documentation. You will not have to worry about concurrency in this chapter as Javalin will handle it for you.

You now have a basic HTTP server running on your computer. It is time to learn more about HTTP!

### Alternatives

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

- Quarkus
- Spring Boot

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

### Resources

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

- Javalin documentation

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

# HTTP

Hyper Text Transfer Protocol (HTTP) is a protocol used to transfer data over the Web based on TCP. It is a client-server protocol based on the request-response pattern: a client (called **user agent** in the HTTP specification) sends a request to a server, the server processes the request and sends a response to the client.

A client can be a web browser, a command line tool, a mobile application, etc.

The client requests a resource from the server. A resource can be a web page, an image, a video, etc.

HTTP was initiated by Tim Berners-Lee at CERN in 1989. It was first used in 1990 to transfer HyperText Markup Language (HTML) documents.

HTML is a markup language used to create web pages that interconnect with each other. It is the primary language used to create web pages.

Over the years, HTTP and HTML have evolved. HTTP is now used to transfer different types of data (HTML, CSS, JavaScript, images, videos, etc.).

Built on top of TCP (until HTTP/2) and UDP (since HTTP/3), HTTP offers numerous features that make it a very powerful protocol.

Servers typically listen on the TCP port 80 for HTTP and 443 for HTTPS.

## HTTP versions

There are several versions of HTTP. The most used are HTTP/1.1, HTTP/2 and HTTP/3.

Each version of HTTP saw the introduction of many features over the years.

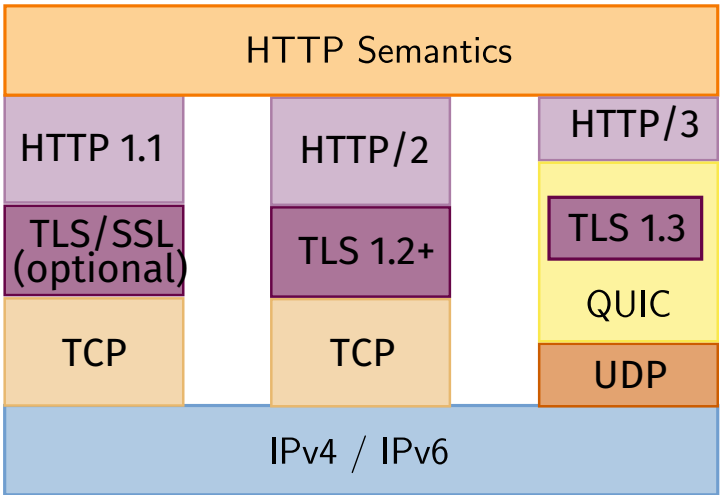
The different versions are:

- HTTP/0.9 (1989)
- HTTP/1.0 (1996)

- HTTP/1.1 (1997)
- HTTP/2 (2015)
- HTTP/3 (2022)

Most features are retro-compatible. This means that a client using HTTP/1.1 can communicate with a server using HTTP/2.

As of today, HTTP/1.1 and HTTP/2 are still the most used versions of HTTP.




### HTTP/0.9

The first version of HTTP was HTTP/0.9 in 1989. It was a very simple protocol only meant to transfer HTML documents.

### HTTP/1.0

HTTP/1.0 was released in 1996. It introduced many features that are still used today (among others):

- HTTP headers
- HTTP status codes
- HTTP methods
- Documents other than HTML (images, videos, etc.) are supported

### HTTP/1.1

HTTP/1.1 was released in 1997, an improved and faster version of HTTP/1.0. It introduced many features that are still used today (among others):

- Persistent connections - The connection between the client and the server is kept alive after the response is sent. This allows the client to send multiple requests over the same connection. Which is more efficient.
- Additional cache control features
- Content negotiation - The client can ask for a specific version of a resource (HTML, CSS, etc.)
- Thanks to the `Host` header, it is possible to host multiple websites on the same server

### HTTP/2

HTTP/2 was released in 2015. The biggest change is the use of a binary protocol instead of a text protocol. This makes it more efficient.

As no much new features were introduced, HTTP/2 was quickly adopted by the industry.

### HTTP/3

HTTP/3 is the latest version of HTTP. It was released in 2022. It is based on the QUIC protocol instead of TCP. QUIC is based on the UDP protocol, making it more efficient.

The main point of HTTP/3 is to make the Web faster and more secure, using a more efficient protocol based on UDP.

## HTTP resources

A resource is identified by an Uniform Resource Locator (URL). A resource can be a web page, an image, a video, etc. A resource can be sometimes be called an **endpoint** (URL + method) or a **route**.

A resource can be static (an JPEG image) or dynamic (a web page generated by a server). A static resource is a file stored on a server. A dynamic resource is generated by a server (data returned in JSON or YAML format - more on this later).

Let's take a look at the following URL (the "Fiche d'unité" of the current course in GAPS):

https://gaps.heig-vd.ch/consultation/fiches/uv/uv.php?id=6573

The URL is composed of the following parts:

- The protocol (`http` or `https`)
- The host (e.g. `gaps.heig-vd.ch`)
- The port number (optional, e.g. `:80` for HTTP or `:443` for HTTPS)
- The path to the resource (e.g. `/consultation/fiches/uv/uv.php`)
- The query parameters (optional, e.g. `?id=6573`)

The URL can also contain the following parts:

- The path parameters (optional, e.g. `/users/{user-id}/view`, where `{user-id}` is a path parameter)
- A subdomain (optional, e.g. `gaps` in `gaps.heig-vd.ch`)

The **host** is sometimes called the **domain name** or the **fully qualified domain name (FQDN)**. It is composed of the following parts:

- The subdomain (optional, e.g. `gaps` in `gaps.heig-vd.ch`)
- The domain name (e.g. `heig-vd`)
- The top-level domain (e.g. `ch`)

This resource is a web page that returns a HTML document. The server will process the request and send the HTML of the web page to the client.

## URL encoding

URLs can only contain a limited set of characters. Some characters are reserved and cannot be used in URLs. For example, the space character cannot be used in URLs.

To send data to the server, you will have to encode the data using the URL encoding (officially known as percent-encoding) format.

URL encoding replaces unsafe ASCII characters with a `%` followed by two hexadecimal digits. URLs cannot contain spaces. Spaces are replaced by `%20`.

For example, the string `Hello world` will be encoded as `Hello%20world`.

Here is a list of the most common characters that must be encoded:

| Character | Encoding |
|---|---|
| Space | %20 |
| ! | %21 |
| " | %22 |
| # | %23 |
| $ | %24 |
| % | %25 |
| & | %26 |
| ' | %27 |
| ( | %28 |
| ) | %29 |
| * | %2A |
| + | %2B |
| , | %2C |
| / | %2F |
| : | %3A |
| ; | %3B |
| = | %3D |
| ? | %3F |
| @ | %40 |
| [ | %5B |
| ] | %5D |

This will be useful in the next sections.

## HTTP request methods

In order to get a resource from a server, the client must send a request to the server.

The request is defined by a method. The most used methods are:

- `GET` - Get a resource (default method - a browser always requests a resource using the HTTP method `GET` by default)
- `POST` - Create a new resource
- `PATCH` - Partially update a resource
- `PUT` - Update a resource (replace the resource - idempotent)
- `DELETE` - Delete a resource

Other methods exist but are out of the scope of this course.

