Harmonogram

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
```

```
    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
```

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."

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.
```

3. 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. Naprawienie problemów w generowaniu 2. Generowanie 3 poziomy opisu (short, medium, long 3. przegląd dokumentacji web + pdf dokumentacja	**1. Nie** **2. Tak** **3. Tak**
12.11.2025	1. Naprawienie problemów w generowaniu 2. Stworzenie dynamicznego interfejsu 3. Generowanie opisu całego projektu do dokumentacji, inerface dokumentacji	**1. Tak** **2. Nie** **3. Tak**
18.11.2025	1. Security 2. Stworzenie dynamicznego interfejsu 3. Możliwość edytowania pliku przed go pobraniem	**1. Tak** **2. Tak** **3. Tak**
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.

```yaml
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:
```

```yaml
    summary: mojiärsk...akcident ...ł.-ʳkcident困り aćärsk...ärskkcident (ärsk...
    description: ο๑◌ʃơlytteniu껍lytte –lytte껍๑◌ʃơ껍lytteja๑◌ʃơlytte zda vulneru epłowniem
      zda.
    operationId: OrderController_getOrder
    parameters:
    - name: id
      in: path
      description: Identyfikator zamówienia.
      required: true
      schema:
        type: string
    responses:
      "200":
        description: waaaaульт껍ʳ08(mærkkcident06(улан๑◌ʃơулан๑◌ʃơʳlytteулан vulner vulner06(.
        content:
          application/json:
            schema:
              type: object
  delete:
    summary: "wa』(улан◌ʃ๓улан-улан (:gachoστάrzy. [...],..."
    description: ο koske껍lytte –껍껍껍๑◌ʃơlytteулан껍껍lytte vulnerlytte - 06(껍๑◌ʃ๑σʃơ껍lytte
      o ...
    operationId: OrderController_delete
    parameters:
    - name: id
      in: path
      description: Identyfikator zamówienia.
      required: true
      schema:
        type: string
    responses:
      "200":
        description: walangsunga๑◌ʃơуланулан pomar08(08( pomar pomarулан๑◌ʃơʳ vulner껍ʳärsklytte:.
        content:
          application/json:
            schema:
              type: object
/api/orders/{orderId}/items:
  post:
    summary: moji -i - - z- opisz zwrot się inicznych. Zobaczy.
```

```yaml
    description: оуланlytteгулан◌ꣿ◌ꣿ◌ꣿ◌ꣿ껍◌ꣿ◌ꣿ◌ꣿ₺₹◌ꣿ◌lytte◌ꣿ◌ꣿ◌ꣿlyttelyttelytte껍◌ꣿ◌lytte껍lytte◌ꣿ◌lytte
      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: "waaaaулан껍гулангулан 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: ο koske◌ꣿ◌lyttenem困ツ lytte koskelytter idгулан id id...껍lytte.
      operationId: UserController_getById
      parameters:
      - name: id
        in: path
        description: Identyfikator użytkownika.
```

```yaml
        required: true
        schema:
          type: string
      responses:
        "200":
          description: waa–~$(уланулан08(-ʳуланулан◌ိဃулан◌ိဃ08(-улан◌ိဃʳулан◌ိဃismissулан.
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/UserResponse"
  /api/users:
    get:
      summary: "moji;\"> . , ..cecece ..."
      description: оулан vulnerулан๑◌ɗσ (๑◌ɗ๑σɗ๑σɗσlytte08(lyttelyttelytte๑◌ɗσуланlyttelytte
        koskelytter:lytte๑◌ɗσ08( 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: waiулан껍lyttelytte껍уланlyttelyttelytteஃஃ껍lytte๑◌ɗσlyttelytte๑◌ɗσ껍껍껍nie.
```

```yaml
        content:
          application/json:
            schema:
              type: object
    post:
      summary: moji;">.. z.
      description: ი koske껍껍껍lytteje껍lytte:껍lytte)rodo껍껍ಿ್ರ껍껍ت خصمشlytte껍lytteулан껍lytte
        -.
      operationId: UserController_create
      requestBody:
        description: Dane użytkownika.
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/CreateUserRequest"
        required: true
      responses:
        "200":
          description: wавзಿ್ರуланlyttelytteнукле lyttelyttelytteуланр ©lyttelytteಿ್ರуланнуклеเขียนเมื่อ.
          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,
leadingComments (komentarze tuż nad deklaracją),
wszystkie inlineComments z ciał metod (z lineNo, type).
2. IR wypełnia pola jak wyżej.
3. RULES:
główny opis = javadoc || leadingComments
inlineComments → streszczenie do notes, surowe → x-impl-notes
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
Usuwam cały kod/konfigurację związaną z mT5 i Transformers w python-nlp.
W CodeToDocsService enum: DescribeMode { PLAIN, RULES, AI } (zamiast MT5).
Zmiana wywołania NLP: POST /describe?mode=ollama&strict=true.
Nowy tryb „AI” (Ollama)
python-nlp:/describe?mode=ollama woła **Ollamę** (/api/generate) i zwraca: mediumDescription, notes, examples (po walidacji JSON).
Brak fallbacku do reguł w trybie AI (żeby widać było „czyste” AI).
openapi.ai.yaml – nowy plik wyjściowy
Reguły bez zmian
openapi.rules.yaml generowane jak dotąd
openapi.plain.yaml bez opisów – bez zmian
Docker
Ollama uruchamiana natywnie na macOS (GPU/Metal), nie w Dockerze
python-nlp dostaje ENV: OLLAMA_BASE_URL=http://host.docker.internal:11434, OLLAMA_MODEL=mistral:instruct (lub llama3.1:8b-instruct).
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 wymusza odpowiedź 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:

```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
```

```yaml
    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.

UPDATE *01.11.2025*

1. Naprawić problemy w generowaniu

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

Tworzenie personalizowanych wersji dokumentacji, w zależności od poziomu doświadczenia użytkownika.

Jak to zostało zrealizowane

1. Wejście poziomu z UI -> Java

W CodeToDocsService.generateYamlFromCode(..., level, ..., mode) poziom trafia do:

- normalizeAudience(level) -> beginner / intermediate / advanced;
- dopisujemy go do metadanych specyfikacji i operacji: x-user-level (na api.info i każdej Operation).

2. Java -> NLP (Ollama)

W callNlp(..., mode, level) poziom jest przekazywany jako audience do /describe.

AI zwraca ustrukturyzowane: mediumDescription, notes, examples (z requestami cURL i przykładową odpowiedzią).
3. NLP (FastAPI app.py) dobiera prompt per poziom
build_prompt_beginner / build_prompt_intermediate / build_prompt_advanced (funkcje w app.py) — każdy ma inne wymagania co do stylu, liczby przykładów cURL i zawartości notes.
4. Twarde reguły w SCHEMA_TEXT wymuszają m.in. brak placeholderów i poprawny JSON; DELETE/void -> 204.
Składanie OpenAPI + sanity pass
Java wstawia opis, notatki i przykłady do Operation (w tym x-request-examples).
5. AiPostProcessor i finalSanity(...) czyszczą duplikaty, przenoszą BODYish do requestBody, normalizują statusy (np. POST->201 gdy „puste 200", DELETE→204), walidują/formatują cURL i dopinają schematy.
6. PDF (renderer)

Jak to działa „w praktyce"

1. Generuję YAML trzy razy dla różnych level (beginner, intermediate, advanced).
2. Każdy run podaje inny prompt do Ollamy, więc różnią się opisy, notatki i liczba/forma przykładów.
3. Na końcu wszystkie trzy wersje przechodzą te same twarde sanity-reguły (statusy 2xx, body vs query, ujednolicone cURL, itp.), więc kontrakt HTTP pozostaje spójny niezależnie od poziomu.

Różnice między 3 dokumentami

Początkujący: prostsze objaśnienia, przykładowe payloady/odpowiedzi wprost, mniej żargonu.
Średniozaawansowany: zwięźlej i bardziej technicznie, często z notatkami implementacyjnymi i bez nadmiaru narracji.
Zaawansowany: najpełniejsza precyzja (kody statusów, przykładowe ciała odpowiedzi „Created"), doprecyzowane opisy.

3. Przegląd dokumentacji web + pdf dokumentacja

Co jest zrobione

PDF (download): POST /api/projects/{id}/docs/pdf — zwraca plik z nagłówkiem Content-Disposition: attachment (pobieranie).
Web-podgląd PDF: GET /api/projects/{id}/docs/pdf/view?level=... — generuje PDF (gdy brak) i zwraca z Content-Disposition: inline oraz Content-Type: application/pdf, otwiera się w przeglądarce.

UPDATE *03.11.2025*

1. Naprawić problemy w generowaniu

Priorytet P0 - kontrakt musi być „prawdziwy"

1. Poprawność i kompletność schematów (blokery dla SDK)

Usunąć placeholders typu ? i „gołe" array bez items.

