**[Harmonogram](#harmon)**

**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).

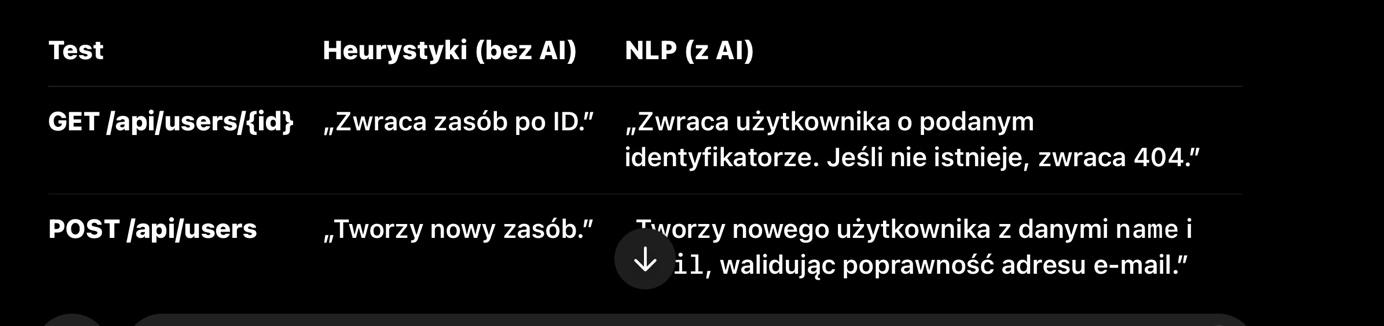
• web-ui/nginx.conf — proxy do /api, /v3, /swagger-ui, /nlp.

• 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 żródłowym w sposób zautomatyzowany i inteligentny, bez konieczności pisania tekstów przez człowieka.



|  |
| --- |
| 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.

1. 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*:

*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

*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'.

UPDATE *20.10.25*:

*openapi*: 3.0.1

*info*:

*title*: Project f76baebc5cc443f9a84dc3713598fcc9-API

*version*: 1.0.0

*paths*:

*/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

*additionalProperties*:

*type*: string

*/api/orders/{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

*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

*/api/orders/{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

*/api/users/{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*:

*$ref*: "#/components/schemas/UserResponse"

*/api/users*:

*get*:

*summary*: "Wyszukuje użytkowników. Typowe kody odpowiedzi: 200."

*description*: "Wyszukuje użytkowników. Typowe kody odpowiedzi: 200."

*operationId*: UserController\_search

*parameters*:

- *name*: q

*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*:

*$ref*: "#/components/schemas/CreateUserRequest"

*required*: ***true***

*responses*:

"*200*":

*description*: Zwraca obiekt `UserResponse`.

*content*:

*application/json*:

*schema*:

*$ref*: "#/components/schemas/UserResponse"

*components*:

*schemas*:

*UserResponse*:

*type*: object

*properties*:

*id*:

*type*: string

*name*:

*type*: string

*email*:

*type*: string

*CreateUserRequest*:

*type*: object

*properties*:

*name*:

*type*: string

*email*:

*type*: string

1. **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."

UPDATE *21.10.2025:*

Po objawach w YAML (podpisy typu “GET /…”, “Operacja …”, “Typowe kody odpowiedzi: .”) do finalnej specyfikacji trafia **fallback rule-based**, a nie teksty z mT5. Dzieją się dwa rzeczy naraz:

1. mT5 zwraca treści, ale „sanityzacja” przycina je zbyt agresywnie i robi z nich null, więc Java bierze fallback.

Funkcja „czyszcząca” (sanityzacja) po stronie Javy odrzucała całe zdania wygenerowane przez mT5, więc w kodzie lądowało null, a potem logika brała fallback rule-based.

2. W niektórych przebiegach mT5 potrafi dorzucić metakomentarz (np. „Instrukcja: …”), który wcześniej wycinam całkowicie, zamiast tylko posprzątać początek.

Poprawić żeby **wymusić użycie mT5** (gdy jest włączony) i nie „zjadać” jego wynik.

Błąd ładowania modelu w PyTorch/Transformers:

Cannot copy out of meta tensor; no data!

Please use torch.nn.Module.to\_empty() instead of torch.nn.Module.to()

when moving module from meta to a different device.

