Języki Skryptowe

Kierunek: Cyberbezpieczeństwo – Studia I stopnia, semestr III

Główny cel przedmiotu:
Zapoznanie studentów z praktycznym zastosowaniem języków skryptowych w kontekście cyberbezpieczeństwa, automatyzacji zadań administracyjnych, przetwarzania danych oraz tworzenia bezpiecznych aplikacji webowych.

Prowadzący:
mgr inż. Mirosław Ossysek

Wprowadzenie do bezpieczeństwa aplikacji webowych

⭐ **Co to znaczy „bezpieczna aplikacja webowa”?**

Bezpieczna aplikacja webowa – aplikacja, która:

- ✓ chroni dane użytkownika,
- ✓ zapobiega nieautoryzowanemu dostępowi,
- ✓ jest odporna na najczęstsze ataki z Internetu,
- ✓ zachowuje prawidłowe działanie nawet przy błędnych wejściach,
- ✓ zabezpiecza logowanie, sesje, API oraz konfigurację.

Dlaczego to ważne?

- ➢ 90% współczesnych ataków dotyczy aplikacji webowych.
- ➢ Najłatwiej zaatakować właśnie warstwę HTTP / API.
- ➢ Flawless code ≠ secure code.

👉 **Flawless code rozwiązuje problem funkcjonalności.**
👉 **Secure code rozwiązuje problem bezpieczeństwa.**
To dwa różne światy. Programista musi myśleć o obu.

⭐ **Modele zagrożeń – co atakuje aplikację?**

Aplikacja webowa może być atakowana przez:

1. Przeglądarkę użytkownika (front-end)

- ❑ XSS
- ❑ CSRF
- ❑ manipulacja formularzami
- ❑ wykradanie cookies

2. Backend / serwer aplikacji

- ❑ SQL Injection
- ❑ Command Injection
- ❑ Path Traversal
- ❑ Brak walidacji danych

3. Infrastrukturę / konfigurację

- ❑ Debug mode
- ❑ Wyciek kluczy API
- ❑ Brak HTTPS
- ❑ Zależności podatne na ataki

Cel wykładu: zrozumieć każdy z tych typów zagrożeń i pokazać, jak *Flask i Python* pomagają im przeciwdziałać.

OWASP Top 10 – fundament bezpieczeństwa

OWASP (Open Web Application Security Project) powstał w **2001 roku** jako otwarty, globalny projekt non-profit zajmujący się poprawą bezpieczeństwa aplikacji. Obecnie działa w ponad **100 krajach**, zrzesza tysiące ekspertów i ma status najbardziej uznawanego autorytetu w dziedzinie bezpieczeństwa webowego.

OWASP ma ogromny wpływ na **praktykę webdevelopmentu** – większość firm, zespołów i audytorów opiera swoje procesy bezpieczeństwa właśnie na OWASP Top 10.

Standard ten jest wykorzystywany w pentestach, audytach bezpieczeństwa aplikacji, wymaganiach projektowych oraz przepisach regulacyjnych.

Projekt OWASP nie ogranicza się tylko do listy Top 10 – tworzy również:

- ✓ **cheat sheety** pokazujące dobre praktyki,
- ✓ **narzędzia** takie jak OWASP ZAP do testów bezpieczeństwa,
- ✓ **standardy** jak Application Security Verification Standard (ASVS),
- ✓ **materiały edukacyjne** i konferencje,
- ✓ **projekty dla API, IoT i mobilnych aplikacji**.

Najważniejsze grupy zagrożeń:

1) **Broken Access Control** – błędna kontrola dostępu
2) **Cryptographic Failures** – złe przechowywanie danych
3) **Injection** – wstrzyknięcia (SQL, command, LDAP...)
4) **Insecure Design** – słabe projektowanie aplikacji
5) **Security Misconfiguration** – błędna konfiguracja
6) **Vulnerable Components** – niebezpieczne zależności
7) **Identification & Authentication Failures**
8) **Software and Data Integrity Failures**
9) **Logging & Monitoring Failures**

OWASP kształtuje sposób, w jaki projektujemy, tworzymy i testujemy aplikacje webowe, a jego wytyczne są traktowane jako obowiązująca baza wiedzy w inżynierii oprogramowania.

Aplikacje tradycyjne vs. SPA

⭐ **Czym różni się aplikacja tradycyjna od SPA?**

🟦 **Aplikacja tradycyjna (Flask + HTML szablony):**
- Backend generuje HTML.
- Formularze wysyłane metodą POST.
- CSRF chronione przez Flask-WTF.
- Walidacja często w backendzie.
- Użytkownik przeładowuje stronę.

🟦 **SPA – Single Page Application:**
- Frontend generuje interfejs w JS.
- Backend → REST API (JSON).
- Walidacja danych w JS *i* Pythonie.
- CSRF wymaga innego mechanizmu niż Flask-WTF.
- JWT zamiast sesji albo w połączeniu z sesjami.

Wniosek: wiele mechanizmów bezpieczeństwa działa inaczej w aplikacjach SPA – i to pokażemy w odpowiednich punktach.

⭐ **Atak – a podatność – a exploit**

Wyjaśnienie ważnych pojęć:

1. Atak (attack)
Aktywność mająca na celu kompromitację aplikacji.

2. Podatność (vulnerability)
Błąd w aplikacji, który można wykorzystać.

3. Exploit
Konkretny kod lub metoda wykorzystywania podatności.

Przykład:
- Podatność: brak walidacji w parametrach SQL
- Attack vector: złośliwy adres URL
- Exploit: `';DROP TABLE users;--`

Exploit można wstrzyknąć wszędzie tam, gdzie aplikacja nie waliduje lub niewłaściwie przetwarza dane wejściowe: SQL, komendy systemowe, pliki, API, parametry URL, ciasteczka.

I. Tożsamość użytkownika, logowanie i kontrola dostępu

⭐ Dlaczego tożsamość użytkownika jest kluczowa?

Tożsamość użytkownika = **informacja o tym, kim jest użytkownik i jakie ma uprawnienia**.

Dlaczego to ważne:

- ➢ większość ataków dotyczy **obejścia logowania** lub **eskalacji uprawnień**,
- ➢ kontrola dostępu chroni **dane**, **funkcje administracyjne** i **operacje krytyczne**,
- ➢ błędy w logowaniu skutkują przejęciem kont użytkowników.

W tym module omówimy:

- ✓ Uwierzytelnianie
- ✓ Sesje i cookies
- ✓ Kontrolę dostępu (autoryzacja)
- ✓ Wsparcie Flaska i ograniczenia

⭐ Uwierzytelnianie (authentication) – definicja

→ proces potwierdzania, że użytkownik jest tym, za kogo się podaje.

Przykłady metod:

- ➢ hasło (najpopularniejsze)
- ➢ token (np. JWT)
- ➢ klucz SSH (np. w paramiko)
- ➢ certyfikaty
- ➢ OAuth / logowanie zewnętrzne

⭐ Autoryzacja (authorization) – definicja

→ proces sprawdzania, do czego użytkownik ma prawo po pomyślnym uwierzytelnieniu.

Przykłady mechanizmów autoryzacji:

- ➢ **role** (np. *user*, *admin*, *moderator*)
- ➢ **poziomy dostępu** (np. *read*, *write*, *delete*)
- ➢ **autoryzacja do zasobu** (np. użytkownik może edytować tylko *swoje* dane)
- ➢ **token z uprawnieniami** (np. JWT z polem `role` lub `scope`)
- ➢ **ACL (Access Control Lists)** – listy uprawnień dla obiektów
- ➢ **RBAC (Role-Based Access Control)** – uprawnienia wynikają z roli

Zrządzanie stanem i mechanizm sesji w Flask

- **Czym jest sesja?**
  Sesja to mechanizm, który pozwala serwerowi zapamiętać dane o użytkowniku między kolejnymi żądaniami HTTP, mimo że sam protokół HTTP jest bezstanowy.

- **Do czego jest wykorzystywana?**
  Sesje są wykorzystywane do przechowywania informacji o stanie użytkownika, takich jak dane logowania, preferencje czy zawartość koszyka, tak aby użytkownik nie musiał podawać ich za każdym razem od nowa.

- **Jak zakłada się sesję?**
  W Flasku sesję zakłada się, zapisując dane do obiektu `session`. Można to zrobić po prostu przypisując wartości pod kluczami, na przykład `session['user'] = ,Janek'.`

- **Jak czyści się sesję?**
  Żeby usunąć pojedynczy klucz z sesji, wystarczy użyć `session.pop('user').` Jeśli chcesz wyczyścić całą sesję, możesz użyć `session.clear().`

- **Jak ustawia się czas wygaśnięcia sesji?**
  Czas wygaśnięcia sesji można ustawić, konfigurując aplikację tak, aby ciasteczko sesyjne miało określony czas życia. Można to zrobić na przykład ustawiając `PERMANENT_SESSION_LIFETIME` w konfiguracji Flaska

- **Jak Flask przechowuje sesję?**

  Flask domyślnie przechowuje sesję w **zaszyfrowanym i podpisanym cookie** po stronie przeglądarki, korzystając z SECRET_KEY do weryfikacji integralności danych. Aby przechowywać dane sesji **na serwerze**, można użyć rozszerzenia **Flask-Session**, które umożliwia zapis sesji w Redisie, bazie danych lub plikach, a w cookie trzyma jedynie identyfikator sesji.