Let's update the `Main.java` file to demonstrate this:

```
package ch.heigvd;

import io.javalin.Javalin;

public class Main {

  public static final int PORT = 8080;

  public static void main(String[] args) {
    Javalin app = Javalin.create();

    app.get("/", ctx ->
        ctx.result("Hello, world from a GET request method!")
    );
    app.post("/", ctx ->
        ctx.result("Hello, world from a POST request method!")
    );
    app.patch("/", ctx ->
        ctx.result("Hello, world from a PATCH request method!")
    );
    app.delete("/", ctx ->
        ctx.result("Hello, world from a DELETE request method!")
    );
```

```
    app.start(PORT);
  }
}
```

We have added a new context for each HTTP method. Each context will respond to the corresponding HTTP method. You might have noticed that we have used the same context for each method. This is because the HTTP method is part of the request.

Run the application and open your browser at http://localhost:8080. You should see the following:

```
Hello, world from a GET request method!
```

Now, let's try to send a POST request to the server using curl:

```
curl -X POST "http://host.docker.internal:8080"
```

The -X option tells curl to use to set the HTTP method.

You should see the following:

```
Hello, world from a POST request method!
```

Try the other methods using curl:

```
# Send a PATCH request to the server
curl -X PATCH "http://host.docker.internal:8080"

# Send a DELETE request to the server
curl -X DELETE "http://host.docker.internal:8080"
```

You should see the different responses.

These methods are used to interact with resources on the server. For example, if you want to create a new user, you will send a POST request to the server. If you want to update a user, you will send a PATCH or PUT request to the server. If you want to delete a user, you will send a DELETE request to the server.

## HTTP request and response format

As seen in the previous section, in order to get a resource from a server, the client must send a request to the server. The request contains the following information:

- The HTTP method
- The URL of the resource
- The supported HTTP version
- Some HTTP headers
- The HTTP body (optional)
- The query parameters (optional)
- The cookies (optional)
- The content type (optional)

The server processes the request and sends a response to the client. The response contains the following information:

- The HTTP version
- The HTTP status code
- The HTTP headers
- The HTTP body (optional)
- The cookies (optional)
- The content type (optional)
- The content length (in HTTP/1.1)
- The content encoding (optional)

You will learn more in details about these elements in the next sections but it is important to understand the structure of a HTTP request and response.

### Structure of a HTTP request

A HTTP request is structured as follows:

<HTTP method> <URL> HTTP/<HTTP version>
<HTTP headers>
<Empty line>
<HTTP body (optional)>

An example of a HTTP request:

```
GET / HTTP/1
Host: gaps.heig-vd.ch
User-Agent: curl/8.1.2
Accept: */*
```

> [!IMPORTANT] Do not forget the empty line!

In this example, we request the resource `/` from the server `gaps.heig-vd.ch` using the HTTP method `GET`.

Some headers as set in the request as well:

- `Host` - The host of the server (gaps.heig-vd.ch in this case)
- `User-Agent` - The user agent that sent the request (curl in this case)
- `Accept` - The content types accepted by the user agent (any type in this case)

You can reproduce this request using curl:

```
# Send a GET request to GAPS
curl -v "http://gaps.heig-vd.ch"
```

The `-v` option tells curl to print the request headers.

The default HTTP method is `GET`. This means that you can omit the HTTP method. A browser always requests a resource using the HTTP method `GET` by default.

The output should be similar to the following:

```
*   Trying 193.134.218.91:80...
* Connected to gaps.heig-vd.ch (193.134.218.91) port 80 (#0)
> GET / HTTP/1.1
> Host: gaps.heig-vd.ch
> User-Agent: curl/8.1.2
> Accept: */*
>
< HTTP/1.1 200 OK
HTTP/1.1 200 OK
< Date: Mon, 27 Nov 2023 17:27:06 GMT
Date: Mon, 27 Nov 2023 17:27:06 GMT
< Server: Apache
Server: Apache
```

```
< Last-Modified: Thu, 23 Feb 2023 15:00:12 GMT
Last-Modified: Thu, 23 Feb 2023 15:00:12 GMT
< ETag: "17df-5f55f450264dd"
ETag: "17df-5f55f450264dd"
< Accept-Ranges: bytes
Accept-Ranges: bytes
< Content-Length: 6111
Content-Length: 6111
< Vary: Accept-Encoding
Vary: Accept-Encoding
< X-Content-Type-Options: nosniff
X-Content-Type-Options: nosniff
< X-Frame-Options: sameorigin
X-Frame-Options: sameorigin
< Content-Type: text/html; charset=ISO-8859-1
Content-Type: text/html; charset=ISO-8859-1

<
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/
TR/html4/loose.dtd">
<html>

<head>
  <title>GAPS/SACHEM</title>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
  <LINK rel="stylesheet" href="style.css" type="text/css">
  <link rel="shortcut icon" href="/img/favicon.ico" />
</head>

<body>
  [...]
</body>

* Connection #0 to host gaps.heig-vd.ch left intact
</html>
```

The > symbol indicates the request headers sent by the client.

The < symbol indicates the response headers sent by the server.

You can notice the first lines of the request that are sent by curl as presented earlier:

```
> GET / HTTP/1.1
> Host: gaps.heig-vd.ch
> User-Agent: curl/8.1.2
> Accept: */*
```

### Structure of a HTTP response

A HTTP response is structured as follows:

```
HTTP/<HTTP version> <HTTP status code> <HTTP status message>
<HTTP headers>
<Empty line>
<HTTP body>
```

An example of a HTTP response from http://gaps.heig-vd:

```
HTTP/1.1 200 OK
Date: Mon, 27 Nov 2023 17:42:47 GMT
Server: Apache
Last-Modified: Thu, 23 Feb 2023 15:00:12 GMT
ETag: "17df-5f55f450264dd"
Accept-Ranges: bytes
Content-Length: 6111
Vary: Accept-Encoding
X-Content-Type-Options: nosniff
X-Frame-Options: sameorigin
Content-Type: text/html; charset=ISO-8859-1

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/
TR/html4/loose.dtd">
<html>

<head>
  <title>GAPS/SACHEM</title>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
  <LINK rel="stylesheet" href="style.css" type="text/css">
  <link rel="shortcut icon" href="/img/favicon.ico" />
</head>
```

```
<body>
  [...]
</body>

</html>
```

In this example, the server responds with the resource / using the HTTP status code 200 (OK).

Some headers as set in the request as well:

- Content-Length - The length of the content in bytes (6111 bytes in this case)
- Content-Type - The content type of the resource (text/html in this case)

The server also sends the content of the resource (the HTML) in the body of the response, separated from the headers by an empty line.

Many other headers are sent by the server. They are not mandatory and some are out of the scope of this course.

You can reproduce this request using curl:

```
# Send a GET request to GAPS
curl -i "http://gaps.heig-vd.ch"
```

The -i option tells curl to print the response headers.

You can notice the first lines of the response that are sent by curl as presented earlier:

```
HTTP/1.1 200 OK
Date: Wed, 06 Dec 2023 18:01:17 GMT
Server: Apache
Last-Modified: Thu, 23 Feb 2023 15:00:12 GMT
ETag: "17df-5f55f450264dd"
Accept-Ranges: bytes
Content-Length: 6111
Vary: Accept-Encoding
X-Content-Type-Options: nosniff
X-Frame-Options: sameorigin
Content-Type: text/html; charset=ISO-8859-1
```

## HTTP response status codes

When a client sends a request to a server, the server processes the request and sends a response to the client.

