

HTTP and curl

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

[Web](#) • [PDF](#)

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

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

This work is licensed under the [CC BY-SA 4.0](#) license.

Objectives

- Understand the basics of HTTP
- Understand the basics of APIs
- Learn how to use curl
- Learn how to design and document a simple API
- Learn how to develop a simple API
- Learn how to use a simple API



Disclaimer

More details for this section in the [course material](#). You can find other resources and alternatives as well.

Disclaimer

- **This is not a course on web development**
- Many many things are not covered
- Focus on HTTP version 1.1
- Javalin used for learning purposes
- For production, use a framework like Quarkus or Spring Boot



Prepare and setup your environment

More details for this section in the [course material](#). You can find other resources and alternatives as well.

curl

- An open source command-line tool
- Used to transfer data using various protocols
 - HTTP/HTTPS
 - FTP
 - etc.
- Used to test APIs



Javalin

- A lightweight web framework for Java and Kotlin
- Easy to learn and use: perfect for learning purposes
- Production ready but not as powerful as Quarkus or Spring Boot

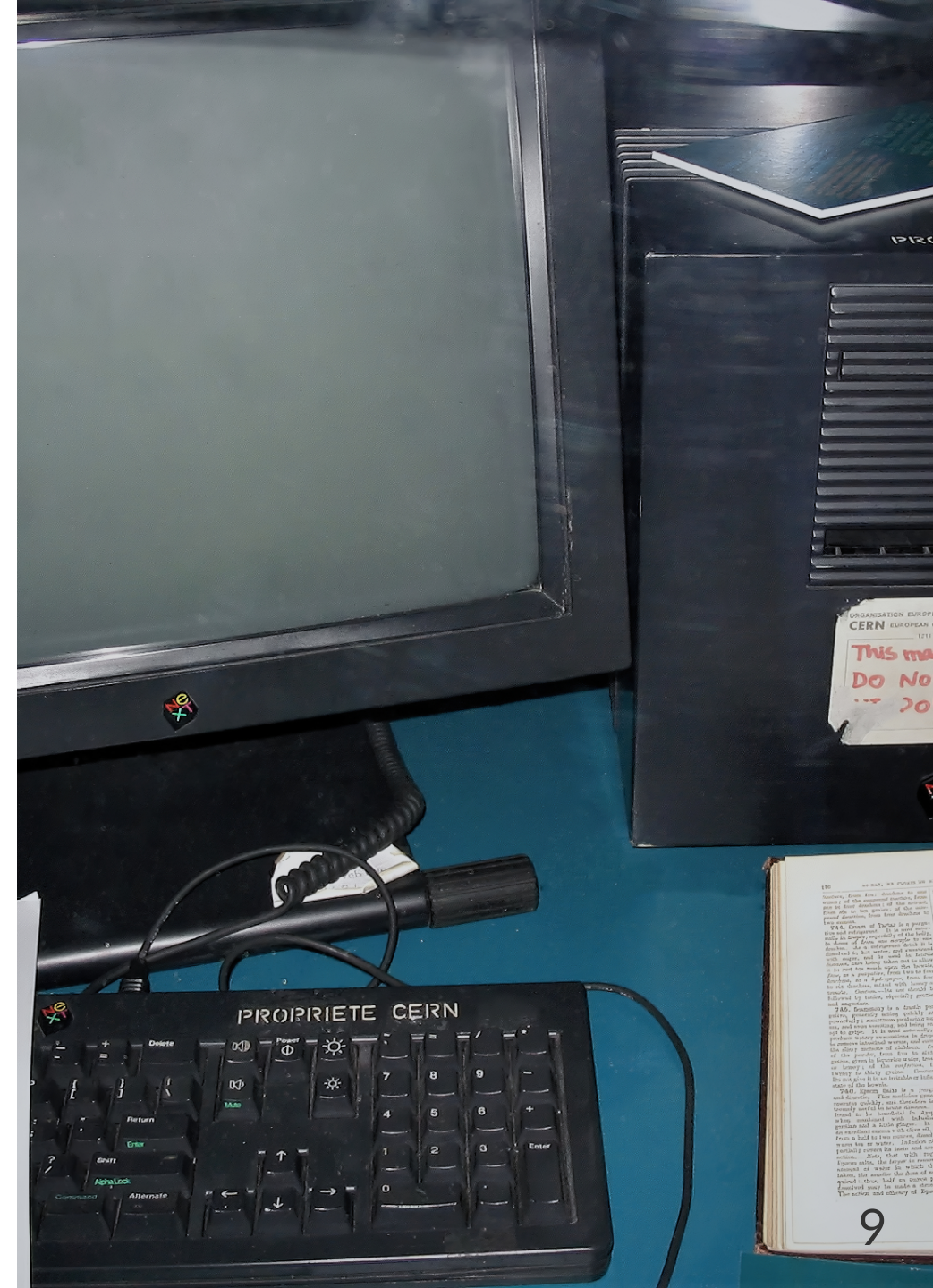


HTTP

More details for this section in the [course material](#). You can find other resources and alternatives as well.

