Programowanie Back-end

Wprowadzenie

mgr inż. Jakub Gogola

Podstawowe informacje

- **Prowadzący** - mgr inż. Jakub Gogola
- **Kontakt** - jakub.gogola@dsw.edu.pl
- **Konsultacje** - *po indywidualnym umówieniu się; za pośrednictwem aplikacji Teams*

Zasady zaliczenia

Wykład
Kolokwium zaliczeniowe (pytanie otwarte + zamknięte) na ostatnim wykładzie (ocena).

Ćwiczenia
Listy zadań (zaliczenie) oraz semestralny projekt zaliczeniowy (ocena) wykorzystujące poznane na wykładzie technologie i wzorce.

Ocena końcowa
Na ocenę semestralną będą składać się oceny końcowe z wykładu oraz z ćwiczeń. Zostanie ona wyliczona według następującego wzoru

$$30\,\%\,o_{\text{wykład}} + 70\,\%\,o_{\text{ćwiczenia}}$$

Zasady zaliczenia

Warunkiem koniecznym uzyskania pozytywnej oceny z kursu jest uzyskanie co najmniej 50% punktów dla każdej z form zaliczenia. Oceny z wykładu, ćwiczeń oraz końcowa zostaną wyliczone według następującej skali (gdzie x oznacza zdobytą liczbę punktów):

- $0\,\% \leq x < 50\,\%$ - 2.0 (niedostateczny),
- $50\,\% \leq x < 60\,\%$ - 3.0 (dostateczny),
- $60\,\% \leq x < 70\,\%$ - 3.5 (dostateczny +),
- $70\,\% \leq x < 80\,\%$ - 4.0 (dobry),
- $80\,\% \leq x < 90\,\%$ - 4.5 (dobry +),
- $90\,\% \leq x < 100\,\%$ - 5.0 (bardzo dobry)

