

# **Laboratorium 1**

Interfejsy WWW

# Spis treści

Wstęp do laboratorium .....	4
Cel laboratorium.....	4
Zakres i charakter laboratorium.....	4
Technologie wykorzystywane w laboratorium.....	5
Windows Subsystem for Linux (WSL).....	5
Visual Studio Code.....	5
Python i FastAPI.....	5
JSON Web Token (JWT).....	5
Docker.....	6
Przebieg laboratorium.....	7
1. Instalacja WSL2 + Ubuntu (na Windows).....	7
1.1. Instalacja / włączenie WSL.....	7
1.2. Zainstaluj Ubuntu (jeśli nie zainstalowało się automatycznie).....	7
1.3. Aktualizacja Ubuntu w WSL.....	7
2. Instalacja nvm w WSL (Ubuntu).....	8
2.1. Instalacja nvm (skrypt instalacyjny).....	8
2.2. Załóż nvm do bieżącej sesji (albo otwórz nowy terminal).....	8
3. Instalacja Node.js (LTS) przez nvm w WSL.....	8
3.1. Zainstaluj najnowsze LTS (możecie równolegle mieć wiele różnych wersji node dzięki nvm).....	8
3.2. Ustaw domyślną wersję Node dla nowych terminali.....	8
3.3. Weryfikacja.....	8
3.4 Szybki test „czy środowisko Node działa”.....	8
4. VS Code + wtyczki (WSL + Python/FastAPI + Angular).....	9
4.1. Pobranie i instalacja VS Code (Windows).....	9
4.2. Instalacja komponentu „Remote – WSL”.....	9
4.3. Otwieranie projektu w kontekście WSL.....	9
4.4. Instalacja rozszerzeń dla backendu Python + FastAPI.....	10
4.5. Instalacja rozszerzeń dla frontendu Angular.....	11
4.6. Docker w VS Code.....	11
4.7. Przydatne opcje w VS Code.....	11
4.8. Test poprawności konfiguracji.....	12
5. Backend – FastAPI + autoryzacja JWT.....	12
5.1. Utworzenie katalogu projektu.....	12
5.2. Utworzenie środowiska wirtualnego (virtualenv).....	12
5.3. Instalacja wymaganych bibliotek.....	13
5.4. Struktura projektu.....	13
5.5. Hardkodowana baza użytkowników.....	13
5.6. Logika JWT i autoryzacji.....	14
5.7. Endpointy API.....	15
5.8. Uruchomienie aplikacji.....	16
5.9. Test w Swagger UI.....	16
6. Instalacja Docker Engine (Docker Desktop + WSL).....	17
6.1. Czym jest Docker w tym scenariuszu.....	17
6.2. Pobranie Docker Desktop.....	17
6.3. Instalacja Docker Desktop.....	17
6.4. Pierwsze uruchomienie Docker Desktop.....	17
6.5. Integracja Dockera z WSL.....	18
6.6. Test działania Dockera w WSL.....	18
6.7. Docker + VS Code.....	18

7: Przygotowanie Dockera do uruchomienia API (Dockerfile).....	19
7.1. Uporządkowanie zależności (requirements.txt).....	19
7.2. Dodaj plik .dockerignore.....	19
7.3. Dockerfile dla FastAPI.....	20
7.4. Budowa obrazu.....	20
8. Uruchomienie i testowanie w Dockerze.....	21
8.1. Uruchom kontener.....	21
8.2. Test w przeglądarce (Swagger).....	21
8.3. Test logowania (JWT).....	21
8.4. Test endpointu chronionego.....	22
8.5. Test negatywny.....	22
8.6. Zarządzanie kontenerem.....	22
9. Zakończenie.....	23

# Wstęp do laboratorium 1

**Temat:** Budowa i konteneryzacja backendu webowego z autoryzacją JWT

## Cel laboratorium

Celem niniejszego laboratorium jest zapoznanie studentów z **pełnym, praktycznym procesem przygotowania nowoczesnego backendu webowego**, począwszy od konfiguracji środowiska developerskiego, poprzez implementację mechanizmu autoryzacji, aż po uruchomienie aplikacji w kontenerze Docker. Laboratorium zostało zaprojektowane w taki sposób, aby odzwierciedlać rzeczywisty ekosystem pracy programisty www, spotykany w projektach komercyjnych oraz zespołowych.

W trakcie zajęć macie samodzielnie przygotować środowisko pracy oparte na systemie Linux uruchamianym w ramach **Windows Subsystem for Linux (WSL)**, zainstalujecie narzędzia niezbędne do tworzenia aplikacji backendowych i frontendowych oraz zbudujecie prostą, ale kompletną aplikację API opartą o framework FastAPI. Kluczowym elementem laboratorium będzie implementacja mechanizmu uwierzytelniania i autoryzacji z wykorzystaniem tokenów JWT (JSON Web Token), który stanowi obecnie jeden z najczęściej stosowanych standardów zabezpieczania usług sieciowych.