HTTP

- Initiated by Tim Berners-Lee at CERN in 1989
- First release in 1990 to transfer HyperText Markup Language (HTML) documents
- Built on top of TCP (HTTP/1.0, HTTP/1.1 and HTTP/2.0) or UDP/QUIC (HTTP/3)
- Ports **80** (HTTP) or **443** (HTTPS)



- Hyper Text Transfer Protocol (HTTP) based on TCP
- Application layer protocol with many features
- Used to transfer data between a client (an **user agent**) and a server
- A client can be a web browser, a mobile application, a command-line tool, household appliance, etc.
- The client requests a **resource** from the server

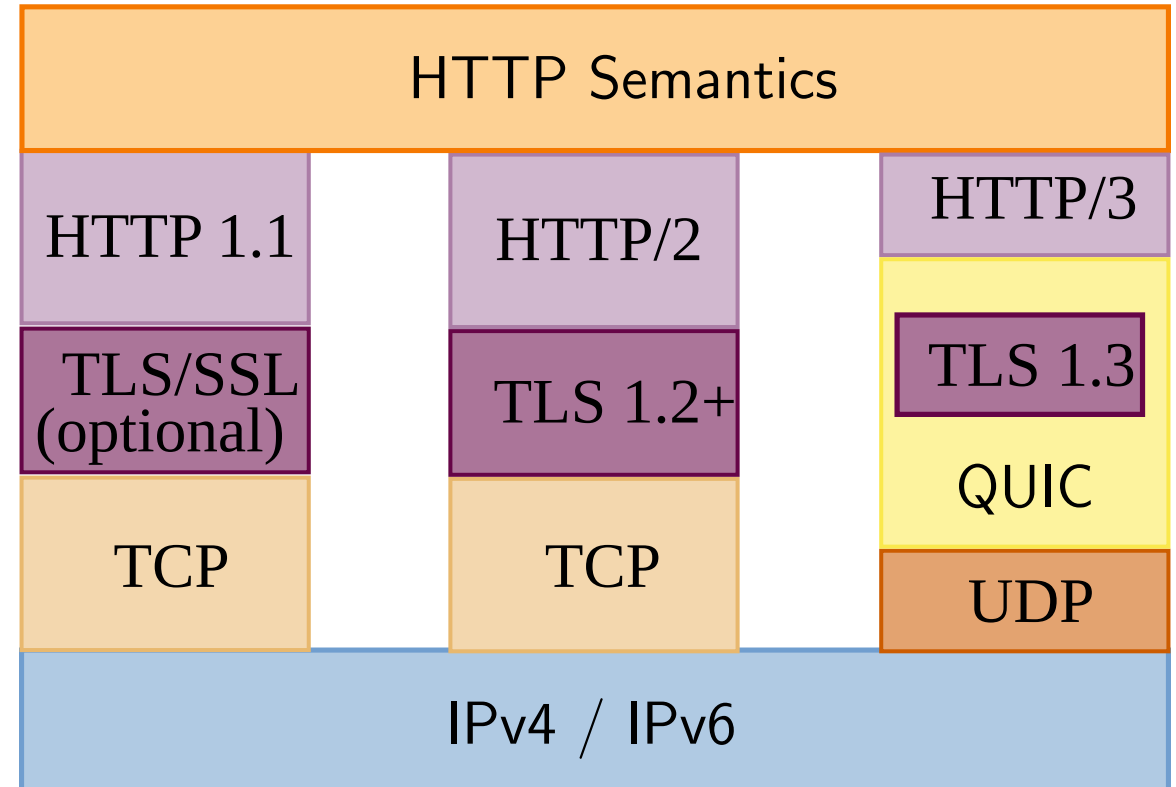


HTTP versions

Multiple versions exist:

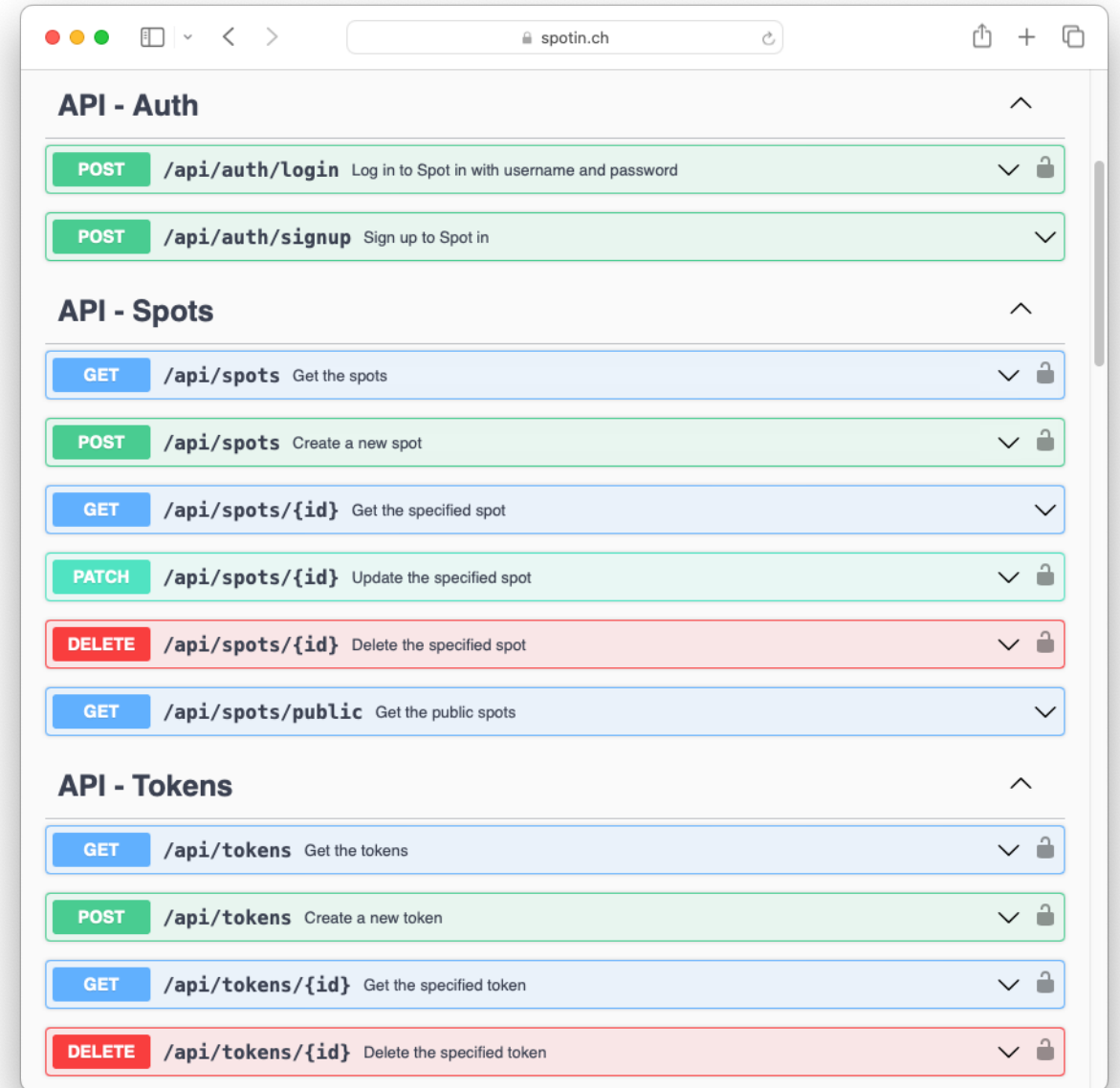
- HTTP/1.0 (1996)
- HTTP/1.1 (1997)
- HTTP/2 (2015)
- HTTP/3 (2022)