- Każda odpowiedź ma konkretny $ref lub jasno zdefiniowany obiekt.

- Szybki przegląd nazw: DEFAUL_AVATAR -> DEFAULT_…, Anonimous… -> Anonymous… (spójnie w modelach i polach).

DoD: OpenAPI przechodzi walidator bez ostrzeżeń, generator klientów (np. TS/Java) buduje się bez ręcznych łat.

UPDATE *04.11.2025*

2. Kody HTTP zgodne z semantyką

- login i wyszukiwarki -> 200 OK.
- Tworzenie zasobu -> 201 Created (+ Location gdy sensowne).
- Akcje „add/remove" bez nowego bytu -> 200/204.
- Nie mieszać GET+query z POST+body w tym samym endpointzie: wybrać jeden wzorzec.

DoD: Każda operacja ma uzasadniony kod sukcesu i jednolity wzorzec (GET=filtry w query, POST=filtry złożone w body).

Do 12.11.2025:

1. Naprawienie problemów w generowaniu
2. Stworzenie dynamicznego interfejsu
3. Generowanie opisu całego projektu do dokumentacji, inerface dokumentacji

UPDATE *05.11.2025*

Priorytet P1 - przewidywalne błędy i auth

1. Globalny model błędu + systematyczne 4xx/5xx

- Dodać components.schemas.ApiError { code, message, details? }.
- Do każdej operacji przypiąć co najmniej: 400, 401, 403, 404, 409/422, 429, 5xx z $ref: ApiError.

DoD: Każdy endpoint ma sekcję błędów; przykładowe payloady błędów są w docs.

2. Security Schemes (JWT) + przypięcie do operacji

- components.securitySchemes.bearerAuth (HTTP bearer, JWT).
- Globalne security: [{ bearerAuth: [] }], a wyjątki (public) odpinać per-endpoint.

DoD: Ścieżki wymagające logowania są oznaczone; publiczne są jawnie bez security.

UPDATE *06.11.2025*

Priorytet P2 - ergonomia klienta

1. Paginacja/filtrowanie jako reużywalne komponenty

- PageableParams (page, size, sort) i PageResponse<T> { content, page, size, totalElements, totalPages }.
- Wymusić limity (size: 1..100) i przykładowe sort.

DoD: Każda lista używa tych samych parametrów i tej samej odpowiedzi.

2. Przykłady (examples) dla request/response

- Minimum: 1 „happy path" per operacja.

DoD: Każdy endpoint ma co najmniej 1 example, a kluczowe — 2–3.

UPDATE *07.11.2025*

Problemy, które występują do tego czasu:

1. Artefakty kodowania: dost#pu, b##d, przechodz# itd. — tego nie może być w produkcie. openapi.ai_advance
2. Bardzo generyczne opisy:
4. /api/auth/login i /api/auth/register opisane jako „Utwórz nowy zasób” zamiast „Zaloguj użytkownika” / „Zarejestruj nowego użytkownika”. openapi.ai_advance
3. Niespójności:
5. Endpoint .../anonymous w różnych poziomach ma inne opisy / inne kody lub dziwnie brzmiące teksty (np. 200 + void vs 204 + opis jakby coś zwracał).
6. Dla publicznych endpointów (login/register) pojawia się 401 z opisem „Wymagany token dostępu”, co jest mylące.
4. Przykłady { } zamiast realnych payloadów — zabijają wartość dodaną Twojego narzędzia.
5. Brakuje wyraźnych „notes” / komentarzy architektonicznych, które rzekomo generuje NLP — w PDF-ach ich praktycznie nie widać jako osobne, wyróżnione sekcje.

Poziom BEGINNER

Co nie gra dla początkującego:

1. Za trudny język jak na „beginner”:

- Suchy, skrótowy styl: „Utwórz nowy zasób”, „Pobierz listę zasobów” — początkujący nie wie *jaki zasób*, do czego to w projekcie służy.
- Brak wyjaśnienia pojęć typu: JWT, bearerAuth, pagination, sort, PageResponse.

2. Za dużo szumu błędów:

Komplet wszystkich 400/401/403/404/409/422/429/500 przy każdym endpoincie to jest overload.

Dla poziomu beginner można:

- zostawić 2–4 najważniejsze kody z prostym, ludzkim opisem;
- resztę skrócić lub podać w tabelce „częste błędy”.

3. Brak konkretnych przykładów użycia:

Dla poziomu beginner powinna być:

- narracja: „Ten endpoint służy do X w Twojej aplikacji (np. pobrania czatu użytkownika)”;
- gotowy przykład curl z komentarzem linijka po linijce;
- mini-scenariusz: „Najpierw zarejestruj, potem zaloguj, potem wywołaj /me”.

Poziom MEDIUM

Minusy:

1. **Za mała różnica względem Beginner / Advanced:**

Ten poziom to w praktyce to samo co advanced, tylko z napisem „Poziom: średniozaawansowany”.

- Brak charakterystycznych elementów:
- brak rozbudowanych przykładów z paginacją, sortowaniem, filtrami,
- brak uwag typu: „użyj 404 zamiast pustej listy?” lub „paginated response ma pola page/size/totalElements”.

2. **Niespójności i bugi w treści:**

W niektórych miejscach opisy nie zgadzają się z tym, co faktycznie robi endpoint (np. anonymous jako „zwraca listę użytkowników” przy schema void lub 204). openapi.ai_medium

Poziom ADVANCED
Minusy:
Brakuje:

- uwag o idempotencji (DELETE, PUT),
- semantyki paginacji (maks. size, domyślne sort),
- szczegółów walidacji (VALIDATION_ERROR – konkretne reguły),
- uwag o bezpieczeństwie (np. kto może wywołać, czy token musi mieć rolę X),
- edge cases, rate limiting, concurrency hints.

Powielone błędy opisów jak w innych poziomach:
login/register opisane jak zwykłe „Utwórz zasób”.
public endpoints + 401 z tekstem „Wymagany token dostępu” – nieprecyzyjne.
Brak dodatkowej wartości eksperckiej:
Dla seniora moje narzędzie mogłoby:
sugerować konwencje (naming, statusy),
zaznaczać niespójności między endpointami,
podkreślać wymagania kontraktowe (np. „ten endpoint zwraca 204 przy sukcesie, nigdy nie zwraca ciała”).

Co konkretnie poprawić w implementacji

5.1. Różnicowanie poziomów – target outcome

Beginner:
Zasada: *„Pomóż juniorowi zrozumieć, co to jest, krok po kroku”*.
Dla każdego endpointu:

7. 1–2 zdania: co robi i w jakim scenariuszu biznesowym.
8. 1 prosty przykład curl (bez zaawansowanych nagłówków).
9. 2–3 główne kody odpowiedzi (200/201, 400, 401/403, 404) wyjaśnione ludzkim językiem.
10. Mini-słowniczek: JWT, bearerAuth, pagination — *opcjonalnie globalnie*, nie przy każdym endpointcie.
11. Usunąć szum (pełne macierze błędów możesz mieć w sekcji globalnej).

Intermediate:
Zasada: *„Dev już zna REST, ale nie zna Twojego API”*.
Dla każdego endpointu:

- Konkretny opis domenowy (np. „Zwraca listę powiadomień zalogowanego użytkownika posortowaną po dacie utworzenia”).

- 1–2 przykłady curl, w tym z paginacją / sortowaniem.
- Pełniejsza lista statusów, ale z krótkimi technicznymi komentarzami („422 – gdy walidacja domenowa nie przeszła (np. zbyt długi tytuł książki)").
- Krótkie „notes" z dobrymi praktykami: jak używać paginacji, jakie są ograniczenia.

Advanced:
Zasada: *„Senior, który chce kontraktów, edge cases i spójności"*.
Dla każdego endpointu:

- Bardzo precyzyjny opis techniczny (idempotency, side effects, constraints).
- Wszystkie statusy z konkretnymi warunkami.
- Jeśli to możliwe: informacje o limitach, concurrency, wersjonowaniu, schematach błędów.
- Zero tłumaczenia podstaw typu „co to jest JWT".
- Dodatkowe „Implementation notes" / „Integration tips".

Fixy techniczno-stylistyczne

Naprawić encoding – w FastAPI/LLM wymusić plain UTF-8 i filtrować # artefakty.
Semantyka opisów:

- Login: „Uwierzytelnia użytkownika i zwraca token JWT".
- Register: „Tworzy nowe konto użytkownika".
- „Utwórz nowy zasób" zostawić dla faktycznych create'ów domenowych, nie auth.

Spójność security:

- Dla endpointów Public (no auth) NIE pisać w 401 „Wymagany token dostępu".
- Jeśli 401 jest dokumentowane przy loginie – opisać: „Nieprawidłowe dane logowania".

Lepsze przykłady:

- Zastąpić { } realnymi payloadami zgodnymi ze schematem.

