Temat pracy dyplomowej inżynierskiej: "Tworzenie dokumentacji projektu z wykorzystaniem metod sztucznej inteligencji."
Celem tego projektu jest stworzenie inteligentnego systemu, który automatycznie generuje dokumentację dla API na podstawie kodu źródłowego oraz komentarzy zawartych w kodzie. Projekt będzie koncentrował się na generowaniu czytelnej dokumentacji technicznej oraz na personalizacji poziomu szczegółowości w zależności od potrzeb i poziomu doświadczenia użytkownika. Użytkownicy będą mogli przeglądać dokumentację przez intuicyjny interfejs i dostosować jej szczegółowość zgodnie ze swoimi preferencjami.

To właśnie etapy które chciałabym zrealizować w ramach tej pracy:
1. Analiza kodu i generowanie podstawowej dokumentacji
- Stworzenie parsera kodu, który będzie analizować kod źródłowy i identyfikować funkcje, klasy, endpointy itp.
- Integracja z systemem Swagger/OpenAPI do wygenerowania szkieletu dokumentacji na podstawie struktury kodu.

Rezultat: Podstawowa wersja dokumentacji API bez zaawansowanych opisów.

Zbudowałam pełną infrastrukturę backendową:
- java-api (Spring Boot) – serwer główny,
- python-nlp (FastAPI) – mikroserwis do przetwarzania języka naturalnego,
- web (Nginx) – reverse proxy, łączący wszystko pod localhost:8080.

Stworzyłam system uploadu projektu (.zip):
- Endpoint /api/projects/upload rozpakowuje projekt i zapisuje go w /uploads/<ID>.
- Obsługuje walidację, błędy i tworzy unikalny identyfikator projektu.

Dodałam detekcję pliku openapi.yaml lub openapi.yml:
Klasa SpecDetector analizuje strukturę ZIP i odnajduje specyfikację.
Jeśli spec nie istnieje, system oferuje generację dokumentacji z kodu.

Zintegrowałam system z OpenAPI / Swaggerem:
EnrichmentService potrafi wczytać istniejący openapi.yaml i wzbogacić go o opisy wygenerowane przez NLP.

Działa endpoint /api/projects/{id}/spec/enriched.

Dodałam parser kodu źródłowego (JavaParser):
JavaSpringParser analizuje pliki .java, wykrywa klasy z @RestController, ich metody i adnotacje (@GetMapping, @PostMapping itd.).
Tworzy pośrednią strukturę EndpointIR, która opisuje endpointy, parametry i typy zwracane.

Zbudowałam moduł „Code → OpenAPI”:

Klasa CodeToDocsService generuje kompletny plik openapi.generated.yaml na podstawie kodu źródłowego.
Integracja z NLP dodaje opis do każdej metody i parametru.
Działa endpoint /api/projects/{id}/docs/from-code.

2. Implementacja NLP do analizy komentarzy i generowania opisów

- Zastosowanie NLP do analizy komentarzy, aby tworzyć jasne, zrozumiałe opisy funkcji i parametrów.
- Użycie modeli NLP do interpretacji kontekstu i generowania opisów na podstawie komentarzy.

Rezultat: Automatycznie generowane, czytelne opisy dla każdej funkcji, co znacznie zwiększa czytelność dokumentacji.

Stworzyłam osobny mikroserwis NLP (python-nlp), który:
przyjmuje strukturę endpointu (symbol, comment, params, returns),
analizuje komentarze i typy parametrów, generuje automatyczne opisy w trzech poziomach szczegółowości:

- shortDescription
- mediumDescription
- longDescription
- dodaje także paramDocs (opis każdego parametru) i returnDoc.

Zintegrowałam NLP z backendem (Spring Boot):

- EnrichmentService wysyła do /nlp/describe dane z kodu i odbiera opisy.
- Wyniki są automatycznie wstawiane do dokumentacji OpenAPI lub pliku YAML.

Zaimplementowałam personalizowany poziom szczegółowości (short/medium/long):
Użytkownik może wybrać poziom, a system automatycznie dopasowuje długość i szczegółowość opisów.

