API Workshop

Hien Ho, Allen Mahdi, Peter Kopylov

23rd Sep. 2024

Contents

1 HTTP

HTTP stands for Hypertext Transfer Protocol. It powers the communication between your web browser and the internet.

1.1 What is HTTP?

First, let's understand what a protocol is. A protocol is a set of rules that dictates how data travels from one system to another. HTTP is one of these protocols, specifically designed for fetching resources like web pages. It's a request-response protocol in the client-server computing model.

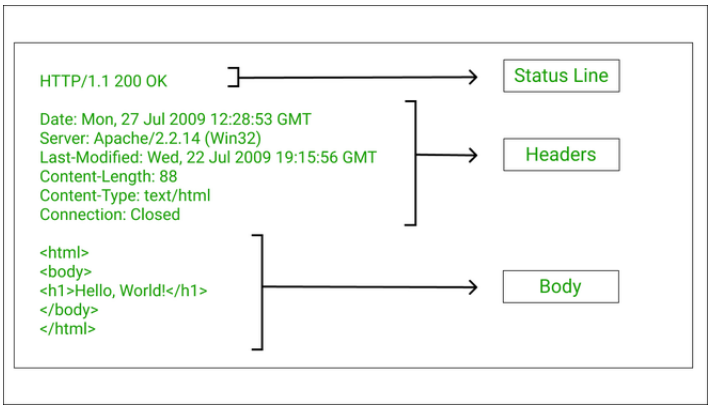
1.2 How does HTTP work?

When you type a website address into your browser, you're sending an HTTP request.

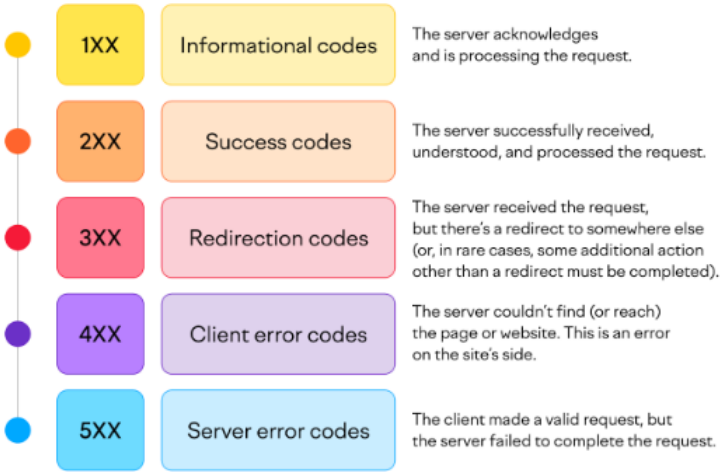
1. You enter the URL in your browser.
2. Your browser sends an HTTP request to the server where the website is hosted.
3. The request is composed of three parts:
    - The first line, which has `GET /index.html HTTP/1.1`, starts with the HTTP method (GET, POST, PUT), the target URL, and the protocol version.
    - Headers - additional information about the request, such as the user agent (helpful for the server).
    - Body - additional data (GET must NOT have a body since it's only requesting data).

The server processes this request and sends back a response, also composed of three parts:

- Status Line - sends back the version, status code, and text.
- Headers - information about the file being sent back.
- Body - the actual resource the server is sending back.



1.3 Status Codes



HTTP is stateless, meaning that once the client sends a request and the server sends a response, the connection is broken and no longer maintained.

2 APIs

An API stands for Application Programming Interface, enabling two or more computers or systems to communicate with each other.

2.1 Why are APIs helpful?

- **Enabling Lightweight Applications:** APIs help in building lightweight applications by offloading heavy processing tasks, data storage, or large computations to external servers or systems. Instead of managing huge datasets or computational resources locally (client side), developers can rely on APIs to access and process data on the server side. This results in more efficient, faster, and less resource-intensive applications.

  Example: An app like YouTube doesn't store its catalog of videos on the user's phone (client side).

- **Access to 3rd-Party Services:** APIs facilitate access to third-party services, allowing developers to integrate external functionalities into their applications easily. By relying on third-party APIs, developers can save time, and resources, as they don't need to build these services from scratch. This approach also helps in maintaining scalability and security as the third-party provider handles much of the backend complexity.

  Example: Shopify's API allows businesses to integrate their online store into various apps. PayPal's API enables secure payment processing in applications. Google Login API allows users to log in to third-party websites or apps using their Google account credentials.