The most used version is HTTP/1.1. Each version is to improve performance.



HTTP resources

- A resource is identified by a Uniform Resource Locator (URL)
- A resource can be a file, a document, a video, etc.
- Sometimes called an **endpoint** or a **route**



An example of a URL is the following:

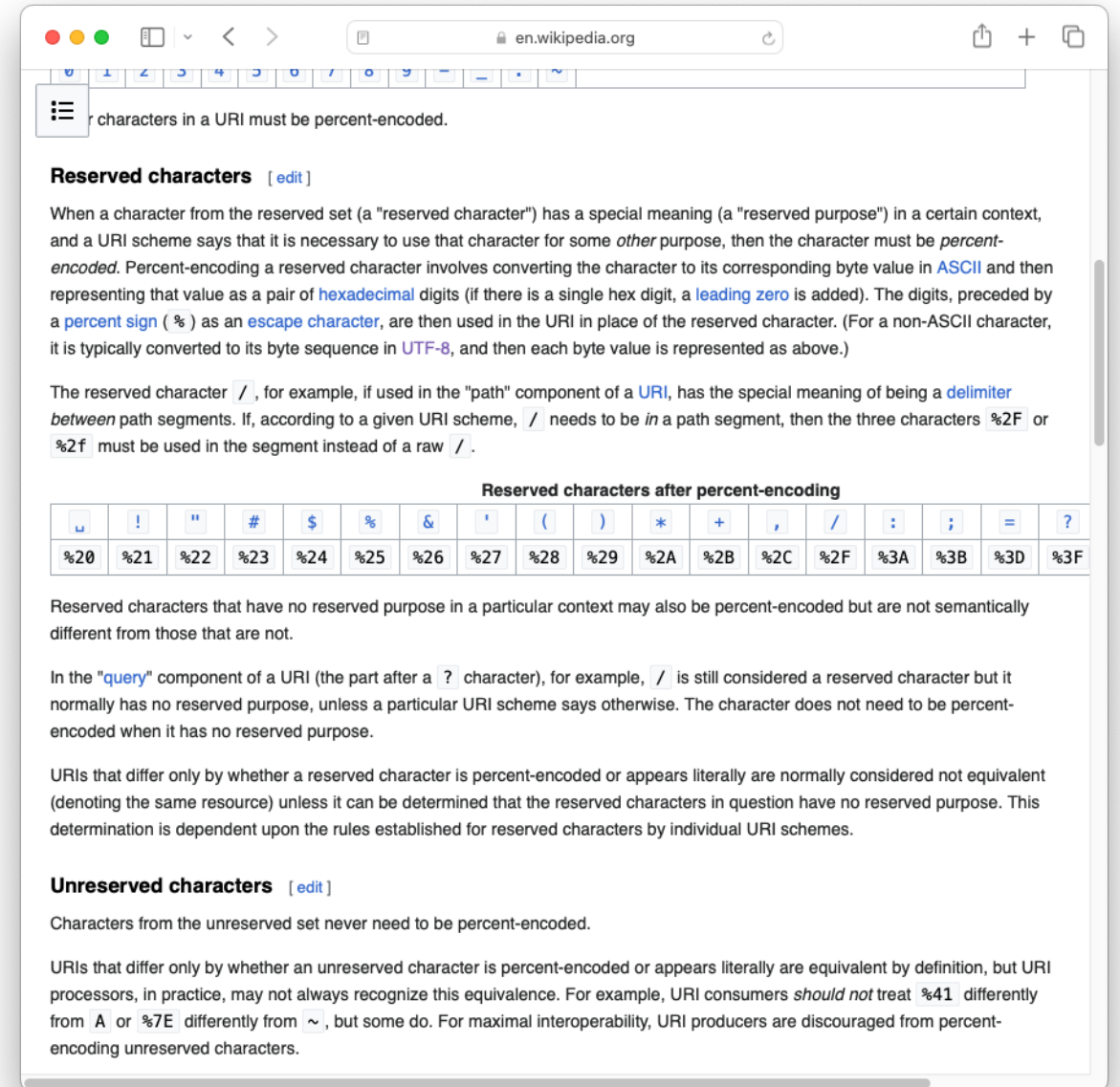
```
https://gaps.heig-vd.ch/consultation/fiches/uv/uv.php?id=6573
```

- Protocol: `http://` or `https://`
- Host: `gaps.heig-vd.ch`
- Port: `:80` or `:443` (optional)
- Path: `/consultation/fiches/uv/uv.php`
- Query parameters: `?id=6573`

This resource returns a HTML document.