The response is defined by a status code. Status codes are grouped into five categories:

- 1xx - Informational responses

  The most common informational response are:

  101 - Switching Protocols (the server switches to a different protocol)
  102 - Processing (the server is processing the request)

- 2xx - Successful responses

  The most common successful responses are:

  200 - OK (the request was successful)
  201 - Created (the request was successful and a new resource was created)
  202 - Accepted (the request was accepted but not yet processed)
  204 - No Content (the request was successful but the server does not send any content)

- 3xx - Redirection messages

  The most common redirection messages are:

  301 - Moved Permanently (the resource has been moved permanently to a new URL)
  302 - Found (the resource has been moved temporarily to a new URL)
  304 - Not Modified (the resource has not been modified since the last request)

- 4xx - Client error responses

  The most common client error responses are:

  400 - Bad Request (the request is malformed)
  401 - Unauthorized (the request requires authentication)
  403 - Forbidden (the request is forbidden)

404 - Not Found (the resource does not exist)
405 - Method Not Allowed (the HTTP method is not allowed for this resource)
409 - Conflict (the request could not be processed because of a conflict)
410 - Gone (the resource is no longer available and has been removed)
429 - Too Many Requests (the client has sent too many requests in a given amount of time)

- 5xx - Server error responses

  The most common server error responses are:

  500 - Internal Server Error (the server encountered an unexpected condition that prevented it from fulfilling the request)
  501 - Not Implemented (the server does not support the functionality required to fulfill the request)
  502 - Bad Gateway (the server received an invalid response from an upstream server)
  503 - Service Unavailable (the server is currently unable to handle the request due to a temporary overload or scheduled maintenance)
  504 - Gateway Timeout (the server did not receive a timely response from an upstream server)

The default status code is 200 (OK). This means that you can omit the status code.

Let's update the Main.java file to demonstrate this:

```
package ch.heigvd;

import io.javalin.Javalin;
import io.javalin.http.HttpStatus;

public class Main {

  public static final int PORT = 8080;

  public static void main(String[] args) {
    Javalin app = Javalin.create();
```

```
    app.get("/", ctx ->
        ctx
          .result("Hello, world from a GET request method with a `HttpStatus.OK` response
              status!")
    );
    app.post("/", ctx ->
        ctx
          .result("Hello, world from a POST request method with a `HttpStatus.CREATED`
              response status!")
          .status(HttpStatus.CREATED)
    );
    app.patch("/", ctx ->
        ctx
          .result("Hello, world from a PATCH request method with a `HttpStatus.OK` response
              status!")
          .status(HttpStatus.OK)
    );
    app.delete("/", ctx ->
        ctx
          .result("Hello, world from a DELETE request method with a
              `HttpStatus.NO_CONTENT` response status!")
          .status(HttpStatus.NO_CONTENT)
    );


    app.start(PORT);
  }
}
```

Now, let's try to send a GET request to the server using curl:

```
curl -v http://host.docker.internal:8080
```

The -v option tells curl to print the request headers.

The output should be similar to the following:

```
*   Trying 127.0.0.1:8080...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET / HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.1.2
> Accept: */*
>
```

```
< HTTP/1.1 200 OK
< Date: Mon, 04 Dec 2023 14:44:01 GMT
< Content-Type: text/plain
< Content-Length: 78
<
* Connection #0 to host localhost left intact
Hello, world from a GET request method with a `HttpStatus.OK` response status!
```

The > symbol indicates the request headers sent by the client.

The < symbol indicates the response headers sent by the server.

You will notice that the response status code is 200 (OK), as expected as no status code was specified.

You will go into more details about HTTP headers in the next sections.

Try the other methods using curl:

```
# Send a POST request to the server
curl -v -X POST "http://host.docker.internal:8080"

# Send a PATCH request to the server
curl -v -X PATCH http://host.docker.internal:8080

# Send a DELETE request to the server
curl -v -X DELETE http://host.docker.internal:8080
```

The status codes should be the same as in the code. They are defined by the HttpStatus class and allow to better understand the responses made by the server and how to handle them.

## HTTP path parameters, query parameters and body

In the previous sections, you have seen how to send a request to a server and how to get a response from the server.

In this section, you will learn how to send data to the server and how to get data from the server.

## HTTP path parameters

A HTTP request can contain path parameters. Path parameters are used to send data to the server such as identifiers, etc.

An example of a path parameter is the following:

```
/users/{user-id}
```

With values:

```
/users/123
```

In this example, the path parameter is `{user-id}`. The server will replace the path parameter with the actual value of the parameter.

Let's update the `Main.java` file to demonstrate this by adding a new context with a path parameter:

```
app.get("/path-parameter-demo/{path-parameter}", ctx -> {
  String pathParameter = ctx.pathParam("path-parameter");

  ctx.result("You just called `/path-parameter-demo` with path parameter '" +
             pathParameter + "'!");
});
```

In this example, we have added a new context with the path `/path-parameter-demo/{path-parameter}`. This context will respond to `GET` requests.

The context will get the path parameter `path-parameter` from the request. The server will then respond with a `200` (OK) status code and a message.

Run the application and open your browser at http://localhost:8080/path-parameter-demo/Hello%20world.

Let's try to send a `GET` request to the server using curl:

```
curl http://host.docker.internal:8080/path-parameter-demo/Hello%20world
```

The output should be similar to the following:

```
You just called `/path-parameter-demo` with path parameter 'Hello world'!
```

You can notice the URL encoding in the URL (`Hello%20world`).

Try to access another URL such as http://localhost:8080/path-parameter-demo/path-parameter/not-found.

The output should be similar to the following:

```
Not Found
```

This is because the server does not know this path. The server will respond with a `404` (Not Found) status code.

Let's try to send a `GET` request to the server using curl to display the response headers:

```
curl -i http://host.docker.internal:8080/path-parameters-demo/path-parameter/not-found
```

The output should be similar to the following:

```
HTTP/1.1 404 Not Found
Date: Wed, 06 Dec 2023 17:18:00 GMT
Content-Type: text/plain
Content-Length: 9

Not Found
```

You can notice the `404` status code in the response headers.

## HTTP query parameters

A HTTP request can contain query parameters. Query parameters are used to send data to the server such as filters, search terms, etc.

An example of some query parameters is the following:

```
/users?firstName=John&lastName=Doe
```

In this example, the query parameters are `firstName` and `lastName`. The server will replace the query parameters with the actual values of the parameters.

The query parameters are separated from the path by a `?` character. The query parameters are separated from each other by a `&` character.

Let's update the `Main.java` file to demonstrate this by adding a new context with some query parameters:

```
app.get("/query-parameters-demo", ctx -> {
  String firstName = ctx.queryParam("firstName");
  String lastName = ctx.queryParam("lastName");

  if (firstName == null || lastName == null) {
    throw new BadRequestResponse();
  }

  ctx.result("Hello, " + firstName + " " + lastName + "!");
});
```

In this example, we have added a new context with the path `/query-parameters-demo`. This context will respond to `GET` requests.

The context will get the query parameters `firstName` and `lastName` from the request. If one of the query parameters is missing, the server will respond with a `400` (Bad Request) status code.

If the query parameters are present, the server will respond with a `200` (OK) status code and a message.

Run the application and open your browser at http://localhost:8080/query-parameters-demo?firstName=John&lastName=Doe.

The output should be similar to the following:

```
Hello, John Doe!
```

You can notice that the query parameters were replaced by the actual values (`John` and `Doe`). You can also notice that the query parameters are separated by a `&` character in the URL.

Now try to access the URL http://localhost:8080/query-parameters-demo.