2.2 API Examples with CMD

Let's say you have a T-shirt shop, and you want to get the sizes in stock:

```
GET /tshirts/123/sizes HTTP/1.1
Host: api.tshirtstore.com
Authorization: Bearer YOUR_ACCESS_TOKEN
```

In response, we get an array of those sizes.

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "id": 1,
    "size": "Small",
    "status": "Out of Stock"
  },
  {
    "id": 2,
    "size": "Medium",
    "status": "In Stock"
  },
  {
    "id": 3,
    "size": "Large",
    "status": "In Stock"
  }
]
```

If "small" is out of stock, we can use a `PUT` request to replace it with "extra small".Our request is going to use PUT operation with the endpoint being /123/flavors/1 to replace ID 1. And parameter body specifying extra small which is the new size it's replacing.

```
PUT /tshirts/123/sizes/1 HTTP/1.1
Host: api.tshirtstore.com
Content-Type: application/json
Authorization: Bearer YOUR_ACCESS_TOKEN

{
  "id": 1,
  "size": "Extra Small",
  "status": "In Stock"
}
```

2.3 Using APIs in Python

First, install the requests library using `pip install requests` in your command line.

2.3.1 Python Methods for HTTP Requests

Method	Description
delete(url, args)	Sends a DELETE request to the specified URL
get(url, params, args)	Sends a GET request to the specified URL
head(url, args)	Sends a HEAD request to the specified URL
patch(url, data, args)	Sends a PATCH request to the specified URL
post(url, data, json, args)	Sends a POST request to the specified URL
put(url, data, args)	Sends a PUT request to the specified URL
request(method, url, args)	Sends a request of the specified method to the specified URL

2.3.2 Example: Using Deck of Cards API

First, you need to import the requests library to handle HTTP requests.

```
import requests
```

Many API endpoints share a common base URL, so it's good practice to define it at the start to avoid repetition.

```
base_url = "https://deckofcardsapi.com/api/deck/"
```

To interact with the API, you need to create a new deck. The API will assign a unique `deck_id` for future interactions (like drawing or shuffling cards).

```
new_deck = f"{base_url}new/shuffle/?deck_count=1"
response = requests.get(new_deck)
deck = response.json()
deck_id = deck['deck_id']
print(deck_id)
```

The API allows you to draw one or more cards from a shuffled deck using a GET request. Here's an example of the JSON output you might receive when drawing cards (https://www.deckofcardsapi.com/api/deck/{deck_id}/draw/?count=2):


```
{
    "success": true,
    "deck_id": "kxozasf3edqu",
    "cards": [
        {
            "code": "6H",
            "image": "https://deckofcardsapi.com/static/img/6H.png",
            "value": "6",
            "suit": "HEARTS"
        },
        {
            "code": "5S",
            "image": "https://deckofcardsapi.com/static/img/5S.png",
            "value": "5",
            "suit": "SPADES"
        }
    ],
    "remaining": 50
```

You can create a function to draw cards, extracting the `success`, `value`, and `suit` for each card.

```
def draw_card(deck_id, count):
    draw_url = f"{base_url}{deck_id}/draw/?count={count}"
    response = requests.get(draw_url)
    cards = response.json().get('cards', [])
    success = response.json().get('success',[])

    # Print only the drawn cards
    for card in cards:
        print(f"{card['value']} of {card['suit']}")
    return success
```

You can also create a function to shuffle the deck. Here is an example of the output: https://www.deckofcardsapi.com/api/deck/{deck_id}/shuffle/

```
{
    "success": true,
    "deck_id": "3p40paa87x90",
    "shuffled": true,
    "remaining": 52
}
```

```
def shuffle_deck(deck_id):
    shuffle_url = f"{base_url}{deck_id}/shuffle/"
    response = requests.get(shuffle_url)
    success = response.json().get('success',[])

    print("Deck shuffled successfully.")
    return success
```

Use the functions like:

```
print(draw_card(deck_id, 3))
print(shuffle_deck(deck_id))
```

2.4 RESTful API

A crucial concept in web development and applications known as RESTful APIs. REST stands for Representational State Transfer. It's a big deal in how web applications communicate with each other.

2.4.1 What is a RESTful API?

A RESTful API is an architectural style for designing networked applications. It relies on a stateless, client-server, cacheable communications protocol and in virtually all cases, the HTTP protocol is used. Core principles include:

- **Client-server architecture**: The client (like your web browser or app) and the server (where the data is stored) operate independently.

- **Statelessness**: Each request from the client to the server must contain all the information the server needs to understand and respond to the request.
- **Cacheable**: Responses should be defined as cacheable or not, to improve client-side performance.
- **Uniform interface**: It means that every request and response follows the same set of rules and patterns, no matter what service or resource you are interacting with. This makes it easier for different systems to communicate with each other because they all use the same consistent structure.

RESTful APIs use standard HTTP methods like `GET`, `POST`, `PUT`, and `DELETE` to interact with resources.

2.4.2 Benefits of RESTful APIs

- **Simple, standardized communication**: You don't have to worry about how to format your data request each time, all standardized and industry used.
- **Scalable and stateless**: As it grows in complexity you can easily make modifications and because they are stateless you don't have to worry about which data is in each state and keep track of that across client and server
- **High performance with caching support**: High performance since it supports caching, as the service gets more complex the performance stays high.

3 FastAPI

3.1 Introduction

FastAPI is a modern, fast (hence the name), and highly efficient web framework for building APIs with Python 3.7+ based on standard Python type hints. FastAPI is designed to be easy to use, highly performant, and developer-friendly, making it an excellent choice for both beginners and experienced developers looking to build robust APIs quickly.

One of the core strengths of FastAPI is its speed. It is built on top of Starlette for the web parts and Pydantic for the data handling parts, allowing it to achieve performance levels on par with Node.js and Go. This makes FastAPI particularly suitable for high-performance applications, such as real-time data processing, machine learning model deployment, and microservices architectures.

FastAPI's efficiency extends to the developer experience. It automatically generates interactive API documentation with OpenAPI and ReDoc, which not only provides a clear overview of the API endpoints but also allows for easy testing directly from the browser. This feature significantly enhances the development workflow, making it easier to prototype, debug, and share APIs with other developers or teams.

Moreover, FastAPI's reliance on Python's type hints means that it offers automatic validation, serialization, and documentation with minimal code. This leads to fewer bugs and a cleaner codebase, which is easier to maintain. The framework's ability to catch errors at the development stage reduces the time spent on debugging and increases overall productivity. For developers, FastAPI's intuitive syntax and automatic generation of API documentation contribute to a more enjoyable coding experience, allowing them to focus on the logic of their applications rather than boilerplate code.

In summary, FastAPI is a powerful and efficient tool for building APIs that not only delivers high performance but also significantly enhances the productivity and experience of developers. Its combination of speed, simplicity, and automatic features makes it an ideal choice for modern API development.

3.2 Installation

The guide was made for a Windows 11 machine, things might be slightly different on your end.

1. Install Anaconda:
   https://www.anaconda.com/download.
2. Open Anaconda Prompt
3. Create a new conda environment named API (or anything you like) with Python 3.11.

```
conda create --name API python=3.11
```

4. Switch to new environment.

```
conda activate API
```

5. Install FastAPI.

```
pip install "fastapi[standard]"
```

6. Install pytest (required for testing).

```
pip install pytest
```

7. Install httpx (required for testing).

```
pip install httpx
```

You can find a conda cheat sheat here.

3.3 Tutorial

FastAPI has lots of different and useful features, and only the very basic will be covered in this tutorial. Nevertheless, FastAPI provides an excellent documentation, available here: https://fastapi.tiangolo.com/learn/.

3.3.1 Hello World

First of all, let's create an endpoint, which will send us a *"Hello World"* back.
Therefore, create a new folder with a file named `tutorial.py`. Type the following:

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    return {"message": "Hello World"}
```

- The line `app = FastAPI()` creates a new FastAPI instance. `app` will be the main point of interaction to create all your API.
- `@app.get` creates a new "GET" endpoint. You can also have `@app.put`, `app.post` and `@app.delete`. Inside, you specify the path for this endpoint, here, it is the root path.

- With `async def root()` you create a new asynchronous function. Refer to the FastAPI docs, if you don't know what that is.
- `return "message": "Hello World"` is what the endpoint will return. You can return a dict, list, singular values as str, int, etc. You can also return Pydantic models (you'll see more about that later).

To run the API, go with your Anaconda prompt (activated on API) to this directory and run:

```
fastapi dev tutorial.py
```

If you want to use an IDE, you might have to configure it, e.g. telling which python environment to use, and creating an FastAPI run configuration. This was very easy in PyCharm.

Now you can access this endpoint in your browser (usually) under the following link:
http://localhost:8000/

Now go to http://localhost:8000/docs. Internet access might be required. You will see the automatic interactive API documentation, provided by OpenAPI. You can directly access your endpoints from here.

3.3.2 Path Parameters

You can declare path parameters like that:

```
@app.get("/items/{item_id}")
async def read_item(item_id: int):
    return {"item_id": item_id}
```

If you go to http://localhost:8000/items/3, you will see the correct response.

While you don't need to use type hinting, it is highly recommended. First of all, this will allow your IDE to provide you with completion and error checks. Second, this will allow FastAPI to automatically validate the input to your API. If you try and access
http://localhost:8000/items/sss, you will get an error message. Also, the type will be automatically added to the documentation.

3.3.3 Query Parameters

Parameters, which are inside the definition but not listed in the path, will be automatically interpreted as query parameters. Therefore, you can mix path and query parameters as you like.

```
@app.get("/items2/{item_id}")
async def read_user_item(
    user_name: str, item_id: str, q: str | None = None, short: bool = False
```

```
):
    item = {"item_id": item_id, "customer_name": user_name}
    if q:
        item.update({"q": q})
    if not short:
        item.update(
            {"description": "This is an amazing item that has a long description"}
        )
    return item
```

Here, `item_id` is a path parameter, while the rest are query parameters.

Every parameter which does not have a default value will be required. The convention to define an optional parameter is to set it to `None` as default. Therefore, `user_name` and `item_id` are required, while `q` is optional and `short` has a default value.

3.3.4 Request Body

When you need to send data from a client (let's say, a browser) to your API, you send it as a request body. A request body is data sent by the client to your API. A response body is the data your API sends to the client. Your API almost always has to send a response body. But clients don't necessarily need to send request bodies all the time, sometimes they only request a path, maybe with some query parameters, but don't send a body. To declare a request body, you use Pydantic models with all their power and benefits.

To send data, you should use one of: POST (the more common), PUT, DELETE or PATCH.
Sending a body with a GET request has an undefined behavior in the specifications, nevertheless, it is supported by FastAPI, only for very complex/extreme use cases.
As it is discouraged, the interactive docs with OpenAPI won't show the documentation for the body when using GET, and proxies in the middle might not support it.

Pydantic is a Python library to perform data validation. You declare the "shape" of the data as classes with attributes. And each attribute has a type. Then you create an instance of that class with some values and it will validate the values, convert them to the appropriate type (if that's the case) and give you an object with all the data. And you get all the editor support with that resulting object.

Body parameters (like all the rest in FastAPI) should be Pydantic (as FastAPI is build around Pydanctic). Therefore, you need to use a Pydantic class or define one by yourself. To do the latter:

```
from pydantic import BaseModel

...

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None

```

```
items = []

@app.post("/items/")
async def create_item(item: Item):
    items.append(item)
    return item
```

With just that Python type declaration, FastAPI will:

- Read the body of the request as JSON.
- Convert the corresponding types (if needed).
- Validate the data. If the data is invalid, it will return a nice and clear error, indicating exactly where and what was the incorrect data.
- Give you the received data in the parameter `item`. As you declared it in the function to be of type `Item`, you will also have all the editor support (completion, etc) for all of the attributes and their types.
- Generate JSON Schema definitions for your model, you can also use them anywhere else you like if it makes sense for your project.
- Those schemas will be part of the generated OpenAPI schema, and used by the automatic documentation. (See in Schemas section of OpenAPI docs).

Like before, you can add other path, query and body parameters and FastAPI will do the rest for you:

- If the parameter is also declared in the path, it will be used as a path parameter.
- If the parameter is of a singular type (like int, float, str, bool, etc) it will be interpreted as a query parameter.
- If the parameter is declared to be of the type of a Pydantic model, it will be interpreted as a request body.

To have a singular value in the body, you can do the following:

```
from typing import Annotated

from fastapi import FastAPI, Body

...

@app.post("/singular_value_in_body")
async def singular_value_in_body(x: Annotated[int, Body()]):
    return x
```

`Annotated` can help you with more complex, partially automated data validation. To learn how, consider the FastAPI docs, especially chapters *Query Parameters and String Validations* and *Path Parameters and Numeric Validations*.

3.3.5 Response Types

You can declare the type used for the response by annotating the path operation function return type.

You can use type annotations the same way you would for input data in function parameters, you can use Pydantic models, lists, dictionaries, scalar values like integers, booleans, etc.

FastAPI will use this return type to:

- Validate the returned data.
  If the data is invalid (e.g. you are missing a field), it means that your app code is broken, not returning what it should, and it will return a server error instead of returning incorrect data. This way you and your clients can be certain that they will receive the data and the data shape expected.
- Add a JSON Schema for the response, in the OpenAPI path operation.
  This will be used by the automatic docs. It will also be used by automatic client code generation tools.
- It will limit and filter the output data to what is defined in the return type.
  Basically, if you have a class User and a ResponseUser, which has the same attributes as User except password, and you return an object of type User but define the return type as ResponseUser, FastAPI will automatically cut the password parameter away.

Therefore, it is recommended to always declare a return type.

```
@app.post("/items/")
async def create_item(item: Item) -> Item:
    items.append(item)
    return item
```

3.3.6 Status Codes

You can return status codes like that:

```
@app.post("/items/", status_code=201)
async def create_item(item: Item) -> Item:
    items.append(item)
    return item
```

To learn about the different status codes, consider:
https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

You can import `status` from `FastAPI` and access status codes like `status.HTTP_201_CREATED`.

3.3.7 Exceptions

To raise an exception, use HTTPException:

```
from fastapi import FastAPI, Body, HTTPException

...

@app.post("/items/", status_code=201)
async def create_item(item: Item) -> Item:
    if item.name in map(lambda x: x.name, items):
        raise HTTPException(status_code=409, detail="Item already exists")
    items.append(item)
    return item
```

3.3.8 Background Tasks

You can define background tasks to be run after returning a response.

This is useful for operations that need to happen after a request, but that the client doesn't really have to be waiting for the operation to complete before receiving the response, like sending an email or processing data.

You can do it the following way:

```
from fastapi import FastAPI, Body, HTTPException, BackgroundTasks

...

@app.get("/background_task", status_code=200)
async def background_task(msg: str, background_tasks: BackgroundTasks):
    background_tasks.add_task(delayed_print, msg)

def delayed_print(msg: str):
    time.sleep(5)
    print(f"Hello World, just 5 seconds late. {msg}")
```

If we performed `delayed_print` without adding it to background tasks, it would slow the entire server down and make every other request wait 5s.

Alternatively, endpoints that require the server to wait (f.e. to contact another server), require the use of concurrency with `async/await` to not block the entire server.

3.3.9 Testing

Thanks to Starlette, testing FastAPI applications is easy and enjoyable. It is based on HTTPX, which in turn is designed based on Requests, so it's very familiar and intuitive. With it, you can use pytest directly with FastAPI.

First of all, make sure to have httpx installed and create a new file named `test_tutorial` for the tests:

Then, create a `TestClient` object. You can make GET, PUT, POST, ... request to it with `client.get`, `client.put`, `client.post`, ... The first argument will be the target URL.

- To pass a path or query parameter, add it to the URL itself.
- To pass a JSON body, pass a Python object (e.g. a `dict`) to the parameter `json`.
- If you need to send Form Data instead of JSON, use the `data` parameter instead.
- To pass headers, use a `dict` in the `headers` parameter.
- For cookies, a `dict` in the `cookies` parameter.

Let us test the `read_item` endpoint:

```
from fastapi.testclient import TestClient

from tutorial import app, Item

client = TestClient(app)

def test_read_item():
    response = client.get("/items/3")
    assert response.status_code == 200
    assert response.json()["item_id"] == 3
```

The status code is stored in the `status_code` parameter of `request`. We can call `json()` on `request` to retreive the response body.

Let us write some more tests, and try creating the same item in a row:

```
my_item = Item(
    name="TV",
    description="A simple TV",
    price=200.00,
    tax=0.1,
)

def test_create_item():
    response = client.post("/items", json=my_item.model_dump())
    assert response.status_code == 201
    ret = response.json()
    assert ret["name"] == my_item.name
    assert ret["description"] == my_item.description
    assert ret["price"] == my_item.price
    assert ret["tax"] == my_item.tax

def test_create_same_item():
    response = client.post("/items", json=my_item.model_dump())
    assert response.status_code == 409
```

To run the tests, simply run:

```
pytest
```

To ensure pytest can automatically detect your test files, name them `test_xyz`.

3.4 A small project

To show how a bigger application would be build and to see some more advanced use cases, we want to create a small project.

We want to create a small user database and offer clients to create users, log in and retrieve their profile information via an API.

All the code is available here:
https://github.com/ZalZarak/FastAPI-Tutorial

3.4.1 Structure

```
project
|
+-- src
|   +-- __init__.py
|   +-- classes.py
|   +-- controller.py
|   +-- database.py
|   +-- handler.py
|   +-- main.py
|   +-- service.py
|
+-- test
    +-- __init__.py
    +-- test.py
```

The project is divided into source code (`src`) and test code (`test`). Both need `__init__.py` files to be recognized as python directories.

3.4.2 `main.py`

`main.py` bundles everything into one app.

```python
"""
This is the main class, which bundles the entire rest into one app.
"""

from fastapi import FastAPI

from src.controller import user_router, login_router

app = FastAPI(swagger_ui_oauth2_redirect_url="/login/token")

# import the defined routers, which themselves root forward to endpoints
app.include_router(user_router)
app.include_router(login_router)
```

3.4.3 controller.py

controller.py defines only the API, and leaves the rest to other components.

```python
"""
Controller implement only the API logic, separating it from the other logic.
"""

from fastapi import APIRouter, Depends
from fastapi.security import OAuth2PasswordRequestForm

from src import classes, handler
from src.classes import UserResponse, UserDB, Token
from src.handler import get_current_user #, get_user

##### USER ENDPOINTS #######
"""
User endpoints will be defined here.
Therefore, a separate router is created. It has the prefix "/users"
All endpoints which are wrapped with this router will start with "/users".
"""

user_router = APIRouter(prefix="/users", tags=["Users"])    # tags define metadata
    for documentation purposes

@user_router.post("/", status_code=201)
async def create_user(user: classes.UserSchema) -> None:
    """
    An endpoint to create a new user.
    """

    return handler.create_user(user)


"""
Multiple "advanced" stuff is happening here.

First of all, the user is automatically retrieved via Depends. Depends is
    essentially a function, which FastAPI will
automatically execute if the endpoint is accessed.
If the user has logged in, they will receive a token. This token must be send in
    the header of the request in the form of:
{"Authorization": f'Bearer {access_token}'}
```

```
OpenAPI Docs will do that for you.
The token contains user information, with witch the user will be retrieved with.
The endpoint is restricted, a "Not Authenticated" exception will be raised if the token is not provided, invalid or expired.

Second, we can see some FastAPI "magic" here:
user is of type UserDB, which is also indicated in the return type. Otherwise we would get a warning and "confuse" the
IDE, so that it would eventually provide wrong code completion information.
However, UserDB contains the hashed password, which we should not return. Therefore, we defined a separate UserResponse
class. It is indicated as response_model in the endpoint wrapper. With that, FastAPI will automatically cast the user
from type UserDB to UserResponse, removing the password in the response.

A "classical" approach is commented out below (calls a commented function in handler).
"""
@user_router.get("/profile", status_code=200, response_model=UserResponse)
async def get_user(user: UserDB = Depends(get_current_user)) -> UserDB:
    """
    Returns the user's profile information. User parameter is automatically resolved. User must be authenticated.
    """

    return user

# Alternative without FastAPI "Magic"
"""@user_router.get("/profile", status_code=200)
async def get_user(user: UserDB = Depends(get_current_user)) -> UserResponse:
    return handler.get_user(user)"""



###### LOGIN ENDPOINT #########
"""
Login endpoints will be defined here.
Therefore, a separate router is created. It has the prefix "/login"
All endpoints which are wrapped with this router will start with "/login".

In bigger applications, you would move each router to a new file.
"""

login_router = APIRouter(prefix="/login", tags=["Login"])

@login_router.post("/token", response_model=Token)
async def login_for_access_token(form_data: OAuth2PasswordRequestForm = Depends()) -> dict[str, str]:
    """
    The endpoint provides a login functionality for users.
    The endpoint expects a form data body (not json) like
    {"username": {user_email}, "password": {password}}

    It returns
    {"access_token": access_token, "token_type": "bearer"}
    """

    return handler.login_for_access_token(form_data)
```