1. **Gotowy harmonogram Pracy Inżynierskiej**

**Harmonogram**

**Aplikacja:**

|  |  |  |
| --- | --- | --- |
| **Data** | **Zadanie** | **Wykonane** |
| **07.10.2025** | Analiza kodu i generowanie podstawowej dokumentacji | **Tak** |
| **14.10.2025** | Implementacja NLP do analizy opisów w specyfikacji OpenAPI i generowania rozszerzonej dokumentacji | **Tak** |
| **21.10.2025** | 1. Dodanie pełnej obsługi generowania dokumentacji na podstawie kodu źródłowego– w sytuacji, gdy projekt nie zawiera pliku openapi.yaml.  2. Wdrożenie modelu mT5 (text-to-text) do inteligentnego generowania dokumentacji  3. Gotowy harmonogram Pracy Inżynierskiej | **1. Tak**  **2. Nie działa**  **3. Tak** |
| **28.10.2025** | 1. Naprawienie i końcowa implementacja mT5  2. Zrobić 3 osobne pliki openapi.generated: 1. Bez opisów, 2. Z fallback base-rules, 3. Z użyciem modelu mT5  3. Dodać odczyt komentarzy (//, /\* \*/, /\*\* \*/) i zapisywać ich do EndpointIR  4. mT5+komentarzy  UPDATE:  5. Analiza i badania innych metod  6. Implementacja modelu | **1. Nie**  **2. Tak**  **3. Tak**  **4. Nie**  **5. Tak**  **6. Tak** |
| **04.11.2025** | 1. Generowanie 3 poziomy opisu (short, medium, long  2. przegląd dokumentacji web + pdf dokumentacja |  |
| **12.11.2025** | 1. Stworzenie dynamicznego interfejsu  2. Generowanie opisu całego projektu do dokumentacji, inerface dokumentacji |  |
| **18.11.2025** | 1. Personalizację bez logowania i bazy – całkowicie „anonimowo”, per-przeglądarka, z wykorzystaniem localStorage + cookie/sessionId i płaskich logów NDJSON |  |
| **25.11.2025** |  |  |

**Praca:**

|  |  |  |
| --- | --- | --- |
| **Data** | **Zadanie** | **Wykonane** |
| **02.12.2025** |  |  |
| **09.12.2025** |  |  |
| **16.12.2025** |  |  |
| **08.01.2026** |  |  |
| **15.01.2026** |  |  |

Do 28.10:

1. Naprawienie i końcowa implementacja mT5

2. Zrobić 3 osobne pliki openapi.generated: 1. Bez opisów, 2. Z fallback base-rules, 3. Z użyciem modelu mT5

3. Dodać odczyt komentarzy (//, /\* \*/, /\*\* \*/) i zapisywać ich do EndpointIR

4. mT5+komentarzy, Generowanie opisu komentarze za pomocą mT5

UPDATE *23.10.2025*

1. Aktualna implementacja działa technicznie (pipeline uruchamia model i zapisuje wyniki), jednak wygenerowane opisy są w dużej mierze niespójne i niezrozumiałe. Konieczne jest dalsze strojenie / dobór promptów, filtrowanie wyjść i walidacja jakości przed uznaniem tego wariantu za produkcyjny.

2. Zaimplementowano generowanie archiwum .zip z trzema osobnymi plikami dokumentacji:

1. openapi.plain.yaml – dokumentacja bez opisów.

2. openapi.rules.yaml – dokumentacja z opisami tworzonymi przez rules-base.

3. openapi.mt5.yaml – dokumentacja z opisami generowanymi przez mT5.

*openapi*: 3.0.1

*info*:

*title*: Project 9ef7280395e44a82b506daac96baee82-API

*version*: 1.0.0

*paths*:

*/hello*:

*get*:

*summary*: "moji,sじみ: -zostałymi o /,zostałotymi i o. a..."

*description*: '....”Example.. undefined>.. »cychdytuj."/>. ....'

*operationId*: HelloController\_hello

*parameters*:

- *name*: name

*in*: query

*description*: Parametr name.

*required*: ***false***

*schema*:

*type*: string

*responses*:

"*200*":

*description*: "....”: Parametr: name. Example."

*content*:

*application/json*:

*schema*:

*type*: object

*additionalProperties*:

*type*: string

*/api/orders/{id}*:

*get*:

*summary*: mojiärsk...akcident ...ł.-⸢kcident困りaćärsk...ärskkcident (ärsk...

*description*: იෝරlytteniu껍lytte –lytte껍ෝර껍lyttejaෝරlytte zda vulneru epłowniem

zda.

*operationId*: OrderController\_getOrder

*parameters*:

- *name*: id

*in*: path

*description*: Identyfikator zamówienia.

*required*: ***true***

*schema*:

*type*: string

*responses*:

"*200*":

*description*: waaaaульт껍⸢08(mærkkcident06(уланෝරуланෝර⸢lytteулан vulner vulner06(.

*content*:

*application/json*:

*schema*:

*type*: object

*delete*:

*summary*: "wa』(уланಿಪулан-улан (:gachοστάrzy. [...],..."

*description*: ი koske껍lytte –껍껍껍ෝරlytteулан껍껍lytte vulnerlytte - 06(껍ෝරෝර껍lytte

o ...

*operationId*: OrderController\_delete

*parameters*:

- *name*: id

*in*: path

*description*: Identyfikator zamówienia.

*required*: ***true***

*schema*:

*type*: string

*responses*:

"*200*":

*description*: walangsungaෝරуланулан pomar08(08( pomar pomarуланෝර⸢ vulner껍⸢ärsklytte:.

*content*:

*application/json*:

*schema*:

*type*: object

*/api/orders/{orderId}/items*:

*post*:

*summary*: moji -i - - z- opisz zwrot się inicznych. Zobaczy.

*description*: იуланlytte⸢уланෝරෝරෝර껍ෝරෝරக்கீෝරlytteෝරෝරlyttelyttelytte껍ෝරlytte껍lytteෝරlytte

koskelytte.

*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*: "waaaaулан껍⸢улан⸢улан vulner vulner08(08(уланlytte1)}08( 1)}к\

омпресс."

*content*:

*application/json*:

*schema*:

*type*: object

*/api/users/{id}*:

*get*:

*summary*: "moji -amiek - którego użytkownika po polsku...,.,. ,."

*description*: ი koskeෝරlyttenem困りlytte koskelytter idrулан id id...껍lytte.

*operationId*: UserController\_getById

*parameters*:

- *name*: id

*in*: path

*description*: Identyfikator użytkownika.

*required*: ***true***

*schema*:

*type*: string

*responses*:

"*200*":

*description*: waa–~$(уланулан08(-⸢улануланಿಪуланಿಪ08(-уланಿಪ⸢уланಿಪismissулан.

*content*:

*application/json*:

*schema*:

*$ref*: "#/components/schemas/UserResponse"

*/api/users*:

*get*:

*summary*: "moji;\"> . , ..cecece ..."

*description*: იулан vulnerуланෝර (ෝරෝරෝරlytte08(lyttelyttelytteෝරуланlyttelytte

koskelytter:lytteෝර08( vulnerlytte e困りlytte g.

*operationId*: UserController\_search

*parameters*:

- *name*: q

*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*: waiулан껍lyttelytte껍уланlyttelyttelytteக்கீ껍lytteෝරlyttelytteෝර껍껍껍nie.

*content*:

*application/json*:

*schema*:

*type*: object

*post*:

*summary*: moji;">.. z.

*description*: ი koske껍껍껍lytteje껍lytte:껍lytte)rodo껍껍ෝර껍껍مشخصاتlytte껍lytteулан껍lytte

-.

*operationId*: UserController\_create

*requestBody*:

*description*: Dane użytkownika.

*content*:

*application/json*:

*schema*:

*$ref*: "#/components/schemas/CreateUserRequest"

*required*: ***true***

*responses*:

"*200*":

*description*: waвзෝරуланlyttelytteнукле lyttelyttelytteуланр ©lyttelytteෝරуланнуклеเขียนเมื่อ.

*content*:

*application/json*:

*schema*:

*$ref*: "#/components/schemas/UserResponse"

*components*:

*schemas*:

*UserResponse*:

*type*: object

*properties*:

*id*:

*type*: string

*name*:

*type*: string

*email*:

*type*: string

*CreateUserRequest*:

*type*: object

*properties*:

*name*:

*type*: string

*email*:

*type*: string

UPDATE *24.10.2025*

3. Dodać odczyt komentarzy (//, /\* \*/, /\*\* \*/) i zapisywać ich do EndpointIR

Obecnie brane pod uwagę:

**głównie Javadoc (/\*\* ... \*/) bezpośrednio nad klasą/metodą/polem:**

* + @param -> parameters[].description
  + opis metody -> operation.description (czasem skrót do summary)
  + Javadoc klasy/pól DTO -> components.schemas.\*.description / properties.\*.description

Ignorowane: zwykłe komentarze // i /\* ... \*/.

Efekt w plikach:

* 1. openapi.plain.yaml – bez opisów.
  2. openapi.rules.yaml – opisy z Javadoc + reguły uzupełniające.
  3. openapi.mt5.yaml – opisy z mT5 (na razie „bełkot”, ale używa treści z IR/Javadoc).

Co uwzględnia generowanie:

* Adnotacje Spring (@GetMapping, @RequestParam, itp.) -> ścieżki, metody, parametry, required.
* Javadoc przy deklaracjach -> opisy operacji/parametrów/DTO.
* Bez Javadoc -> opisy są generyczne („Parametr q.”), kody odpowiedzi nie są wylistowane.

**Po implementacji zapisywania innych komentarze // oraz /\* ... \*/:**

Jeśli zebrać komentarz bezpośrednio nad deklaracją (np. nad metodą) i zapisać do IR:

* można użyć go jako fallback opisu, gdy brak Javadoc
* trafi do operation.description / parameters[].description analogicznie jak Javadoc

Komentarze // wpływają łagodniej: trafiają do pól pomocniczych, mogą być streszczane (rules/mT5), żeby nie „zalać” dokumentacji.

**Przepływ:**

1. Parser (JavaParser/Spoon) zbiera:

javadoc nad klasami/metodami/polami,

* 1. leadingComments (komentarze tuż nad deklaracją),
  2. wszystkie inlineComments z ciał metod (z lineNo, type).
  3. 2. IR wypełnia pola jak wyżej.
  4. 3. RULES:
  5. główny opis = javadoc || leadingComments
  6. inlineComments → streszczenie do notes, surowe → x-impl-notes
  7. TODO/FIXME → x-todos

// walidacja ilości

/\* sprawdź dostępność SKU w katalogu \*/

// TODO: dodać audit log

paths***:***

/api/orders/{orderId}/items***:***

post***:***

description***:*** "Dodaje pozycję do zamówienia. Typowe kody: 200."

x-impl-notes***:***

- "walidacja ilości"

- "sprawdź dostępność SKU w katalogu"

x-todos***:***

- "dodać audit log"

1 && 4. Implementacja modelu mT5

Dlaczego surowy mT5 okazał się złym dopasowaniem do mojego programu oraz jakie podejścia AI dadzą stabilny, ludzki opis API (z przykładami) przy zachowaniu jakości i kontroli:

**1. Dlaczego mT5 „nie gra” w moim use-case:**

**1. Nie jest instruction-tuned**

mT5 (google/mt5-small/base) był uczony głównie na rekonstrukcji brakujących fragmentów (span-corruption). Nie był szkolony na pary polecenie-> odpowiedź.

Efekt: na prośbę „opisz endpoint po polsku, dodaj przykłady” model nie ma silnej „intuicji” formatu/tonu.

**2. Wspólny, wielojęzyczny słownik != gwarancja jednego języka**

mT5 używa jednego SentencePiece dla wielu języków. Gdy sygnał „po polsku” jest za słaby, dekoder potrafi „zeskoczyć” w inne skrypty (cyrylica, znaki obce). Dlatego w logach widzimy mieszankę alfabetów. Instruction-tuned modele są zwykle bardziej „posłuszne” instrukcji dot. języka.

**3. Format wyjścia (JSON) i kontrola**

Chcę strukturalny output (opis, implNotes, examples). mT5, bez specjalnych ograniczeń, często łamie format. W moim serwisie włączyłam walidację - ta odcina śmieci -> zostaje pusty opis.

**Wniosek:** surowy mT5 nie daje gwarancji krótkiego, ludzkiego, polskiego opisu w stałym formacie. Dokładnie tego potrzebuję.

**5. Analiza i badania innych metod**

**Co się sprawdzi lepiej i dlaczego:**

**Model:** Mistral-7B-Instruct **albo** Llama-3.1-8B-Instruct

**Dlaczego:**

1. Wyższa jakość parafraz i przykładów

2. Bardzo dobre „instruction following”, sensowny polski

3. Dają krótkie, spójne „ludzkie” opisy i poprawne przykłady JSON.

**Rezygnacja z mT5**

* 1. Usuwam cały kod/konfigurację związaną z mT5 i Transformers w python-nlp.
  2. W CodeToDocsService enum: DescribeMode { PLAIN, RULES, AI } (zamiast MT5).
  3. Zmiana wywołania NLP: POST /describe?mode=ollama&strict=true.

**Nowy tryb „AI” (Ollama)**

* 1. python-nlp:/describe?mode=ollama woła **Ollamę** (/api/generate) i zwraca: mediumDescription, notes, examples (po walidacji JSON).
  2. **Brak fallbacku** do reguł w trybie AI (żeby widać było „czyste” AI).

openapi.ai.yaml – nowy plik wyjściowy

**Reguły bez zmian**

* 1. openapi.rules.yaml generowane jak dotąd
  2. openapi.plain.yaml bez opisów – bez zmian

**Docker**

Ollama uruchamiana natywnie na macOS (GPU/Metal), nie w Dockerze

* 1. python-nlp dostaje ENV: OLLAMA\_BASE\_URL=http://host.docker.internal:11434,
  2. OLLAMA\_MODEL=mistral:instruct (lub llama3.1:8b-instruct).
  3. Healthcheck python-nlp sprawdza /healthz.

|  |  |  |
| --- | --- | --- |
| Kryterium | Llama-3.1-8B-Instruct | Mistral-7B-Instruct |
| Posłuszeństwo instrukcjom / JSON | lepsze | dobre |
| Jakość polskiego, klarowność | bardzo dobra | dobra+ |
| Szybkość / zasoby (q4) | trochę wolniejsza/cięższa | lżejsza/szybsza |
| Stabilność w przykładach API | bardzo dobra | dobra |

|  |  |  |
| --- | --- | --- |
| **Kryterium** | **Llama-3.1-8B-Instruct (przez Ollama)** | **mT5 (Transformers)** |
| Typ modelu | LLM instruct-tuned (chat/komendy) | Seq2Seq do tłum./streszczeń (nie-instruct bez dodatkowego strojenia) |
| Trzymanie formatu (JSON) | Bardzo dobre – łatwo wymusić „zwróć tylko JSON” | Słabe/niestałe – skłonność do metatekstu i markerów <extra\_id\_…> |
| Jakość krótkich opisów PL | Wysoka (zwięzłe, „ludzkie” 1–3 zdania) | Zmienna; częściej „bełkot” bez dostrojenia |
| Przykłady (curl/response) | Stabilne i użyteczne, mniejsze halucynacje | Częste odchylenia i łamanie schematu |
| Uruchomienie na Mac (GPU) | Proste: Ollama + Metal, ollama pull/run | Złożone: PyTorch+MPS, wersje HF, cache, brak GGUF/Ollama |
| Offline / wdrożenie | Świetne: GGUF, lokalny serwer REST | Możliwe, ale cięższe (cache HF, zależności) |
| Parametry inference | Łatwe przez Ollamę (temperature, num\_predict, …) | Elastyczne, ale więcej „kabli” (Transformers/torch) |
| Zużycie zasobów | Q4 (int4) działa płynnie na M-serii | Często wolniej (CPU) lub wrażliwe na MPS |
| Integracja z Twoją architekturą | Idealna: REST do hosta z kontenerów, prosty glue w Pythonie | Więcej kodu i kruchości środowiska |
| Fallback/walidacja | Mniej potrzebny, ale i tak robimy walidację JSON | Konieczny agresywny filtr + fallbacki |
| Najlepsze zastosowanie u Ciebie | Finalne opisy/notes/przykłady do openapi.ai.yaml | Eksperymenty/badania; nie do produkcji w tym use-case |

macOS (M-serie, Metal):

* Docker Desktop na Macu uruchamia kontenery w maszynie wirtualnej z Linuxem (HyperKit/AppleHV).
* GPU Apple (Metal) nie jest „przepuszczany” do tej VM – Docker nie udostępnia akceleracji GPU dla kontenerów na macOS.
* Efekt: kontener z Ollamą na Macu działa CPU-only (wolniej). Dlatego zalecane Ollamę natywnie na hoście i tylko łączyć się do niej z kontenerów.

Linux (NVIDIA):

* Na Linuksie jest oficjalny NVIDIA Container Toolkit, który pozwala na passthrough GPU do kontenerów
* Kontener widzi sterowniki CUDA i może realnie używać GPU.
* Efekt: Ollama w Dockerze na Linuksie ma pełną akcelerację GPU.

Podsumowanie:

Mac: kontenery nie mają dostępu do GPU -> Ollama w kontenerze = CPU-only -> uruchamiaj Ollamę poza Dockerem (Metal).

**web (Nginx) – reverse proxy pod http://localhost:8080:**

* routuje do java-api i serwuje UI,
* zależy od zdrowia python-nlp i startu java-api.

**java-api (Spring Boot) – serwer główny:**

* parsuje kod (JavaParser → IR),
* generuje: openapi.plain.yaml, openapi.rules.yaml, openapi.ai.yaml,
* dla trybu AI woła python-nlp:/describe?mode=ollama.

**python-nlp (FastAPI) – mikroserwis NLP:**

* buduje prompt z IR (opis, parametry, typy, notatki),
* wywołuje Ollamę (POST /api/generate) z parametrami z ENV,
* waliduje JSON (schema: mediumDescription, notes[], examples),
* zwraca tylko poprawne dane (albo puste pola).

**Ollama (na HOŚCIE, poza Dockerem – macOS):**

* model: llama3.1:8b-instruct-q4 (domyślnie),
* endpoint: http://localhost:11434,
* Dockerowe serwisy łączą się przez http://host.docker.internal:11434.

**Kluczowe porty**

1. 8080 – Nginx (wejście z przeglądarki),

2. 8080 – java-api,

3. 8000 – python-nlp,

4. 11434 – Ollama (na hoście).

UPDATE *27.10.2025*

**6. Implementacja modelu**

**Dlaczego llama3.1:8b-instruct-q4:**

1. Dobra uległość instrukcjom i trzymanie formatu JSON (ważne przy schemacie mediumDescription/notes/examples).

2. Multijęzyk: polski działa stabilnie; styl neutralny, techniczny.

3. Lekka kwantyzacja Q4 -> mieści się w ~10–11 GB VRAM i daje przyzwoite opóźnienia (kilka sekund na odpowiedź) na M-serii.