Zasady zaliczenia

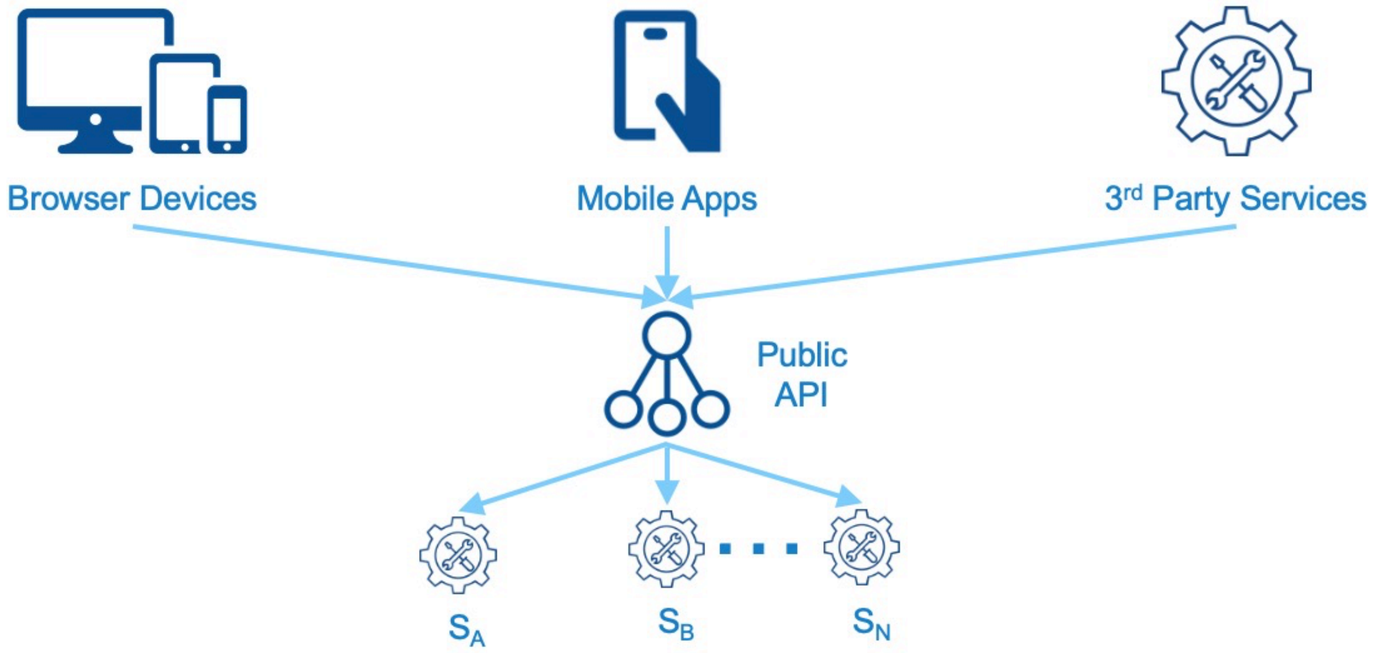
Uwagi

- Ocena końcowa z ćwiczeń będzie *de facto* odpowiadać ocenie za projekt semestralny.
- Uzyskanie pozytywnej oceny z ćwiczeń będzie wymagało zaliczenia wszystkich list oraz uzyskania pozytywnej oceny z projektu semestralnego.
- Przy zaliczaniu projektu semestralnego student **musi** umieć odpowiedzieć na pytania prowadzącego oraz wykazać się znajomością aplikacji. Nieumiejętność odpowiedzenia na pytania lub rażąca niekompetencja będzie skutkować obniżeniem oceny lub niezaliczeniem projektu, a w konsekwencji - kursu.
- Nie przewiduje się poprawkowych/dodatkowych terminów kolokwium zaliczeniowego (poza usprawiedliwionymi i uzasadnionymi przypadkami).

Plan wykładu

- Wprowadzenie do technologii REST API.
- Wprowadzenie do narzędzi ORM.
- Wprowadzenie do narzędzi zarządzania uprawieniami; uwierzytelnienia i autoryzacji.
- Praktyczne wykorzystanie wzorców projektowania aplikacji serwerowych
  - architektura aplikacji,
  - Współbieżność i asynchroniczność,
  - *Command-Query Responsibility Segregation*,
  - *Domain-Driven Design*.
- Mikroserwisy i komunikacja między nimi.
- Aplikacje serwerowe a infrastruktura.

Czym jest backend?



Czym jest backend i jaką rolę pełni w aplikacjach webowych?

Backend to część aplikacji odpowiedzialna za logikę biznesową, przetwarzanie danych i komunikację z bazą danych. Jest niewidoczny dla użytkownika końcowego, ale kluczowy dla działania aplikacji.

Przykłady operacji backend’owych:

- Obsługa użytkowników (logowanie, rejestracja, autoryzacja)
- Przetwarzanie danych (pobieranie, walidacja, zapis)
- Komunikacja z bazami danych i innymi serwisami
- Integracje z systemami zewnętrznymi

Frontend + Backend

Funkcje

Frontend – odpowiada za interfejs użytkownika (UI) i doświadczenie użytkownika (UX), np. React, Vue.js, Angular.

Backend – odpowiada za logikę aplikacji, obsługę żądań, operacje na bazach danych, np. FastAPI, Django, Express.js.

Frontend + Backend

Współdziałanie

1. Użytkownik wysyła zapytanie (np. logowanie) przez frontend.
2. Frontend wysyła żądanie HTTP do backendu.
3. Backend przetwarza żądanie, sprawdza bazę danych i zwraca odpowiedź.
4. Frontend prezentuje otrzymane dane użytkownikowi.

Popularne technologie backend’owe

Frameworki REST API

Python: FastAPI, Django, Flask
JavaScript: Express.js, NestJS, Fastify
Java: Spring Boot
PHP: Laravel, Symfony
Go: Gin, Echo, Mux

API

Co to jest?