Podsumowując, sesja w Flasku to po prostu sposób na przechowywanie danych użytkownika między żądaniami.

Możesz ją założyć, dodając dane do sesji, usunąć je, gdy są już niepotrzebne, i określić, po jakim czasie sesja ma wygasnąć.

Ciasteczka (HTTP Cookies)

- **Czym są ciasteczka (cookies)?**
  Ciasteczka to małe fragmenty danych (klucz-wartość), które przeglądarka przechowuje na komputerze użytkownika. Służą do przechowywania pewnych informacji związanych z daną stroną internetową, na przykład preferencji użytkownika, stanu logowania czy zawartości koszyka.
- **Do czego są wykorzystywane?**
  Ciasteczka są wykorzystywane do śledzenia stanu użytkownika, zapamiętywania jego preferencji, utrzymywania sesji, a także do celów analitycznych czy reklamowych. Dzięki nim strona "pamięta" użytkownika między kolejnymi wizytami.
- **Jak się je zakłada i kto je zakłada?**
  Ciasteczka są tworzone przez serwer, który wysyła je w nagłówku HTTP (nagłówek `Set-Cookie`) w odpowiedzi na żądanie przeglądarki. Przeglądarka przechowuje te ciasteczka i automatycznie odsyła je do serwera przy każdym kolejnym żądaniu do tej samej domeny.
- **Jak wygląda mechanizm wymiany ciasteczek?**
  Kiedy serwer chce ustawić ciasteczko, wysyła je w odpowiedzi HTTP. Przeglądarka zapisuje to ciasteczko i przy kolejnych żądaniach do tego samego serwera automatycznie dołącza nagłówek `Cookie`, zawierający te zapisane informacje. W ten sposób serwer może "rozpoznać" użytkownika i przywrócić jego stan.

Podsumowując, ciasteczka to małe dane zapisywane przez serwer i przechowywane przez przeglądarkę, które pozwalają na utrzymanie pewnych informacji między kolejnymi wizytami na stronie.
Serwer je tworzy, przeglądarka przechowuje i odsyła, a cała wymiana odbywa się automatycznie przy kolejnych żądaniach HTTP.

Ciasteczka, sesje – ustawienia bezpieczeństwa

- **Ustawienie klucza SECRET_KEY**:
  Najważniejszą rzeczą jest ustawienie bezpiecznego klucza `SECRET_KEY`, który Flask wykorzystuje do podpisywania ciasteczek sesyjnych. Powinien to być losowy, trudny do odgadnięcia ciąg znaków, na przykład:
  ```
  SECRET_KEY = 'super_tajny_klucz_z_losowymi_znakami'
  ```

- **Konfiguracja sesji**:
  Możesz ustawić, jak długo sesja ma być ważna, ustawiając `PERMANENT_SESSION_LIFETIME`. Na przykład:
  ```
  from datetime import timedelta

  PERMANENT_SESSION_LIFETIME = timedelta(minutes=30)
  ```
  ✓ To sprawi, że sesja wygaśnie po 30 minutach bezczynności.

- **Bezpieczeństwo ciasteczek sesyjnych**:
  Warto włączyć atrybuty `SESSION_COOKIE_HTTPONLY` i `SESSION_COOKIE_SECURE`, żeby ciasteczko sesyjne było dostępne tylko przez HTTP i wysyłane tylko po bezpiecznym połączeniu HTTPS:
  ```
  SESSION_COOKIE_HTTPONLY = True
  SESSION_COOKIE_SECURE = True
  ```

- **Inne ustawienia cookies**:
  Jeśli używasz innych ciasteczek, które nie są związane z sesją, możesz ustawić im atrybuty Secure i SameSite, na przykład:
  ```
  SESSION_COOKIE_SAMESITE = 'Lax'  # albo 'Strict' jeśli chcesz bardziej restrykcyjnie
  ```

Podsumowując, w pliku config.py ustawiasz bezpieczny klucz SECRET_KEY, określasz czas życia sesji, a także włączasz bezpieczne opcje dla ciasteczek, żeby dane były dobrze chronione.
Dzięki temu sesje i ciasteczka są konfigurowane w bezpieczny sposób.

Sesja i ciasteczka - przykład

1

```python
from flask import Flask, session, request, make_response
from datetime import datetime, timedelta

class Config:
    # klucz do podpisywania ciasteczek sesyjnych
    SECRET_KEY = "twoj_super_tajny_klucz"
    # czas życia sesji (np. 30 minut)
    PERMANENT_SESSION_LIFETIME = timedelta(minutes=30)
    SESSION_COOKIE_SECURE = True # Ciasteczko dostępne tylko przez HTTPS
    SESSION_COOKIE_HTTPONLY = True # Niedostępne dla JavaScript
    SESSION_COOKIE_SAMESITE = "Lax" # Ochrona przed CSRF; może być też "Strict" lub "None"

app = Flask(__name__)
app.config.from_object(Config) # Ustawienia z klasy Config

@app.route("/")
def index():
    # ---- 1. Zapis do sesji ----
    session.permanent = True
    session["username"] = "Janek"
    # ---- 2. Zwykłe ciasteczko (bez podpisu) ----
    resp = make_response("Sesja ustawiona + zwykłe ciasteczko zapisane!")
    resp.set_cookie(
        "ulubiony_kolor",
        "zielony",
        max_age=60 * 60 * 24 * 7,   # czas życia: 7 dni, po którym przeglądarka usunie ciasteczko
        secure=True,                # tylko HTTPS
        httponly=False,             # można odczytać w JS aby pokazać kolor
        expires=datetime.date.today() + timedelta(days=7), # data wygaśnięcia
        #lub użyć max_age zamiast expires: max_age=60*60*24*7, #Czas życia: 7 dni
        #jeśli nie podamy expires/max_age, ciasteczko będzie usunięte po zamknięciu przeglądarki
    )
    # Zwracamy odpowiedź z ustawionym własnym ciasteczkiem, które nie jest podpisane
    # oraz sesją zapisaną w ciasteczku sesyjnym, które jest podpisane z użyciem SECRET_KEY
    return resp
```

2

```python
...
@app.route("/odczytaj")
def odczytaj():
    # ---- 1. Odczyt danych z sesji ----
    username = session.get("username", "brak sesji")

    # ---- 2. Odczyt zwykłego ciasteczka ----
    kolor = request.cookies.get("ulubiony_kolor", "brak ciasteczka")

    return f"Sesja: {username}, ulubiony kolor: {kolor}"

@app.route("/wyloguj")
def wyloguj():
    # Usuwamy dane z sesji
    session.pop("username", None)
    return "Wylogowano, sesja usunięta."

if __name__ == "__main__":
    app.run(debug=True)
```

`httponly=False` – dostępne w JavaScript

- to ciasteczko przechowuje *preferencję użytkownika,*
- takie preferencje czytamy w JS: `document.cookie`

Wsparcie Flask dla kontroli dostępu

Flask jest mikro-frameworkiem, co oznacza, że w swojej podstawowej formie nie posiada wbudowanego mechanizmu zarządzania użytkownikami ani logowania.

Aby zaimplementować te funkcjonalności, standardem w ekosystemie Flaska jest użycie rozszerzenia **Flask-Login** oraz narzędzi bezpieczeństwa z biblioteki **Werkzeug** (która jest częścią rdzenia Flaska).

Kluczowe moduły i biblioteki:

`Flask-Login`:

- **Opis**: Najważniejsze rozszerzenie. Zarządza sesją użytkownika (zapamiętuje, że użytkownik jest zalogowany po przeładowaniu strony). Dostarcza helpery do logowania, wylogowywania i sprawdzania statusu użytkownika.

- **Kluczowe elementy**:
  - **`UserMixin`**: Klasa bazowa dla modelu użytkownika, implementująca wymagane metody (np. is_authenticated).
  - **`login_user`**(): Funkcja logująca użytkownika (tworzy sesję).
  - **`logout_user()`**: Funkcja wylogowująca (czyści sesję).
  - **`@login_required`**: Dekorator chroniący widoki (endpointy) przed niezalogowanymi użytkownikami.
  - **`current_user`**: Proxy do obiektu aktualnie zalogowanego użytkownika.

`Werkzeug.security:`

- **Opis**: Moduł wbudowany we Flaska, służący do bezpiecznego haszowania haseł. **Nigdy nie przechowujemy haseł otwartym tekstem.**
- **Kluczowe funkcje**:
  - **`generate_password_hash():`** Tworzy bezpieczny skrót hasła.
  - **`check_password_hash():`** Porównuje podane hasło ze skrótem w bazie.

`Flask-SQLAlchemy` (Opcjonalnie, ale standardowo):

- Używany do przechowywania danych użytkowników w bazie danych.