4. Szerokie wsparcie w Ollamie i community -> łatwiejszy tuning (parametry, Modelfile, stop-sekwencje).

5. Konserwatywne domyślne zachowania (mniej „halucynacji” niż w wielu małych modelach).

**Alternatywy**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Model (Ollama tag)** | **Parametry** | **Plusy** | **Minusy / Ryzyka** | **Kiedy brać** |
| **mistral:7b-instruct** | 7B | Szybki, lekki, sprawny w instrukcjach | Czasem mniej stabilny JSON niż Llama 3.1 | Gdy liczysz każdą sekundę i chcesz bardzo szybki model |
| **qwen2.5:7b-instruct** | 7B | Bardzo dobry **multijęzyk**, często świetny w JSON | Styl bywa bardziej „werbalny” (trzeba trzymać na smyczy) | Gdy polski/słownictwo domenowe wypadają lepiej niż u Llamy |
| **gemma2:9b-instruct** | 9B | Ładny styl, dobre instrukcje | Trochę cięższy; na M4 może być wolniej | Gdy chcesz ciut „ładniejszy” język kosztem szybkości |
| **phi3.5:mini/medium-instruct** | 3–14B\* | Bardzo szybki **mini**, tani w RAM, często zaskakująco dobry | **Mini** mniej precyzyjny semantycznie | Gdy priorytetem jest szybkość i koszt, a opis ma być krótki |
| **mixtral:8x7b-instruct** | MoE | Mocny model (jakość 12–20B) | Za ciężki na M-Air; niepraktyczny lokalnie | Raczej nie na M-Air |
| **llama3.1:70b-instruct** | 70B | Jakość top-tier | Wymaga serwera/GPU – **nie** na M-Air | Tylko chmura/serwer z GPU |

**O kwantyzacji (q4 vs q5 vs q8)**

1. Q4: najmniejszy ślad pamięci, najszybszy, delikatnie gorsza jakość -> świetny do prototypu i CI.

2. Q5: kompromis – trochę lepsza jakość, nadal szybki, większe zużycie RAM/VRAM.

3. Q8/fp16: najwyższa jakość lokalnie, ale zbyt ciężkie na M-Air dla 8–9B bez „przytyków”.

*python-nlp*:

*build*: ./python-nlp

*image*: praca/python-nlp:dev

*container\_name*: python-nlp

*expose*:

- "8000"

*environment*:

*NLP\_MODE*: "ollama"

*OLLAMA\_BASE\_URL*: "http://host.docker.internal:11434"

*OLLAMA\_MODEL*: "llama3.1:8b-instruct-q4"

# Parametry generacji (bazowe)

*OLLAMA\_TEMPERATURE*: "0.3"

*OLLAMA\_TOP\_P*: "0.9"

*OLLAMA\_TOP\_K*: "60"

*OLLAMA\_REPEAT\_PENALTY*: "1.15"

*OLLAMA\_NUM\_CTX*: "4096"

*OLLAMA\_NUM\_PREDICT*: "256"

*NLP\_DEBUG*: "true"

*healthcheck*:

*test*: ["CMD", "python", "-c", "import urllib.request; urllib.request.urlopen('http://localhost:8000/healthz', timeout=3)"]

*interval*: 15s

*timeout*: 5s

*retries*: 20

*start\_period*: 45s

**1. OLLAMA\_TEMPERATURE (0.0–1.0)**

Steruje kreatywnością. Niższa = bardziej zachowawczo, mniejsze ryzyko błędów w JSON. 0.2–0.4 do dokumentacji. Zwiększę jeśli tekst jest zbyt suchy.

**2. OLLAMA\_TOP\_P (0–1)**

Filtr nucleus sampling – bierze tylko najbardziej prawdopodobne tokeny sumujące się do tego progu. 0.8–0.95. Mniejsze ->stabilniejszy, krótszy język.

**3. OLLAMA\_TOP\_K (liczba całkowita)**

Ogranicza rozważaną liczbę kolejnych tokenów do K. 40–100. Mniejsze -. bardziej deterministycznie, czasem ubogi język.

**4. OLLAMA\_REPEAT\_PENALTY (~1.0–1.3)**

Kara za powtarzanie n-gramów. 1.1–1.2 zmniejsza „zapętlanie” bez psucia sensu. 1.0 = brak kary.

**5. OLLAMA\_NUM\_CTX (kontekst, tokeny)**

Ile tokenów wejścia (prompt+historia) model „pamięta”. Większy kontekst = większe zużycie RAM/VRAM. 4096 to dobry start na MacBook Air. Jeśli pojawią się błędy pamięci, obniżyć do 3072/2048.

**6. OLLAMA\_NUM\_PREDICT (limit odpowiedzi, tokeny)**

Maksymalna długość generowanej odpowiedzi. Wprost wpływa na czas odpowiedzi. 200–300 wystarcza dla 2–3 zdań + przykładu JSON. Jeśli ucinane, podnieść; jeśli za długie, obniżyć.

**Implementacja modelu i wykorzystanie go do generowania dokumentacji**

**Co zostało zrobione**

**Zmiana podejścia do NLP:**

Zrezygnowałam z mT5 i wdrożyłam LLM przez Ollama (Llama 3.1 8B Instruct, wariant q4\_K\_M).

Ollama działa lokalnie na macOS (Metal), a moja usługa python-nlp (FastAPI) komunikuje się z nim po HTTP.

W docker-compose dodałam zmienne środowiskowe (MODEL, TEMPERATURE, itp.) i healthcheck.

**Nowy endpoint NLP i kontrakt JSON:**

W python-nlp zaimplementowałam endpoint /describe, który buduje prompt z danych IR (operationId, ścieżka, parametry, zwracany typ, notatki) i wymuszaodpowiedź w ściśle określonym schemacie JSON:

mediumDescription + notes[] + examples{ requests[curl], response{status, body} }.