Spójny opis kontrowersyjnych endpointów:

- /api/profile/anonymous – zdecydować: lista? 204? Zwraca co? Ujednolić we wszystkich poziomach.

Włączyć widoczne „notes" per poziom:
W promptach wymusić sekcję np.:

- notesForBeginner
- implementationNotes
- advancedConsiderations

I potem w Java ładnie je wyrenderuj jako bloki tekstu w PDF.

UPDATE *08.11.2025*

1. Opisy endpointów (poziom beginner)
Przed:
Opisy były:

- bardzo krótkie, często generyczne („Zwraca przywitanie…", „Pobiera szczegóły zamówienia o podanym ID."), czasem powielone w summary/description.

- mało „junior-friendly” – brakowało kontekstu biznesowego („po co użyć tego endpointu”).

Brak lub szczątkowe przykłady:

- brak spójnych przykładów cURL,
- response examples często {} albo wcale,
- brak jednoznacznej polityki dla 201/204.

W treści mogły pojawiać się placeholdery / słabe jakościowo teksty z AI.

Po:

Zmieniony app.py wymusza na modelu:

- generowanie sensownych, pełnych opisów (mediumDescription)
- prosty, scenariuszowy język pod juniora (co robi endpoint, w jakiej sytuacji go użyć),
- twarde zasady: brak placeholderów, brak fantazji spoza wejścia, poprawne kody (GET-> 200, POST -> 201, DELETE/void -> 204).

CodeToDocsService:

wstrzykuje wygenerowane przez model opisy i przykłady bez modyfikowania ich treści merytorycznej,

dla każdego endpointu dokłada:

- spójny przykład cURL (lub używa tego z modelu),
- przykładowe body request/response w oparciu o schematy,
- standardowe odpowiedzi błędów oparte na ApiError.

CodeToDocsService

W PDF dla beginnera:

- endpointy mają czytelny opis + tabelki parametrów + konkretne przykłady,
- są wyświetlane tylko kluczowe kody (200/201/204, 400, 401, 403, 404), dzięki czemu dokument nie przytłacza początkującego.

2. Standardowe błędy i spójny model ApiError

Przed:

Obsługa błędów była mniej spójna, brak wyraźnej, jednolitej tabeli dla beginnera.

Po:

Dodano centralny schemat ApiError i mechanizm dokładania standardowych odpowiedzi 4xx/5xx do operacji.

W PDF beginner:

- pojawia się globalna tabela standardowych kodów błędów z opisem w ludzkim języku,
- w sekcji Components opisany jest format ApiError.

3. Spójność techniczna i bezpieczeństwo treści

Dodatkowo:

Wymuszone:

- zwracanie czystego JSON z NLP (bez Markdown, komentarzy),
- odrzucanie placeholderów,
- poprawne mapowanie typów (204 dla DELETE/void itp.).
- Dodane sanity-checki i fallbacki:
- jeśli model coś zepsuje, generowany jest sensowny deterministyczny opis,

- ale gdy JSON jest poprawny – treść modelu wchodzi do OpenAPI/PDF bez ręcznego przepisywania.

UPDATE *10-14.11.2025*

Poziomy dokumentacji: zrezygnowałam z trzech wariantów -> zostawiłam tylko beginner i advanced.
Dwa poziomy pokrywają realne potrzeby użytkowników (junior vs. senior), upraszczają UX i znacząco obniżają koszt utrzymania oraz liczbę błędów bez utraty wartości merytorycznej.
Dlaczego to lepsze:
Mniej złożoności = mniej błędów.
Czytelny podział odbiorców. Beginner -> skrót i kontekst; Advanced -> precyzja techniczna. Trzeci wariant nie miał wyraźnie innej persony.
Niższy koszt utrzymania. Mniej warunków w promptach i rendererze (mniej testów, mniej regresji).
Mniej decyzji po stronie użytkownika. Prostszy interfejs -> mniej pytań typu „który poziom wybrać?".
Łatwiejsze QA. Dwie ścieżki jakości zamiast trzech - łatwiej porównać i standaryzować wynik.

Java – kontrolery:
- ProjectDocsFromCodeController: generowanie YAML i PDF z kodu; dodane endpointy pod podgląd i pobieranie YAML (inline + download).
- ProjectsController: bez zmian w logice uploadu; czyszczenie komunikatów i statusów.

Java – PdfDocService:
- uproszczony nagłówek i badge poziomu (wykorzystanie jednej metody/metadanych poziomu),
- w renderOp(...) usunięte „inteligentne" dopiski (fallbacki, filtry beginner) dla summary/description – teraz drukujemy tylko to, co jest w OpenAPI,
- pozostawione neutralne renderowanie parametrów, request body, odpowiedzi, security i schemas,
- drobne porządki: helpery schemaToLabel, paramConstraints – bez generowania treści „z powietrza".

Frontend (React):
- dodane przyciski „Podgląd YAML" i „Pobierz YAML" obok PDF,
- drobne UX (status/timer)

Python NLP (FastAPI + Ollama):
- usunięte wszystkie fallbacki treści: brak rule_based, brak automatycznych opisów parametrów,
- uproszczone _build_param_docs – tylko to, co przyszło w IR (zero dopisków),
- wyrzucone SCHEMA_TEXT – wymagany schemat JSON wbudowany bezpośrednio w prompty build_prompt_beginner/advanced,
- walidacja _validate_ai_doc: wymagany mediumDescription; notes sanitizowane; examples akceptowane tylko jako słownik; brak dopisywania czegokolwiek,
- call_ollama(...): wycinanie pierwszego bloku JSON z odpowiedzi;

Efekt
Dokumentacja (YAML/PDF) wiernie odzwierciedla dane z kodu i z AI bez żadnych automatycznych dopisków.
Kod uproszczony.

Do 18.11.2025:

1. Security
2. Stworzenie dynamicznego interfejsu
3. Możliwość edytowania pliku przed go pobraniem

UPDATE *15.11.2025*

1. Security

Z punktu widzenia dokumentacji potrzebujemy głównie:

- Czy endpoint jest publiczny, czy wymaga auth.
- Jeśli wymaga auth – jakiego typu:
  - HTTP Bearer (JWT),
  - session / cookie,
  - basic auth,
  - coś custom?
- Ewentualnie: jakiekolwiek info o rolach (hasRole("ADMIN"), hasAuthority("SCOPE_read") itp.).

Na poziomie OpenAPI można to wyrazić jako:

- components.securitySchemes – np. bearerAuth, sessionCookie, basicAuth.
- security globalne – np. [{ "bearerAuth": [] }].
- operation.security – dla wyjątków (public endpoint → [], inny scheme → inna nazwa).

Celem jest: zbudować te 3 rzeczy na podstawie kodu Spring Security.

Robimy:

1. szukamy metod zwracających SecurityFilterChain,
2. z ciała metody wyciągamy:
   - requestMatchers(...), antMatchers(...), mvcMatchers(...),
   - to, czy jest permitAll(), authenticated(), hasRole(...), hasAnyRole(...), itd.,
   - anyRequest()…. jako globalny fallback,
   - wykrywamy mechanizm auth:
   - .oauth2ResourceServer().jwt() → traktujemy jako **Bearer JWT**,
   - .httpBasic() → basic,
   - .formLogin() / .sessionManagement() → session.

To jest realne czytanie konfiguracji, nie zgadywanie po nazwach endpointów.
Nie obiecuję:
że ogarniemy każdy możliwy, kreatywny kod typu:

- endpointMatcherProvider.openEndpoints() zwracający dynamicznie listę,

konfigurację w innych modułach, do których parser nie zagląda,
szczegółowe reguły pisane przez custom DSL-e,
że zrozumiemy całą logikę Spring Security 1:1 tak jak runtime.

1. Spring Security (JavaSecurityParser):
Parser skanuje wszystkie pliki .java w projekcie.
Szuka metod zwracających SecurityFilterChain i analizuje ich ciało.
Na tej podstawie wykrywa:

- mechanizm uwierzytelniania (BEARER_JWT, BASIC, SESSION, OTHER, NONE),
- reguły autoryzacji z authorizeHttpRequests():
  requestMatchers(...).permitAll(),
  requestMatchers(...).authenticated(),
  requestMatchers(...).hasRole("...") / hasAuthority("..."),
  anyRequest().authenticated() jako fallback.

Reguły są zapisywane jako SecurityRule (metoda HTTP, pattern ścieżki, typ reguły, role).

2. Budowanie security w OpenAPI (CodeToDocsService)
Na podstawie SecurityModel generator tworzy:

- components.securitySchemes (np. bearerAuth dla JWT, basicAuth dla HTTP Basic),
- globalne security (np. - bearerAuth: []), które oznacza, że domyślnie wszystkie endpointy wymagają auth.

Dla każdego endpointu sprawdzane jest, czy pasuje do reguły permitAll (metoda + ścieżka, z użyciem AntPathMatcher):

- jeśli tak → w OpenAPI dostaje security: [], x-security: public
- jeśli nie → dziedziczy globalne security i jest traktowany jako zabezpieczony (x-security: secured).

3. Prezentacja w PDF (PdfDocService):
Na podstawie OpenAPI PDF pokazuje:

- przy każdym endpointzie etykietę:
- Security: publiczny (brak uwierzytelniania) dla security: [],
- Security: Bearer JWT albo HTTP Basic dla endpointów wymagających auth.
- Na początku dokumentu generowane jest krótkie podsumowanie wykrytych mechanizmów bezpieczeństwa i liczby endpointów publicznych vs zabezpieczonych.

UPDATE *16.11.2025*

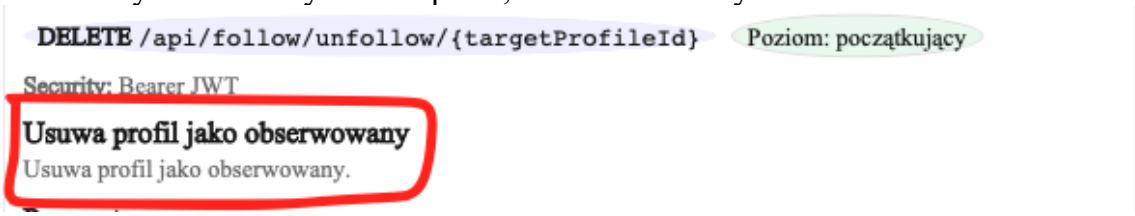
Naprawienie:
1. Przykłady cURL vs rzeczywiste ścieżki/metody
Widać kilka problemów w przykładach:
1. część curl-i ma inne ścieżki niż opisany endpoint (np. /api/chat vs /api/chat/group, /api/chats vs /api/chat/allChats, czy /api/profile vs /api/profile/me),

2. miejscami metoda z nagłówka to POST, a przykład sugeruje GET lub odwrotnie,
3. w przykładach JSON zdarzają się lekko pogięte stringi („has#o”, „ksi##ka” – to widać jako efekt escape/encoding i dzielenia wierszy).

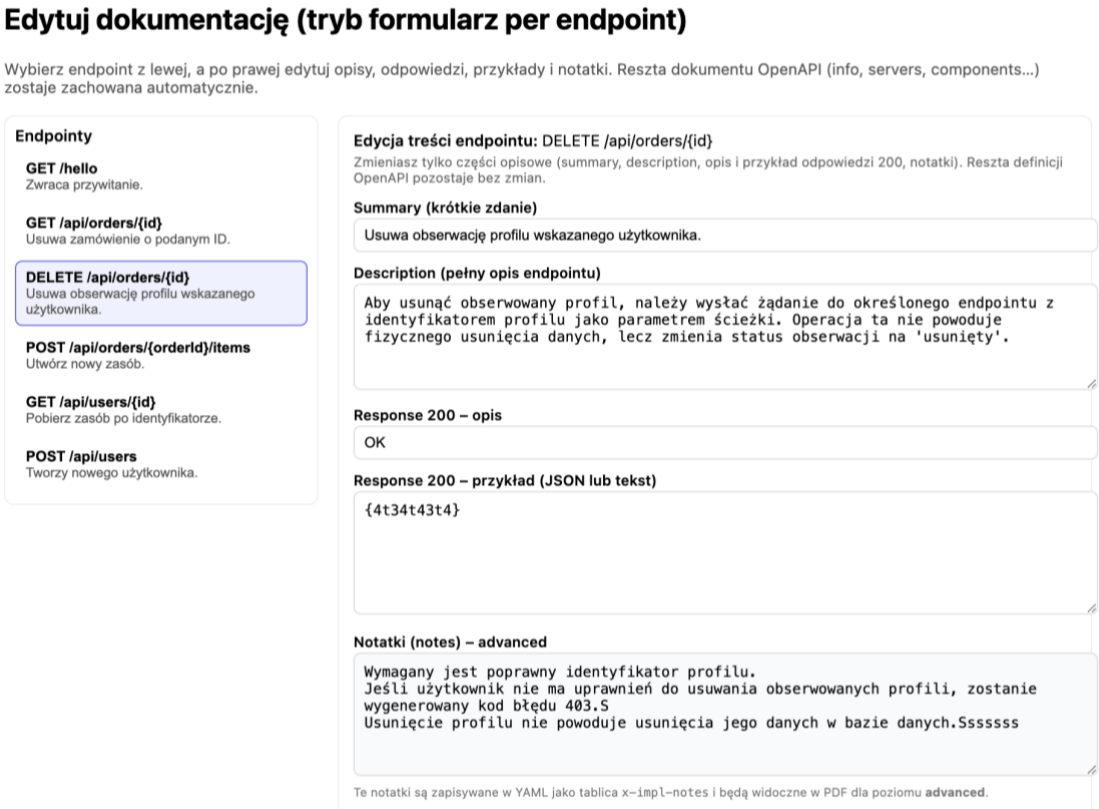
2. Brak nagłówka Authorization w cURL dla zabezpieczonych endpointów
Teraz:
globalny opis tłumaczy, że do chronionych endpointów używamy Authorization: Bearer <token>, ale w większości przykładów cURL dla chronionych endpointów nie ma tego nagłówka.

3. Schematy odpowiedzi „#/components/schemas/?”, void, { }
W kilku miejscach w tabeli odpowiedzi widać:
Schema: #/components/schemas/? lub po prostu { }.

4. Dublikaty w summary i description, nie ma summary



Naprawiono:
Przykłady cURL vs rzeczywiste ścieżki/metody
Ujednolicono wszystkie przykłady cURL z definicjami endpointów – metoda HTTP i ścieżka w przykładzie są teraz kopiowane 1:1 z OpenAPI (np. /api/chat/group, /api/chat/allChats, /api/profile/me). Zmieniono font.
Nagłówek Authorization dla chronionych endpointów
Dla wszystkich endpointów oznaczonych jako zabezpieczone (Bearer JWT) dodano w przykładach cURL nagłówek: Authorization: Bearer <token>. Z przykładów dla endpointów publicznych nagłówek został usunięty, żeby nie sugerować wymogu autoryzacji tam, gdzie jej faktycznie nie ma.
Schematy odpowiedzi „#/components/schemas/?”, void, { }
W miejscach, gdzie generator OpenAPI wstawił placeholder #/components/schemas/? lub pusty obiekt { }, dodano jednoznaczną informację tekstową dla użytkownika:„Typ odpowiedzi nie został jednoznacznie określony podczas generowania OpenAPI (użyto symbolicznego placeholdera ?).” Dzięki temu jest jasne, że ograniczenie wynika z definicji OpenAPI, a nie z błędu dokumentacji.
Duplikaty w summary i description / brak summary
Przegenerowano opisy tak, aby każdy endpoint miał wypełnione pole summary (krótkie, jednowierszowe podsumowanie) oraz dłuższe description/mediumDescription.
3. Możliwość edytowania pliku przed go pobraniem



Po wgraniu projektu ZIP backend generuje dokumentację OpenAPI w formacie YAML i udostępnia ją pod endpointem GET /api/projects/{id}/docs/editable. Frontend pobiera ten YAML i zapisuje w stanie aplikacji.

Następnie komponent EditableDocsPanel (React + TypeScript, js-yaml) parsuje YAML do obiektu i dla każdego endpointu (paths + methods) wyświetla formularz z polami: summary, description, opis odpowiedzi 200, przykład odpowiedzi 200 oraz notatki (x-impl-notes dla poziomu advanced). Użytkownik edytuje treść w czytelnych

inputach/textarea, a każda zmiana jest od razu nanoszona na obiekt OpenAPI i ponownie zapisywana do YAML (yaml.dump).
W ten sposób w stanie aplikacji zawsze znajduje się aktualna, już zmodyfikowana wersja dokumentacji.
Przycisk „Pobierz edytowany PDF" wysyła ten zaktualizowany YAML w żądaniu POST /api/projects/{id}/docs/edited/pdf.
Backend zapisuje go jako openapi_<level>_edited.yaml, generuje z niego PDF (pdfDocService.renderPdfFromYaml(...)) i zwraca do pobrania. Dzięki temu użytkownik pobiera PDF zawierający wszystkie ręczne poprawki wykonane w edytorze.

UPDATE *17.11.2025*

Funkcjonalność aplikacji

Aplikacja służy do automatycznego generowania dokumentacji API z kodu projektu Java Spring z wykorzystaniem modelu LLM, z możliwością ręcznej korekty treści przed wygenerowaniem końcowego PDF.
Główne funkcje:

1. Wgrywanie projektu (ZIP)
   - Użytkownik wgrywa paczkę ZIP z projektem backendowym.
   - Backend zapisuje projekt na dysku i nadaje mu identyfikator projectId.
2. Parsowanie kodu i generowanie IR
   - Serwer analizuje kod (kontrolery, metody HTTP, ścieżki, parametry, typy odpowiedzi) i buduje wewnętrzną reprezentację endpointów (EndpointIR).
   - Na tej podstawie przygotowywane są dane wejściowe dla modelu (NLP input).
3. Wykorzystanie modelu LLM do stworzenia opisów
   - Dla każdego endpointu generowane są opisy (summary, description, opisy odpowiedzi 200, przykłady) oraz – dla poziomu advanced – notatki developerskie (notes).
   - Model może pracować w różnych trybach „poziomu odbiorcy": beginner oraz advanced.
4. Generowanie dokumentacji OpenAPI (YAML)

Na podstawie IR + odpowiedzi modelu powstaje plik openapi_<poziom>.yaml.
Użytkownik może:

- pobrać YAML,
- podejrzeć YAML w przeglądarce.

5. Generowanie dokumentacji PDF

Z pliku YAML tworzony jest PDF (przy użyciu PdfDocService).
Dostępne są dwie opcje:

- pobranie PDF,
- podgląd PDF inline w nowej karcie przeglądarki.

6. Podgląd danych wejściowych/wyjściowych modelu (debug / test NLP)
   - Aplikacja umożliwia podejrzenie, jakie dokładnie dane są wysyłane do modelu (NLP input) – w formie tabeli z listą endpointów.
   - Dla wybranego endpointu można też podejrzeć wygenerowany prompt oraz surową odpowiedź modelu.
7. Edycja dokumentacji przed pobraniem (tryb przyjazny użytkownikowi)

Po wygenerowaniu YAML użytkownik może przełączyć się w tryb edycji.
Komponent EditableDocsPanel:

- wyświetla listę wszystkich endpointów po lewej,
- po prawej umożliwia edycję w formularzu, zamiast pracy na surowym YAML:
  - summary (krótkie zdanie),
  - description (pełny opis),
  - opis odpowiedzi 200,
  - przykład odpowiedzi 200 (JSON lub tekst),
  - dla poziomu advanced – notatki (zapisywane jako x-impl-notes w YAML).

Każda zmiana na froncie jest od razu nanoszona na strukturę OpenAPI i z powrotem serializowana do YAML.

8. Pobranie edytowanego PDF
   - Zaktualizowany YAML jest wysyłany do endpointu POST /api/projects/{id}/docs/edited/pdf.
   - Backend zapisuje go jako openapi_<poziom>_edited.yaml, generuje nowy PDF i odsyła do pobrania.

Dzięki temu użytkownik dostaje końcowy PDF zawierający zarówno treści wygenerowane przez model, jak i ręczne poprawki.

2. Stworzenie dynamicznego interfejsu

1. Dla *kogo* i *po co* jest UI

1. Kto używa?
   - dev, który wrzuca projekt i chce szybko mieć PDF
   - tech-writer / analityk, który chce edytować opisy
2. Jakie główne zadania?
   - Wgrać ZIP → zobaczyć status → mieć dokumentację.
   - Przejrzeć endpointy → poprawić opisy → pobrać PDF.
   - (opcjonalnie) zajrzeć do „debug" modelu: input / output.

To się przekłada na 3 główne ekrany / sekcje:

- Upload + status
- Lista endpointów + edycja
- Debug NLP (dla „zaawansowanego" użytkownika)

2. Zaplanować strukturę ekranu (layout)

Przykładowy layout dla mojej aplikacji

Header (góra)

- nazwa aplikacji
- status + timer
- przełącznik: Beginner / Advanced

Główny obszar (podzielony na 2–3 kolumny):

lewy panel – projekt / nawigacja:

- upload ZIP
- info o projekcie (nazwa, poziom)
- przyciski: „Generuj PDF", „Pobierz YAML", „Tryb edycji"

środkowy panel – lista endpointów

- filtr (np. po path / metodzie)

- lista klikanych endpointów

prawy panel – szczegóły wybranego endpointu

- w trybie edycji: formularze do edycji (summary, description, response, notes)
- w trybie „podgląd” – doc-style view (ładnie sformatowane)

3. Rozbić UI na sensowne komponenty

Zamiast jednego App.tsx + mega UploadBox, zrobić:

- AppLayout.tsx – ogólny szkielet (header, main, panele)
- ProjectBar.tsx – upload + info o projekcie + przyciski (PDF/YAML/tryb edycji)
- EndpointList.tsx – lista endpointów (z filtrem)
- EndpointEditor.tsx – to, co masz teraz w EditableDocsPanel (formularz)
- NlpDebugPanel.tsx – to, co masz z „danymi wejściowymi/wyjściowymi” dla modelu
- StatusBar.tsx – pasek „Status / czas”

4. Zaplanować „stany aplikacji”

- globalne:
  - status, elapsed
  - currentProjectId, projectName
  - level (beginner/advanced)
  - mode: np. "view" vs "edit"
- związane z projektem:
  - uploadResult
  - isLoadingYaml, isLoadingPdf, isSavingEdits
  - editableYaml (cały dokument)
  - selectedEndpointKey
- związane z UI:
  - activeTab: "edit" | "preview" | "nlp-debug"

Stan początkowy → ZIP wgrany → YAML wygenerowany → tryb edycji → PDF po edycji.

5. Zachowanie przy akcjach (UX flow)

Przyjazny, dynamiczny front = przewidywalne zachowanie przy akcjach:
Po wgraniu ZIP:

- pokazuje info „Projekt X, poziom Y, możesz wygenerować dokumentację”
- przyciski do generowania stają się aktywne
- Po wejściu w „Tryb edycji”:
- automatycznie ładuje się YAML (jeśli jeszcze go nie ma)
- otwiera się panel z listą endpointów i edytorem

Przy edycji:

- inputy reagują od razu (tak jak teraz)
- można dodać mały „badge”: *Zmieniono* / *Zapisano w pamięci* (lokalnej – w state)
- przy wyjściu / pobraniu PDF – krótkie info „Generuję PDF na podstawie aktualnych zmian”

Przy przełączaniu level beginner/advanced:

- pytam: „Przełączenie poziomu wygeneruje nowy YAML. Kontynuować?”

6. Warstwa wizualna / design system

ustalić kolory bazowe (fiolet dla akcji, szarości dla tła),
spójne style:

- ten sam styl przycisków (rounded, padding),
- te same odstępy (8 / 12 / 16 px),

typografia:

- h1 – 24px, h2 – 18px, tekst – 13–14px,

drobne **mikroanimacje**:

- :hover na przyciskach,
- cursor: progress przy long-running akcjach.

7. Przygotować „makiety”

1. Narysować :
   - ekran po wgraniu projektu,
   - ekran w trybie edycji,
   - ekran z debugiem NLP.
2. Zaznaczyć:
   - gdzie są przyciski,
   - jak użytkownik przełącza tryby,
   - gdzie widzi komunikaty.

8. Plan

1. **Wyciągnąć logikę statusu + uploadu do osobnych komponentów**:
   - StatusBar
   - ProjectUploadPanel
2. **Dodać zakładki (tabs)**:
   - Edycja
   - Podgląd PDF
   - NLP debug (prosty useState("edit" | "preview" | "debug")).
3. **Dodać mały „status zmian”**:
   - badge „Zapisano lokalnie” vs „Masz niezapisane zmiany w PDF (trzeba pobrać nowy)”.

Wszystkie endpointy:

1. Upload projektu

POST /api/projects/upload

- Body: multipart/form-data z polem file (ZIP)
- Zwraca: UploadResult (id projektu, status, wykryta specyfikacja itd.)
- Opis: wgrywa ZIP z projektem (kod źródłowy).

2. Generowanie dokumentacji z kodu (YAML)

POST /api/projects/{id}/docs/from-code?level=beginner|advanced

- Body: brak

- Zwraca: plik openapi_<level>.yaml jako attachment
- Opis: parsuje kod źródłowy, generuje OpenAPI YAML i zwraca jako plik.

3. PDF z kodu (bez edycji)

POST /api/projects/{id}/docs/pdf?level=beginner|advanced
- Body: brak
- Zwraca: openapi_<level>.pdf jako pobierany plik
- Opis: generuje YAML z kodu, potem PDF i zwraca go do pobrania.

GET /api/projects/{id}/docs/pdf?level=beginner|advanced
- Produces: application/pdf
- Zwraca: PDF jako inline (do podglądu w przeglądarce)
- Opis: to samo, ale do podglądu w nowej karcie.

4. YAML (podgląd + pobranie)

GET /api/projects/{id}/docs/yaml?level=beginner|advanced
- Produces: text/yaml
- Zwraca: YAML jako inline (wyświetlany w przeglądarce)

GET /api/projects/{id}/docs/yaml/download?level=beginner|advanced
- Zwraca: openapi_<level>.yaml jako pobierany plik.

5. Tryb edycji dokumentacji (YAML per endpoint)

GET /api/projects/{id}/docs/editable?level=beginner|advanced
- Produces: text/plain
- Zwraca: pełny YAML (openapi_<level>.yaml) jako tekst
- Opis: frontend pobiera ten YAML, rozbija go na endpointy i pokazuje formularz do edycji (summary, description, response, notes).

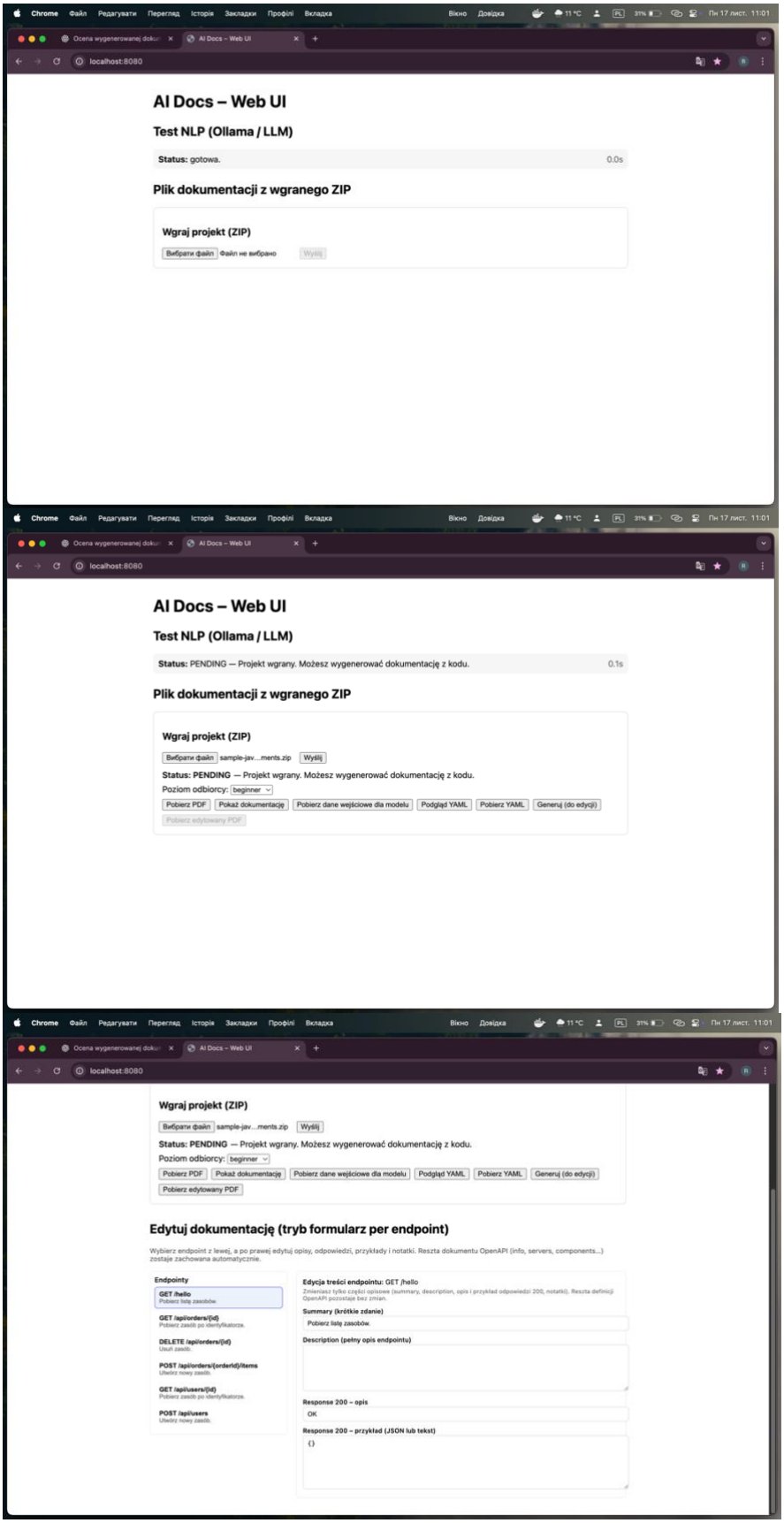
POST /api/projects/{id}/docs/edited/pdf?level=beginner|advanced
- Consumes: text/plain
- Body: edytowany YAML (cały dokument po zmianach w UI)
- Zwraca: openapi_<level>_edited.pdf jako pobierany plik
- Opis: generuje PDF z edytowanej wersji dokumentacji.

6. NLP – dane wejściowe / wyjściowe (debug)

GET /api/projects/{id}/docs/nlp-input?level=beginner|advanced&mode=ollama
- Produces: application/json
- Zwraca: tablicę obiektów IR (DescribeIn) dla każdego endpointu
- Opis: dane wejściowe przekazywane do modelu (symbol, params, returns, comment, notes itd.).

POST /api/nlp/output-preview
- Body: JSON z jednym obiektem DescribeIn (dla wybranego endpointu)
- Zwraca: { prompt, raw } – prompt wysłany do modelu + surowa odpowiedź
- Opis: proxy do serwisu NLP (Python), używane w „Test NLP (Ollama / LLM)”.

Na razie:



1. Główny layout

1. Jedna strona **AI Docs – Web UI**.
2. U góry pasek **Status + timer**, który pokazuje, co się dzieje (wysyłanie ZIP, generowanie PDF, gotowe itp.).

2. Blok „Plik dokumentacji z wgranego ZIP”

W tym bloku użytkownik może:

1. Wgrać ZIP z projektem Spring (input type="file" + Wyślij).
2. Zobaczyć status wgrania (PENDING / ERROR / OK + ewentualny komunikat).
3. Wybrać Poziom odbiorcy: beginner / advanced.
4. Używać przycisków:
   - **Pobierz PDF** – pobiera standardowy PDF z kodu.
   - **Pokaż dokumentację** – otwiera PDF w nowej karcie (inline preview).
   - **Pobierz dane wejściowe dla modelu** – pobiera IR dla NLP i pokazuje je w tabeli (z możliwością podglądu promptu i odpowiedzi modelu).
   - **Podgląd YAML** – otwiera wygenerowany OpenAPI YAML w nowej karcie.
   - **Pobierz YAML** – pobiera YAML jako plik.
   - **Generuj (do edycji)** – generuje YAML, który ma być edytowany w UI.
   - **Pobierz edytowany PDF** – pobiera PDF wygenerowany z edytowanego YAML.

3. Panel edycji dokumentacji (EditableDocsPanel)

Pojawia się po kliknięciu **„Generuj (do edycji)”**:

Lewa kolumna: lista **endpointów** (GET /hello, GET /api/orders/{id} itd.) – wybór endpointu.

Prawa kolumna: **formularz per endpoint**, w którym użytkownik edytuje tylko część opisową:

1. Summary (krótkie zdanie)
2. Description (pełny opis endpointu)
3. Response 200 – opis
4. Response 200 – przykład (JSON lub tekst) – to, co później widzisz w kolumnie „Przykład” w PDF.
5. dla poziomu advanced dodatkowo: Notatki (notes) – advanced → mapowane na tablicę x-impl-notes w YAML.

Wszystkie zmiany:

1. od razu aktualizują strukturę YAML w pamięci,
2. od razu są zapisywane w stanie editableYaml w App.tsx,
3. później ten YAML idzie na backend i z niego generowany jest edytowany PDF.

4. Sekcja NLP (IR + preview odpowiedzi)

Tabela z wierszami dla każdego endpointu: method, path, params, returns, komentarze, requestBody itd.

Przycisk w każdym wierszu „Pokaż dane wyjściowe”:

- wysyła pojedynczy IR do /api/nlp/output-preview,
- pokazuje z lewej wygenerowany prompt, z prawej surową odpowiedź modelu.