Implementacja kontroli dostępu we Flask

1. **Konfiguracja modelu użytkownika (Tożsamość)**
Model użytkownika musi implementować kilka metod wymaganych przez `Flask-Login` (np. czy konto jest aktywne). Najłatwiej to zrobić dziedzicząc po `UserMixin`.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from config import Config
from flask_login import UserMixin
from werkzeug.security import generate_password_hash, check_password_hash

app = Flask(__name__)
app.config.from_object(Config)

db = SQLAlchemy(app)

# Model Użytkownika
class User(UserMixin, db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(100), unique=True, nullable=False)
    password_hash = db.Column(db.String(200), nullable=False)
    role = db.Column(db.String(20), default='user') # Do kontroli dostęp

    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

    def check_password(self, password):
        return check_password_hash(self.password_hash, password)
```

Logowanie

2. Konfiguracja Flask-Login

Należy zainicjować `LoginManager` i zdefiniować funkcję `user_loader`, która mówi Flaskowi, jak pobrać użytkownika z bazy na podstawie jego ID zapisanego w sesji (ciasteczku).

```
from flask_login import LoginManager

login_manager = LoginManager()
login_manager.init_app(app)

# Gdzie przekierować niezalogowanego użytkownika
login_manager.login_view = 'login’

# Funkcja ładująca użytkownika z bazy (potrzebna dla sesji)
@login_manager.user_loader
def load_user(user_id):
    # Flask-Login wysyła ID jako string, więc konwertujemy na int
    return User.query.get(int(user_id))
```

3. Logowanie (Uwierzytelnianie)

Proces logowania polega na:

1. Pobraniu danych z formularza.
2. Znalezieniu użytkownika w bazie.
3. Sprawdzeniu poprawności hasła (check_password).
4. Utworzeniu sesji za pomocą login_user.

```
from flask import request, redirect, url_for, flash, render_template
from flask_login import login_user, logout_user, current_user

@app.route('/login', methods=['GET', 'POST'])
def login():
    if current_user.is_authenticated:
        return redirect(url_for('dashboard'))

    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        user = User.query.filter_by(username=username).first()

        # Sprawdzamy czy user istnieje i czy hasło jest poprawne
        if user and user.check_password(password):
            login_user(user) # Tutaj dzieje się magia sesji
            return redirect(url_for('dashboard'))
        else:
            flash('Błędna nazwa użytkownika lub hasło')
    return render_template('login.html')

@app.route('/logout')
def logout():
    logout_user() # Czyści sesję
    return redirect(url_for('login'))
```

Kontrola dostępu

4. Kontrola dostępu (Autoryzacja podstawowa)

Aby zablokować dostęp do widoku dla osób niezalogowanych, używamy dekoratora `@login_required`.

```
from flask_login import login_required

@app.route('/dashboard')
@login_required # Tylko zalogowani wejdą tutaj
def dashboard():
    return f'Witaj, {current_user.username}! To jest panel użytkownika.’
```

5. Zaawansowana kontrola dostępu (Role)

`Flask-Login` sprawdza tylko, *czy* ktoś jest zalogowany. Jeśli chcesz sprawdzić, *kim* jest (np. Admin vs User), musisz napisać własny dekorator lub sprawdzić to w widoku.

Przykład własnego dekoratora dla roli Admina:

```
from functools import wraps
from flask import abort

def admin_required(f):
    @wraps(f) # Dekorator zachowujący informacje o funkcji oryginalnej f
    def decorated_function(*args, **kwargs):
        # Najpierw sprawdzamy czy zalogowany, potem czy ma rolę admin
        if not current_user.is_authenticated or current_user.role != 'admin':
            abort(403) # Błąd 403 Forbidden
        return f(*args, **kwargs)
    return decorated_function

@app.route('/admin_panel')
@login_required # Najpierw wymuś logowanie
@admin_required # Potem wymuś rolę
def admin_panel():
    # Dzięki @wraps zachowujemy nazwę i docstring oryginalnej funkcji admin_panel
    return "To widzi tylko Administrator."
```

Brute-force i zabezpieczenie logowania

⭐ **Atak brute-force = wielokrotne zgadywanie hasła.**

➢ Minimalne zabezpieczenia:
✓ **Limit prób logowania** – ogranicza liczbę możliwych nieudanych logowań w krótkim czasie, utrudniając ataki brute-force.
✓ **Czasowe blokowanie konta** – po przekroczeniu ustalonego limitu nieudanych prób konto zostaje na pewien czas zablokowane, co znacząco spowalnia atakującego.
✓ **Użycie Flask-Limiter** – automatycznie nakłada ograniczenia na liczbę żądań HTTP na endpoint logowania, co chroni serwer przed masowymi próbami zgadywania hasła.
✓ **Logowanie nieudanych prób** – zapisuje każde nieudane logowanie w logach bezpieczeństwa, umożliwiając wykrycie podejrzanych działań i generowanie alertów.

⭐ **Co to jest Flask-Limiter?**
Flask-Limiter to rozszerzenie Flaska, które pozwala łatwo wdrażać *rate limiting*, czyli ograniczenia liczby żądań do danego endpointu w określonym czasie.

Zapobiega to:
✓ atakom brute-force na logowanie,
✓ przeciążaniu serwera,
✓ skanowaniu API i automatycznym botom.

⭐ **Jak działa dekorator `@limiter.limit("5 per minute")`?**
Oznacza to:
„Pozwól wykonać ten endpoint maksymalnie 5 razy w ciągu jednej minuty z jednego IP / użytkownika (w zależności od konfiguracji).”

```python
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

limiter = Limiter(get_remote_address, app=app)

@limiter.limit("5 per minute")
@app.post("/login")
def login():
    ...
```

Identyfikacja klienta w *rate limiting*

🔹 1. Po co jest identyfikacja klienta?

Rate limiting musi wiedzieć, *kogo ograniczać* – czyli dla każdego klienta prowadzi oddzielny licznik żądań.
Identyfikator klienta może być:

➢ adresem IP, ID zalogowanego użytkownika, tokenem API, albo innym, dowolnym kluczem.

🔹 2. Co robi `get_remote_address`?

`get_remote_address` zwraca **adres IP klienta** na podstawie obiektu `request`
(czyli `request.remote_addr`).

Użycie:

```
limiter = Limiter(get_remote_address, app=app)
```

oznacza: *„Ograniczaj żądania **per IP** — każdy adres IP ma własny limit."*

🔹 3. Własna funkcja identyfikująca użytkownika

Można napisać własną funkcję, która zwróci dowolny identyfikator

Przykład – **limit per użytkownik (gdy zalogowany), inaczej per IP**:

```
from flask_login import current_user
from flask import request

def user_or_ip():
    if current_user.is_authenticated:
        return str(current_user.id)   # limit dla konkretnego usera
    return request.remote_addr        # limit dla IP, jeśli nie zalogowany