API *(Application Programming Interface)* to interfejs umożliwiający komunikację między różnymi systemami, aplikacjami lub komponentami oprogramowania.

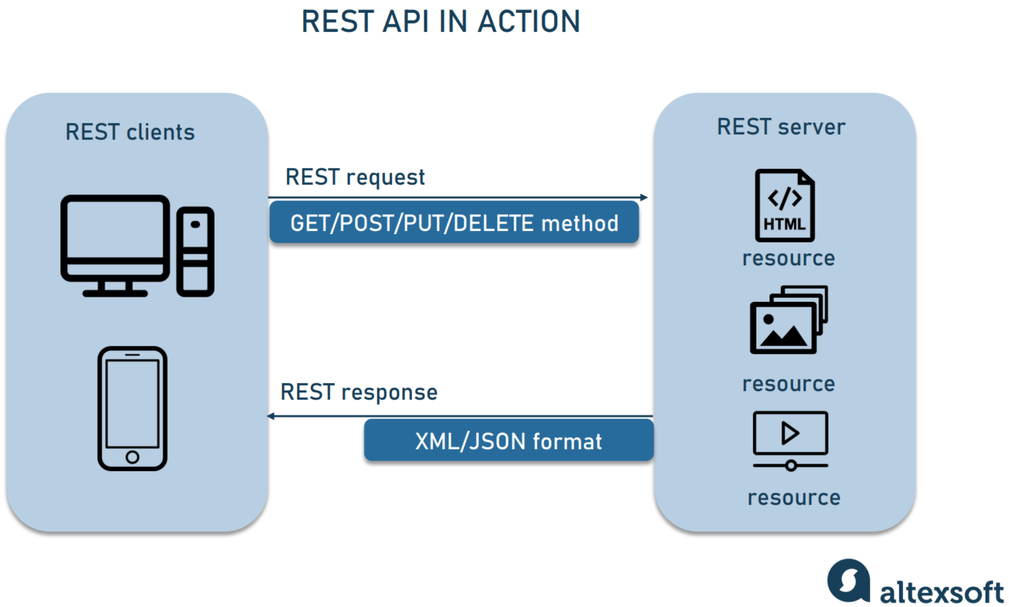
Przykłady użycia API:

- Aplikacje mobilne komunikujące się z serwerem
- Pobieranie prognozy pogody z zewnętrznego serwisu
- Logowanie do aplikacji za pomocą Google lub Facebooka

REST API

Co to jest?

REST *(Representational State Transfer)* to styl architektoniczny, który definiuje sposób komunikacji między klientem a serwerem za pomocą standardowych metod HTTP.



REST API

Koncepcja

REST to zbiór zasad projektowania zdalnego API opartego na HTTP.

Stateless – Serwer nie przechowuje stanu klienta między żądaniami. Każde żądanie musi zawierać wszystkie informacje potrzebne do jego obsługi.
Client-Server – Serwer i klient są od siebie niezależne. Klient (frontend) nie musi znać szczegółów implementacyjnych backend'u.
Uniform Interface – Spójność interfejsu API:

- Identyfikacja zasobów przez **URI** (np. /users/123)
- Wykorzystanie standardowych metod HTTP
- Jednolity format odpowiedzi (np. JSON)

Cacheability – Możliwość buforowania odpowiedzi w celu zwiększenia wydajności.
Layered System – Możliwość stosowania pośrednich warstw (np. *load balancers, proxy, cache*).
Code on Demand – Możliwość pobierania i wykonywania kodu na kliencie.