The output should be similar to the following:

```
Bad Request
```

This is because one or both query parameters are missing. The server will respond with a `400` (Bad Request) status code.

Let's try to send a `GET` request to the server using curl to display the response headers:

```
curl -i http://host.docker.internal:8080/query-parameters-demo
```

The output should be similar to the following:

```
HTTP/1.1 400 Bad Request
Date: Wed, 06 Dec 2023 17:32:31 GMT
Content-Type: text/plain
Content-Length: 11

Bad Request
```

You can notice the `400` status code in the response headers.

## HTTP body

A HTTP request can contain a body. The body is used to send data to the server.

A HTTP response can also contain a body. The body is used to send data to the client.

The body is optional. It is not mandatory to send a body with a request or a response. The body is not limited to text. It can contain any type of data (text, images, videos, etc.) and can be of any size, encoded in any format.

The body is separated from the headers by an empty line.

An example of a body is the following:

```
Hello, world!
```

Let's update the `Main.java` file to demonstrate this:

```
app.post("/body-demo", ctx -> {
  String data = ctx.body();

  ctx.result("You just called `/body-demo` with data '" + data + "'!");
});
```

In this example, we have added a new context with the path `/body-demo`. This context will respond to `POST` requests.

The context will get the body from the request. The server will then respond with a `200` (OK) status code and a message.

As this context responds to POST requests, you will have to use curl to send a POST request to the server (as your browser sends GET requests by default):

```
curl -X POST -d "Hello, world!" http://host.docker.internal:8080/body-demo
```

The -d option tells curl to use to set the body.

The output should be similar to the following:

```
You just called `/body-demo` with data 'Hello, world!'!
```

Let's display the request and response headers:

```
curl -i -v -X POST -d "Hello, world!" http://host.docker.internal:8080/body-demo
```

The output should be similar to the following:

```
You just called `/body-demo` with data 'Hello, world!'!~ $ curl -i -v -X POST -d "Hello,
world!" http://host.docker.internal:8080/body-demo
Note: Unnecessary use of -X or --request, POST is already inferred.
*   Trying 192.168.65.254:8080...
* Connected to host.docker.internal (192.168.65.254) port 8080
> POST /body-demo HTTP/1.1
> Host: host.docker.internal:8080
> User-Agent: curl/8.4.0
> Accept: */*
> Content-Length: 13
> Content-Type: application/x-www-form-urlencoded
>
< HTTP/1.1 200 OK
HTTP/1.1 200 OK
< Date: Wed, 06 Dec 2023 17:42:13 GMT
Date: Wed, 06 Dec 2023 17:42:13 GMT
< Content-Type: text/plain
Content-Type: text/plain
< Content-Length: 55
Content-Length: 55

<
* Connection #0 to host host.docker.internal left intact
You just called `/body-demo` with data 'Hello, world!'!
```

You can notice the POST method in the request headers and the 200 status code in the response headers. You can also notice the Content-Length and Content-Type headers in the request and response headers.

## HTTP headers

HTTP headers are used to send additional information in a HTTP request or response.

HTTP headers are separated from the body by an empty line.

An example of a HTTP header is the following:

```
Content-Type: text/plain
```

In this example, the HTTP header is Content-Type and the value of the header is text/plain.

HTTP headers are case-insensitive. This means that Content-Type and content-type are the same.

Here is a list of the most common HTTP headers:

| Header | Description |
|---|---|
| Accept | The media types accepted by the client |
| Content-Type | The media type of the body sent by the client or from the server |
| Content-Length | The length of the body sent by the client or the server |
| User-Agent | The user agent of the client (the browser name and version, the curl version, etc.) |
| Host | The host of the server |
| Set-Cookie | The cookies set by the server |

## HTTP content negotiation

Thanks to the Accept header, the client can ask for a specific version of a resource (HTML, CSS, etc.).

The Accept header is used to specify the media types accepted by the client.

An example of a Accept header is the following:

```
Accept: text/html
```

In this example, the client accepts the media type text/html.

You can find a list of the most common media types on the IANA website.

When requesting a resource, the client can specify the media types it accepts. The server will then respond with the resource in the media type that is the closest to the media types accepted by the client.

HTTP **does not transfer objects**, it **transfers representations of objects**. This means that the server can send the same resource in different representations.

The same resource, i.e. the URL /my-resource, can be sent in different representations:

- Format: HTML, JSON, YAML, PNG, JPEG, etc.
- Language: English, French, German, etc.
- Encoding: UTF-8, UTF-16, etc.

Let's update the Main.java file to demonstrate this:

```
app.get("/content-negotiation-demo", ctx -> {
  String acceptHeader = ctx.header("Accept");

  if (acceptHeader == null) {
    throw new BadRequestResponse();
  }

  if (acceptHeader.contains("text/html")) {
    ctx.contentType("text/html");
    ctx.result("<h1>Hello, world!</h1>");
  } else if (acceptHeader.contains("text/plain")) {
    ctx.contentType("text/plain");
    ctx.result("Hello, world!");
  } else {
    throw new NotAcceptableResponse();
  }
});
```

In this example, we have added a new context with the path `/content-negotiation-demo`. This context will respond to `GET` requests.

The context will get the `Accept` header from the request. If the `Accept` header is missing, the server will respond with a `400` (Bad Request) status code.

If the `Accept` header is present, the server will check if the client accepts the media type `text/html`. If the client accepts the media type `text/html`, the server will respond with a `200` (OK) status code and a HTML message.

If the client does not accept the media type `text/html`, the server will check if the client accepts the media type `text/plain`. If the client accepts the media type `text/plain`, the server will respond with a `200` (OK) status code and a plain text message.

If the client does not accept the media type `text/plain`, the server will respond with a `406` (Not Acceptable) status code.

Run the application and open your browser at http://localhost:8080/content-negotiation-demo.

You should see the following:

```
Hello, world!
```

Your browser accepts the media type `text/html` by default. This is why you see the HTML message.

Now, try to access the URL http://localhost:8080/content-negotiation-demo with curl:

```
curl -v -i http://host.docker.internal:8080/content-negotiation-demo
```

The output should be similar to the following:

```
Not Acceptable
```

This is because curl does not accept the media type `text/html` by default. The server will respond with a `406` (Not Acceptable) status code.

Let's specify the `Accept` header to tell the server that we accept the media type `text/plain`:

```
curl -H "Accept: text/plain" http://host.docker.internal:8080/content-negotiation-demo
```

The output should be similar to the following:

```
Hello, world!
```

Let's display the request and response headers:

```
curl -v -i -H "Accept: text/plain" http://host.docker.internal:8080/content-negotiation-
                demo
```

The output should be similar to the following:

```
*   Trying 192.168.65.254:8080...
* Connected to host.docker.internal (192.168.65.254) port 8080
> GET /content-negotiation-demo HTTP/1.1
> Host: host.docker.internal:8080
> User-Agent: curl/8.4.0
> Accept: text/plain
>
< HTTP/1.1 200 OK
HTTP/1.1 200 OK
< Date: Wed, 06 Dec 2023 18:11:55 GMT
Date: Wed, 06 Dec 2023 18:11:55 GMT
< Content-Type: text/plain
Content-Type: text/plain
< Content-Length: 13
Content-Length: 13

<
* Connection #0 to host host.docker.internal left intact
Hello, world!
```

You can notice the client `Accept` header in the request headers and the `Content-Type` header in the response headers.

This feature is very useful to serve different versions of a resource to different clients.