3.4.4 handler.py

handler.py implements the "higher" application logic.

```python
"""
The handler implements "higher-level" logic which is not connected to API.
"""

from datetime import timedelta, datetime

from fastapi import HTTPException, Depends
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm

from src import service
from src.classes import UserSchema, UserDB, UserResponse
from src.service import add_user_to_db, get_user_from_db, verify_password

from jose import jwt
from jose.exceptions import ExpiredSignatureError, JWTError


####### USER HANDLER #########
"""
The functions, which the User Controller calls are defined here.
"""


def create_user(user: UserSchema) -> None:
    """
    Create a user, raise HTTPException if exists.
    """

    user = UserDB(**user.model_dump())  # cast user from UserSchema to UserDB

    user.email = user.email.lower()     # make email lower case
    user.password = service.encode_password(user.password)  # hash password

    try:
        add_user_to_db(user)
    except KeyError:
        raise HTTPException(409, "User already exists")


# for an alternative without FastAPI "Magic"
"""def get_user(user: UserDB) -> UserResponse:
    response_user = UserResponse(**user.model_dump(exclude={"password"}))
    return response_user"""


####### LOGIN HANDLER #########
"""
The functions, which the Login Controller and restricted endpoints call (as
    Depends()), are defined here.

In bigger applications, you would move this handler to a new file.
"""

# some security definitions

# the access token is encrypted with this key
```

```python
# DON'T do it like that, create random JWT_KEY and read it from a file.
JWT_KEY = "4976bc345151db1c35c2923a2463f0bf870b083a41afdf2b8e3f5057e61589ea"

# encryption algorithm for token
ALGORITHM = "HS256"

# Defines the scheme, which clients have to follow if the want to access restricted endpoints.
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="login/token")

# Defines expiration time for token
ACCESS_TOKEN_EXPIRE_MINUTES = 30

"""
This function creates an encoded token. It takes the provided data, calculates the expiration time and
encodes that using the defined algorithm and jwt_key.
"""
def create_access_token(data: dict, expires_delta: timedelta = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)) -> str:
    to_encode = data.copy()
    expire = datetime.utcnow() + expires_delta
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, JWT_KEY, algorithm=ALGORITHM)
    return encoded_jwt

"""
This function is called if a user tries to login in.
form_data is of type OAuth2PasswordRequestForm, which has the parameters username and password.
In security, it is advisable to use industry standards like OAuth2, instead of creating your own solution, especially
if you are not exactly sure what you are doing.
"""
def login_for_access_token(form_data: OAuth2PasswordRequestForm) -> dict[str, str]:
    try:
        user = get_user_from_db(form_data.username)
    except KeyError:
        raise HTTPException(401, "User not found")

    verified = verify_password(form_data.password, user.password)

    if not verified:
        raise HTTPException(
            status_code=401,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )

    access_token = create_access_token(
        {"sub": user.email},
        expires_delta=timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES),
    )

    return {"access_token": access_token, "token_type": "bearer"}   # This is just how you do it


"""
This is the function Depends calls.
token is defined as another Depends of the above oath2_scheme. Here, FastApi will
```

```
    automatically resolve it, so that
it reads the token from the provided header.
"""
def get_current_user(token: str = Depends(oauth2_scheme)) -> UserDB:
    try:
        payload = jwt.decode(token, JWT_KEY, algorithms=[ALGORITHM])    # decode
    the token with the key.
        email: str = payload.get("sub")     # read the email from it, which we put
     there when user logged in.
        if email is None or email == "":
            raise HTTPException(
                status_code=401,
                detail="Invalid username or password"
            )
    except ExpiredSignatureError:
        raise HTTPException(
            status_code=401,
            detail="Token expired"
        )
    except JWTError:
        raise HTTPException(
            status_code=401,
            detail="Could not validate credentials"
        )

    return get_user_from_db(email)  # return the user associated with this mail
```

3.4.5 service.py

service.py implements the "lower" application logic, like basic database interactions.

```
"""
Service implements low-level logic, like (fake)-database transaction/calls
"""

from passlib.context import CryptContext

from src.classes import UserDB
from src.database import fake_user_db

# essentially the algorithm to hash the password
# again, don't create your own solutions, use existing ones - if you are not
    exactly sure what you are doing.
pwd_context = CryptContext(schemes=["argon2"], deprecated="auto")

def encode_password(password: str) -> str:
    return pwd_context.hash(password)

def verify_password(plain_password: str, hashed_password: str) -> bool:
    return pwd_context.verify(plain_password, hashed_password)

def add_user_to_db(user: UserDB) -> None:
    # raise error if user exists
    if user.email in fake_user_db.keys():
        raise KeyError("User already registered")

    # save user with key email
```

```
    fake_user_db[user.email] = user

def get_user_from_db(email: str) -> UserDB:
    return fake_user_db[email]
```