limiter = Limiter(key_func=user_or_ip, app=app)
```

Co to daje?

✓ zalogowany użytkownik ma swój własny limit, niezależny od IP,
✓ niezalogowany użytkownik jest limitowany per IP.

⭐ Co zawiera obiekt `request`?

`request` to struktura, która zawiera m.in.:

✔ Dane przesłane przez użytkownika

- `request.args` – parametry w URL (`/search?query=abc`)
- `request.form` – dane z formularza POST
- `request.json` – dane JSON wysłane przez SPA
- `request.files` – wysłane pliki

✔ Informacje o połączeniu

- `request.remote_addr` – adres IP klienta
- `request.method` – metoda (GET, POST, PUT...)
- `request.headers` – nagłówki HTTP
- `request.cookies` – ciasteczka wysłane przez klienta

✔ Ścieżka i URL

- `request.path`
- `request.url`

Inne metody uwierzytelniania

🔐 1. Klucze SSH

- Metoda logowania **bez hasła**, oparta o **parę kluczy kryptograficznych**:
  - **klucz prywatny** (private key) – trzymany na komputerze użytkownika,
  - **klucz publiczny** – zapisany na serwerze.
- Stosowana głównie do logowania na serwer (np. przez `ssh`, `paramiko`).
- Bardzo bezpieczna, trudna do złamania metodą brute-force.

🔐 2. Certyfikaty (TLS/SSL)

- Certyfikat X.509 to dokument kryptograficzny używany do:
  - potwierdzania tożsamości serwera (HTTPS),
  - szyfrowania połączeń,
  - w niektórych systemach – logowania użytkowników (tzw. client certificates).
- Uwierzytelnianie certyfikatem jest znacznie silniejsze niż hasło.

Przykład użycia:

- ✓ przeglądarka weryfikuje certyfikat serwera HTTPS,
- ✓ klient VPN loguje się za pomocą certyfikatu.

🔐 3. OAuth 2.0 (logowanie z zewnętrznego dostawcy)

- Protokół umożliwiający logowanie przez:
  - Google,
  - Facebook,
  - GitHub,
  - Microsoft.
- W OAuth aplikacja **nie dostaje hasła użytkownika** – otrzymuje jedynie **token dostępu**.

Zalety:

- ✓ użytkownik nie musi tworzyć nowego konta,
- ✓ bezpieczeństwo hasła po stronie dostawcy,
- ✓ łatwe zarządzanie wylogowaniem i autoryzacjami.

W Flasku:

- biblioteki: `Authlib`, `Flask-Dance`.

W praktyce webowej hasła i sesje są najczęstsze,
SPA często korzysta z JWT,

a duże aplikacje i systemy korporacyjne wykorzystują także klucze SSH, certyfikaty i OAuth 2.0.

Tożsamość użytkownika, logowanie i kontrola dostępu - Podsumowanie

Zagadnienia:

- Uwierzytelnianie → potwierdzanie tożsamości
- Sesje → przechowywanie stanu użytkownika
- Cookies → muszą być odpowiednio ustawione
- Hashowanie → obowiązkowe
- Kontrola dostępu → backend zawsze rozstrzyga
- SPA wymaga tokenów + backendowej autoryzacji
- Flask część rzeczy robi automatycznie, ale nie wszystkie!

⭐ **Rola Flaska i Pythona w bezpieczeństwie**

Co Flask daje automatycznie:

- ✓ escaper XSS (Jinja2)
- ✓ HttpOnly dla cookies
- ✓ integrację z Werkzeug (hashowanie haseł)
- ✓ strukturalne wsparcie dla sesji
- ✓ integrację z blueprintami (separacja modułów)

⭐ **Rola Flaska i Pythona w bezpieczeństwie**

Czego Flask NIE robi automatycznie:

- nie waliduje JSON
- nie chroni przed SQL Injection (robi to SQLAlchemy)
- nie zapewnia CSRF w SPA
- nie wymusza HTTPS
- nie wykrywa ataków ani nie monitoruje ich
- nie zapewnia rate limiting
- nie zarządza rolami użytkowników

II. Walidacja danych i ataki na wejście

⭐ Dlaczego walidacja danych jest kluczowa?

Walidacja danych to proces sprawdzenia, czy dane wejściowe są poprawne, spójne i bezpieczne, zanim trafią do aplikacji.

Brak walidacji powoduje najpoważniejsze rodzaje ataków:

- SQL Injection
- Command Injection
- Path Traversal
- przepełnienia, DoS
- błędy w działaniu aplikacji

Zasada nr 1 bezpieczeństwa:
➡️ *Nigdy nie ufamy danym od użytkownika – nawet jeśli wyglądają poprawnie.*

⭐ Dwa modele wejścia danych w aplikacji webowej

🟦 **Aplikacja tradycyjna (Flask + HTML)**
- formularze HTML wysyłane POST-em
- backend używa **Flask-WTF + WTForms**
- walidacja odbywa się *po stronie backendu*
- CSRF automatycznie obsługiwane przez Flask-WTF

🟥 **Aplikacja SPA (frontend JS)**
- dane są wysyłane jako **JSON**
- walidacja w dwóch miejscach:
  - w JS (UI),
  - oraz w backendzie (np. **marshmallow**)
- Flask-WTF **nie działa** z JSON
- CSRF wymaga innego mechanizmu

SQL Injection – jak wyeliminować ryzyko

⭐ SQL Injection: błędny kod

❌ **NIEBEZPIECZNY kod z formatowaniem stringa:**

```
name = request.args.get("name")
query = f"SELECT * FROM users WHERE name = '{name}'"
db.session.execute(query)
```

Atakujący wysyła:

```
/users?name=' OR '1'=,1
```

I dostaje *wszystkich użytkowników*.

⭐ Dlaczego ORM i parametryzacja eliminują SQL Injection?

ORM (np. SQLAlchemy) i parametryzowane zapytania:
✓ NIE łączą danych i SQL w jeden string,
✓ przekazują dane **osobno**, jako „parametry”,
✓ baza traktuje te parametry **jak dane**, a nie jak kod SQL.

⭐ SQL Injection: bezpieczny kod w SQLAlchemy (ORM)

✔ SQLAlchemy ORM automatycznie **parametryzuje** zapytania.

```
name = request.args.get("name")
user = User.query.filter_by(name=name).first()
```

✔ Bezpieczne zapytanie SQL z parametrami:

```
from sqlalchemy import text

db.session.execute(
    text("SELECT * FROM users WHERE name = :name"),
    {"name": name}
)
```

Co jest robione automatycznie?
→ ORM SQLAlchemy **chroni przed SQL Injection**.

Walidacja danych wejściowych

⭐ Walidacja danych w aplikacji tradycyjnej (Flask-WTF + WTForms)

Flask-WTF integruje:
✓ **WTForms** – definicje formularzy i walidacji,
✓ **CSRF** – automatyczny token,
✓ obsługę `form.validate_on_submit()`

Przykład **formularza**:

```
class RegisterForm(FlaskForm):
    username = StringField("Username", validators=[DataRequired(), Length(min=3)])
    age = IntegerField("Age", validators=[NumberRange(min=1, max=120)])
```

Przykład **widoku**:

```
@app.route("/register", methods=["GET", "POST"])
def register():
    form = RegisterForm()
    if form.validate_on_submit():
        # dane są zwalidowane
        ...
    return render_template("register.html", form=form)
```

✔️ Walidacja odbywa się **automatycznie**.
❌ Nie działa w SPA.

⭐ Walidacja danych w SPA (JSON + backend)

Flask-WTF nie działa z JSON – trzeba skorzystać z walidatora JSON (`Pydantic` lub `Marshmallow`).

Przykład w **marshmallow**:

```
from marshmallow import Schema, fields
...
class UserSchema(Schema):
    username = fields.Str(required=True)
    age = fields.Int(required=True, validate=lambda x: 0 < x < 120)

schema = UserSchema()
data = schema.load(request.get_json())
```

W SPA robimy **walidację podwójną**:
✓ w JavaScript (dla UX),
✓ *obowiązkowo* w backendzie.

⭐ Walidacja w JS (SPA)

Frontend **nigdy nie jest zaufany**, ale robi pierwszą warstwę walidacji:

```
if (username.length < 3) {
    alert("Username too short!");
    return;
}
```

➡️ Backend waliduje *ponownie* – zawsze.

Upload plików

⭐ Upload plików: ryzyka bezpieczeństwa

Zagrożenia:

- **Path traversal** → zapis pliku poza katalogiem
- wgranie pliku jako malware
- nadpisanie plików
- zbyt duży plik → DoS

Przykład path traversal:

```
filename = "../../../../etc/passwd"
```

⭐ Upload plików: błędny kod

❌ Niebezpieczne:

```
file = request.files["file"]
file.save("/uploads/" + file.filename)
```

Ryzyka:

- path traversal
- nadpisywanie plików
- ataki przez unicode

⭐ Upload plików: poprawny kod

✔ Używanie `secure_filename`:

- Zamienia nazwę pliku na bezpieczną wersję, usuwając nielegalne znaki, normalizując ścieżkę i uniemożliwiając ataki typu path traversal (np. `../../etc/passwd`).

Przykład:

```
from pathlib import Path
from werkzeug.utils import secure_filename

file = request.files["file"]
filename = secure_filename(file.filename)

upload_dir = Path(app.config["UPLOAD_FOLDER"])
filepath = upload_dir / filename

file.save(filepath)
```

✔ Zaleca się także:

```
# Zmiana konfiguracji aplikacji Flask:
# maksymalny rozmiar przesyłanego pliku
app.config["MAX_CONTENT_LENGTH"] = 2 * 1024 * 1024  # limit 2 MB
```

Command Injection

⭐ Command Injection (definicja)

Command Injection polega na wykonaniu przez aplikację poleceń systemowych z wstrzykniętą treścią od użytkownika.

Skutki:

- usunięcie plików,
- wyciek danych,
- przejęcie systemu.

⭐ Command Injection: błędny kod (subprocess)

❌ Bardzo niebezpieczne:

```
cmd = request.args.get("cmd")
subprocess.run(cmd, shell=True)
```

Atakujący może wysłać:

```
; rm -rf /
```

⭐ Command Injection: poprawny kod

✔ Nigdy nie używaj `shell=True` z wejściem użytkownika.