## HTTP sessions (stateless vs. stateful)

Stateless and stateful are two terms that are often used in computer science.

A stateless application is an application that loses its state when it is restarted. This means that the application does not store any data on the disk or a database or stores the data in memory (RAM).

A stateful application is an application that keeps its state when it is restarted. This means that the application stores data on the disk or a database.

This concept can be applied to network protocols such as HTTP as well.

**HTTP is a stateless protocol**: for each request, the server does not know who the client is or what the client did before. This means that the server does not keep track of the state of the client over time and there is no way to know who made a request.

We need to keep track of the state of the client in order to build applications that are stateful such as a login system, a shopping cart, etc.

Let's illustrate this with an example. Imagine that you have a website with a few pages that you can access:

- A homepage (`/` - public)
- A login page (`/login` - public)
- A profile page (`/profile` - private)

The profile page returns the information of the user. This page is private. This means that you need to log in to access it.

In a stateful protocol, the server keeps track of the state of the client. This means that the server knows who you are. This means that the server knows what you did before:

1. You arrive on the homepage. You can click on a link to access the login page.
2. You fill in your username and password and you click on the "Login" button.
3. The server checks your credentials and logs you in.
4. You are now logged in. You can click on a link to access your profile page.
5. The server knows who you are. The server knows that you are logged in. The server can send you the profile page.

HTTP is **not a stateful protocol**. This means that the server does not keep track of the state of the client.

Using the same example as before, in a stateless protocol, the server does not know who you are. This means that the server does not know what you did before:

1. You arrive on the homepage. You can click on a link to access the login page.
2. You fill in your username and password and you click on the "Login" button.
3. The server checks your credentials and logs you in.
4. You are now logged in. You can click on a link to access your profile page.
5. The server does not know who you are. The server does not know that you are logged in. The server cannot send you the profile page.

The example above is almost the same as the previous one. The only difference is that with HTTP, it is up to the developer to implement a way to keep track of the state of the client and send it to the server with each request.

In order to achieve this, the server can use HTTP sessions. There are many ways to implement HTTP sessions. Here are two of them:

- Using a query parameter
- Using cookies

### HTTP sessions using a query parameter

Using the previous example, the server can use a query parameter to keep track of the state of the client:

1. Once a user is logged in, the server generates a random token and stores it in a database.
2. The server sends the token to the client in a query parameter.
3. The client stores the token and sends it back to the server with each request.
4. The server checks if the token is valid and retrieves the user from the database.
5. The server can then send the profile page to the client.

This would look like the following:

```
C -> S: POST /login
S -> C: 302 Found (redirect to /profile?token=1234567890)
C -> S: GET /profile?token=1234567890
S -> C: 200 OK (profile page)
```

This is a very simple way to implement HTTP sessions. It is not very secure as the token is sent in clear text in the URL. This means that anyone can see the token and use it to access the profile page.

### HTTP sessions using cookies

In a very similar way as the previous example, the server can use cookies to keep track of the state of the client.

Cookies are small pieces of data sent by the server to the client and sent back to the server with each request.

It is part of the HTTP protocol and is meant to store data on the client side. A cookie is identified by a name and a value and is sent in the `Set-Cookie` header.

The MDN Web Docs has a very good documentation on cookies: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies.

The server can use cookies to keep track of the state of the client:

1. Once a user is logged in, the server generates a random token and stores it in a database.
2. The server sends the token to the client in a cookie.
3. The client stores the cookie and sends it back to the server with each request.

This would look like the following:

```
C -> S: POST /login
S -> C: 302 Found (redirect to /profile and set a cookie with the token)
C -> S: GET /profile (the cookie is sent by the client)
S -> C: 200 OK (profile page)
```

Let's update the `Main.java` file to demonstrate this:

```
app.get("/cookie-demo", ctx -> {
  String cookie = ctx.cookie("cookie");

  if (cookie == null) {
    ctx.cookie("cookie", "cookie-demo");

    ctx.result("You just called `/cookie-demo` without a cookie. A cookie is now set!");
  } else {
    ctx.result("You just called `/cookie-demo` with a cookie. Its value is '" + cookie + "'!");
  }
});
```

In this example, we have added a new context with the path `/cookie-demo`. This context will respond to `GET` requests.

The context will get the cookie `cookie` from the request. If the cookie is missing, the server will set a new cookie and respond with a `200` (OK) status code and a message.

If the cookie is present, the server will respond with a `200` (OK) status code and a message with the value of the cookie.

Run the application and open your browser at http://localhost:8080/cookie-demo.

The first time you access the page, you should see the following:

```
You just called `/cookie-demo` without a cookie. A cookie is now set!
```

The server has set a cookie with the name `cookie` and the value `cookie-demo`.

Now, refresh the page. You should see the following:

```
You just called `/cookie-demo` with a cookie. Its value is 'cookie-demo'!
```

The browser has sent the cookie with the name `cookie` and the value to the server.

You can access the cookies in your browser using the developer tools:

- Firefox: https://developer.mozilla.org/en-US/docs/Tools/Storage_Inspector
- Chromium-based browsers (Chrome, Edge, etc.): https://developer.chrome.com/docs/devtools/application/cookies

- Safari: https://developer.apple.com/safari/tools/

Try to update the value of the cookie and refresh the page. You should see the new value of the cookie.

If you update the value of the cookie to something else than `cookie-demo`, the server will display its new value. Just be aware that some characters are not allowed in cookies. See the MDN Web Docs documentation on cookies for more information: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie#cookie-namecookie-value.

Cookies are very useful to keep track of the state of the client. They are automatically sent by the browser with each request. There is no need to do anything.

# API design

Now that you have a better understand of the basics of HTTP, you will learn how to design and develop APIs.

Developing a web application is not easy. You need to think about the architecture of the application, the database, the security, etc.

In order to make it easier to develop web applications, developers have developed some patterns to follow.

An Application Protocol Interface (API) is a set of functions and procedures that allow the creation of applications that access the features or data of an operating system, application or other service.

An API is a contract between the client and the server. It defines how the client and the server should interact with each other.

Using an API, the client can send requests to the server and get responses from the server. This allow to use many different clients (web browsers, mobile applications, desktop applications, etc.) with the same server.

Most APIs are based on the HTTP protocol. This means that they use the HTTP methods (GET, POST, PUT, PATCH, DELETE, etc.) to define the operations that can be performed on the resources.

The common format to send data to the server and get data from the server is JSON. JSON is a lightweight data interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate.

JSON is the standard format used by most APIs. It is not mandatory to use JSON but it is recommended.

Javalin allows the serialization and deserialization of JSON. This means that you can transform a Java Object to JSON and vice versa.

## Simple APIs with CRUD operations

Designing an API is not a trivial task. Most of the time, APIs are designed around the CRUD pattern.

CRUD stands for Create, Read, Update and Delete. It is a pattern that is used to design APIs.

The CRUD pattern is based on the four basic operations of persistent storage:

- Create - Create a new resource
- Read - Read a resource
- Update - Update a resource
- Delete - Delete a resource

The CRUD pattern is very simple and easy to understand. It is very useful to design APIs.

## REST APIs

REST stands for Representational State Transfer. It is a pattern that is used to design APIs but has to follow strict rules and are more complex to design than simple CRUD APIs.