3.4.6 classes.py

classes.py defines all classes used.

```
from typing import Annotated

from pydantic import BaseModel, EmailStr, Field


# define base user, which other user classes will inherit from
class UserBase(BaseModel):
    email: EmailStr     # validate automatically that string is email-like

# Database User
class UserDB(UserBase):
    password: str   # will be stored as hash
    personal_info: str|None

# User for Input (creating new user)
class UserSchema(UserBase):
    password: Annotated[str, Field(min_length=8)]   # validate automatically, that password has minimum length of 8
    personal_info: str|None

# Will be used in Responses
class UserResponse(UserBase):
    personal_info: str|None


"""
this is the token class. An object is returned if user logs in.
To access restricted endpoints, user must send this token in the header in the form of:
{"Authorization": f'Bearer {access_token}'}
This is done automatically if using OpenAPI Docs.
"""
class Token(BaseModel):
    access_token: str
    token_type: str
```

3.4.7 database.py

database.py defines/creates the database.

```
# Initialize db
# Will be stored as user.email:user

fake_user_db = {}
```

3.4.8 test.py

`test.py` implements all the tests.

```
import copy

from starlette.testclient import TestClient

from src.classes import UserSchema, UserDB
from src.main import app

"""
in an application which uses a persistent database (which is not deleted on restart), you would create a
testing session of your database and reset it at the end, or create a new database, so that your tests don't alter
the existing database
Tutorial: https://fastapi.tiangolo.com/advanced/testing-database/#create-the-new-database-session
here it is not necessary, since our fake database is not persistent
"""
from src.database import fake_user_db

client = TestClient(app)

# create test user
my_user = UserSchema(email='my_user@email.com', password='12345678', personal_info="I am just a user.")

# global variable to store the login header across the different functions
global user_access_header

def test_create_user_password_to_short():
    my_user2 = copy.deepcopy(my_user)
    my_user2.password = '1234567'

    # model_dump() gives you the dictionary of a pydantic object.
    # body are submitted as json
    response = client.post('users/', json=my_user2.model_dump())

    assert response.status_code == 422, ("status code should be 422 - unprocessable entity, if the user tries to create "
                                         "a password which is not long enough")
    assert len(fake_user_db) == 0

    """
    in a bigger project, you should use assert with error messages, so that developers can see directly
    what went wrong. Like:

    assert response.status_code == 422, ("status code should be 422 - unprocessable entity, if the user tries to create "
                                         "a password which is not long enough")

    Here I left it out for simplicity.
    """


def test_create_user():
    response = client.post('users/', json=my_user.model_dump())
```

```
50
51      assert response.status_code == 201
52
53      db_user: UserDB = fake_user_db[my_user.email]
54
55      assert db_user.email == my_user.email
56      assert db_user.personal_info == my_user.personal_info
57      assert db_user.password != my_user.password
58
59
60  def test_create_user_again():
61      response = client.post('users/', json=my_user.model_dump())
62
63      assert response.status_code == 409
64      assert len(fake_user_db) == 1
65
66
67  def test_login_nonexistent_user():
68      form_data = {"username": "random@email.com", "password": "12345678"}
69
70      # this endpoint expects form_data, so pass it as data parameter instead
71      response = client.post('/login/token', data=form_data)
72
73      assert response.status_code == 401
74      assert "access_token" not in response.json()
75
76
77  def test_login_wrong_password():
78      form_data = {"username": my_user.email, "password": "123456789"}
79
80      response = client.post('/login/token', data=form_data)
81
82      assert response.status_code == 401
83      assert "access_token" not in response.json()
84
85
86  def test_login():
87      form_data = {"username": my_user.email, "password": my_user.password}
88
89      response = client.post('/login/token', data=form_data)
90
91      assert response.status_code == 200
92      assert "access_token" in response.json()
93
94      global user_access_header   # access this variable from global scope
95      # this is just how the header has to look like
96      user_access_header = {"Authorization": f'Bearer {response.json()["access_token
        "]}'}
97
98
99  def test_get_user():
100     # header goes in header
101     response = client.get('/users/profile', headers=user_access_header)
102
103     assert response.status_code == 200
104
105     ret = response.json()
106
107     assert ret["email"] == my_user.email
108     assert ret["personal_info"] == my_user.personal_info
109     assert "password" not in ret.keys()
110
```

```
111
112 def test_get_user_wrong_token():
113     wrong_token = copy.deepcopy(user_access_header)
114     wrong_token['Authorization'] = wrong_token['Authorization'] + '1'
115
116     response = client.get('/users/profile', headers=wrong_token)
117
118     assert response.status_code == 401
```