3. Stworzenie systemu personalizacji dokumentacji

Umożliwiłam ręczny wybór poziomu szczegółowości dokumentacji (short, medium, long) – użytkownik decyduje, jak rozbudowane mają być opisy.

- Implementacja mechanizmów śledzenia interakcji użytkownika, aby rozpoznać wzorce zachowań. (Śledzenie kliknięć i wyborów, czas spędzony na poszczególnych sekcjach, śledzenie wyszukiwań, interakcje z poziomem szczegółowości)
- Zastosowanie uczenia maszynowego do klasyfikacji użytkowników jako początkujących lub zaawansowanych.
- Tworzenie personalizowanych wersji dokumentacji, w zależności od poziomu doświadczenia użytkownika.

Rezultat: Dokumentacja dostosowana do poziomu wiedzy użytkownika, z możliwością wyboru poziomu szczegółowości.

4. Budowa interaktywnego interfejsu użytkownika
- Stworzenie dynamicznego interfejsu użytkownika, który umożliwia przeglądanie dokumentacji, filtrowanie i przeszukiwanie.
- Integracja interfejsu z backendem oraz systemem personalizacji.

Rezultat: Funkcjonalny interfejs użytkownika, który umożliwia wygodne przeglądanie dokumentacji i dostosowywanie poziomu szczegółowości.

5. Testowanie i optymalizacja
- Przeprowadzenie testów użyteczności i optymalizacji pod kątem wydajności.
- Testowanie algorytmów personalizacji i dopasowywanie ich do realnych potrzeb użytkowników.

Rezultat: Stabilna i zoptymalizowana wersja systemu gotowa do wdrożenia.

07.10.2025:

Java: Spring Boot, springdoc-openapi
Python: FastAPI, do NLP: spaCy / Hugging Face
Frontend: React + TypeScript
Wspólne: Docker

Co działa teraz:
• java-api - serwis backendowy (Spring Boot),
• python-nlp - mikroserwis AI (FastAPI),
• web - serwer Nginx (reverse proxy), który spina wszystko razem i wystawia publiczny adres http://localhost:8080.

1. **Środowisko uruchomieniowe (Docker + Nginx)**
• Trzy serwisy odpalane razem: java-api (Spring Boot), python-nlp (FastAPI), web (Nginx reverse proxy).
• Jeden punkt dostępu: http://localhost:8080 (Nginx przekazuje /api, /v3, /swagger-ui, /nlp do właściwych serwisów).

2. **Java API – szkielety i dokumentacja**
- springdoc-openapi podłączony: automatyczna specyfikacja OpenAPI: /v3/api-docs (JSON), /v3/api-docs.yaml (YAML),
- Swagger UI: /swagger-ui/index.html.
- OpenApiConfig: ładny tytuł, opis, contact, license (MIT)

Endpointy demo (do dokumentowania i testów)
- GET /api/hello?name=: szybki test.
- GET /api/users/{id}: przykładowy odczyt (DTO w odpowiedzi).
- POST /api/users (JSON body + walidacja): pełny przepływ request body: response:
- 400 Bad Request z czytelnymi błędami walidacji, gdy dane są niepełne.

Java API będzie wysyłać surowe dane (nazwy funkcji, parametry, komentarze) do serwisu python-nlp, żeby otrzymać opisy w języku naturalnym.

3. **Python NLP – gotowy mikroserwis**
• GET /nlp/healthz (przez Nginx jako /nlp/healthz) — healthcheck.
• POST /nlp/describe — zwraca short/medium/long (szkielet pod późniejsze NLP).
• Nginx ma poprawne proxy dla /nlp/*, więc UI/Java mogą go wołać bez CORS.

• Java API będzie wysyłać do niego „surowe dane z parsera" (nazwy metod, komentarze),
• on będzie zwracał czytelne opisy,
• dane te trafią z powrotem do dokumentacji OpenAPI.