Istotnym celem zajęć jest również uświadomienie Wam różnicy pomiędzy:

- uruchamianiem aplikacji lokalnie w środowisku developerskim,
- a uruchamianiem jej w kontenerze Dockera, który zapewnia powtarzalność, izolację i przenośność aplikacji.

Na zakończenie laboratorium będziecie potrafili:

- uruchomić własne API zarówno lokalnie, jak i w kontenerze,
- przetestować jego działanie,
- nie bać się stosowania konteneryzacji - dziś standardu w procesach CI/CD i wdrożeniowych.

## Zakres i charakter laboratorium

Laboratorium ma charakter praktyczny i zadaniowy. Zamiast skupiać się na pojedynczych, oderwanych od siebie przykładach, realizujecie jeden spójny scenariusz, w którym kolejne etapy logicznie wynikają z poprzednich. Dzięki temu macie możliwość prześledzenia pełnej ścieżki:

*od pustego środowiska systemowego do działającej usługi webowej dostępnej przez przeglądarkę.*

Zastosowane rozwiązania są celowo uproszczone tam, gdzie nie jest to kluczowe z punktu widzenia dydaktycznego (np. hardkodowana baza użytkowników), a jednocześnie zachowują te same mechanizmy, które stosuje się w systemach produkcyjnych (np. haszowanie haseł, weryfikacja tokenów, separacja odpowiedzialności w kodzie).

# Technologie wykorzystywane w laboratorium

## Windows Subsystem for Linux (WSL)

Podstawą Waszego środowiska pracy będzie **WSL**, czyli mechanizm umożliwiający uruchamianie systemu Linux bezpośrednio na Windowsie. Dzięki temu możecie:

- pracować w środowisku zgodnym z serwerami produkcyjnymi,
- korzystać z narzędzi linuksowych (bash, apt, curl),
- jednocześnie zachowując wygodę pracy w systemie Windows.

W kontekście tworzenia aplikacji webowych WSL pozwala Wam uniknąć wielu problemów wynikających z różnic między środowiskiem developerskim a serwerowym – przy okazji będziecie mieli łagodne wprowadzenie w korzystanie z Linux-a

## Visual Studio Code

Visual Studio Code będzie Waszym głównym edytorem kodu i środowiskiem pracy. Dzięki rozszerzeniu *Remote – WSL* możecie pracować bezpośrednio na plikach znajdujących się w systemie Linux, jednocześnie korzystając z funkcji nowoczesnego IDE.

W trakcie laboratorium VS Code posłuży Wam do:

- edycji kodu w TypeScript oraz (okazjonalnie) w Pythonie,
- pracy z terminaliem WSL,
- integracji z Dockerem,
- testowania i uruchamiania aplikacji.

## Python i FastAPI

Naszym celem jest tworzenie frontendów – ale zaczniemy od kadłubkowego backendu do późniejszego wykorzystania przez nasz frontend. Backend aplikacji przygotujecie w języku Python, wykorzystując framework FastAPI. Jest to nowoczesny framework do tworzenia API HTTP, który:

- pozwala tworzyć wydajne aplikacje,
- automatycznie generuje dokumentację OpenAPI (Swagger),
- wymusza czytelną i logiczną strukturę kodu,
- ułatwia implementację mechanizmów autoryzacji.

Dzięki FastAPI możecie skupić się na logice aplikacji, jednocześnie pracując z narzędziem, które bardzo dobrze oddaje realia współczesnych backendów webowych.

## JSON Web Token (JWT)

Jednym z najważniejszych elementów laboratorium będzie implementacja mechanizmu autoryzacji z wykorzystaniem JWT. Tokeny JWT są powszechnie stosowane w architekturach REST API, zwłaszcza w systemach, w których backend i frontend są od siebie niezależne.

Podczas zajęć:

- zaimplementujecie proces logowania użytkownika,
- wygenerujecie token JWT,
- zabezpieczycie wybrane endpointy,
- przetestujecie zachowanie aplikacji w przypadku braku lub nieprawidłowego tokenu.

## Docker

Ostatnim, ale bardzo istotnym elementem laboratorium jest Docker. Docker pozwala uruchamiać aplikacje w izolowanych kontenerach, co znaczco upraszcza proces ich wdrażania i uruchamiania na różnych maszynach.