Dodałam walidację Pydantic i „sanity checks” (np. przycinanie notatek, fallback rule-based gdy model zwróci coś niezgodnego ze schematem).

**Jaki jest wpływ na dokumentację**

**Więcej treści „dla człowieka”:** opisy są naturalne językowo, krótkie i rzeczowe (summary + 2–3 zdania description), a notatki implementacyjne są syntetycznymi punktami „na co uważać”.

**Przykłady użycia od razu w YAML:** w trybie AI dołączam przykłady cURL i przykładowe body odpowiedzi – tego nie było w prostym rules.

Openapi.ai.yaml:

*openapi*: 3.0.1

*info*:

*title*: Project 27a20c011f50408796cb91b09f41b3ee-API

*version*: 1.0.0

*paths*:

*/hello*:

*get*:

*summary*: Zwraca przywitanie.

*description*: Zwraca przywitanie. Opcjonalnie można podać imię użytkownika.

*operationId*: HelloController\_hello

*parameters*:

- *name*: name

*in*: query

*description*: "(opcjonalnie) imię użytkownika, np. /hello?name=Anna"

*required*: ***false***

*schema*:

*type*: string

*responses*:

"*200*":

*description*: OK

*content*:

*application/json*:

*schema*:

*type*: object

*additionalProperties*:

*type*: string

*example*:

*greeting*: Cześć Anna!

*x-impl-notes*:

- Ustaw domyślną wartość gdy brak name

- Prosta odpowiedź

- Trim logic

*x-request-examples*:

- /hello?name=Anna

*/api/orders/{id}*:

*get*:

*summary*: Pobiera zamówienie o podanym identyfikatorze UUID.

*description*: Pobiera zamówienie o podanym identyfikatorze UUID.

*operationId*: OrderController\_getOrder

*parameters*:

- *name*: id

*in*: path

*description*: identyfikator zamówienia (UUID)

*required*: ***true***

*schema*:

*type*: string

*responses*:

"*200*":

*description*: OK

*content*:

*application/json*:

*schema*:

*type*: object

*example*: {}

*x-impl-notes*:

- Sprawdź poprawność formatu UUID

- W przypadku braku zamówienia zwrócone zostanie 404 status

*x-request-examples*:

- GET /api/orders/123e4567-e89b-12d3-a456-426614174000

*delete*:

*summary*: Usuwanie zamówienia (soft delete) poprzez podanie identyfikatora zasobu.

*description*: "Usuwanie zamówienia (soft delete) poprzez podanie identyfikatora\

\ zasobu. Aby usunąć zamówienie, należy najpierw sprawdzić uprawnienia."

*operationId*: OrderController\_delete

*parameters*:

- *name*: id

*in*: path

*description*: Identyfikator zasobu.

*required*: ***true***

*schema*:

*type*: string

*responses*:

"*200*":

*description*: OK

*content*:

*application/json*:

*schema*:

*type*: object

*example*: {}

*x-impl-notes*:

- Użycie metody DELETE zamiast soft delete może spowodować trwałe usunięcie

danych.

- Należy pamiętać o sprawdzeniu uprawnień przed wykonaniem akcji usuwania.

*x-request-examples*:

- "curl -X DELETE 'https://example.com/api/orders/123' -H 'Authorization: Bearer\

\ token'"

*/api/orders/{orderId}/items*:

*post*:

*summary*: Dodaje pozycję do zamówienia.

*description*: Dodaje pozycję do zamówienia. Wymagane są identyfikator zamówienia

oraz kod produktu (SKU) lub ilość.

*operationId*: OrderController\_addItem

*parameters*:

- *name*: orderId

*in*: path

*description*: identyfikator zamówienia

*required*: ***true***

*schema*:

*type*: string

- *name*: sku

*in*: query

*description*: kod produktu (SKU)

*required*: ***false***

*schema*:

*type*: string

- *name*: qty

*in*: query

*description*: ilość (>0)

*required*: ***false***

*schema*:

*type*: integer

*format*: int32

*responses*:

"*200*":

*description*: OK

*content*:

*application/json*:

*schema*:

*type*: object

*example*: {}

*x-impl-notes*:

- Identyfikator zamówienia jest wymagany.

- Ilość musi być większa od zera.

- Sprawdzana dostępność kodu produktu (SKU).

*x-request-examples*:

- curl -X POST 'https://example.com/api/orders/ORD-123/items?sku=PROD-001&qty=2'

*/api/users/{id}*:

*get*:

*summary*: Zwraca użytkownika po podanym ID.

*description*: Zwraca użytkownika po podanym ID.

*operationId*: UserController\_getById

*parameters*:

- *name*: id

*in*: path

*description*: identyfikator użytkownika

*required*: ***true***

*schema*:

*type*: string

*responses*:

"*200*":

*description*: OK

*content*:

*application/json*:

*schema*:

*$ref*: "#/components/schemas/UserResponse"

*example*:

*id*: string

*name*: string

*email*: string

*x-impl-notes*:

- Użytkownik musi istnieć w bazie danych.