The REST pattern is based on the six following principles:

1. Client / server architecture: client and server are completely separated and only interact through the API
2. Stateless: the server does not retain any session information. Requests from the client must include all the information necessary to process it.
3. Cache-ability: a REST API should support caching of responses by the client and control which responses can be cached and which not.
4. Layered system: it should be able to add intermediate systems (cache, load balancer, security gateway) without any impact for the client
5. Uniform interface:
   Use URIs/URLs to identify resources
   Server responses use a standard format that includes all information required by the client to process the data (modify or delete the resource's state)
   Server responses include links that allow the client to discover how to interact with a resource

6. Code on demand (optional): responses may include executable code to customize functionality of the client

All REST APIs are APIs but not all APIs are REST APIs.

REST APIs are hard to implement correctly. In this course, we will stay with CRUD APIs. We mention REST APIs for completeness.

## Simple API with CRUD operations example

Let's design a simple CRUD API for a user resource.

We will architecture the application using the Domain Driven Design (DDD) approach with two domains:

- The `users` domain that contains the user resource
- The `auth` domain that contains the authentication logic

The user resource has the following properties:

- `id` - The unique identifier of the user
- `firstName` - The first name of the user
- `lastName` - The last name of the user
- `email` - The email address of the user
- `password` - The password of the user

The user resource has the following operations:

- Create a new user
- Get many users that you can filter by first name and/or last name
- Get one user by its ID
- Update a user
- Delete a user

The user resource has the following endpoints:

- `POST /users` (or `PUT /users`) - Create a new user
- `GET /users` - List all users
- `GET /users?firstName={firstName}&lastName={lastName}` - Search users by first name and/or last name
- `GET /users/{id}` - Read a user
- `PUT /users/{id}` (or `PATCH /users/{id}`) - Update a user
- `DELETE /users/{id}` - Delete a user

In order to demonstrate the previous example of a stateful/stateless application, the auth domain will contain the following operations:

- Login a user
- Logout a user
- Get the current user (the user that is logged in)

The auth domain has the following endpoints:

- `POST /login` - Login a user
- `POST /logout` - Logout a user
- `GET /profile` - Get the current user

Let's implement this API using Javalin.

Start by creating a new directory `src/main/java/ch/heigvd/users` and a new file `User.java`:

```
package ch.heigvd.users;

public class User {

    public Integer id;
    public String firstName;
    public String lastName;
    public String email;
    public String password;

    public User() {
        // Empty constructor for serialisation/deserialization
    }
}
```

This class represents a user. It has the following properties:

- `id` - The unique identifier of the user
- `firstName` - The first name of the user
- `lastName` - The last name of the user
- `email` - The email address of the user
- `password` - The password of the user

It has one constructor to allow the serialization and deserialization of the user. In a production application, you would certainly make all fields private and use getters and setters. For the sake of simplicity, we will use public fields.

Now, let's create a new class UsersController.java in the same directory:

```
package ch.heigvd.users;

import io.javalin.http.*;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicInteger;

public class UsersController {

    private final ConcurrentHashMap<Integer, User> users;
    private final AtomicInteger userId = new AtomicInteger();

    public UsersController(ConcurrentHashMap<Integer, User> users) {
        this.users = users;
    }

    public void create(Context ctx) {
        User newUser = ctx.bodyValidator(User.class)
                .check(obj -> obj.firstName != null, "Missing first name")
                .check(obj -> obj.lastName != null, "Missing last name")
                .check(obj -> obj.email != null, "Missing email")
                .check(obj -> obj.password != null, "Missing password")
                .get();

        for (User user : users.values()) {
            if (user.email.equals(newUser.email)) {
                throw new ConflictResponse();
            }
        }

        User user = new User();
```

```
        user.id = userId.getAndIncrement();
        user.firstName = newUser.firstName;
        user.lastName = newUser.lastName;
        user.email = newUser.email;
        user.password = newUser.password;

        users.put(user.id, user);

        ctx.status(HttpStatus.CREATED);
        ctx.json(user);
    }

    public void getOne(Context ctx) {
        Integer id = ctx.pathParamAsClass("id", Integer.class)
                .check(userId -> users.get(userId) != null, "User not found")
                .getOrThrow(message -> new NotFoundResponse());

        User user = users.get(id);

        ctx.json(user);
    }

    public void getMany(Context ctx) {
        String firstName = ctx.queryParam("firstName");
        String lastName = ctx.queryParam("lastName");

        List<User> users = new ArrayList<>();

        for (User user : this.users.values()) {
            if (firstName != null && !user.firstName.equals(firstName)) {
                continue;
            }

            if (lastName != null && !user.lastName.equals(lastName)) {
                continue;
            }

            users.add(user);
        }
```

```
        ctx.json(users);
    }

    public void update(Context ctx) {
        Integer id = ctx.pathParamAsClass("id", Integer.class)
                .check(userId -> users.get(userId) != null, "User not found")
                .getOrThrow(message -> new NotFoundResponse());

        User updateUser = ctx.bodyValidator(User.class)
                .check(obj -> obj.firstName != null, "Missing first name")
                .check(obj -> obj.lastName != null, "Missing last name")
                .check(obj -> obj.email != null, "Missing email")
                .check(obj -> obj.password != null, "Missing password")
                .get();

        User user = users.get(id);

        user.firstName = updateUser.firstName;
        user.lastName = updateUser.lastName;
        user.email = updateUser.email;
        user.password = updateUser.password;

        users.put(id, user);

        ctx.json(user);
    }

    public void delete(Context ctx) {
        Integer id = ctx.pathParamAsClass("id", Integer.class)
                .check(userId -> users.get(userId) != null, "User not found")
                .getOrThrow(message -> new NotFoundResponse());

        users.remove(id);

        ctx.status(HttpStatus.NO_CONTENT);
    }
}
```

This class represents the controller of the users resource. It has a `ConcurrentHashMap` to store the users.

It has the following methods:

- create - Create a new user
- getOne - Get one user by its ID
- getMany - Get many users that you can filter by first name and/or last name
- update - Update a user
- delete - Delete a user

Using a bit more advanced features of Javalin, we can validate the request body and the path parameters to ensure that the request is valid with the help of the bodyValidator and pathParamAsClass methods.

These methods will check if the request body and the path parameters are valid and automatically deserialize the request body and the path parameters to the specified class.

If not, we can throw an exception that will be handled by Javalin and will respond with the appropriate status code.

The createUser method will respond with a 201 (Created) status code if successful. However, the method will respond with a 409 (Conflict) status code if the user already exists by their email address.

Now let's implement the auth domain. Start by creating a new directory src/main/java/ch/heigvd/auth and a new file AuthController.java:

```
package ch.heigvd.auth;

import ch.heigvd.users.User;
import io.javalin.http.*;

import java.util.concurrent.ConcurrentHashMap;

public class AuthController {

  private final ConcurrentHashMap<Integer, User> users;

  public AuthController(ConcurrentHashMap<Integer, User> users) {
    this.users = users;
  }
```

```
    public void login(Context ctx) {
        User loginUser = ctx.bodyValidator(User.class)
                .check(obj -> obj.email != null, "Missing email")
                .check(obj -> obj.password != null, "Missing password")
                .get();

        for (User user : users.values()) {
            if (user.email.equals(loginUser.email) &&
                user.password.equals(loginUser.password)) {
                ctx.cookie("user", user.id.toString());
                ctx.status(HttpStatus.NO_CONTENT);
                return;
            }
        }

        throw new UnauthorizedResponse();
    }

    public void logout(Context ctx) {
        ctx.removeCookie("user");
        ctx.status(HttpStatus.NO_CONTENT);
    }

    public void profile(Context ctx) {
        String userId = ctx.cookie("user");

        if (userId == null) {
            throw new UnauthorizedResponse();
        }

        User user = users.get(Integer.parseInt(userId));

        if (user == null) {
            throw new UnauthorizedResponse();
        }

        ctx.json(user);
    }
}
```

This class represents the controller of the auth domain. It has a `ConcurrentHashMap` to store the users.