W ramach laboratorium:

- przygotujecie własny obraz Dockera,
- uruchomicie backend FastAPI w kontenerze,
- przetestujecie działanie API w identyczny sposób jak w środowisku lokalnym.

Dzięki temu zrozumiecie, dlaczego Docker stał się standardowym narzędziem w nowoczesnym tworzeniu i wdrażaniu aplikacji backendowych.

Laboratorium stanowi kompleksowe wprowadzenie do praktycznych aspektów budowy backendu webowego. Łączy ono zagadnienia systemowe, programistyczne i wdrożeniowe, pokazując Wam, że współczesna aplikacja webowa to nie tylko kod, ale również środowisko, bezpieczeństwo oraz sposób uruchamiania aplikacji.

Umiejętności zdobyte w trakcie tych zajęć będą solidną podstawą do dalszej nauki technologii webowych oraz do pracy nad bardziej rozbudowanymi projektami w przyszłości.

# Przebieg laboratorium

Laboratorium macie podzielone na 9 kroków:

## 1. Instalacja WSL2 + Ubuntu (na Windows)

### 1.1. Instalacja / włączenie WSL

1. Otwórz **Windows PowerShell (Admin)**. Uruchamianie jako admin – klikacie prawym przyciskiem myszy na programie i wybieracie odpowiednią opcję
2. Wpisz:

```
wsl --install
```

To polecenie przygotuje podsystem Windows System for Linux – czyli w uproszczeniu maszynę wirtualną do uruchamiania aplikacji Linux-owych w Windows. Po zainstalowaniu możemy dostać prośbę o restart, finalnie możecie sprawdzić status po zainstalowaniu poleceniem:

```
wsl --status  
wsl -l -v
```

Jeśli masz dystrybucję na WSL1, ustaw WSL2:

```
wsl --set-default-version 2
```

### 1.2. Zainstaluj Ubuntu (jeśli nie zainstalowało się automatycznie)

Najprościej ze sklepu Microsoft Store (Ubuntu 24.04.1 LTS) Link (Microsoft Store):

```
https://apps.microsoft.com/detail/9nz3klhxjdjp5
```

Po instalacji uruchom „Ubuntu” z menu Start i dokończ inicjalizację (utwórz użytkownika + hasło).

Ogólnie – lista dostępnych dystrybucji Linux-a jest możliwa do uzyskania poprzez:

```
wsl --list -online
```

Zachęcam też do zapoznania się z innymi poleceniami ułatwiającymi korzystanie z WSL:

```
https://learn.microsoft.com/pl-pl/windows/wsl/basic-commands
```

### 1.3. Aktualizacja Ubuntu w WSL

Terminal Ubuntu uruchamiacie z menu start. W terminalu Ubuntu aktualizujecie system:

```
sudo apt update  
sudo apt -y upgrade
```

Dodatkowo warto zainstalować narzędzia do budowania aplikacji – niekoniecznie będą wymagane przez nasz projekt – ale może się zdarzyć, że node doinstalowywając pakiet będzie budował coś ze źródeł:

```
sudo apt -y install build-essential curl git
```

## 2. Instalacja nvm w WSL (Ubuntu)

**Źródło oficjalne:** repozytorium nvm-sh/nvm na GitHub. [GitHub](#)

### 2.1. Instalacja nvm (skrypt instalacyjny)

W terminalu Ubuntu wprowadź:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.3/install.sh | bash
```

To jest zalecana droga instalacji nvm – zawsze warto sprawdzić aktualną wersję nvm na github.  
Alternatywnie instalacja jest możliwa przez wget:

```
wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.3/install.sh | bash
```

### 2.2. Załóż nvm do bieżącej sesji (albo otwórz nowy terminal)

Po instalacji nie będziecie mieli zaktualizowanego z automatu środowiska systemowego.  
Najprościej je zaktualizować poprzez restart maszyny Ubuntu, co w WSL sprowadza się do zamknięcia i ponownego otwarcia okna Linux-a. Jeśli chcecie bez zamazywania wydajecie polecenie:

```
source ~/.bashrc
```

Następnie należy przetestować NVM czy działa:

```
command -v nvm  
nvm --version
```

## 3. Instalacja Node.js (LTS) przez nvm w WSL

### 3.1. Zainstaluj najnowsze LTS (możecie równolegle mieć wiele różnych wersji node dzięki nvm)

```
nvm install --lts
```

### 3.2. Ustaw domyślną wersję Node dla nowych terminali

```
nvm alias default 'lts/*'
```

### 3.3. Weryfikacja

```
node -v  
npm -v  
which node  
which npm
```

### 3.4 Szybki test „czy środowisko Node działa”