Pliki/elementy, które powstały:
• java-api/pom.xml — zależności: springdoc-openapi-starter-webmvc-ui, walidacja.
• java-api/src/main/java/.../config/OpenApiConfig.java — tytuł/opis/contact/license.
• java-api/src/main/java/.../controller/HelloController.java — prosty endpoint.
• java-api/src/main/java/.../controller/UsersController.java — GET/POST z JSON body.
• java-api/src/main/java/.../dto/CreateUserRequest.java i UserResponse.java — DTO (walidacja + schematy w OpenAPI).
• web-ui/nginx.conf — proxy do /api, /v3, /swagger-ui, /nlp.
• docker-compose.yml — definicje trzech kontenerów i ich sieci.

Zastosowany mikroserwis python-nlp będzie wykorzystywać model językowy mT5 (Multilingual Text-to-Text Transfer Transformer), opracowany przez Google Research.
Model ten przetwarza dane wejściowe w postaci komentarzy i nazw metod, a następnie generuje opisy w języku naturalnym w kilku wariantach (krótki, średni, szczegółowy).
Dzięki temu możliwe jest tworzenie dokumentacji technicznej opartej na kodzie źródłowym w sposób zautomatyzowany i inteligentny, bez konieczności pisania tekstów przez człowieka.

Test	Heurystyki (bez AI)	NLP (z AI)
GET /api/users/{id}	„Zwraca zasób po ID."	„Zwraca użytkownika o podanym identyfikatorze. Jeśli nie istnieje, zwraca 404."
POST /api/users	„Tworzy nowy zasób."	Tworzy nowego użytkownika z danymi name i il, walidując poprawność adresu e-mail."

google/mt5-small

Co się dzieje pod spodem:
1. Plik trafia do backendu (java-api - / api/upload).
2. Mój system rozpakowuje ZIP-a, analizuje kod:
   - wykrywa klasy, kontrolery, funkcje, parametry, adnotacje, komentarze;
   - tworzy surowy opis kodu.

3. Dla kazdego endpointu (np. GET /api/ users/{id}) wysyła zapytanie do mikroserwisu python-nlp, który analizuje komentarze i generuje teksty opisowe (short, medium, long).

Swagger daje strukturę, a NLP daje semantykę i naturalny język
Przed: surowe dane

```
/api/hello:
  get:
    responses:
      "200":
        description: OK
```

Po:

```
/api/hello:
  get:
    summary: Zwraca powitanie użytkownika.
    description: Endpoint zwraca powitanie z imieniem przekazanym w parametrze
`name`.
    responses:
      "200":
        description: Poprawna odpowiedź z wiadomością powitalną.
```

14.10.2025:

2. Implementacja NLP do analizy opisów w specyfikacji OpenAPI i generowania rozszerzonej dokumentacji

W ramach tego etapu wdrożono mikroserwis NLP, który analizuje istniejące opisy i komentarze w pliku OpenAPI (openapi.yaml) oraz automatycznie generuje bardziej rozbudowane, naturalne i zrozumiałe opisy funkcji, parametrów i odpowiedzi.

W odróżnieniu od klasycznego podejścia, gdzie analiza odbywa się bezpośrednio na kodzie źródłowym, system wykorzystuje strukturę OpenAPI jako pośrednią warstwę semantyczną. Dzięki temu możliwe jest automatyczne wzbogacanie dokumentacji wygenerowanej z dowolnego projektu zawierającego specyfikację API, niezależnie od języka programowania.

Mikroserwis NLP, oparty na frameworku FastAPI i modelach językowych, generuje opisy w trzech poziomach szczegółowości (short, medium, long). Wyniki są automatycznie wstawiane do sekcji description w obiektach paths, parameters i responses specyfikacji OpenAPI.

Rezultat: dokumentacja API staje się pełniejsza, spójna i bardziej zrozumiała dla użytkownika końcowego, bez konieczności ręcznego uzupełniania opisów w kodzie.