URL encoding

- URLs can only contain a limited set of characters
- Some characters are reserved for special purposes
- Some characters must be encoded: (Space -> %20)
- For example, Hello world becomes Hello%20world



characters in a URI must be percent-encoded.

Reserved characters [\[edit \]](#)

When a character from the reserved set (a "reserved character") has a special meaning (a "reserved purpose") in a certain context, and a URI scheme says that it is necessary to use that character for some *other* purpose, then the character must be *percent-encoded*. Percent-encoding a reserved character involves converting the character to its corresponding byte value in [ASCII](#) and then representing that value as a pair of [hexadecimal](#) digits (if there is a single hex digit, a [leading zero](#) is added). The digits, preceded by a [percent sign](#) (%) as an [escape character](#), are then used in the URI in place of the reserved character. (For a non-ASCII character, it is typically converted to its byte sequence in [UTF-8](#), and then each byte value is represented as above.)

The reserved character / , for example, if used in the "path" component of a [URI](#), has the special meaning of being a [delimiter between](#) path segments. If, according to a given URI scheme, / needs to be *in* a path segment, then the three characters %2F or %2f must be used in the segment instead of a raw / .

!	"	#	\$	%	&	'	(	)	*	+	,	/	:	;	=	?	
%20	%21	%22	%23	%24	%25	%26	%27	%28	%29	%2A	%2B	%2C	%2F	%3A	%3B	%3D	%3F

Reserved characters that have no reserved purpose in a particular context may also be percent-encoded but are not semantically different from those that are not.

In the "query" component of a URI (the part after a ? character), for example, / is still considered a reserved character but it normally has no reserved purpose, unless a particular URI scheme says otherwise. The character does not need to be percent-encoded when it has no reserved purpose.

URIs that differ only by whether a reserved character is percent-encoded or appears literally are normally considered not equivalent (denoting the same resource) unless it can be determined that the reserved characters in question have no reserved purpose. This determination is dependent upon the rules established for reserved characters by individual URI schemes.

Unreserved characters [\[edit \]](#)

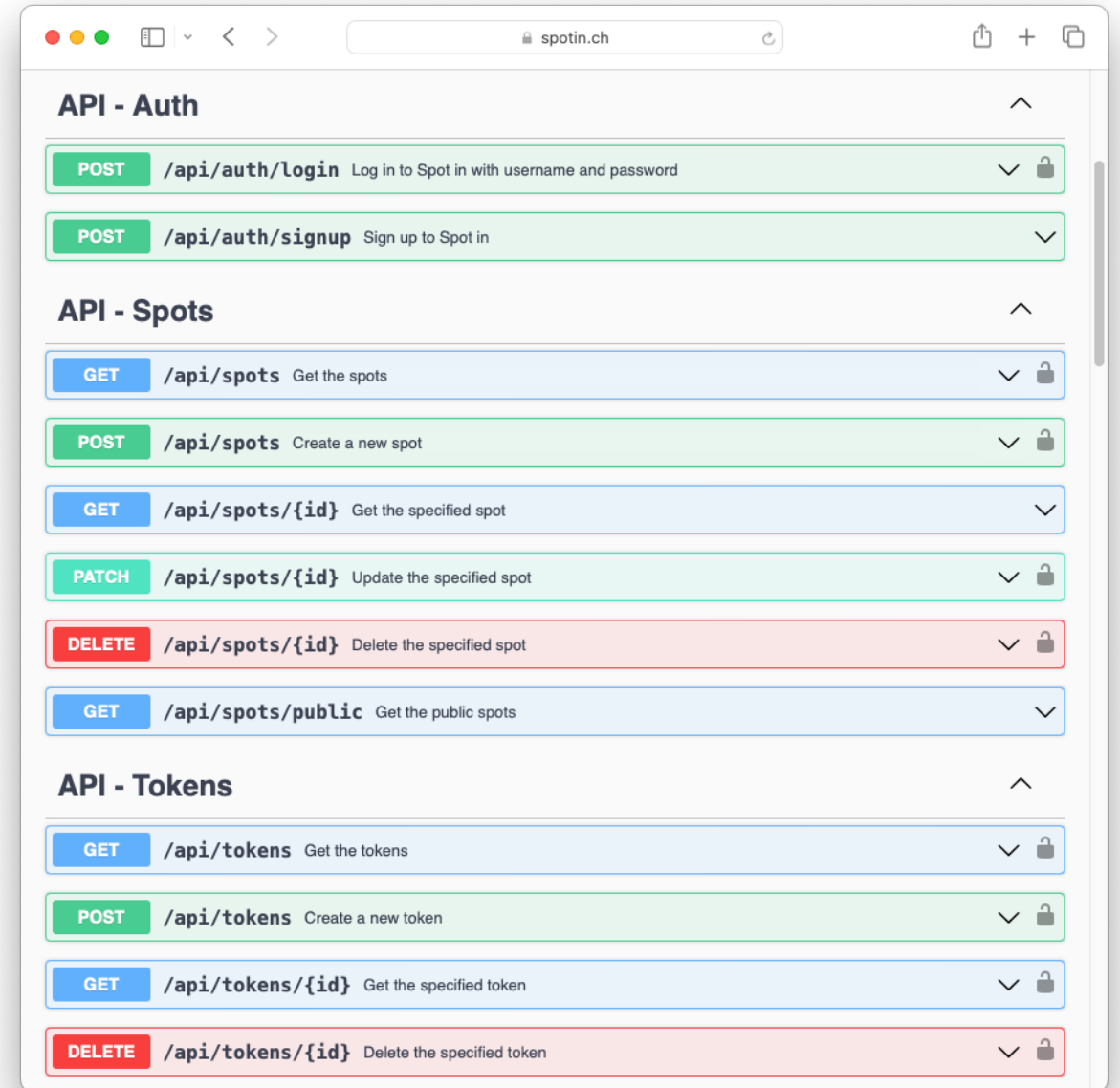
Characters from the unreserved set never need to be percent-encoded.

URIs that differ only by whether an unreserved character is percent-encoded or appears literally are equivalent by definition, but URI processors, in practice, may not always recognize this equivalence. For example, URI consumers *should not* treat %41 differently from A or %7E differently from ~ , but some do. For maximal interoperability, URI producers are discouraged from percent-encoding unreserved characters.

HTTP request methods

- GET - Get a resource
- POST - Create a resource
- PATCH / PUT - Update a resource
- DELETE - Delete a resource

A browser does GET methods by default.



HTTP request and response format

- To request a resource, a client sends a HTTP request to a server
- The server processes the request and sends back a HTTP response

HTTP is based on a **request-response** model.



The HTTP request and response are composed of:

- A **start line** with:
 - The **method**
 - The **URL**
 - The **version**
- **Headers** with metadata
- An **empty line**
- An optional **body** with data

Structure of a HTTP request:

```
<HTTP method> <URL> HTTP/<HTTP version>  
<HTTP headers>  
<Empty line if there is a body>  
<HTTP body (optional)>
```

Structure of a HTTP response:

```
HTTP/<HTTP version> <HTTP status code> <HTTP status message>  
<HTTP headers>  
<Empty line if there is a body>  
<HTTP body>
```

A HTTP request example:

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

A HTTP response example:

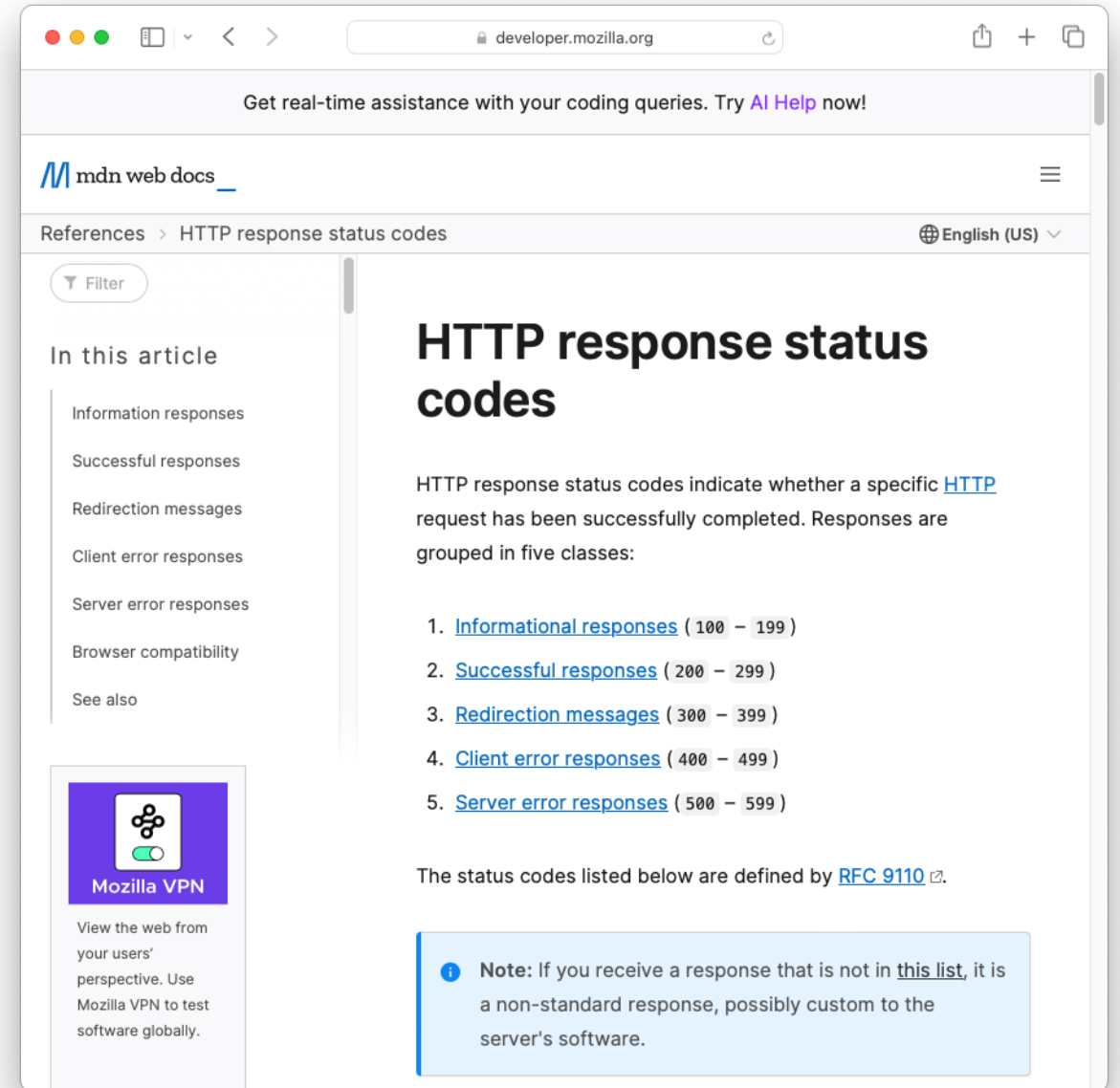
```
HTTP/1.1 200 OK
Content-Type: text/html; charset=ISO-8859-1
Content-Length: 6111

<!DOCTYPE HTML>
...
```