- W przypadku błędu zwracany jest kod HTTP 404.

*x-request-examples*:

- GET /api/users/123

*/api/users*:

*post*:

*operationId*: UserController\_create

*requestBody*:

*description*: dane użytkownika

*content*:

*application/json*:

*schema*:

*$ref*: "#/components/schemas/CreateUserRequest"

*required*: ***true***

*responses*:

"*200*":

*description*: OK

*content*:

*application/json*:

*schema*:

*$ref*: "#/components/schemas/UserResponse"

*components*:

*schemas*:

*UserResponse*:

*type*: object

*properties*:

*id*:

*type*: string

*name*:

*type*: string

*email*:

*type*: string

*CreateUserRequest*:

*type*: object

*properties*:

*name*:

*type*: string

*email*:

*type*: string

**Co mam na teraz**

Na podstawie wygenerowanego **openapi.ai.yaml**:

Dla **/hello** – naturalny summary/description, notatki, oraz prosty przykład wywołania (x-request-examples).

Dla **GET /api/orders/{id}** – opis z akcentem na UUID, notatki o walidacji i 404, przykład cURL.

Dla **DELETE /api/orders/{id}** – zwięzły opis soft-delete + notatki o uprawnieniach i charakterze operacji.

Dla **POST /api/orders/{orderId}/items** – jasne wymagania (qty > 0, SKU wymagane), **status 201** i przykład cURL.

Dla **GET /api/users/{id}** – opis + warunek istnienia w bazie, przykłady wywołań.

Dla **POST /api/users** – uproszczone odpowiedzi 200 w tej próbce;

**Różnice: openapi.ai.yaml vs openapi.rules.yaml**

|  |  |  |
| --- | --- | --- |
| **Obszar** | **openapi.rules.yaml (reguły)** | **openapi.ai.yaml (LLM/Ollama)** |
| Styl opisów | Sztywne, mechaniczne zdania („Typowe kody: …”). | Naturalny, zwięzły, „ludzki” opis dla developera. |
| Notatki | Często przeniesione żywcem z komentarzy; mogą być długie. | Zsyntetyzowane 3 krótkie punkty w x-impl-notes. |
| Przykłady | Zazwyczaj brak lub bardzo ogólne. | Konkretne curl w x-request-examples, plus example body w odpowiedzi. |
| Statusy | Domyślnie 200 (mało kontekstu). | Może pojawić się 201/404 (zależnie od operacji i promptu). |
| Język | Często powtarzalne frazy, mało kontekstu. | Kontekstowy opis (np. UUID, walidacje), krótkie i czytelne. |
| Źródło treści | Heurystyki i proste reguły. | Model językowy, który „parafrazuje” i uzupełnia na podstawie IR i promptu. |

Do **04.11.2025:**

**1. Największe problemy do poprawy**

1. Placeholdery w opisach – model wstawił szablon „string (1–3 zdania, po polsku, zwięźle)”, np. w profilach i setupie. To sygnał, że prompt nie został w pełni wypełniony lub walidator dopuścił placeholder.

2. Powtórzenia i dublujące się opisy — „Zwraca …” podwójnie (summary=description), np. /api/chat/{chatId} i inne.

3. Parametry w złych miejscach — pola typu request, avatarFile trafiają jako query, gdy realnie powinny być w requestBody (application/json). Np. /api/profile/setup ma request jako query i osobno body — to niespójne

3. Niejednoznaczne/„puste” odpowiedzi — masa odpowiedzi 200 OK {} bez schematów lub przykładu, nawet tam gdzie semantycznie powinno być 201 lub 204. Np. wiele POST zwraca „201 OK { }” bez modelu.

4. Język i format w przykładach — łamanie linii, ucieczki znaków w curl (np. \n, \") i „Ksi##ka” → problemy tokenizacji/escapingu.

5. Brak sekcji bezpieczeństwa — brak securitySchemes i security na ścieżkach, mimo że są wzmianki o autoryzacji (Bearer). Np. przykłady używają Authorization: Bearer token, ale spec nie ma definicji.

**Implementacja AiPostProcessor.java:**

Co dokładnie robi (automatycznie, po wygenerowaniu openapi.ai.yaml):

1. Wycina placeholdery typu string (1–3 zdania…). Gdy je wykryje, podmienia summary/description na wersje z fallbacku (rules/Javadoc).

2. Gdy summary == description, skraca summary do pierwszego zdania.

3. Jeśli operacja ma requestBody, usuwa zdublowane query paramy (request, payload, body, dto, file, avatar, …) – dane zostają tylko w body.

4. Normalizuje przykłady curl: od-escape’owuje \n, \" i składa wielolinijkowe polecenia z \.

5. Porządkuje statusy: dla POST preferuje 201, dla DELETE dopuszcza 204 bez body.

**2. Generowanie 3 poziomy opisu (short, medium, long**

**3. przegląd dokumentacji web + pdf dokumentacja**