It has the following methods:

- `login` - Login a user
- `logout` - Logout a user
- `profile` - Get the current user (the user that is logged in)

The `profile` method will respond with a `401` (Unauthorized) status code if the user is not logged in. If logged in, the method will respond with a `200` (OK) status code and the user.

The final step is to update the `Main.java` file to define the endpoints/routes that will be used by the API using the same controllers we have just created:

```
package ch.heigvd;

import ch.heigvd.auth.AuthController;
import ch.heigvd.users.User;
import ch.heigvd.users.UsersController;
import io.javalin.Javalin;

import java.util.concurrent.ConcurrentHashMap;

public class Main {

  public final static int PORT = 8080;

  public static void main(String[] args) {
    Javalin app = Javalin.create();

    // This will serve as our database
    ConcurrentHashMap<Integer, User> users = new ConcurrentHashMap<>();

    // Controllers
    AuthController authController = new AuthController(users);
    UsersController usersController = new UsersController(users);

    // Auth routes
    app.post("/login", authController::login);
    app.post("/logout", authController::logout);
```

```
    app.get("/profile", authController::profile);

    // Users routes
    app.post("/users", usersController::create);
    app.get("/users", usersController::getMany);
    app.get("/users/{id}", usersController::getOne);
    app.put("/users/{id}", usersController::update);
    app.delete("/users/{id}", usersController::delete);

    app.start(PORT);
  }
}
```

Run the application and open your browser at http://localhost:8080/users. It should return an empty array.

Let's try to create a new user using curl:

```
curl -i -X POST -H "Content-Type: application/json" -d
        '{"firstName":"John","lastName":"Doe","email":"john.doe@example.com","password":"secret"}'
        http://host.docker.internal:8080/users
```

The output should be similar to the following:

```
HTTP/1.1 201 Created
Date: Fri, 08 Dec 2023 10:48:15 GMT
Content-Type: application/json
Content-Length: 96

{"id":
0,"firstName":"John","lastName":"Doe","email":"john.doe@example.com","password":"secret"}
```

The user has been successfully created with ID `0` and a status code `201`. Let's try to create the same user again:

```
curl -i -X POST -H "Content-Type: application/json" -d
        '{"firstName":"John","lastName":"Doe","email":"john.doe@example.com","password":"secret"}'
        http://host.docker.internal:8080/users
```

The output should be similar to the following:

```
HTTP/1.1 409 Conflict
Date: Fri, 08 Dec 2023 10:46:58 GMT
Content-Type: text/plain
Content-Length: 8
```

Conflict

As the user already exists, the server responds with a 409 (Conflict) status code.

Let's create a user with a missing field:

```
curl -i -X POST -H "Content-Type: application/json" -d
        '{"firstName":"Johanna","lastName":"Dane","email":"johanna.dane@example.com"}'
        http://host.docker.internal:8080/users
```

The output should be similar to the following:

```
HTTP/1.1 400 Bad Request
Date: Mon, 11 Dec 2023 17:04:05 GMT
Content-Type: application/json
Content-Length: 170

{"REQUEST_BODY":[{"message":"Missing password","args":{},"value":
{"id":null,"firstName":"Johanna","lastName":"Dane","email":"johanna.dane@example.com","pa
```

As the password is missing, the server responds with a 400 (Bad Request) status code. The response body contains the error message and the request body that was sent by the client.

Refresh the page at http://localhost:8080/users. You should see the user you have just created in an array. You can do the same with curl:

```
curl -i http://host.docker.internal:8080/users
```

The output should be similar to the following:

```
HTTP/1.1 200 OK
Date: Fri, 08 Dec 2023 10:48:58 GMT
Content-Type: application/json
Content-Length: 194

[{"id":
0,"firstName":"John","lastName":"Doe","email":"john.doe@example.com","password":"secret"},
{"id":
1,"firstName":"John","lastName":"Doe","email":"john.doe@example.com","password":"secret"}]
```

Let's get the user we have just created (the same can be done in the browser):

```
curl -i http://host.docker.internal:8080/users/0
```

Let's try to update the user using curl:

```
curl -i -X PUT -H "Content-Type: application/json" -d
        '{"firstName":"Jane","lastName":"Doe","email":"jane.doe@example.com","password":"secret"}'
        http://host.docker.internal:8080/users/0
```

The output should be similar to the following:

```
HTTP/1.1 200 OK
Date: Fri, 08 Dec 2023 12:19:13 GMT
Content-Type: application/json
Content-Length: 95

{"id":
0,"firstName":"Jane","lastName":"Doe","email":"jane.doe@example.com","password":"secret"}
```

The user has been successfully updated with ID `0` and a status code `200`.

Let's try to login the user using curl:

```
curl -i -X POST -H "Content-Type: application/json" -d
        '{"email":"jane.doe@example.com","password":"secret"}' http://
        host.docker.internal:8080/login
```

The output should be similar to the following:

```
HTTP/1.1 200 OK
Date: Fri, 08 Dec 2023 12:21:06 GMT
Content-Type: text/plain
Set-Cookie: user=0; Path=/
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Length: 0
```

You can notice the `Set-Cookie` header in the response headers. This means that the server has set a cookie with the name `user` and the value `0`.

Let's use the cookie to get the current user:

```
curl -i --cookie user=0 http://host.docker.internal:8080/profile
```

The --cookie option allows to send a cookie with the request. A browser does this automatically.

The output should be similar to the following:

```
HTTP/1.1 200 OK
Date: Fri, 08 Dec 2023 12:23:21 GMT
Content-Type: application/json
Content-Length: 95

{"id":
0,"firstName":"Jane","lastName":"Doe","email":"jane.doe@example.com","password":"secret"}
```

Let's try to get the current user without the cookie:

```
curl -i http://host.docker.internal:8080/profile
```

The output should be similar to the following:

```
HTTP/1.1 401 Unauthorized
Date: Fri, 08 Dec 2023 12:23:49 GMT
Content-Type: text/plain
Content-Length: 12

Unauthorized
```

As the cookie is missing, the server responds with a 401 (Unauthorized) status code.

Let's try to logout the user using curl:

```
curl -i -X POST http://host.docker.internal:8080/logout
```

The output should be similar to the following:

```
HTTP/1.1 204 No Content
Date: Fri, 08 Dec 2023 12:24:42 GMT
Content-Type: text/plain
Set-Cookie: user=; Path=/; Expires=Thu, 01-Jan-1970 00:00:00 GMT; Max-Age=0
Expires: Thu, 01 Jan 1970 00:00:00 GMT
```

You can notice the server 204 (No Content) status code and the Set-Cookie header.

The header has an empty value for the cookie `user`. This means that the server asks the client to remove the cookie with the name `user`.

Other information about the cookie is also present in the `Set-Cookie` header to ensure the cookie is invalidated.

Let's try to delete the user using curl:

```
curl -i -X DELETE http://host.docker.internal:8080/users/0
```

The output should be similar to the following:

```
HTTP/1.1 204 No Content
Date: Fri, 08 Dec 2023 12:27:43 GMT
Content-Type: text/plain
```

The user has been successfully deleted with ID `0` and a status code `204` (No Content).