HTTP response status codes

Grouped by categories:

- 1xx: Informational
- 2xx: Success
- 3xx: Redirection
- 4xx: Client error
- 5xx: Server error



HTTP path parameters, query parameters and body

These elements are used to pass data to the server.

Path parameters and query parameters are part of the URL.

Headers are part of the HTTP request or response.

HTTP path parameters

An example of a path parameter is the following:

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

The `{user-id}` part is a path parameter.

With values: `/users/123` -> `123` is the user ID.

HTTP query parameters

An example of a query parameter is the following:

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

The `?` character separates the path from the query parameters.

The `&` character separates query parameters.

Each query parameter is composed of a key and a value.

HTTP body

An example of a HTTP body is the following:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=ISO-8859-1
Content-Length: 6111

<!DOCTYPE HTML [...]>
<html>

[...]

</html>
```

HTTP headers

HTTP headers are used to pass metadata to/from the server.

- **Accept** - The media types accepted by the client
- **Content-Type** - The media type of the body
- **Content-Length** - The length of the body
- **User-Agent** - The user agent of the client
- **Host** - The host of the server
- **Set-Cookie** - The cookies set by the server

HTTP content negotiation

The `Accept` header is used to negotiate the content type between the client and the server. These are based on the MIME types:

- `Accept: text/html` - HTML
- `Accept: application/json` - JSON
- etc.

The same URL can return different content types based on the `Accept` header.

HTTP sessions

As HTTP is based on the request-response model, each request is independent of the others. This is called a stateless protocol.

This means that the server cannot know/identify who is the author of each request without additional information. Let's take an example:

1. A first user access the homepage of a website
2. A second user access the homepage of the same website

Who is the author of each request?

Why does the server not know who you are?

It is because you have not stated who you are. In other words, you do not have a session with the server.

They are two ways to maintain a session:

- Using a query parameter
- Using cookies



HTTP sessions using a query parameter

A query parameter can be used to maintain a session:

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

Advantages: Easy to implement.

Disadvantages: The token is visible in the URL (security issue).

HTTP sessions using a cookie

A cookie can be used to maintain a session:

```
C: POST /login  
S: 200 OK (set a cookie with the token)  
C: GET /profile (the cookie is sent by the client)  
S: 200 OK (profile page)
```

Advantages: The token is not visible in the URL (more secure).

Disadvantages: A bit more complex to implement.

Do all websites use sessions?

Not all applications need a session.

For example, a calculator application that waits for a calculation and directly sends the result does not need to keep track of the client:

- Each request is independent of the others
- The server can directly respond to each request

The server does not have to know who is the author of the request: it can send the result directly to the client.

HEIG-VD FDS AI Course 2025-2026 - CC BY-SA 4.0

In this case, the server does not need to use HTTP sessions and is, 31

On the other hand, an e-commerce application where the user can add items to a shopping cart needs to keep track of the client:

- The user can add items to the shopping cart
- The user can remove items from the shopping cart
- The user can buy the items in the shopping cart

The server must know who is the author of each request in order to maintain the shopping cart.

In this case, the server must use HTTP sessions and is, therefore, stateful.

API design

More details for this section in the [course material](#). You can find other resources and alternatives as well.

API design

Developing a web application is not easy.

In order to make it easier, we follow patterns and a set of rules such as an **Application Programming Interface** (API).

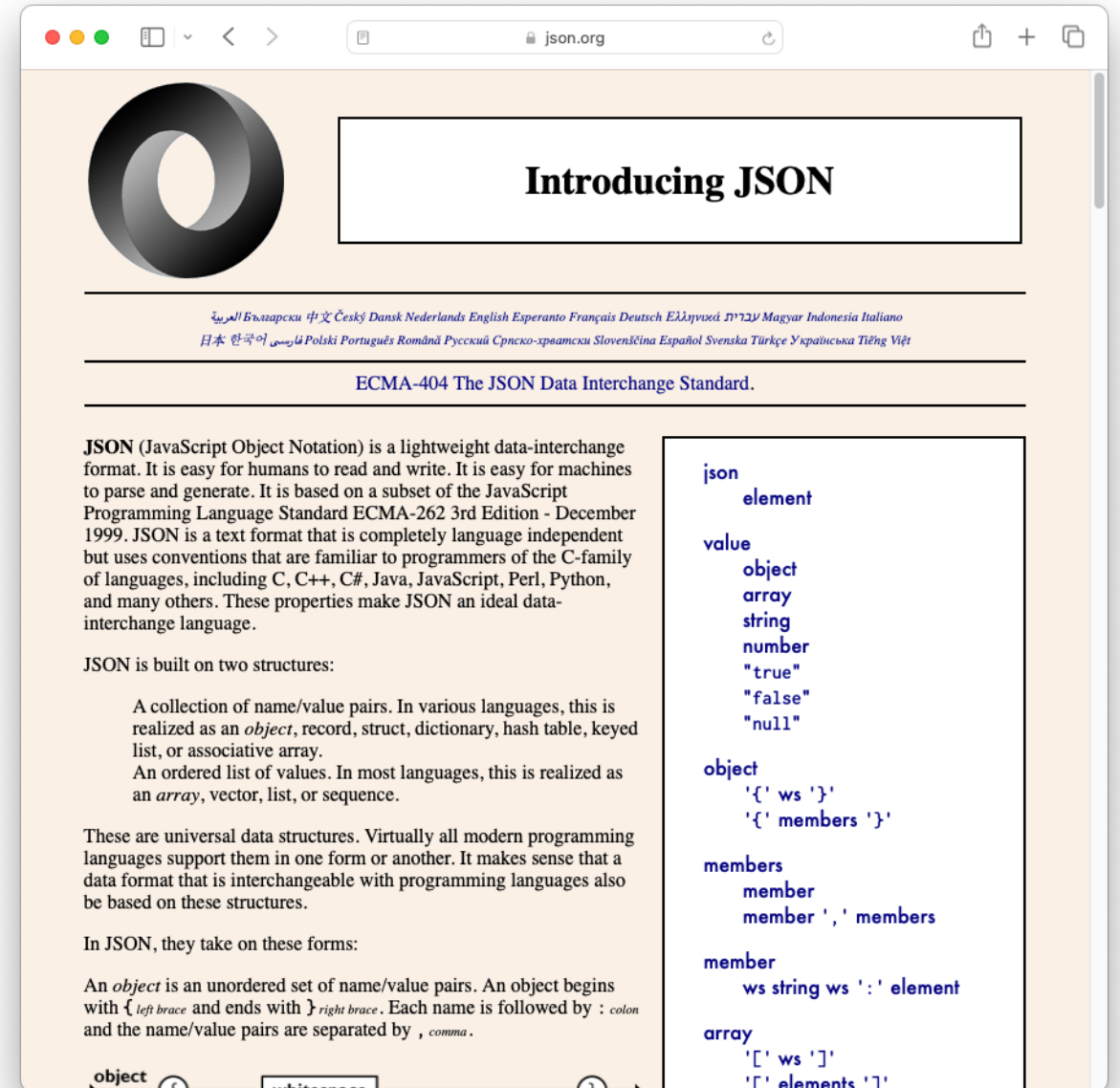
An API is a contract between the client and the server that must be documented.

Most APIs are based on HTTP and exchange data in JSON format, the most used format for APIs.

JSON is an easy to read and write format for humans. It is also easy to parse for computers.

Example of a JSON document:

```
{  
  "firstName": "John",  
  "lastName": "Doe",  
  "age": 42  
}
```



Simple APIs with CRUD operations

CRUD stands for:

- **C**reate
- **R**ead
- **U**ppdate
- **D**eleate

CRUD APIs are used to manage data.

A simple API with CRUD operations to manage users will expose the following endpoints:

- `POST /users` - Create a new user
- `GET /users` - List all users
- `GET /users/{id}` - Read a user
- `PUT /users/{id}` - Update a user
- `PATCH /users/{id}` - Partially update a user
- `DELETE /users/{id}` - Delete a user

REST APIs

A REST API is a set of (strict) rules to design APIs.

The REST pattern is based around a six following principles.

Is an improvement over CRUD APIs.

Not all APIs are REST APIs but all REST APIs are APIs.

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

How to document an API

- Documentation is important for developers as well as users
- An API exposes the features of an application to the outside world
- There exist many tools to document APIs such as [OpenAPI](#)
- As these tools are complex, we will use a simple solution: a text file [#Define an application protocol](#)



How to persist data

- In the course material, we store data in memory
- Data can be (and should be!) stored in a database
- Out of scope for this course but you can find resources in the course material



How to secure an API

- Not all APIs are public
- Some APIs are private and require authentication
- Out of scope for this course but you can find resources in the course material.

It is more important to understand the basics of how to design, how to develop and how to document an API.



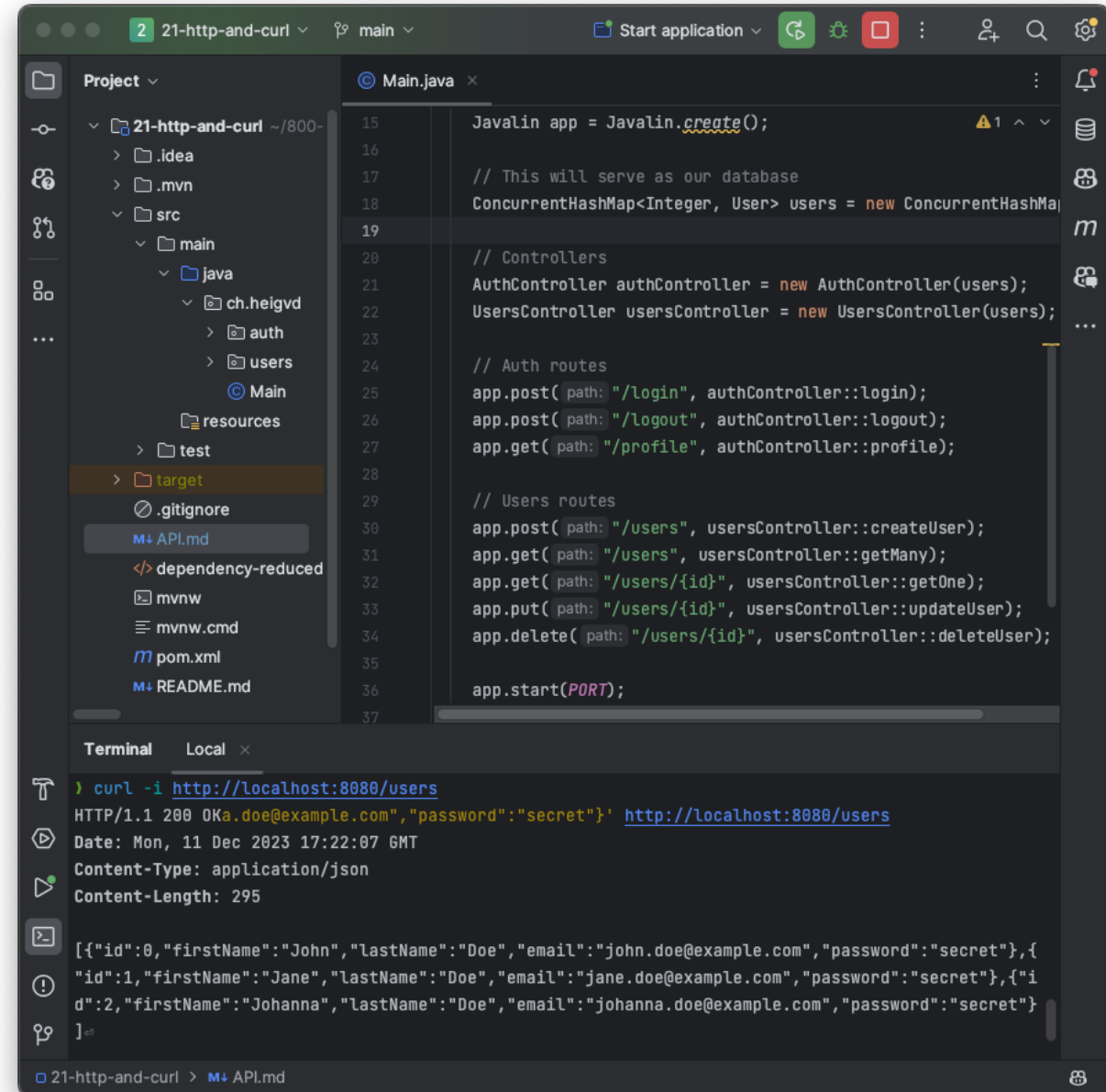
Questions

Do you have any questions?

Practical content

What will you do?

- Try out all the main concepts of HTTP (methods, status codes, headers, JSON, etc.)
- Learn how to use curl
- Build a simple web application to manage users
- Learn how to design and document a simple API



The screenshot shows an IDE window with a project named "21-http-and-curl". The project structure includes a "src" directory with "main" and "test" subdirectories, and a "target" directory. The "Main.java" file is open, showing a Javalin web application. The code defines a database of users, controllers for authentication and user management, and routes for login, logout, profile, and user management. The application is started on a specific port.

```
15 Javalin app = Javalin.create();
16
17 // This will serve as our database
18 ConcurrentHashMap<Integer, User> users = new ConcurrentHashMap<>();
19
20 // Controllers
21 AuthController authController = new AuthController(users);
22 UsersController usersController = new UsersController(users);
23
24 // Auth routes
25 app.post(path: "/login", authController::login);
26 app.post(path: "/logout", authController::logout);
27 app.get(path: "/profile", authController::profile);
28
29 // Users routes
30 app.post(path: "/users", usersController::createUser);
31 app.get(path: "/users", usersController::getMany);
32 app.get(path: "/users/{id}", usersController::getOne);
33 app.put(path: "/users/{id}", usersController::updateUser);
34 app.delete(path: "/users/{id}", usersController::deleteUser);
35
36 app.start(PORT);
37
```

The terminal window shows the output of a curl command, displaying a JSON response with three user objects.

```
> curl -i http://localhost:8080/users
HTTP/1.1 200 OK
Date: Mon, 11 Dec 2023 17:22:07 GMT
Content-Type: application/json
Content-Length: 295

[{"id":0,"firstName":"John","lastName":"Doe","email":"john.doe@example.com","password":"secret"}, {"id":1,"firstName":"Jane","lastName":"Doe","email":"jane.doe@example.com","password":"secret"}, {"id":2,"firstName":"Johanna","lastName":"Doe","email":"johanna.doe@example.com","password":"secret"}]
```

Find the practical content

You can find the practical content for this chapter on [GitHub](#).



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?
 - How to secure web applications?

Sources

- Main illustration by [Ashley Knedler](#) on [Unsplash](#)
- Illustration by [Aline de Nadai](#) on [Unsplash](#)
- Illustration by [Bernard Hermant](#) on [Unsplash](#)
- Illustration by [Walling](#) on [Unsplash](#)
- Illustration by [Pavan Trikutam](#) on [Unsplash](#)
- Illustration by [Chien Nguyen Minh](#) on [Unsplash](#)
- Illustration by [Iñaki del Olmo](#) on [Unsplash](#)

- Illustration by [Jan Antonin Kolar](#) on [Unsplash](#)
- Illustration by [Amol Tyagi](#) on [Unsplash](#)
- Illustration by [Nicolas Picard](#) on [Unsplash](#)