Kolory:

Tło i ramki:

- Tło: #f4f4f4

- Karty / boxy / panele: #FFFFFF
- Ramki/linie: #dbdded

Tekst

- Główny tekst: #111827
- Opisy dodatkowe / hinty: #575b64
- Tekst w statusie „błąd”: #B91C1C

Kolor akcentu (przyciski, zaznaczone endpointy, linki)

- Główny akcent: #4F46E5 (indigo)
- Hover: #4338CA
- Bardzo delikatne tło zaznaczenia (np. wybrany endpoint): #EEF2FF

Statusy

- Sukces (np. „PDF wygenerowany”): tło #DCFCE7
- Ostrzeżenie: tło #FEF3C7
- Błąd: tło #FEE2E2

Nazwa aplikacji: | Code2Docs AI |

Slogan: „Code2Docs AI – od kodu do dokumentacji w kilka kliknięć.” / “ Code2Docs AI – from code to documentation in a few clicks”

Logo:

Font: Koulen
Color: #4F46E5 (indigo)

Code2Docs AI – Java API Documentation Generator

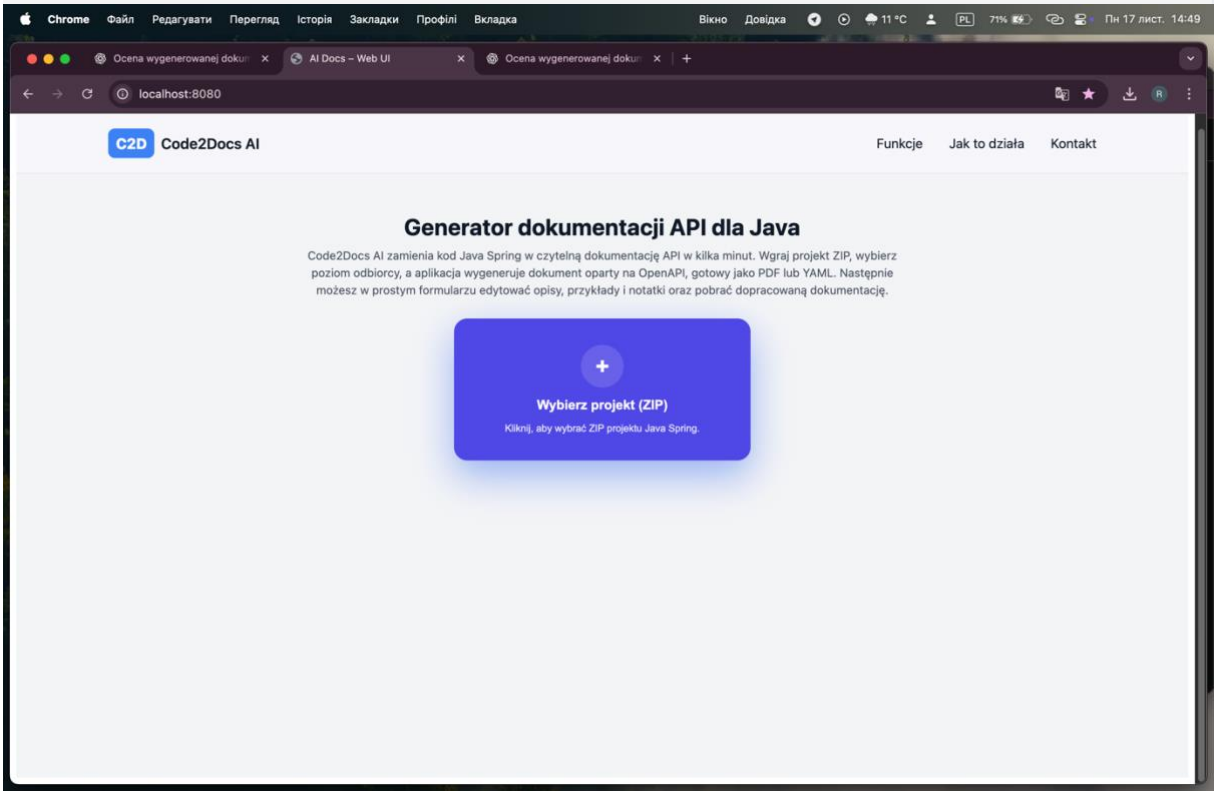
Code2Docs AI turns your Java Spring code into clean API documentation in minutes. Just upload a ZIP with your project, choose the audience level, and the app generates OpenAPI-based docs, ready as PDF or YAML. You can edit summaries, descriptions, examples and notes in a friendly form – then download the final, polished documentation for your project.

Code2Docs AI – generator dokumentacji API dla Java

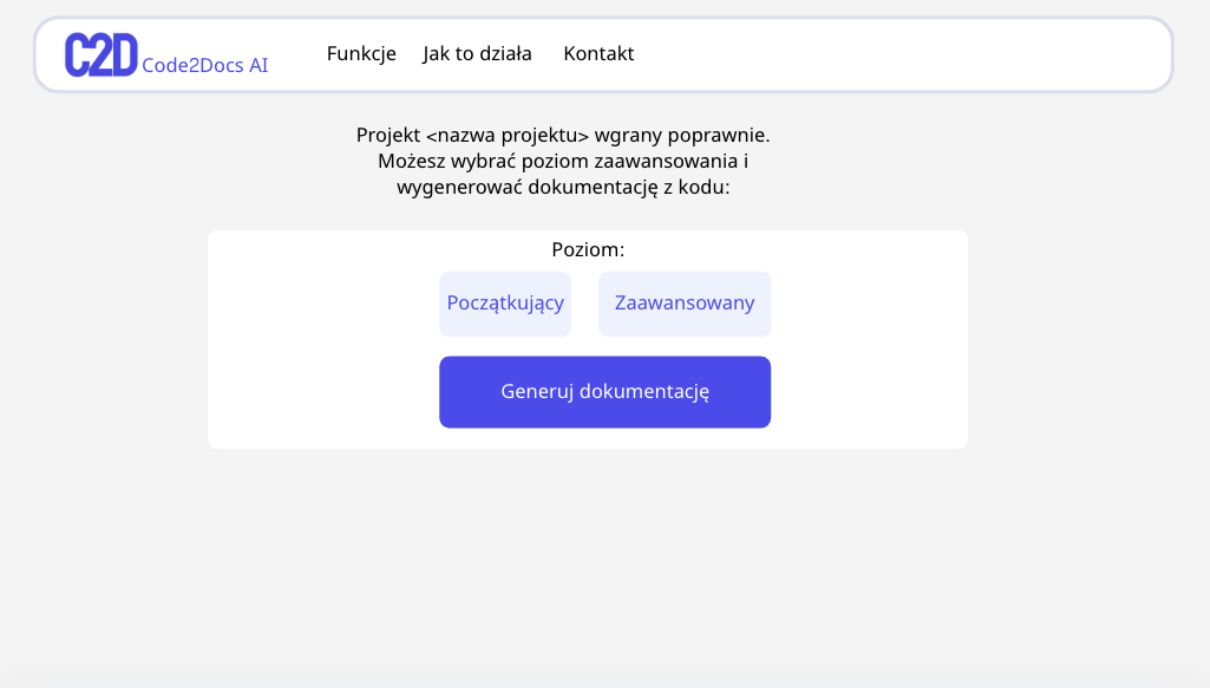
Code2Docs AI zamienia kod Java Spring w czytelną dokumentację API w kilka minut. Wgraj projekt ZIP, wybierz poziom odbiorcy, a aplikacja wygeneruje dokument oparty na OpenAPI, gotowy jako PDF lub YAML. Możesz w prostym formularzu edytować opisy, przykłady i notatki, a następnie pobrać gotową, dopracowaną dokumentację.

1. Krok

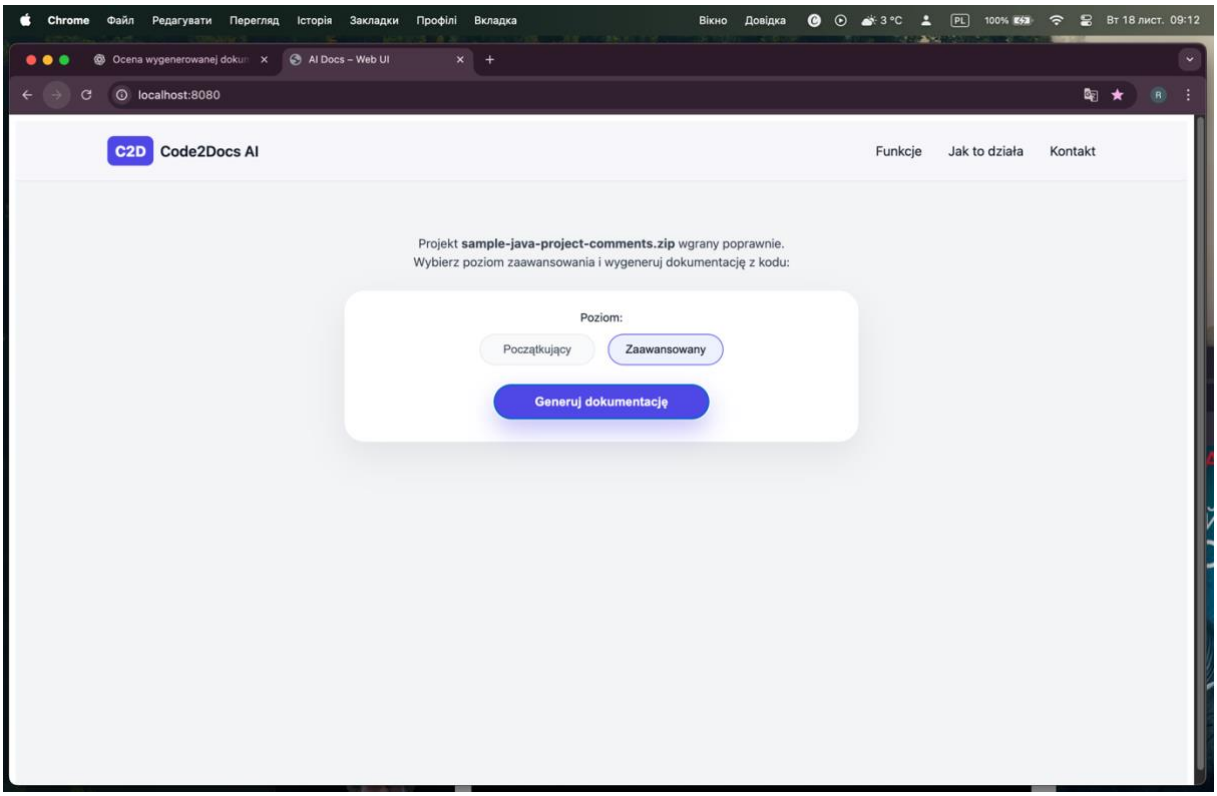
StartBar, HelloText, ProjectUploadPanel oraz StatusBar.