REST API

Zasoby i operacje HTTP

Metoda HTTP	Funkcja	Przykład użycia
GET	Pobiera dane	`GET /users` (lista użytkowników)
POST	Tworzy zasób	`POST /users` (stworzenie nowego użytkownika)
PUT	Aktualizuje cały zasób	`PUT /users/:userId` (aktualizacja wszystkich danych użytkownika)
PATCH	Aktualizuje część zasobu	`PATCH /users/:userId` (aktualizacja części danych użytkownika)
DELETE	Usuwa cały zasób	`DELETE /users/:userId` (usuwa wszystkie dane użytkownika)

REST API

Przykładowe zapytanie i odpowiedź

```
GET /users/123 HTTP/1.1
Host: api.example.com
Authorization: Bearer token123
```

```
{
  "id": 123,
  "name": "John Doe",
  "email": "john@example.com"
}
```

Alternatywa dla REST API

GraphQL

Ceha	REST	GraphQL
Struktura	Zasoby i kendpointy	Zapytania i typy danych
Pobieranie danych	Wiele żądań do różnych endpoint’ów	Jeden endpoint, zwraca tylko wymagane dane
Typowanie danych	Brak wbudowanego typowania	Silne typowanie
Optymalizacja zapytań	Możliwe nadmiarowe dane	Możliwość ograniczenia żądanych danych

REST API vs GraphQL

Zapytania i odpowiedzi

```
query {
  user(id: 123) {
    id
    name
    email
    orders {
      id
      total
      status
    }
  }
}
```


```
{
  "data": {
    "user": {
      "id": 123,
      "name": "John Doe",
      "email": "john@example.com",
      "orders": [
        {
          "id": 1,
          "total": 99.99,
          "status": "Shipped"
        },
        {
          "id": 2,
          "total": 49.50,
          "status": "Processing"
        }
      ]
    }
  }
}
```


1. Pobranie danych użytkownika

   `GET /users/:userId`

2. Pobranie zamówień użytkownika

   `GET /users/:userId/orders`

Open API

Co to jest?

OpenAPI to otwarty standard do definiowania interfejsów REST API. Umożliwia opisanie struktury API w sposób czytelny zarówno dla ludzi, jak i dla maszyn, co ułatwia dokumentację, testowanie i generowanie kodu.

Open API

Co to jest?

OpenAPI (dawniej Swagger) to specyfikacja służąca do opisywania API w formacie **YAML** lub **JSON**.

Jest niezależna od języka programowania i wspierana przez wiele frameworków, np. **FastAPI, Flask, Spring Boot, Express.js**.

Open API

Zalety

- Automatyczna dokumentacja API (Swagger UI, Redoc)
- Walidacja poprawności API
- Generowanie kodu klientów i serwerów
- Łatwiejsze testowanie API

Open API

Struktura

```yaml
openapi: 3.0.0
info:
  title: Coffee API
  version: 1.0.0
servers:
  - url: https://api.example.com
paths:
  /coffee:
    get:
      summary: Pobierz listę kaw
      responses:
        "200":
          description: Lista dostępnych kaw
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: "#/components/schemas/Coffee"
components:
  schemas:
    Coffee:
      type: object
      properties:
        id:
          type: integer
        name:
          type: string
        origin:
          type: string
```

Co oznaczają poszczególne sekcje?

- **`openapi`** – wersja specyfikacji (np. 3.0.0)
- **`info`** – informacje o API (tytuł, wersja, opis)
- **`servers`** – lista serwerów, z których można korzystać
- **`paths`** – definicja endpoint’ów i obsługiwanych metod HTTP
- **`components`** – zdefiniowane schematy danych (np. model Coffee)

Open API

Definicja endpoint’ów - paths

- Każdy endpoint API ma swoją specyfikację

```yaml
paths:
  /coffee:
    get:
      summary: Pobierz listę kaw
      responses:
        "200":
          description: Lista kaw
```

Open API

Parametry w URL - `parameters`

```yaml
paths:
  /coffee/{id}:
    get:
      summary: Pobierz szczegóły kawy
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: integer
```

Open API

Struktura odpowiedzi - `responses`

```yaml
responses:
  "200":
    description: Sukces
    content:
      application/json:
        schema:
          $ref: "#/components/schemas/Coffee"
```

Open API

Schematy danych - schemas

```yaml
components:
  schemas:
    Coffee:
      type: object
      properties:
        id:
          type: integer
        name:
          type: string
        origin:
          type: string
```

Open API

Generowanie kodu

OpenAPI umożliwia **automatyczne generowanie kodu** dla różnych języków i framewor'ków:

- Generowanie **klienta API** (np. dla React, Python, Java)
- Generowanie **serwera API** (np. FastAPI, Flask, Spring Boot)

Open API

Przykład zastosowania

Repozytorium z przykładami: https://github.com/JakubGogola/dsw-backend-programming/tree/main/lectures/lecture-1

FastAPI

Co to jest?