```
node -e "console.log('WSL + Node działa ')"  
npm -v
```

Jeśli powyższe nie zadziała – szukaj wsparcia u prowadzącego, cioci Google czy wujka GPT ;)

## 4. VS Code + wtyczki (WSL + Python/FastAPI + Angular)

Założenia: Masz już działające WSL (Ubuntu) i dostęp do terminala Linux (z punktu 1) oraz Node jest instalowany w WSL, więc VS Code ma pracować „w środku” WSL.

### 4.1. Pobranie i instalacja VS Code (Windows)

Zainstaluj VS Code

1. Wejdź na stronę pobierania VS Code:

```
https://code.visualstudio.com/download
```

2. Wybierz wersję **Windows** (User Installer albo System Installer).

Instalujesz edytor na Windows, ale dzięki integracji z WSL będzie on uruchamiał terminal, interpretery i narzędzia (Python/Node) po stronie Linuxa.

Zainstaluj VS Code

1. Uruchom instalator.
2. Zaznacz opcje:
  - Add to PATH (ułatwia uruchamianie `code` z terminala)
  - Add “Open with Code” / integracje z menu kontekstowym (opcjonalnie)
3. Zakończ instalację.

VS Code instaluje się jako aplikacja Windows. Dodatkowo dodaje polecenie `code` do PATH.

### 4.2. Instalacja komponentu „WSL”

Zainstaluj rozszerzenie WSL

1. Otwórz VS Code.
2. Otwórz panel Extensions (**Ctrl+Shift+X**).
3. Wyszukaj: **WSL** (autor: Microsoft).
4. Kliknij **Install**.

To rozszerzenie pozwala VS Code „wejść” do środowiska WSL i uruchamiać narzędzia po stronie Linuxa. W praktyce: edytor działa na Windows, ale narzędzia do backend (Python, pip, uvicorn) i frontend (node, npm) działają w WSL.

### 4.3. Otwieranie projektu w kontekście WSL

Uruchom VS Code z poziomu WSL (rekomendowane)

1. Otwórz terminal Ubuntu (WSL).
2. Przejdź do folderu roboczego, np.:

```
cd ~  
mkdir -p lab_iwww  
cd lab_iwww
```

3. Uruchom VS Code poleceniem:

```
code .
```

- `code .` uruchamia VS Code (Windows), ale od razu w trybie „Remote: WSL”.
- VS Code doinstaluje w WSL mały komponent **VS Code Server** (automatycznie).
- Dzięki temu debugowanie, terminal, python interpreter i node będą „linuxowe”, a nie windowsowe.

Jeśli `code` nie działa: uruchom VS Code normalnie w Windows, potem **Ctrl+Shift+P** - wpisz WSL: Reopen Folder in WSL.

Sprawdź, czy na pewno jesteś w WSL

W VS Code zobaczysz w lewym dolnym rogu zielony pasek: **“WSL: Ubuntu”** (lub podobnie). Ten zielony wskaźnik oznacza, że rozszerzenia i terminal będą działały w WSL, a nie na Windows.

#### 4.4. Instalacja rozszerzeń dla backendu Python + FastAPI

Zainstaluj rozszerzenia Pythona (w kontekście WSL)

W **Ctrl+Shift+X** zainstaluj:

##### 1. **Python** (Microsoft)

Marketplace:

```
https://marketplace.visualstudio.com/items?itemName=ms-python.python
```

##### 2. **Pylance** (Microsoft) – jeśli nie zainstaluje się z automatu wraz z Pythonem

Marketplace:

```
https://marketplace.visualstudio.com/items?itemName=ms-python.vscode-pylance
```

##### 3. **Python Debugger** (Microsoft) – jeśli nie zainstaluje się z automatu wraz z Pythonem

Marketplace:

```
https://marketplace.visualstudio.com/items?itemName=ms-python.debugpy
```

- **Python**: wykrywanie interpreterów, uruchamianie plików, terminale, linting.

- **Pylance**: inteligentne podpowiedzi, typowanie, szybka analiza kodu (duży komfort pracy).

- **debugpy**: umożliwia debugowanie (breakpointy, wykonanie step by step).

Uwaga: rozszerzenia instalują się osobno dla Windows i osobno dla WSL. Instaluj je, gdy VS Code jest w trybie “WSL: Ubuntu”.

Ustaw interpreter Pythona z WSL

1. **Ctrl+Shift+P**
2. Python: Select Interpreter
3. Wybierz interpreter z WSL (np. `/usr/bin/python3` albo `.venv/bin/python` jeśli w projekcie będzie `virtualenv`).

VS Code musi wiedzieć którego Pythona używać (w WSL). Bez tego może próbować użyć interpretera z Windows – a to nie zadziała tak jakbyśmy chcieli (nie będzie m.in dostępu do paczek zainstalowanych w linux-ie, i nie da się tego łatwo skonteneryzować).

#### 4.5. Instalacja rozszerzeń dla frontendu Angular

Zainstaluj rozszerzenia Angular/TypeScript

W Ctrl+Shift+X zainstaluj:

##### 1. Angular Language Service

Marketplace:

<https://marketplace.visualstudio.com/items?itemName=Angular.ng-template>

##### 2. ESLint (dla TS/JS)

Marketplace:

<https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint>

##### 3. (opcjonalnie) Prettier - Code formatter

Marketplace:

<https://marketplace.visualstudio.com/items?itemName=esbenp.prettier-vscode>

- Angular Language Service daje podpowiedzi w HTML templates, sprawdza bindy, komponenty itd.
- ESLint wyłapuje błędy i złe praktyki w TypeScript.
- Prettier automatycznie formatuje kod (mniej „walki” w zespołach).

#### 4.6. Docker w VS Code

Zainstaluj rozszerzenie Docker

W Ctrl+Shift+X:

##### 1. Docker (Microsoft)

Marketplace:

<https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-docker>

VS Code dostaje panel do podglądu obrazów/kontenerów, logów i helpery do Dockerfile. To nie zastępuje Dockera – to tylko integracja.

#### 4.7. Przydatne opcje w VS Code

Włącz automatyczny format przy zapisie - w VS Code:

- File | Preferences | Settings
- wyszukaj: Format On Save → włącz go

VS Code uruchamia formattera przy zapisie pliku – kod jest od razu „elegancki” – nie ważne jak brzydko go napiszecie.

Ustaw domyślny formatter - w ustawieniach wyszukaj Default Formatter i wybierz:

- dla Python: (zwykle Python/Pylance lub Black jeśli będziesz używać)
- dla TS/HTML: Prettier (jeśli zainstalowany)

Gdy masz kilka formatterów, VS Code musi wiedzieć, którego użyć.

#### 4.8. Test poprawności konfiguracji

Sprawdź terminal i środowisko

W VS Code: Terminal → New Terminal i wpisz:

```
uname -a  
python3 --version  
node -v  
npm -v
```

- `uname -a` powinno pokazać Linux (WSL).
- Python/Node powinny być zainstalowane w WSL. To potwierdza, że pracujesz w dobrym środowisku.

## 5. Backend – FastAPI + autoryzacja JWT

Cel zadania:

- uruchomić backend FastAPI,
- zaimplementować pełny proces logowania z JWT,
- zabezpieczyć endpoint wymagający autoryzacji,
- przetestować całość w Swagger UI.

### 5.1. Utworzenie katalogu projektu

Utwórz katalog projektu: W terminalu WSL (w VS Code):

```
cd ~/lab_iwww  
mkdir backend  
cd backend
```

Tworzymy osobny katalog na backend API. To dobra praktyka – backend i frontend są logicznie rozdzielone.

### 5.2. Utworzenie środowiska wirtualnego (virtualenv)

Zainstaluj pythona (jeśli go nie masz w WSL):

```
sudo apt install python3-pip
```

Utwórz virtualenv

```
python3 -m venv .venv
```

Aktywuj virtualenv

```
source .venv/bin/activate
```

Po aktywacji w terminalu pojawi się:

```
(.venv)
```

Virtualenv izoluje biblioteki Pythona dla tego projektu. Dzięki temu:

- nie „zaśmiecamy” systemowego Pythona,
- Docker później dostanie dokładnie te same zależności.

### 5.3. Instalacja wymaganych bibliotek

Instalacja FastAPI i narzędzi JWT

```
pip install fastapi uvicorn python-jose passlib[bcrypt]
```

- **fastapi** – framework API
- **uvicorn** – serwer ASGI (uruchamia aplikację)
- **python-jose** – tworzenie i weryfikacja JWT
- **passlib[bcrypt]** – bezpieczne haszowanie haseł

Trochę w tym zabawy – w hardkodowanej bazie też nie będziemy trzymali jawnie haseł ;)

### 5.4. Struktura projektu

Utwórz pliki projektu

```
mkdir app
touch app/main.py app/auth.py app/users.py
```

Struktura:

```
backend/
  └── app/
    ├── main.py
    ├── auth.py
    └── users.py
  .venv/
```

Rozdzielimy odpowiedzialności:

- **users.py** – „baza danych” użytkowników
- **auth.py** – logika JWT
- **main.py** – endpointy API

### 5.5. Hardkodowana baza użytkowników

Zawartość **users.py**