Do 21.10:

Następnym krokiem w rozwoju systemu będzie

1. **dodanie pełnej obsługi generowania dokumentacji na podstawie kodu źródłowego i komentarzy w kodzie – w sytuacji, gdy projekt nie zawiera pliku openapi.yaml.**

Jeśli użytkownik wgra projekt bez gotowej specyfikacji OpenAPI, system:

- automatycznie wykryje brak pliku openapi.yaml,
- przeanalizuje kod źródłowy (Java, a w przyszłości także Python),
- odczyta komentarze, typy danych i endpointy,
- wygeneruje kompletną dokumentację API przy użyciu NLP,
- zapisując ją jako openapi.generated.yaml.

Dzięki temu użytkownik nie musi samodzielnie pisać pliku OpenAPI, dokumentacja zostanie stworzona na podstawie kodu i komentarzy.

UPDATE *18.10.25*:

Co zostało zrobione:

Zaimplementowałam mechanizm automatycznego generowania dokumentacji API w formacie OpenAPI na podstawie kodu źródłowego projektu (Java) w sytuacji, gdy użytkownik nie dostarcza własnego pliku openapi.yaml.
System analizuje kod, odczytuje komentarze (Javadoc), typy danych oraz adnotacje kontrolerów Springa, a następnie generuje kompletny plik openapi.generated.yaml.

Jak to działa:

1. Użytkownik wysyła projekt jako archiwum ZIP.
2. System sprawdza, czy w projekcie znajduje się plik openapi.yaml. Jeśli go brak — uruchamiany jest moduł Code -> OpenAPI.
3. Klasa JavaSpringParser analizuje wszystkie pliki .java:
   - wykrywa klasy oznaczone adnotacjami @RestController lub @Controller,
   - rozpoznaje metody z adnotacjami @GetMapping, @PostMapping, @RequestMapping itd.,
   - odczytuje ścieżki, typy metod HTTP, parametry oraz komentarze Javadoc (@param, @return).

Wynik zapisywany jest jako struktura pośrednia EndpointIR.

4. Klasa CodeToDocsService przetwarza te dane i generuje gotową specyfikację OpenAPI 3.0:
   - dodaje sekcje paths, parameters, requestBody, responses,
   - uzupełnia opisy metod i parametrów przy pomocy NLP,
   - zapisuje wynik jako plik openapi.generated.yaml.
5. Użytkownik może pobrać wygenerowany plik.