- Szybki i nowoczesny framework do budowy REST API w Pythonie.
- Oparty na **Starlette** (wydajny serwer ASGI) i **Pydantic** (walidacja danych).
- Automatycznie generuje dokumentację OpenAPI i Swagger UI.
- Obsługuje **asynchroniczne operacje** (async / await).

FastAPI

Prosty serwer

```python
from fastapi import FastAPI
from pydantic import BaseModel
from fastapi import HTTPException

app = FastAPI()

@app.get("/")
async def read_root():
    return {"message": "Hello, FastAPI!"}

@app.get("/users/{user_id}")
async def get_user(user_id: int):
    return {"user_id": user_id, "name": "John Doe"}

class User(BaseModel):
    name: str
    email: str

@app.post("/users")
async def create_user(user: User):
    return {"message": "User created", "user": user}

@app.get("/items/{item_id}")
async def get_item(item_id: int):
    if item_id != 1:
        raise HTTPException(status_code=404, detail="Item not found")
    return {"item_id": item_id, "name": "Sample Item"}

@app.get("/users/{user_id}", summary="Get user by ID", description="Returns user details.")
async def get_user(user_id: int):
    return {"user_id": user_id, "name": "John Doe"}
```

FastAPI

Prosty endpoint GET

```python
@app.get("/users/{user_id}")
async def get_user(user_id: int):
    return {"user_id": user_id, "name": "John Doe"}
```

FastAPI

Endpoint POST z walidacją danych

```
from pydantic import BaseModel

class User(BaseModel):
    name: str
    email: str

@app.post("/users")
async def create_user(user: User):
    return {"message": "User created", "user": user}
```

FastAPI

Obsługa błędów

```python
from fastapi import HTTPException

@app.get("/items/{item_id}")
async def get_item(item_id: int):
    if item_id != 1:
        raise HTTPException(status_code=404, detail="Item not found")
    return {"item_id": item_id, "name": "Sample Item"}
```

FastAPI

Automatyczna dokumentacja

```python
@app.get("/users/{user_id}", summary="Get user by ID", description="Returns user details.")
async def get_user(user_id: int):
    return {"user_id": user_id, "name": "John Doe"}
```

FastAPI

Middleware

Middleware to warstwa pośrednicząca, przez którą przechodzą wszystkie żądania i odpowiedzi. Może być używana np. do logowania, dodawania nagłówków, obsługi CORS itp.

FastAPI

Middleware

```python
from fastapi import FastAPI, Request
import time

app = FastAPI()

@app.middleware("http")
async def log_requests(request: Request, call_next):
    start_time = time.time()
    response = await call_next(request)
    duration = time.time() - start_time
    print(f"Request: {request.method} {request.url} | Time: {duration:.4f}s")
    return response
```

FastAPI

Global error handlers

```python
from fastapi import FastAPI, Request, HTTPException
from fastapi.responses import JSONResponse

app = FastAPI()

@app.exception_handler(Exception)
async def global_exception_handler(request: Request, exc: Exception):
    return JSONResponse(
        status_code=500,
        content={"message": "Internal Server Error", "detail": str(exc)},
    )

@app.exception_handler(HTTPException)
async def http_exception_handler(request: Request, exc: HTTPException):
    return JSONResponse(
        status_code=exc.status_code,
        content={"message": "HTTP Error", "detail": exc.detail},
    )
```

Dziękuję za uwagę!