```
subprocess.run(["ls", "-la"]) # stałe polecenie, bez wejścia użytkownika.
```

✔ Jeśli komenda musi zależeć od użytkownika – stosujemy **whitelist**:

```
allowed = {"status", "info"}

# Sprawdzamy, czy podane polecenie jest na liście dozwolonych
cmd = request.args.get("cmd")

if cmd not in allowed:      # ! blokujemy wszystko, czego nie znamy
    abort(400)

# Mapowanie dozwolonych poleceń na ich rzeczywiste wywołania
if cmd == "status":
    subprocess.run(["systemctl", "status"])   # stałe, znane polecenie
elif cmd == "info":
    subprocess.run(["uname", "-a"])
```

Biblioteka	Ryzyko	Przykład zabezpieczenia
subprocess	Command Injection	brak shell=True, whitelist
paramiko	przechowywanie hasła, słabe klucze	klucze prywatne zamiast haseł
pandas	złośliwe CSV / DoS przez duże pliki	limit rozmiaru, walidacja źródła
psutil	dostęp do zasobów systemu	nie przekazywać danych z zew.

III. Ataki na przeglądarkę: XSS, CSP, CSRF

⭐ Wprowadzenie: ataki na przeglądarkę

Większość ataków w tym obszarze polega na:
- ❖ **wstrzyknięciu szkodliwego JavaScriptu** (XSS),
- ❖ **zmuszeniu przeglądarki do wykonania niechcianej akcji** (CSRF),
- ❖ **błędnej konfiguracji zabezpieczeń przeglądarki** (CSP).

To ataki na **warstwę front-end**, a nie na backend czy bazę.

⭐ Co to jest XSS?

XSS polega na wstrzyknięciu złośliwego JavaScriptu do strony wyświetlanej przez przeglądarkę.
Cel:
- ❖ kradzież cookies,
- ❖ przejęcie sesji,
- ❖ phishing w obrębie strony,
- ❖ przejęcie konta użytkownika,
- ❖ manipulacja interfejsem.

Rodzaje XSS:
- ➢ **Stored XSS** – szkodliwy kod zapisany w bazie,
- ➢ **Reflected XSS** – kod pojawia się w odpowiedzi serwera z parametru URL,
- ➢ **DOM XSS** – JS działa po stronie frontendu.

Ochrona przed XSS

⭐ **TRADYCYJNA aplikacja Flask: autoescape w Jinja2**
Autoescape – automatyczna ochrona przed XSS

Domyślnie `{{ zmienna }}` w Jinja2 stosuje **escape**, czyli zamienia:

- `<` → `<`
- `>` → `>`
- `'` → `&#39;`
- `"` → `&quot;`
- `&` → `&`

Cel: aby kod użytkownika nie wykonał się jako JavaScript.
Na przykład użytkownik wpisuje:

```
<script>alert('XSS')</script>
```

Jinja2 wyrenderuje:

```
&lt;script&gt;alert('XSS')&lt;/script&gt;
```

→ **Skrypt się nie wykona.**

⭐ **Niebezpieczne wyłączenie escape: `|safe`**

❌ Jeśli programista użyje `|safe`, wyłącza ochronę:

```
<p>{{ comment | safe }}</p>    <!-- bardzo niebezpieczne -->
```

Wtedy XSS działa w 100%.

💡 **Reguła:** Nigdy nie używaj `|safe` dla danych pochodzących od użytkownika.

⭐ **SPA (JavaScript): XSS inny niż w Flasku**
W SPA:

- nie używasz Jinja2 → **brak autoescape**,
- szablony są tworzone w JS, często manipulacją DOM lub frameworkiem.

Najczęstszy błąd:

```
document.getElementById("msg").innerHTML = userInput;  // ❌ XSS
```

Bezpieczna wersja:

```
document.getElementById("msg").innerText = userInput; //✔brak interpretacji HTML
```

⭐ **SQL Injection vs XSS**
SQL Injection – atak na backend (bazę danych).
XSS – atak na frontend (przeglądarkę).

Content Security Policy (CSP) w aplikacjach webowych

1. Definicja i cel stosowania

Content Security Policy (CSP) to dodatkowa warstwa bezpieczeństwa implementowana poprzez nagłówek HTTP (Content-Security-Policy), która pomaga w wykrywaniu i łagodzeniu skutków ataków, w szczególności **Cross-Site Scripting (XSS)** oraz ataków polegających na wstrzykiwaniu danych (Data Injection).

Zasada działania opiera się na modelu **białej listy (allowlist)**. Przeglądarka internetowa, otrzymując politykę CSP od serwera, blokuje wszelkie zasoby (skrypty, obrazy, arkusze stylów, połączenia sieciowe), które nie zostały jawnie dozwolone w konfiguracji.

Dzięki temu, nawet jeśli atakującemu uda się wstrzyknąć złośliwy kod <script> do widoku aplikacji (np. przez lukę w walidacji wejścia), przeglądarka odmówi jego wykonania, ponieważ źródło tego skryptu nie będzie zgodne z polityką bezpieczeństwa.

2. Kluczowe dyrektywy CSP

Polityka składa się z zestawu dyrektyw określających dozwolone źródła dla różnych typów zasobów:

- default-src: Dyrektywa domyślna, stosowana dla typów zasobów, które nie mają własnej definicji.
- script-src: Określa dozwolone źródła plików JavaScript (kluczowe dla ochrony przed XSS).
- style-src: Określa dozwolone źródła arkuszy stylów (CSS).
- img-src: Określa dozwolone źródła obrazów.
- connect-src: Określa domeny, z którymi skrypty mogą nawiązywać połączenia (np. przez fetch czy WebSockets). Jest to krytyczne dla warstwy API w aplikacjach hybrydowych.

3. Implementacja w środowisku Flask

W ekosystemie Flaska standardem branżowym do zarządzania nagłówkami bezpieczeństwa jest biblioteka **Flask-Talisman**. Automatyzuje ona proces dodawania nagłówków CSP, HSTS oraz X-Content-Type-Options.

Wyzwania w architekturze hybrydowej

Aplikacje hybrydowe (łączące render_template z dynamicznym JS korzystającym z API) napotykają dwa specyficzne problemy przy wdrażaniu CSP:

- **Skrypty Inline (Inline Scripts):** W szablonach Jinja2 często umieszcza się kod JavaScript bezpośrednio w pliku HTML (np. przekazanie zmiennych z backendu do frontendu). Restrykcyjna polityka CSP (script-src 'self') domyślnie blokuje taki kod.
- **Połączenia do API:** Skrypty frontendowe muszą mieć zezwolenie na łączenie się z endpointami API (często na tej samej domenie, ale czasem zewnętrznymi)

Przykład CSP z Flask-Talisman

```python
from flask import Flask, render_template
from flask_talisman import Talisman
import os

app = Flask(__name__)

# Definicja polityki CSP
csp = {
    'default-src': '\'self\'',
    'script-src': [
        '\'self\'',
        'https://code.jquery.com',  # Zaufane zewnętrzne CDN
        # Ważne: to pozwoli na użycie mechanizmu nonce dla skryptów inline
        # Talisman automatycznie podmieni to na właściwy nonce
    ],
    'style-src': [
        '\'self\'',
        'https://stackpath.bootstrapcdn.com' # Zaufane style (Bootstrap)
    ],
    # connect-src kontroluje, gdzie JS może wysyłać zapytania (API)
    'connect-src': ['\'self\'', 'https://api.partner.com']
}

# Inicjalizacja Talisman z konfiguracją CSP
# content_security_policy_nonce_in=['script-src'] instruuje Talisman,
# aby wygenerował nonce i dodał go do nagłówka dla skryptów.
talisman = Talisman(
    app,
    content_security_policy=csp,
    content_security_policy_nonce_in=['script-src']
)

@app.route('/dashboard')
def dashboard():
    # Renderowanie szablonu. Zmienna 'nonce' jest dostępna w szablonie
    # automatycznie dzięki Talismanowi.
    return render_template('dashboard.html')
```

```html
<!-- dashboard.html -->

<!-- 1. Skrypt zewnętrzny (dozwolony przez 'self' lub whitelistę CDN) -->
<script src="/static/js/app.js"></script>

<!-- 2. Skrypt Inline (wymaga nonce) -->
<!-- Flask-Talisman udostępnia funkcję csp_nonce() -->
<script nonce="{{ csp_nonce() }}">
    // Ten kod wykona się, ponieważ nonce zgadza się z nagłówkiem HTTP.
    const userId = {{ current_user.id }};

    // Próba połączenia z API (musi być zgodna z connect-src)
    fetch('/api/data')
        .then(response => response.json())
        .then(data => console.log(data));
</script>
<!-- 3. Atak XSS (zablokowany) -->
<!-- Jeśli atakujący wstrzyknie ten kod, nie będzie on znał poprawnego nonce. -->
<script>
    alert('Hacked!'); // Przeglądarka zablokuje wykonanie
</script>
```

CSRF – Cross-Site Request Forgery

⭐ **Co to jest CSRF?**
CSRF polega na tym, że atakujący **zmusza zalogowaną przeglądarkę** do wykonania żądania, którego użytkownik nie chciał wykonać.

Przykład:
1. użytkownik zalogowany do banku,
2. wchodzi na stronę z obrazkiem:

```
<img src="https://bank.com/transfer?to=hacker&amount=1000">
```

Przeglądarka automatycznie wysyła cookies → przelew wykonany.

Atak nie wymaga znajomości hasła!
Tylko wykorzystuje **automatyczne wysyłanie ciasteczek**.

⭐ **Aplikacja Flask: Flask-WTF chroni automatycznie**
Każdy formularz HTML generowany przez Flask-WTF:

```
<input type="hidden" name="csrf_token" value="...">
```

A backend wymaga prawidłowego tokenu:

```
if form.validate_on_submit():# ✔automatyczna walidacja CSRF
```

Flask-WTF:
- ✓ generuje CSRF token,
- ✓ przechowuje go w ciasteczku,
- ✓ porównuje każdy request POST/PUT/DELETE,
- ✓ odrzuca żądania bez tokenu.

To ochrona w 100% (dla aplikacji tradycyjnej).

Zabezpieczenie CSRF z `Flask-WTF` (formularze)

`Flask-WTF` to rozszerzenie, które automatycznie generuje tokeny CSRF dla formularzy. To bardziej klasyczne podejście, gdy masz zwykłe formularze HTML, a nie typowe SPA.

W szablonie Jinja2 po prostu wstawiasz `{{ form.hidden_tag() }}`, a `Flask-WTF` automatycznie doda ukryte pole z tokenem CSRF:
`<input type="hidden" name="csrf_token" value="...">`

Backend (`Flask + Flask-WTH`)

```python
from flask import Flask, render_template
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField
from wtforms.validators import DataRequired

app = Flask(__name__)
app.secret_key = 'twoj_super_tajny_klucz'

class ExampleForm(FlaskForm):
    name = StringField('Name', validators=[DataRequired()])
    submit = SubmitField('Submit')

@app.route('/form', methods=['GET', 'POST'])
def form():
    form = ExampleForm()
    if form.validate_on_submit():
        # Tutaj obsługa formularza, token CSRF jest automatycznie sprawdzany
        return 'Formularz przesłany poprawnie!'
    return render_template('form.html', form=form)
```

Frontend (`HTML`)

```html
<form method="post">
    {{ form.hidden_tag() }}
    {{ form.name.label }} {{ form.name(size=20) }}
    {{ form.submit() }}
</form>
```

Ochrona przed CSRF w SPA

⭐ **SPA (JavaScript) – CSRF działa inaczej**
W SPA: dane wysyłasz JSON-em, żądania robisz fetch, formularzy HTML nie ma, Flask-WTF nie działa automatycznie.

⭐ **Token CSRF** : token CSRF, który jest generowany na backendzie, a następnie wstrzykiwany do każdego żądania wysyłanego przez frontend.

- Token CSRF to zwykle **losowy, kryptograficznie bezpieczny ciąg znaków (np. hex lub Base64)**

```python
from flask import Flask, session, jsonify
import secrets

app = Flask(__name__)
app.secret_key = 'twoj_super_tajny_klucz'

@app.route('/get-csrf-token')
def get_csrf_token():
    # Generujemy losowy token CSRF
    csrf_token = secrets.token_hex(16)
    # Zapisujemy go w sesji więc zostanie wysłane w ciasteczku sesyjnym
    session['csrf_token'] = csrf_token
    # Zwracamy także token w odpowiedzi JSON aby frontend mógł go pobrać
    return jsonify({'csrf_token': csrf_token})
```

```javascript
// Przykład: pobranie tokena i wysłanie go w fetchu
fetch('/get-csrf-token')
  .then(response => response.json())
  .then(data => {
    const csrfToken = data.csrf_token;
    fetch('/some-protected-endpoint', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        'X-CSRF-Token': csrfToken
      },
      body: JSON.stringify({ /* dane żądania */ })
    });
  });
```

```javascript
// Pobranie tokena z ciasteczka (jeśli nie jest HttpOnly)
function getCookie(name) {
    return document.cookie
        .split('; ')
        .find(row => row.startsWith(name + '='))
        ?.split('=')[1];
}

const csrfToken = getCookie("csrf_token");

fetch('/some-protected-endpoint', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json',
        'X-CSRFToken': csrfToken
    },
    body: JSON.stringify({ ... })
});
```

Rola JWT w kontroli dostępu

🟦 Co to jest JWT? (JSON Web Token)

JWT (JSON Web Token) to **samowystarczalny token** służący do uwierzytelniania i autoryzacji użytkownika.
Przechowuje dane w formacie JSON i jest **podpisany kryptograficznie**, więc nie można go podrobić ani zmodyfikować.
Składa się z 3 części:

1. **Header** – typ tokena, algorytm (np. HS256)
2. **Payload** – dane użytkownika (np. username, role, exp)
3. **Signature** – podpis obliczony z użyciem tajnego klucza serwera

🟦 Dlaczego JWT jest „samowystarczalny”?

Serwer **nie musi przechowywać sesji** użytkownika.
Wszystkie potrzebne informacje są w tokenie **podpisane kluczem**.
Serwer jedynie **weryfikuje podpis** i ufa danym w środku.
To czyni JWT **bezstanowym mechanizmem autoryzacji** (stateless authentication).

🟦 Kiedy NIE używać JWT?

Nie stosujemy JWT gdy:
✓ Aplikacja korzysta z klasycznego modelu sesyjnego
✓ Serwer renderuje HTML (Jinja2)
✓ Mamy normalne logowanie + sesja oparta o cookies
✓ Korzystamy z **Flask-Login**, który już “załatwia” sesje

Wniosek:
👉 W typowych aplikacjach monolitycznych oraz w modelu mieszanym (Render Template + fetch w JS), w pełni wystarczające jest korzystanie z Flask-Login i sesji Flaska.

Kiedy stosować JWT?

🟦 **JWT ma sens, gdy:**

✓ **Tworzysz SPA (React, Vue, Angular)**
Backend to API → frontend przetrzymuje token, a każde zapytanie ma nagłówek:
`Authorization: Bearer <token>`

✓ **Tworzysz aplikację mobilną** JWT działa na Android/iOS tak samo.

✓ **Budujesz mikroserwisy** Token „przenosi" tożsamość między serwisami.

✓ **Chcesz łatwo skalować aplikację (load balancing)** Brak sesji po stronie serwera = brak konieczności współdzielenia pamięci.

✓ **Tworzysz publiczne REST API** JWT to standard w autoryzacji API.

🟦 **Jak działa autoryzacja JWT? (flow)**

1. Użytkownik wysyła login + hasło → /login
2. Backend weryfikuje dane i generuje JWT
3. Frontend zapisuje JWT (np. localStorage)
4. Przy każdym zapytaniu frontend wysyła nagłówek:
   `Authorization: Bearer <jwt>`
5. Backend weryfikuje podpis JWT
6. Jeśli podpis OK → użytkownik jest zalogowany
7. Brak potrzeby „sesji" po stronie backendu.

Mechanizm CORS w przeglądarkach i aplikacjach

CORS (Cross-Origin Resource Sharing) nie jest atakiem. Jest to mechanizm bezpieczeństwa zaimplementowany w przeglądarkach, który pozwala na kontrolowane rozluźnienie domyślnych restrykcji. Kwestia bezpieczeństwa sprowadza się zatem do **poprawnej konfiguracji**, aby nie narazić aplikacji na nieautoryzowany dostęp.

Fundament: Polityka Samego Pochodzenia (SOP)

- Aby zrozumieć CORS, należy zdefiniować **SOP (Same-Origin Policy)**. Jest to domyślna zasada bezpieczeństwa przeglądarek internetowych.
- Zakłada ona, że skrypt JavaScript pobrany z jednego źródła (origin) może łączyć się wyłącznie z serwerem, z którego pochodzi. Próba połączenia z innym serwerem zostanie zablokowana przez przeglądarkę.

Definicja "Pochodzenia" (Origin):
Na pochodzenie składają się trzy elementy: Protokół + Domena + Port.

- http://localhost:5000 oraz http://localhost:5000 -> **To samo pochodzenie.**
- http://localhost:5000 oraz http://localhost:3000 -> **Różne pochodzenia** (różne porty).
- http://serwis.pl oraz https://serwis.pl -> **Różne pochodzenia** (różny protokół).

Czym jest CORS?

- ✓ CORS to standard pozwalający serwerowi na jawne zezwolenie przeglądarce na pominięcie blokady SOP.
- ✓ Jest to informacja dla przeglądarki, że żądanie pochodzące z innej domeny jest zaufane i powinno zostać przetworzone.

CORS – Mechanizm działania

Mechanizm działania:

1. Przeglądarka wykrywa, że kod JavaScript próbuje wykonać żądanie do innej domeny.
2. (Opcjonalnie) Przeglądarka wysyła żądanie wstępne typu OPTIONS (tzw. **Preflight request**), weryfikując, czy serwer zezwala na daną akcję.
3. Serwer Flask odpowiada, dołączając nagłówki CORS, np.:
   - `Access-Control-Allow-Origin: http://localhost:3000`
   - `Access-Control-Allow-Methods: GET, POST`
4. Jeśli nagłówki są zgodne z żądaniem, przeglądarka zezwala na odczyt odpowiedzi przez skrypt.

Funkcjonowanie w aplikacji hybrydowej (Flask)

Przypadek A: Widoki Tradycyjne (render_template)

Gdy użytkownik nawiguje do adresu `https://domena.pl/dashboard`, Flask renderuje kod HTML.

Status CORS: Mechanizm ten **nie ma tutaj zastosowania**.

Przypadek B: API + JavaScript (/api/users)

W tym przypadku skrypt JS na załadowanej stronie wykonuje żądanie `fetch('/api/users')`.

Wyróżnia się tu trzy scenariusze:

SCENARIUSZ 1: Produkcja / Monolit (Jedna domena)

Frontend oraz API są serwowane z tego samego adresu i portu (np. `https://domena.pl`).

Status: Polityka SOP zezwala na połączenie.

SCENARIUSZ 2: Środowisko Deweloperskie (Frontend oddzielnie)

Frontend uruchamiany jest na porcie 3000, a Flask na porcie 5000.

Status: Polityka SOP blokuje połączenie (różne porty).

Wniosek: Należy włączyć CORS we Flasku, ograniczając go wyłącznie do środowiska deweloperskiego.

SCENARIUSZ 3: API Publiczne / Integracje B2B

Zewnętrzny serwis (np. partner.com) ma pobierać dane z API.

Status: Polityka SOP blokuje połączenie.

Wniosek: Należy włączyć CORS, jawnie wskazując domenę partnera na białej liście

```python
from flask import Flask, render_template, jsonify
from flask_cors import CORS

app = Flask(__name__)

# BŁĄD KONFIGURACYJNY (Ryzyko wycieku danych):
# CORS(app)  <-- Ustawia Access-Control-Allow-Origin: *
# Pozwala dowolnej domenie na dostęp do API.

# POPRAWNA KONFIGURACJA (Whitelisting):
# Zezwolenie wyłącznie zaufanym domenom na dostęp do zasobów
CORS(app, resources={
    r"/api/*": {  # Ograniczenie CORS wyłącznie do ścieżek API
        "origins": ["http://localhost:3000", "https://zaufany-partner.pl"]
    }
})

# Endpointy renderowane (HTML)
@app.route('/')
def home():
    # Tutaj nagłówki CORS nie są dodawane (i nie są potrzebne)
    return render_template('index.html')

# Endpointy API
@app.route('/api/data')
def api_data():
    # Flask doda nagłówki CORS tylko jeśli żądanie pochodzi
    # z domen zdefiniowanych w konfiguracji.
    return jsonify({"data": "dane chronione"})
```

IV. Konfiguracja, sekrety, DevSecOps

Ten blok dotyczy wszystkiego, co *nie jest kodem aplikacji*, ale ma **krytyczne znaczenie dla bezpieczeństwa** – konfiguracji środowiska, zarządzania sekretami, zależnościami, ustawieniami produkcyjnymi, logowaniem.

⭐ Co to jest „sekret" w aplikacji?

Sekrety (Secrets) to dane, których nie wolno trzymać w repozytorium ani ujawniać:

- SECRET_KEY (dla Flask sesji)
- API Keys (np. do OpenWeather, Stripe)
- Hasła do baz danych
- Tokeny OAuth
- Klucze szyfrujące
- Hasła do usług trzecich (SMTP, Redis)

Sekret = każda wartość, która daje dostęp do zasobów.

.gitignore:

```
.env
```

✓ `.env` nie trafia do repozytorium
✓ sekret kontrolowany przez środowisko

⭐ Jak przechowywać sekrety w Pythonie / Flask?

❌ **Błędnie – sekret w kodzie:**

```
app.config["SECRET_KEY"] = "123456789"
```

Problemy:

- trafia do GitHuba,
- każdy może go odczytać,
- nie da się zmienić bez edycji kodu,
- złamanie klucza sesji → przejęcie sesji użytkowników.

✔ **Poprawnie – sekrety w `.env` + `python-dotenv`**

Plik `.env`:

```
SECRET_KEY = 'bardzo-tajny-klucz-lab-7'
SQLALCHEMY_DATABASE_URI = "sqlite:///../instance/lab7.db„
```

Aplikacja Flask:

```
from dotenv import load_dotenv
import os

load_dotenv()

app.config["SECRET_KEY"] = os.getenv("SECRET_KEY")
app.config["SQLALCHEMY_DATABASE_URI"] = os.getenv("DATABASE_URL")
```

V. Zabezpieczenie zależności i Logging

⭐ **Zabezpieczanie zależności**
Dlaczego to ważne?
- luka w zależności = luka w Twojej aplikacji, nawet jeśli Twój kod jest poprawny
- biblioteki Pythona mają CVE (np. XXE, SSRF, SQL Injection)

CVE = Common Vulnerabilities and Exposures
(*Powszechnie Znane Słabości i Podatności*)
To **globalny system identyfikatorów podatności bezpieczeństwa**.
Każda poważna luka dostaje **unikalny numer CVE.**
(luka w bibliotekach Pythona, systemach operacyjnych, serwerach, frameworkach, narzędziach)

🔧 **Narzędzie: `pip-audit`**
Instalacja:
```
pip install pip-audit
```

Użycie:
```
pip-audit
```

Wynik:
- lista zależności z podatnościami,
- możliwe aktualizacje,
- identyfikatory CVE.

Pozytywny wynik to: No known vulnerabilities found

⭐ **Logging: jakie logi musi mieć bezpieczna aplikacja?**
1. Logi błędów
- ✓ wyjątki,
- ✓ błędne żądania,
- ✓ błędy API,
- ✓ problemy z bazą danych.

NIE WOLNO:
- ❖ logować haseł,
- ❖ logować tokenów JWT,
- ❖ logować numerów kart, PESEL itp.

2. Logi bezpieczeństwa
- ➢ nieudane logowania,
- ➢ zablokowane IP,
- ➢ przekroczenie limitów,
- ➢ próby dostępu do zasobów admina,
- ➢ CSRF failure.

Przykład:
```
app.logger.warning(
    f"Failed login attempt from IP
{request.remote_addr}"
)
```

VI. DevSecOps: cykl bezpieczeństwa

Development

- ✓ **Walidacja danych** – każdy endpoint i formularz sprawdza typy, zakresy i format danych wejściowych, zamiast ufać użytkownikowi.
- ✓ **Testy jednostkowe bezpieczeństwa** – piszemy testy, które sprawdzają np. brak dostępu do danych bez uprawnień albo blokowanie zbyt długich wejść.
- ✓ **Analiza kodu** – przeglądy kodu (code review) pod kątem błędów bezpieczeństwa, wzorców typu „shell=True", „innerHTML" itp.
- ✓ **Statyczna analiza kodu (SAST, np. Bandit)** – automatyczne skanery analizują kod źródłowy i wykrywają niebezpieczne konstrukcje zanim trafią do repozytorium. (SAST = Static Application Security Testing)

Security

- ✓ **pip-audit** – regularnie skanujemy zależności pod kątem znanych podatności (CVE) i planujemy aktualizacje.
- ✓ **Testy penetracyjne** – zespół lub zewnętrzny pentester próbuje „złamać" aplikację jak napastnik i raportuje luki.
- ✓ **Analiza API (AuthZ, AuthN)** – sprawdzamy, czy każde wywołanie API poprawnie uwierzytelnia użytkownika (AuthN) i egzekwuje jego uprawnienia (AuthZ).

Operations

- ✓ **Logi bezpieczeństwa** – zapisujemy próby logowania, błędy autoryzacji, blokady IP i inne zdarzenia istotne z punktu widzenia ataków.
- ✓ **Monitorowanie** – śledzimy metryki (ruch, błędy, czas odpowiedzi) i szukamy nietypowych wzorców, które mogą oznaczać atak.
- ✓ **Alerty** – system automatycznie powiadamia (mail, Slack, SMS), gdy dzieje się coś podejrzanego, zamiast liczyć na ręczne sprawdzanie logów.

Proces ciągły

- ✓ **Rotacja sekretów** – klucze, hasła i tokeny są regularnie zmieniane, a nie używane latami w tej samej postaci.
- ✓ **Aktualizacja zależności** – biblioteki i frameworki są systematycznie podnoszone do bezpiecznych wersji, zamiast tkwić na starych release'ach.
- ✓ **Automatyczne skanowanie CI/CD** – pipeline przy każdym buildzie uruchamia testy, SAST, pip-audit i inne skanery, blokując wdrożenie z lukami.

Podsumowanie i mapa zagrożeń

⭐ Najważniejsze kategorie zagrożeń (OWASP Style)

1. **Injection** (SQL, Command, Path)
2. **XSS** (DOM, Reflected, Stored)
3. **CSRF**
4. **Broken Authentication**
5. **Broken Access Control (AuthZ)**
6. **Security Misconfiguration**
7. **Sensitive Data Exposure**
8. **API Security (Auth, Rate Limits, CORS)**
9. **SSRF / Path Traversal**
10. **Logging & Monitoring Failures**

⭐ Co Flask robi automatycznie?

- ✓ **Autoescape w Jinja2**
  (chroni przed XSS w HTML)
- ✓ **Sesje podpisane SECRET_KEY**
  (brak manipulacji ciastkiem)
- ✓ **Flask-WTF** → automatyczny CSRF (w tradycyjnych aplikacjach)
- ✓ **Parametryzacja SQL w SQLAlchemy**
  (chroni przed SQL Injection)
- ✓ **Bezpieczne pliki dzięki `secure_filename()`**
- ✓ **Obsługa wyjątków i logowanie**

⭐ Co MUSI zrobić programista?

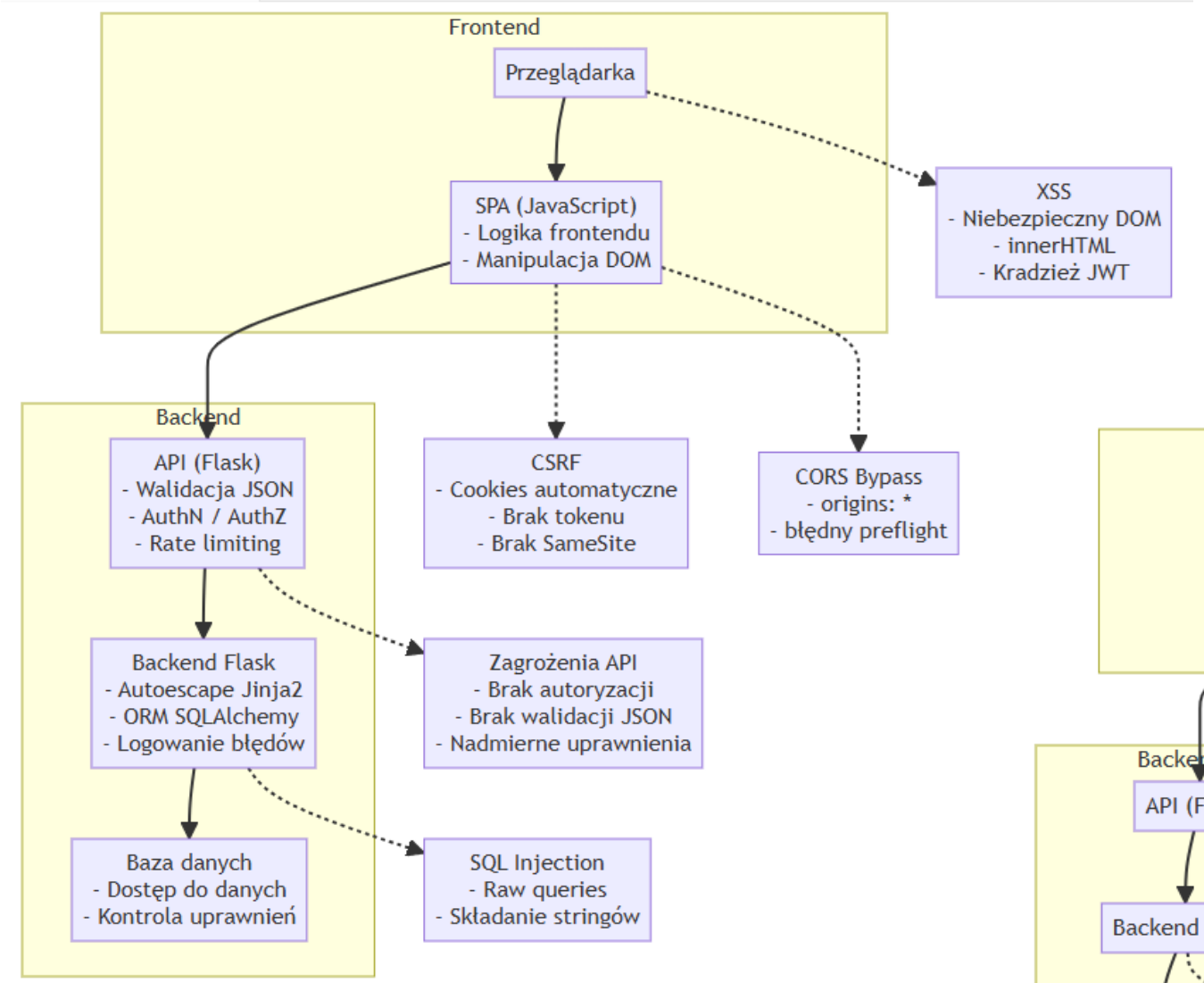
- ➢ walidacja danych wejściowych: (WTForms / marshmallow)
- ➢ ochrona API: autoryzacja, role, JWT/session management
- ➢ obsługa CORS (SPA)
- ➢ limitowanie requestów (Flask-Limiter)
- ➢ konfiguracja środowiska (debug off)
- ➢ przechowywanie sekretów
- ➢ rotacja kluczy
- ➢ bezpieczne uploady
- ➢ brak użycia `innerHTML` w JS
- ➢ ochrona JWT przed XSS

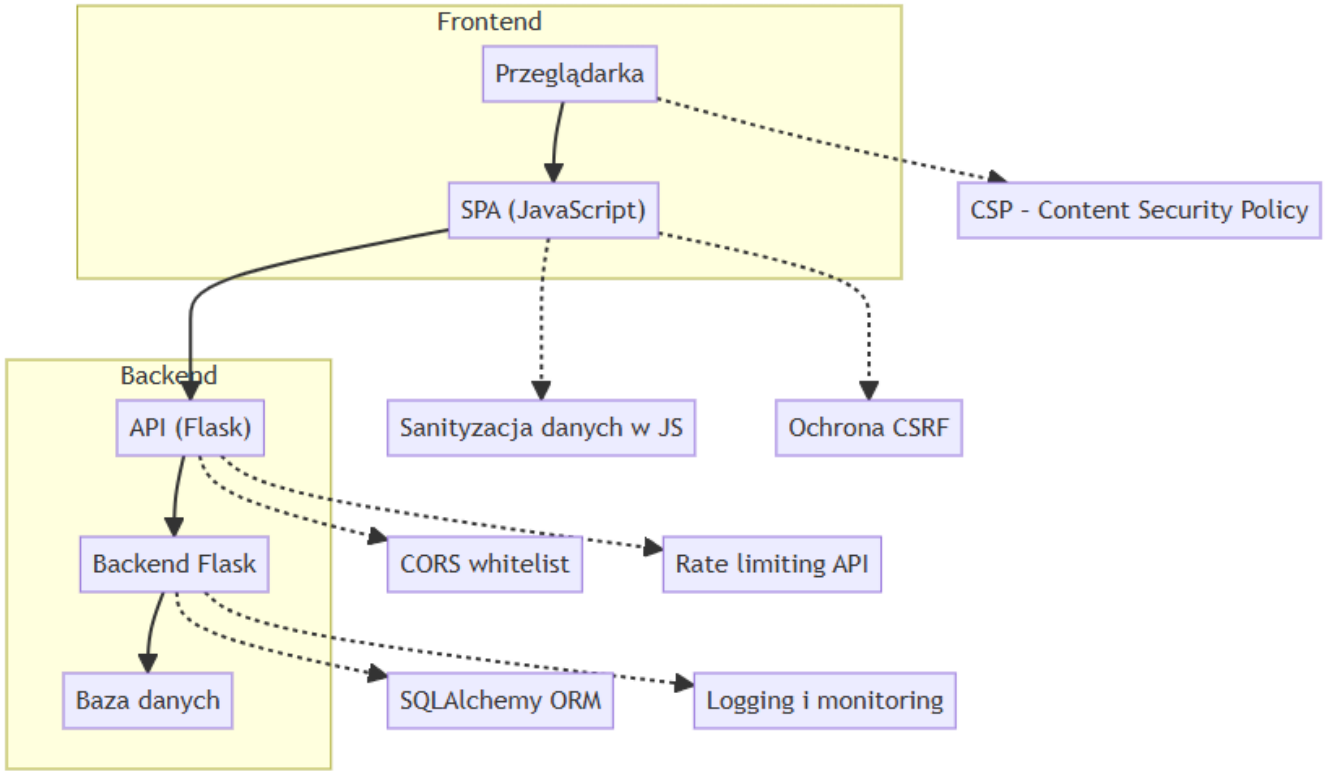
⭐ Co w SPA wymaga dodatkowej logiki w JavaScript?

SPA usuwa część automatycznych zabezpieczeń, które daje Flask.
W JS trzeba zrobić:

- ✔ **Ochronę przed XSS**
  - nie używać `innerHTML`
  - używać `textContent`
  - sanitizacja HTML (DOMPurify)
- ✔ **Ochronę API**
  - wysyłanie nagłówków `Authorization`
  - trzymanie dostępu/refresh tokenów w bezpiecznym miejscu
- ✔ **CSRF (jeśli SPA używa cookies)**
  - pobieranie tokenu CSRF
  - dodawanie nagłówka w `fetch()`
  - trzymanie kopii tokenu w cookie
- ✔ **Obsługę błędów API**
  - reagowanie na 401/403
  - automatyczne odświeżanie JWT
- ✔ **CORS**
  - backend musi być skonfigurowany – ale SPA musi wysyłać requesty zgodnie z zasadami (np. `credentials: "include"`).

Mapy zagrożeń i bezpieczeństwa





Link do testu:

https://tinyurl.com/WykladCyber-8

Literatura:

- **Oficjalny przewodnik bezpieczeństwa Flask**: https://flask.palletsprojects.com/en/stable/web-security
- **Jak zabezpieczyć aplikacje Flask – artykuł Snyk:** https://snyk.io/blog/secure-python-flask-applications
- **OWASP Cheat Sheets – Web Security + API Security:** https://cheatsheetseries.owasp.org/

Niektóre treści w prezentacji zostały wygenerowane przy użyciu ChatGPT oraz Google AI Studio.