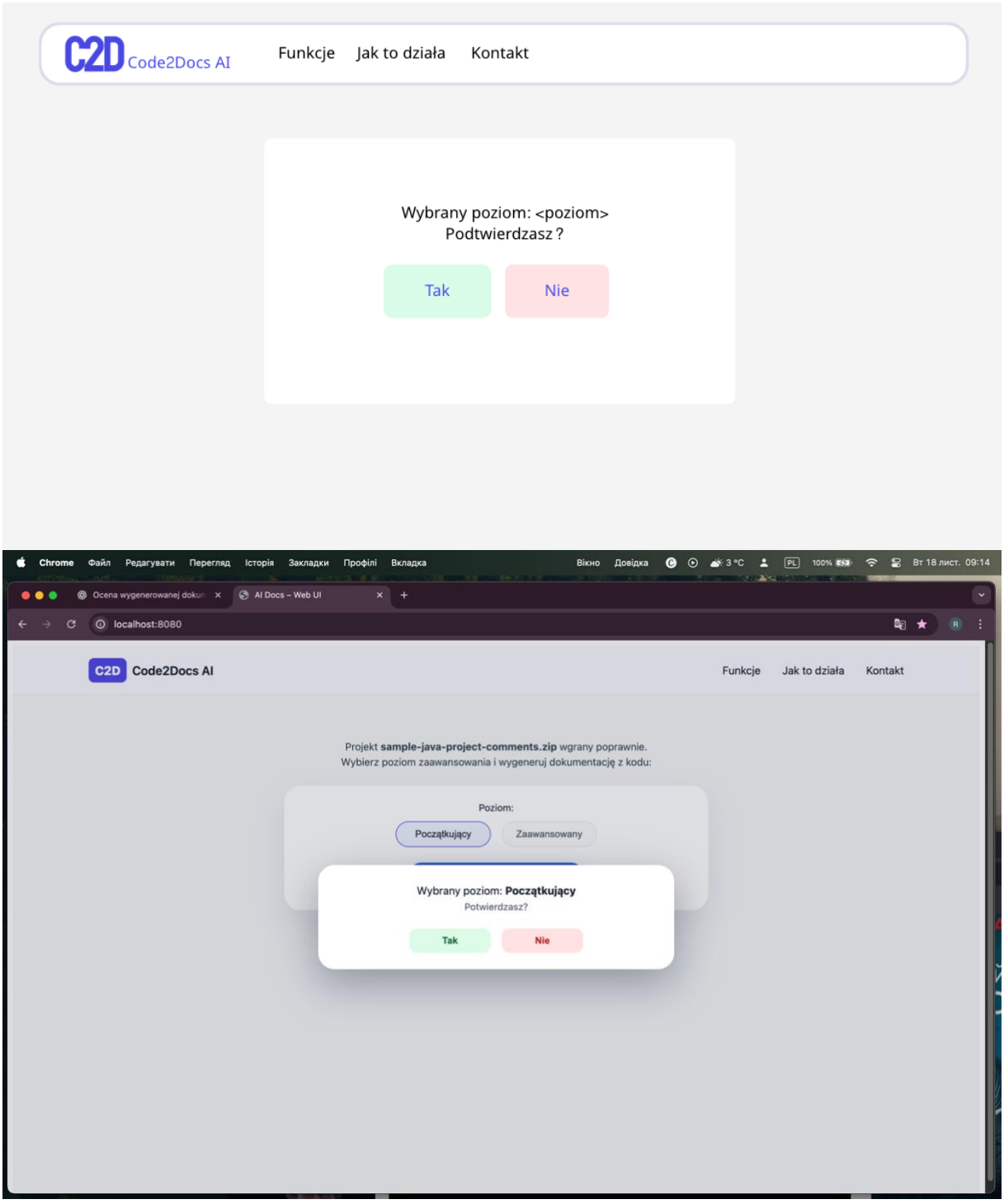
2. Krok LevelPanel



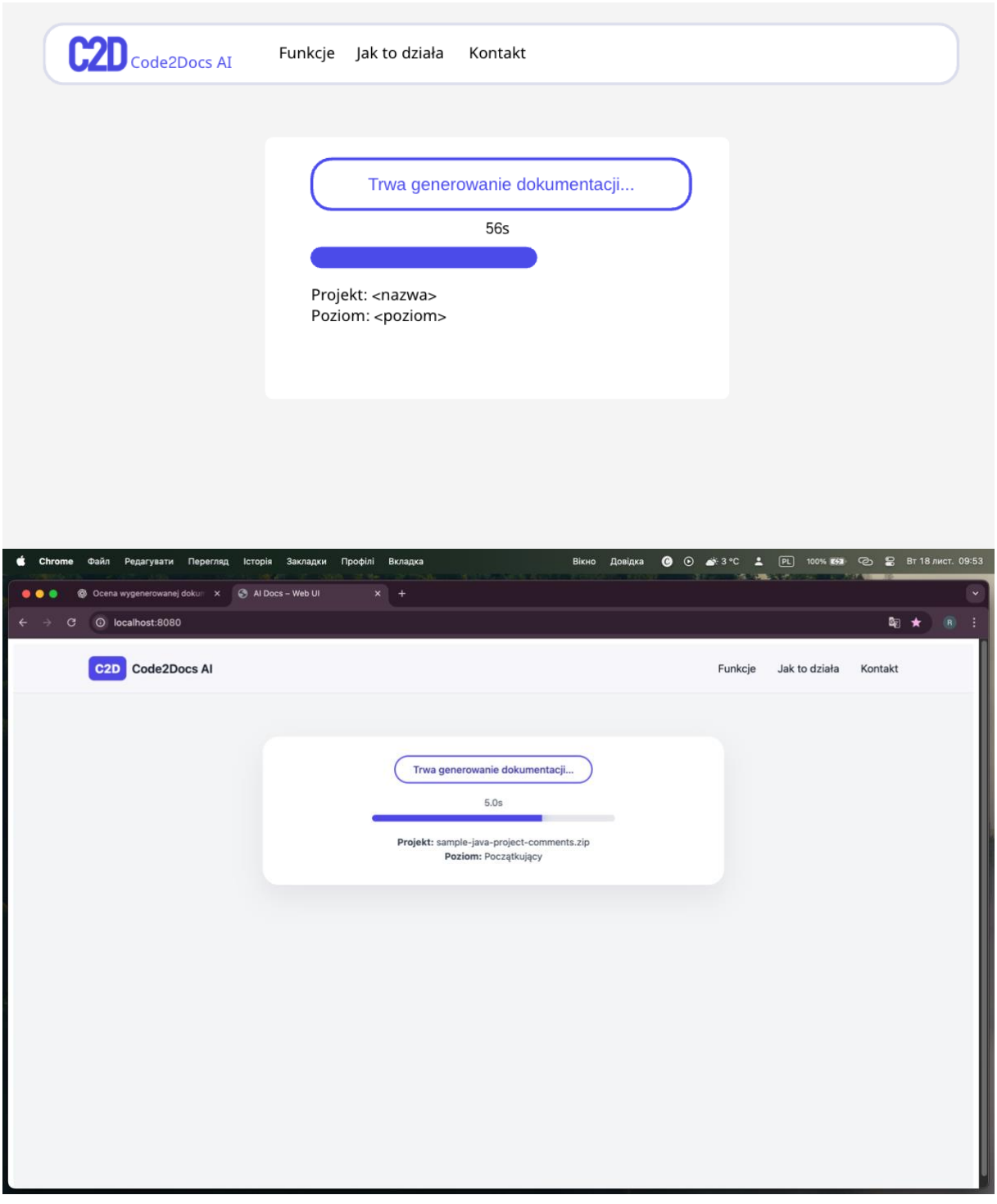
UPDATE *18.11.2025*



3. Krok showConfirm



4. Krok StatusGenerate



5. Krok DocsActionsPanel