```
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
```

```

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

users_db = {
    "admin": {
        "username": "admin",
        "hashed_password": hash_password("admin123"),
        "role": "admin",
    },
    "user": {
        "username": "user",
        "hashed_password": hash_password("user123"),
        "role": "user",
    },
}

```

- Tworzymy „bazę danych” w postaci słownika.
- Hasła są hashowane przy starcie aplikacji.
- W realnym systemie hashe byłyby zapisane w bazie danych, ale mechanizm jest identyczny.

## 5.6. Logika JWT i autoryzacji

Zawartość auth.py

```

from datetime import datetime, timedelta
from jose import jwt, JWTError
from passlib.context import CryptContext
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer

from app.users import users_db

SECRET_KEY = "super-secret-key"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="login")

def verify_password(plain_password, hashed_password):
    return pwd_context.verify(plain_password, hashed_password)

def authenticate_user(username: str, password: str):
    user = users_db.get(username)
    if not user:
        return None
    if not verify_password(password, user["hashed_password"]):
        return None
    return user

def create_access_token(data: dict, expires_delta: timedelta | None = None):
    to_encode = data.copy()
    expire = datetime.utcnow() + (expires_delta or timedelta(minutes=15))
    to_encode.update({"exp": expire})
    return jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)

def get_current_user(token: str = Depends(oauth2_scheme)):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
    
```

```

username: str = payload.get("sub")
if username is None:
    raise HTTPException(status_code=401)
except JWTError:
    raise HTTPException(status_code=401)
user = users_db.get(username)
if user is None:
    raise HTTPException(status_code=401)
return user

```

- JWT zawiera informację o użytkowniku (sub)
- Token ma czas ważności
- Każdy request do endpointu chronionego jest weryfikowany
- Brak lub zły token jest sygnalizowane przez kod http - 401 Unauthorized

## 5.7. Endpointy API

Zawartość main.py

```

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

from app.auth import authenticate_user, create_access_token, get_current_user

app = FastAPI(title="FastAPI JWT Demo")

@app.post("/login")
def login(form_data: OAuth2PasswordRequestForm = Depends()):
    user = authenticate_user(form_data.username, form_data.password)
    if not user:
        raise HTTPException(status_code=401, detail="Invalid credentials")

    token = create_access_token({"sub": user["username"]})
    return {"access_token": token, "token_type": "bearer"}

@app.get("/me")
def read_me(current_user: dict = Depends(get_current_user)):
    return {
        "username": current_user["username"],
        "role": current_user["role"],
    }

```

- /login:
  - przyjmuje login i hasło,
  - generuje JWT,
  - zwraca token.
- /me:
  - wymaga JWT,
  - zwraca dane aktualnie zalogowanego użytkownika.

## 5.8. Uruchomienie aplikacji

Start serwera

```
uvicorn app.main:app --reload
```

- `uvicorn` uruchamia serwer ASGI,
- `--reload` powoduje automatyczny restart po zmianie kodu (dev-mode).

Aplikacja dostępna pod:

```
http://127.0.0.1:8000
```

## 5.9. Test w Swagger UI

Otwórz Swagger, wejdź w przeglądarce na:

```
http://127.0.0.1:8000/docs
```

FastAPI automatycznie generuje dokumentację OpenAPI + interaktywny interfejs testowy.

Test logowania

- Endpoint: `POST /login`
- Dane:
  - `username: admin`
  - `password: admin123`

Skopiuj `access_token`.

Autoryzacja

- Kliknij **Authorize**
- Wpisz:

```
Bearer <TOKEN>
```

Test endpointu chronionego

- `GET /me`
- Oczekiwany wynik:

```
{  
    "username": "admin",  
    "role": "admin"  
}
```

## 5.10 Zadanie domowe

Zmieńcie to API autoryzacji do poziomu w pełni profesjonalnego – dodajcie drugi token (refresh), oraz endpoint do odświeżania access token (to ten który już macie) bez konieczności dodatkowego logowania – przy wykorzystaniu refresh token.

Standardowo – backend po zalogowaniu powinien Wam odpowiedzieć dwoma tokenami – jeden wykorzystujecie na bieżąco (access), drugi pamiętacie (refresh). Jak ważność access się skończy – odwołujecie się do endpointu nazwijmy go `/renew` – i tam podajecie refresh, w zwrotce dostajecie nowy access, ważny na kolejne  $x$  minut

## 6. Instalacja Docker Engine (Docker Desktop + WSL)

### 6.1. Czym jest Docker w tym scenariuszu

Docker Desktop działa na Windows, kontenery uruchamiamy z WSL, a Docker Engine faktycznie działa w tle w WSL2. Co jest ważne:

- nie uruchamiamy Dockera „w PowerShell”,
- nie instalujemy dockera bezpośrednio przez `apt`,
- korzystamy z Docker Desktop + WSL integration.

### 6.2. Pobranie Docker Desktop

Pobierz Docker Desktop – wejdź na oficjalną stronę:

<https://www.docker.com/products/docker-desktop/>

Kliknij Download for Windows. Pobierasz aplikację, która:

- instaluje Docker Engine,
- konfiguruje integrację z WSL2,
- zapewnia narzędzia CLI (`docker`, `docker compose`).

### 6.3. Instalacja Docker Desktop

Uruchom instalator

1. Uruchom pobrany plik `.exe`.
2. Zostaw zaznaczone domyślne opcje, w szczególności:
  - Use WSL 2 instead of Hyper-V
3. Dokoncz instalację.

Docker Desktop:

- instaluje backend Dockera,
- konfiguruje WSL2 jako środowisko uruchomieniowe,
- dodaje polecenie `docker` do systemu.

Restart systemu (jeśli wymagany)

Po instalacji zrestartuj Windows, jeśli instalator o to poprosi.

## **6.4. Pierwsze uruchomienie Docker Desktop**

Uruchom Docker Desktop

1. Uruchom Docker Desktop z menu Start.
2. Poczekaj, aż status zmieni się na: “Docker Desktop is running” lub podobnie.

Docker uruchamia swoje usługi w tle (w tym WSL2-based engine). Bez uruchomionego Docker Desktop polecenie `docker` nie będzie działać.

## **6.5. Integracja Dockera z WSL**

Włącz integrację z Ubuntu:

1. W Docker Desktop wejdź w:  
`Settings | Resources | WSL Integration`
2. Włącz:
  - Enable integration with my default WSL distro
  - lub konkretnie z wybraną dystrybucją
3. Kliknij `Apply & Restart`.

Docker udostępnia swoje gniazdo (`/var/run/docker.sock`) do WSL. Dzięki temu możesz uruchamiać kontenery bezpośrednio z terminala Ubuntu.

## **6.6. Test działania Dockera w WSL**

Sprawdź Dockera w terminalu WSL: Otwórz terminal Ubuntu (najlepiej w VS Code):

```
docker --version
```

Oczekiwany wynik (przykładowy):

```
Docker version 26.x.x, build ...
```

Uruchom testowy kontener

```
docker run hello-world
```

Oczekiwany efekt:

- Docker pobiera obraz `hello-world`
- Wyświetla komunikat powitalny

Co się dzieje :

1. Docker pobiera gotowy obraz z Docker Hub.
2. Tworzy kontener.
3. Uruchamia go.
4. Kontener wypisuje tekst i kończy działanie.

To jest minimalny dowód, że Docker działa poprawnie.

## 6.7. Docker + VS Code

Sprawdź integrację w VS Code

1. Otwórz VS Code (w trybie **WSL: Ubuntu**).
2. W panelu bocznym kliknij ikonę **Docker**.

VS Code:

- komunikuje się z Docker Engine,
- pokazuje obrazy, kontenery i logi,
- w kolejnych punktach pomoże w pracy z Dockerfile.

## 7: Przygotowanie Dockera do uruchomienia API (Dockerfile)

Teraz chcemy:

- spakować aplikację FastAPI do obrazu Dockera,
- uruchamiać ją identycznie na każdym komputerze,
- przygotować podstawy pod wdrożenia.

### 7.1. Uporządkowanie zależności (requirements.txt)

Upewnij się, że jesteś w virtualenv. W terminalu (WSL) w katalogu backend/:

```
cd ~/lab_iwww/backend  
source .venv/bin/activate
```

W ten sposób włączasz środowisko, w którym były instalowane biblioteki. Dzięki temu wygenerujesz poprawną listę zależności.

Wygeneruj requirements.txt

```
pip freeze > requirements.txt
```

pip freeze zapisuje *konkretne wersje* bibliotek, które są zainstalowane. Docker później użyje tej listy, aby odtworzyć identyczne środowisko w kontenerze.

### 7.2. Dodaj plik .dockerignore

W katalogu backend/ utwórz plik .dockerignore:

```
cat > .dockerignore << 'EOF'  
.venv  
__pycache__  
*.pyc  
*.pyo  
*.pyd  
.git  
.vscode  
EOF
```

.dockerignore mówi Dockerowi, czego nie kopiować do obrazu. Nie chcemy kopiować virtualenv ani cache, bo:

- powiększają obraz,
- mogą powodować konflikty platform (Linux vs Windows),
- są zbędne do uruchomienia aplikacji.

### 7.3. Dockerfile dla FastAPI

W katalogu backend/:

```
cat > Dockerfile << 'EOF'
FROM python:3.12-slim

# 1) Ustaw katalog roboczy w kontenerze
WORKDIR /app

# 2) Skopiuj listę zależności i zainstaluj je
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# 3) Skopiuj kod aplikacji
COPY app ./app

# 4) Otwórz port aplikacji (informacyjnie)
EXPOSE 8000

# 5) Komenda startowa
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
EOF
```

1. **FROM python:3.12-slim** – bierzemy lekką bazę z Pythonem.
2. **WORKDIR /app** – wewnątrz kontenera pracujemy w /app.
3. **COPY requirements.txt + RUN pip install** – instalujemy biblioteki w kontenerze.
4. **COPY app** – kopujemy kod aplikacji (tylko to, co potrzebne).
5. **CMD uvicorn ...** – kontener po uruchomieniu startuje serwer FastAPI.

--host 0.0.0.0 jest kluczowe: inaczej serwer nasłuchiwał tylko lokalnie w kontenerze i Windows nie zobaczył portu.

### 7.4. Budowa obrazu

Zbuduj obraz. W katalogu backend/:

```
docker build -t fastapi-jwt .
```

Docker:

- czyta Dockerfile,
- pobiera bazowy obraz Pythona,
- instaluje zależności,

- kopiuje kod,
- zapisuje wynik jako lokalny obraz o nazwie **fastapi-jwt**.

## 8. Uruchomienie i testowanie w Dockerze

Zbliżamy się do końca. Teraz naszym celem jest:

- uruchomić kontener z API,
- przetestować logowanie i endpoint chroniony,
- zweryfikować, że działa identycznie jak lokalnie.

### 8.1. Uruchom kontener

Uruchom API w kontenerze

```
docker run --rm -p 8000:8000 --name fastapi-jwt-api fastapi-jwt
```

- **--rm** usuwa kontener po zakończeniu (czyściej na labach).
- **-p 8000:8000** mapuje port: **Windows/host:8000** na **kontener:8000**
- **--name** nadaje nazwę, żeby łatwo zatrzymać/oglądać logi.

Po uruchomieniu powinieneś zobaczyć w terminalu logi uvicorn (serwer działa).

### 8.2. Test w przeglądarce (Swagger)

Wejdź na Swagger UI - w przeglądarce (Windows):

```
http://127.0.0.1:8000/docs
```

Przeglądarka łączy się z portem 8000 na hoście, który jest przekierowany do portu 8000 kontenera.

### 8.3. Test logowania (JWT)

Przetestuj `/login`. W Swagger:

- wybierz **POST /login**
- Kliknij Try it out
- Wpisz:
  - username: **admin**
  - password: **admin123**
- Kliknij Execute

Odpowiedź powinna zawierać:

```
{  
    "access_token": "...",  
    "token_type": "bearer"  
}
```

API w kontenerze sprawdza dane w hardkodowanej bazie, generuje JWT podpisany kluczem SECRET\_KEY, i finalnie zwraca token.

#### 8.4. Test endpointu chronionego

Autoryzuj się w Swagger

1. Kliknij **Authorize** (góra po prawej w Swagger UI)
2. Wklej:

```
Bearer <TWOJ_TOKEN>
```

3. Kliknij **Authorize** i zamknij okno.

Swagger od tej pory dodaje nagłówek:

```
Authorization: Bearer <token>
```

do requestów.

Przetestuj GET /me

- wybierz GET /me
- Execute

Oczekiwane:

```
{  
    "username": "admin",  
    "role": "admin"  
}
```

FastAPI wyciąga token z nagłówka, weryfikuje podpis i datę ważności, i zwraca dane użytkownika.

#### 8.5. Test negatywny

Sprawdź, że bez tokenu jest 401

1. Kliknij ponownie Authorize
2. Kliknij Logout (lub usuń token)
3. Uruchom GET /me

Oczekiwane:

- kod HTTP: 401 Unauthorized

API poprawnie blokuje dostęp bez JWT – i tak miało być, to jest sens autoryzacji.

#### 8.6. Zarządzanie kontenerem

Zobacz działające kontenery w drugim terminalu:

```
docker ps
```

Podejrzyj logi

```
docker logs fastapi-jwt-api
```

Zatrzymaj kontener - w terminalu gdzie działa kontener: **Ctrl+C** albo:

```
docker stop fastapi-jwt-api
```

- `docker ps` pokazuje procesy/kontenery.
- `docker logs` pomaga, gdy „coś nie działa”.
- `docker stop` kończy kontener.

## 9. Zakończenie

Laboratorium kończy się sukcesem jak macie działający docker z backendem, i pełne środowisko uruchomieniowe. Następnym krokiem będzie wprowadzenie do Angulara.