## How to document an API

As seen in the Define an application protocol chapter, it is important to document a protocol to allow other developers to understand how to use it.

Documenting an API is thus very important as well: as an API exposes the features of an application to the outside world, it is important to document it properly for other developers to understand how to use it.

There are many ways to document an API. The most common way is one of the following:

- Using the OpenAPI Specification
- Describe the API using a simple text file

OpenAPI is a very useful and powerful specification to document an API. It is, however, a bit complex to use in the context of this course with the time allowed (you can check the Go further section if you want to implement if yourself).

Let's use a simple text file to document the API we have just created.

As HTTP is way more structured than other protocols, it is easy to document an API using a simple text file.

Let's create a new file API.md at the root of the project:

```
# Users API

The users API allows to manage users. It uses the HTTP protocol and the JSON
format.

The API is based on the CRUD pattern. It has the following operations:

- Create a new user
- Get many users that you can filter by first name and/or last name
- Get one user by its ID
- Update a user
- Delete a user

Users are also able to login and logout. They can also access their profile to
validate their information using a cookie.

## Endpoints

### Create a new user

- `POST /users`

Create a new user.

#### Request

The request body must contain a JSON object with the following properties:

- `firstName` - The first name of the user
- `lastName` - The last name of the user
- `email` - The email address of the user
- `password` - The password of the user

#### Response

The response body contains a JSON object with the following properties:

- `id` - The unique identifier of the user
```

- `firstName` - The first name of the user
- `lastName` - The last name of the user
- `email` - The email address of the user
- `password` - The password of the user

#### Status codes

- `201` (Created) - The user has been successfully created
- `400` (Bad Request) - The request body is invalid
- `409` (Conflict) - The user already exists

### Get many users

- `GET /users`

Get many users.

#### Request

The request can contain the following query parameters:

- `firstName` - The first name of the user
- `lastName` - The last name of the user

#### Response

The response body contains a JSON array with the following properties:

- `id` - The unique identifier of the user
- `firstName` - The first name of the user
- `lastName` - The last name of the user
- `email` - The email address of the user
- `password` - The password of the user

#### Status codes

- `200` (OK) - The users have been successfully retrieved

### Get one user

- `GET /users/{id}`

Get one user by its ID.

#### Request

The request path must contain the ID of the user.

#### Response

The response body contains a JSON object with the following properties:

- `id` - The unique identifier of the user
- `firstName` - The first name of the user
- `lastName` - The last name of the user
- `email` - The email address of the user
- `password` - The password of the user

#### Status codes

- `200` (OK) - The user has been successfully retrieved
- `404` (Not Found) - The user does not exist

### Update a user

- `PUT /users/{id}`

Update a user by its ID.

#### Request

The request path must contain the ID of the user.

The request body must contain a JSON object with the following properties:

- `firstName` - The first name of the user
- `lastName` - The last name of the user
- `email` - The email address of the user
- `password` - The password of the user

#### Response

The response body contains a JSON object with the following properties:

- `id` - The unique identifier of the user
- `firstName` - The first name of the user
- `lastName` - The last name of the user
- `email` - The email address of the user
- `password` - The password of the user

#### Status codes

- `200` (OK) - The user has been successfully updated
- `400` (Bad Request) - The request body is invalid
- `404` (Not Found) - The user does not exist

### Delete a user

- `DELETE /users/{id}`

Delete a user by its ID.

#### Request

The request path must contain the ID of the user.

#### Response

The response body is empty.

#### Status codes

- `204` (No Content) - The user has been successfully deleted
- `404` (Not Found) - The user does not exist

### Login

- `POST /login`

Login a user.

#### Request

The request body must contain a JSON object with the following properties:

- `email` - The email address of the user
- `password` - The password of the user

#### Response

The response body is empty. A `user` cookie is set with the ID of the user.

#### Status codes

- `204` (No Content) - The user has been successfully logged in
- `400` (Bad Request) - The request body is invalid
- `401` (Unauthorized) - The user does not exist or the password is incorrect

### Logout

- `POST /logout`

Logout a user.

#### Request

The request body is empty.

#### Response

The response body is empty. The `user` cookie is removed.

#### Status codes

- `204` (No Content) - The user has been successfully logged out

### Profile

- `GET /profile`

Get the current user (the user that is logged in).

#### Request

The request body is empty.

#### Response

The response body contains a JSON object with the following properties:

- `id` - The unique identifier of the user
- `firstName` - The first name of the user
- `lastName` - The last name of the user
- `email` - The email address of the user
- `password` - The password of the user

#### Status codes

- `200` (OK) - The user has been successfully retrieved
- `401` (Unauthorized) - The user is not logged in

## How to persist data

In the previous example, we have used a ConcurrentHashMap to store the users.

In a production application, you would certainly use a database to store the users (PostgreSQL, MySQL, MongoDB, etc.).

While it would be possible to use a database with Javalin, it is out of the scope of this course. You can check the Go further section if you want to implement if yourself.

As already mentioned, Javalin is perfect to create prototypes and proof of concepts but we would recommend you to use a web framework such as Quarkus or Spring Boot to create a production application.

## How to secure an API

You might have noticed that the API we have just created is not very secure: we do not want to allow anyone to create, read, update or delete users.

Some APIs are public and do not require any authentication. However, most APIs are private and require authentication.

Securing an API is not an easy task and is out of the scope of this course (you can check the Go further section if you want to implement if yourself). You will learn how to secure a web application in future courses.

In the context of this course, it is not important that the API is secure. It is more important to understand the basics of how to design, how to develop and how to document an API.

# Share your project

Create a new Git repository and push your code to it. Do not forget all the files you have created or modified during this chapter and the best practices you have learned.

Share your project in the GitHub Discussions of this organization: https://github.com/orgs/heig-vd-dai-course/discussions.

Create a new discussion with the following information:

- **Title**: DAI 2023-2024 - Users API - First name Last Name
- **Category**: Show and tell
- **Description**: The link to your GitHub repository.

This will notify us that you have completed the exercise and we can check your work.

You can compare your solution with the official one stated in the Solution section, however, **we highly recommend you to try to complete the practical content by yourself first to learn the most**.

# Go further

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

- Are you able to document the API you have just created using the OpenAPI Specification? You can use the official documentation to help you: https://javalin.io/tutorials/openapi-example.
- Are you able to secure the API you have just created? You can use the official documentation to help you: https://javalin.io/tutorials/auth-example.
- Are you able to persist the data in a database? You can use the official documentation to help you: https://javalin.io/tutorials/jetty-session-handling.

# Conclusion

## What did you do and learn?

In this chapter, you have learned about the extended features of HTTP.

You have learned how endpoints/routes are defined with their methods and status codes.

You have learned how to send and get data to/from the server and about content negotiation.

Using cookies, you have learned how to keep track of the state of the client.

And finally, you have learned how to design and develop APIs with the HTTP protocol.

## Test your knowledge

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

- What are HTTP methods?
- What are HTTP status codes? How are they classified?
- What is content negotiation?
- What are cookies? How are they used?
- What is an API? How to design and develop an API?

# 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?

In the next chapter, you will learn the following topics:

- Web infrastructures
    - How to run and maintain web applications on the Internet?
    - How to scale web applications?

# Additional resources

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

- Evolution of HTTP

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

# Solution

You can find the solution to the practical content in the heig-vd-dai-course/heig-vd-dai-course-solutions repository.

If you have any questions about the solution, feel free to open an issue to discuss it!

# Sources

- Main illustration by Ashley Knedler on Unsplash