Efekt: Dzięki temu system automatycznie tworzy pełną dokumentację API nawet wtedy, gdy projekt nie zawiera gotowego pliku openapi.yaml. Użytkownik nie musi jej pisać ręcznie — dokumentacja jest generowana dynamicznie na podstawie kodu i komentarzy.

```
openapi: 3.0.1
info:
```

```yaml
  title: Project bb587ae5001842b3aa59a8623c9ee7a8-API
  version: 1.0.0
paths:
  /@GetMapping("/hello"):
    get:
      summary: "Zwraca obiekt `Map<String,String>`. Typowe kody odpowiedzi: 200."
      description: "Zwraca obiekt `Map<String,String>`. Typowe kody odpowiedzi: 200."
      operationId: HelloController_hello
      parameters:
      - name: name
        in: query
        description: Parametr name.
        required: false
        schema:
          type: string
      responses:
        "200":
          description: "Zwraca obiekt `Map<String,String>`."
          content:
            application/json:
              schema:
                type: object
  /@RequestMapping("/api/orders")/@GetMapping("/id"):
    get:
      summary: "Pobiera zamówienie po ID. Typowe kody odpowiedzi: 200."
      description: "Pobiera zamówienie po ID. Typowe kody odpowiedzi: 200."
      operationId: OrderController_getOrder
      parameters:
      - name: id
        in: path
        description: Identyfikator zamówienia.
        required: true
        schema:
          type: string
      responses:
        "200":
          description: Zwraca obiekt `Object`.
          content:
            application/json:
              schema:
```

```
            type: object
/@RequestMapping("/api/orders")/id:
  delete:
    summary: "Usuwa zamówienie (przykład użycia RequestMapping z metodą). Typowe\
      \ kody odpowiedzi: 200."
    description: "Usuwa zamówienie (przykład użycia RequestMapping z metodą). Typowe\
      \ kody odpowiedzi: 200."
    operationId: OrderController_delete
    parameters:
    - name: id
      in: path
      description: Identyfikator zamówienia.
      required: true
      schema:
        type: string
    responses:
      "200":
        description: Zwraca obiekt `Object`.
        content:
          application/json:
            schema:
              type: object
/@RequestMapping("/api/orders")/@PostMapping("/orderId/items"):
  post:
    summary: "Dodaje pozycję do zamówienia. Typowe kody odpowiedzi: 200."
    description: "Dodaje pozycję do zamówienia. Typowe kody odpowiedzi: 200."
    operationId: OrderController_addItem
    parameters:
    - name: orderId
      in: path
      description: Identyfikator zamówienia.
      required: true
      schema:
        type: string
    - name: sku
      in: query
      description: Kod produktu.
      required: false
      schema:
        type: string
```

```
    - name: qty
      in: query
      description: Ilość.
      required: false
      schema:
        type: integer
        format: int32
    responses:
      "200":
        description: Zwraca obiekt `Object`.
        content:
          application/json:
            schema:
              type: object
  /@RequestMapping("/api/users")/@GetMapping("/id"):
    get:
      summary: "Zwraca użytkownika po ID. Typowe kody odpowiedzi: 200."
      description: "Zwraca użytkownika po ID. Typowe kody odpowiedzi: 200."
      operationId: UserController_getById
      parameters:
      - name: id
        in: path
        description: Identyfikator użytkownika.
        required: true
        schema:
          type: string
      responses:
        "200":
          description: Zwraca obiekt `UserResponse`.
          content:
            application/json:
              schema:
                type: object
  /@RequestMapping("/api/users")/:
    get:
      summary: "Wyszukuje użytkowników. Typowe kody odpowiedzi: 200."
      description: "Wyszukuje użytkowników. Typowe kody odpowiedzi: 200."
      operationId: UserController_search
      parameters:
      - name: q
```

```yaml
    in: query
    description: Fraza wyszukiwania.
    required: false
    schema:
      type: string
  - name: page
    in: query
    description: Numer strony.
    required: false
    schema:
      type: integer
      format: int32
  - name: size
    in: query
    description: Rozmiar strony.
    required: false
    schema:
      type: integer
      format: int32
  responses:
    "200":
      description: Zwraca obiekt `Object`.
      content:
        application/json:
          schema:
            type: object
post:
  summary: "Tworzy nowego użytkownika. Typowe kody odpowiedzi: 200, 400, 409."
  description: "Tworzy nowego użytkownika. Typowe kody odpowiedzi: 200, 400, 409."
  operationId: UserController_create
  requestBody:
    description: Dane użytkownika.
    content:
      application/json:
        schema:
          type: object
    required: true
  responses:
    "200":
      description: Zwraca obiekt `UserResponse`.
```

```
    content:
      application/json:
        schema:
          type: object
```

Dodać:
Dodać prosty parser klas DTO.
Wydobyć z każdej klasy pola (String name, int age, itp.) i dodać je do components/schemas.
Zamiast schema: object używać $ref: '#/components/schemas/NazwaKlasy'.

2. **wdrożenie modelu mT5 (text-to-text) do inteligentnego generowania dokumentacji**
   - Integracja modelu mT5 w mikroserwisie python-nlp przy użyciu biblioteki transformers (Hugging Face).
   - Model mT5 będzie przetwarzać dane w formacie:

Wejście:
"Komentarz: Zwraca użytkownika po ID. Parametr: id - identyfikator użytkownika."
Wyjście:
"Endpoint służy do pobierania danych użytkownika na podstawie jego identyfikatora. Jeśli użytkownik nie zostanie znaleziony, zwracany jest kod 404."

3. **Gotowy harmonogram Pracy Inżynierskiej**

Do 28.10: