Spis treści

Opis aplikacji

Utworzona aplikacja stanowi prototyp API *dashboardu* do zarządzania projektami. Zastosowane podejście stanowi fundament do utworzenia wysoko skalowalnej aplikacji z wyraźną separacją odpowiedzialności poszczególnych mikrousług składających się na utworzony system. Odbyło się to między innymi poprzez rozdzielenie bazy danych pomiędzy mikrousługami: projektów (*project microservice*) oraz zadań (*task microservice*).

Aplikacje *dashboard* pozwalają na monitorowanie postępów wykonywanych zadań. Jest to funkcjonalność niezbędna w każdym przedsiębiorstwie. Podstawową funkcjonalnością aplikacji *dashboard* jest możliwość oszacowania postępu realizowanych prac.

Oferowane funkcjonalności

Podstawowym zadaniem systemu jest śledzenie postępu w wykonywanych projektach. W celu realizacji tego zadania zaimplementowano:

- Operacje CRUD dotyczące indywidualnych zadań i projektów.
- Procedurę zliczania postępu projektu, zlecane mikrousłudze zadań.
- Mechanizm nasłuchiwania aktualizacji zadań i ich postępów.
- Mechanizm przekazywania zmiany stanu zadań z mikrousługi zadań do mikrousługi projektów
- Procedurę tworzenia listy osób pracujących nad zadaniami należącymi do danego projektu.
- Mechanizm cache w mikrousłudze zadań.

Powyższe funkcjonalności pozwalają na modyfikowanie stanu projektów i zadań, a także na wyświetlenie kompletnego podsumowania projektu i śledzenie procentowego postępu projektu w czasie rzeczywistym.

Zastosowane narzędzia

System oparty jest o architekturę mikrousługową, wymagającą: komunikację międzyusługową, zgodność z *Restful* API, mechanizm *cache* oraz *Gateway*. Do realizacji tych celów wykorzystano następujące mechanizmy i narzędzia:

- *Framework* webowy – *Spring Boot*.
- Baza danych i mechanizm persistence – *Postgres*.
- Odkrywanie i mapowanie mikrousług – *Netflix Eureka Service Discovery*.
- Mechanizm *cacheowania* w mikrousłudze zadań – *Caffeine*.
- Dostawcę mechanizmu *Gateway* – *Spring Cloud Routing*.
- Komunikacja międzyusługowa – *RabbitMQ*.
- Konteneryzacja usług i baz danych – *Docker*.

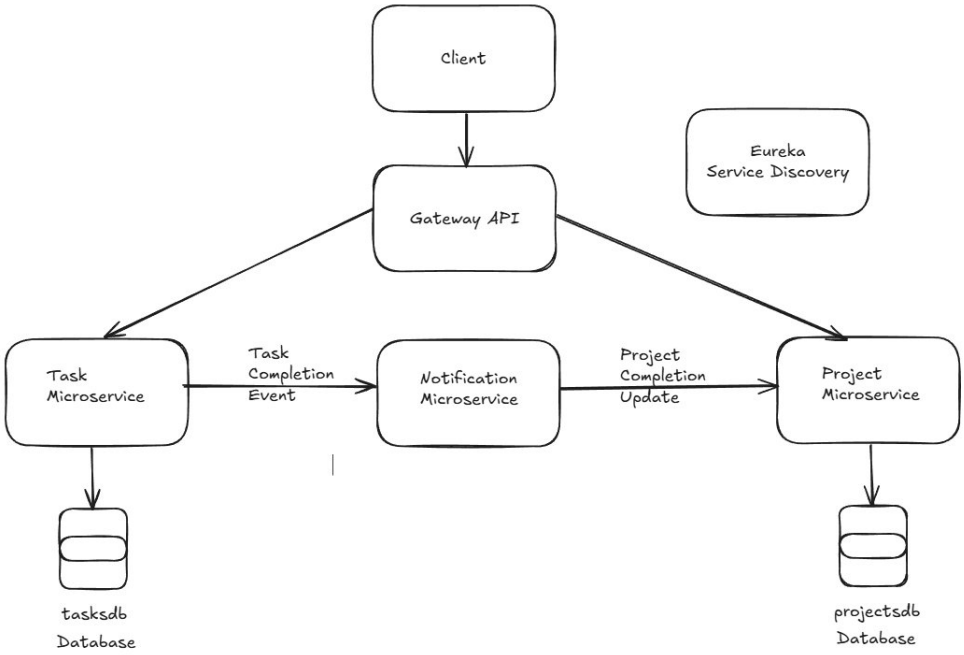
Budowa utworzonego systemu *dashboard*

Przygotowana aplikacja składa się z 3 mikrousług odpowiedzialnych za podstawowe funkcjonalności systemu:

- Mikrousługa projektów – system stanowiący zalążek systemu *dashboard*. Posiada wydzieloną bazę danych zawierającą 1 tabelę (*projects*), która przechowuje ogólne informacje na temat stanu projektów, takie jak postęp, termin i opis.
- Mikrousługa zadań – system zarządzania zadaniami pozwalający na przypisywanie zadań do osób i projektów. Posiada dedykowaną bazę danych z 3 tabelami (*tasks*, *persons*, task_*assignments*), które opisują indywidualne zadania oraz łączą je z projektami i osobami.
- Mikrousługa powiadomień – system śledzący zmiany stanu zadań i przekazujący informację o globalnym postępie danego projektu między mikrousługami zadań oraz projektów.

Powyższe mikrousługi wspierane są przez systemy: odkrywania usług *Eureka* oraz system przekierowujący *Gateway*. Dokładna budowa aplikacji została przedstawiona na diagramie 1.

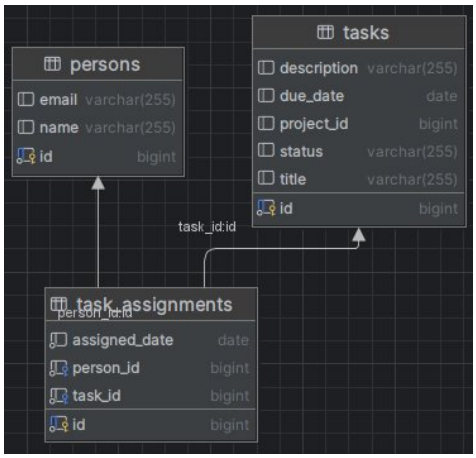
Rysunek 1. Budowa mikrousługowego systemu dashboard



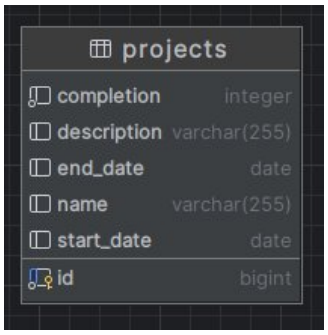
Budowa bazy danych

Aplikacja posiada 2 bazy danych: *tasksdb* (mikrousługa zadań) oraz *projectsdb* (mikrousługa projektów). Baza *tasksdb* (rys. 2) posiada 3 tabele: osób, zadań, oraz tabelę transferową *task_assignments* pozwalającą zlecanie pojedynczego zadania wielu osobom. Stan realizacji zadania *status* mierzony jest przy pomocy enumeratora opisującego progres przy pomocy 4 poziomów: *TODO*, *IN_PROGRESS*, *COMPLETED*, *BLOCKED*. W celu ułatwienia pracy z narzędziem utworzono odrębny *endpoint* API do edycji statusu danego zadania. Zastosowano mechanizm obiektu DTO (*TaskStatusUpdateDto*), który pozwala na wprowadzanie statusu zadania w postaci zwykłego łańcucha znaków.

Rysunek 2. Diagram ERD bazy danych tasksdb



Rysunek 3. Diagram ERD bazy danych projectsdb

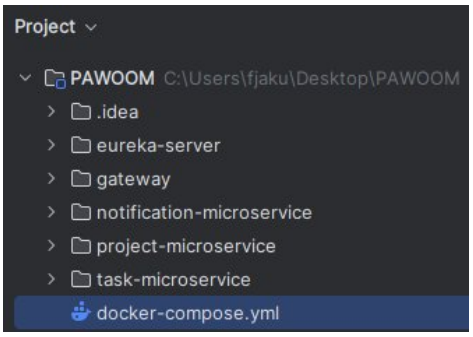


Struktura projektu i instrukcja uruchomienia

Projekt składa się z 5 podprojektów wewnątrz jednego wspólnego katalogu *root* zawierającego plik *docker-compose.yml* do konteneryzacji aplikacji. Każdy podprojekt zawiera plik *Dockerfile*. Do uruchomienia skonteneryzowanej aplikacji należy:

- Uruchomić program *Docker*.
- Przejść do katalogu root całego projektu.
- Uruchomić komendę *docker-compose up --build*
- Podgląd mikroserwisów widoczny jest na stronie *Eureki*: *localhost:8761*.
- Przykładowe żądanie może zostać wysłane jako GET *localhost:8090/projects* – serwer *Gateway* przekieruje je na port 8081 mikroserwisu projektów 8081.

Rysunek 4. Podgląd struktury projektu



Poniżej przedstawiono definicję mikroserwisu zadań w skrypcie *docker-compose.yml*.

```
task-microservice:
  build: ./task-microservice
  container_name: task-microservice
  ports:
    - "8080:8080"
  depends_on:
    - tasksdb
    - eureka-server
    - rabbitmq
```

```
  environment:
    SPRING_DATASOURCE_URL: jdbc:postgresql://tasksdb:5432/tasksdb
    SPRING_DATASOURCE_USERNAME: admin
    SPRING_DATASOURCE_PASSWORD: admin
  networks:
    - micro-net
```

Wykorzystane porty

- 5432: baza danych *tasksdb*.
- 5433: baza danych *projectsdb*.
- 5672: serwer *RabbitMQ*.
- 8080: mikrousługa zadań *TASK-MICROSERVICE*.
- 8081: mikrousługa projektów *PROJECT-MICROSERVICE*.
- 8082: mikrousługa powiadomień *NOTIFICATION-MICROSERVICE*.
- 8090: *GATEWAY-SERVER*.
- 8761: serwer *Eureka*.

Implementacja *cacheowania*

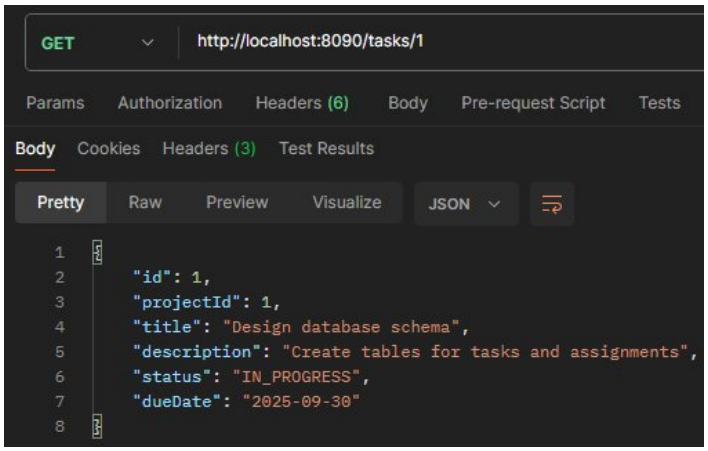
Cacheowanie po stronie serwera zostało zaimplementowane z użyciem biblioteki *Caffeine*. W szczególności dotyczy to wyników dla tabeli zadań *tasks*, które wraz z rozbudową projektu będą wymagały możliwie najkrótszych czasów przetwarzania. Podczas operacji takich jak usuwanie również następuje aktualizacja pamięci *cache*. Poniżej przedstawiono fragment kodu klasy *TaskService* wykorzystującej *cacheowanie*.

```
@CacheEvict(value = "tasks", key = "#id")
public void deleteTask(Long id) {
    taskRepository.deleteById(id);
}

@Cacheable("tasks")
public List<Task> findAll() {
    try{
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return taskRepository.findAll();
}
```

Wyniki realizacji projektu

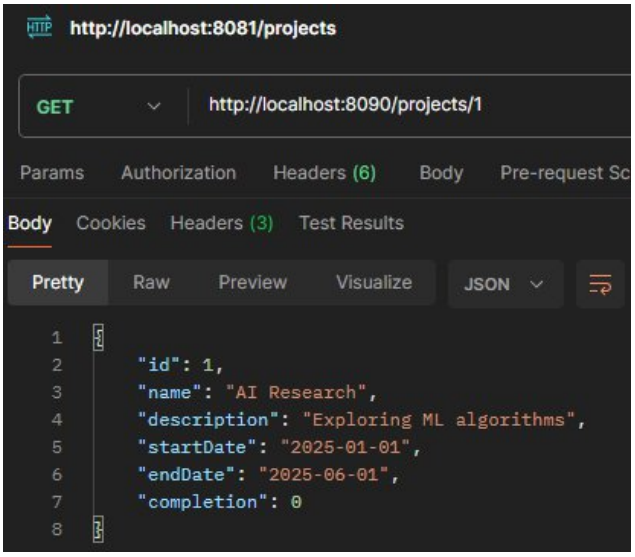
Poniżej przedstawiono rezultat żądania *GET* z wynikiem w postaci przykładowego zadania (*task*) należącego do projektu z id = 1. Widoczne jest również działania serwera *Gateway*, który przekierował żądanie z portu *Gateway* 8090 na port *TASK_MICROSERVICE* 8080 zmapowany z użyciem *Eureka Service Discovery*.

Rysunek 5. Pobranie zadania o indeksie 1



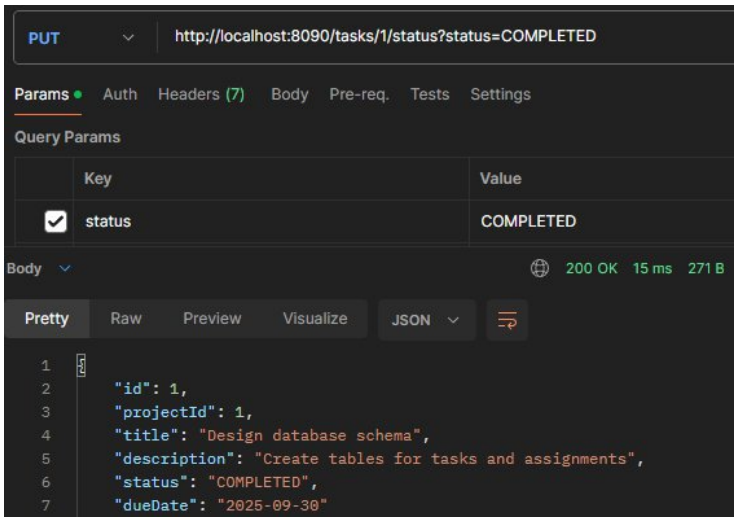
Poniżej przedstawiono obiekt projektu o id = 1. Widoczny jest poziom ukończenia wynoszący 0.

Rysunek 6. Pobranie projektu o indeksie 1



Następnie wysłano żądanie PUT aktualizacji postępu zadania należącego do projektu z id = 1.

Rysunek 7. Aktualizacja statusu ukończonego zadania o indeksie 1

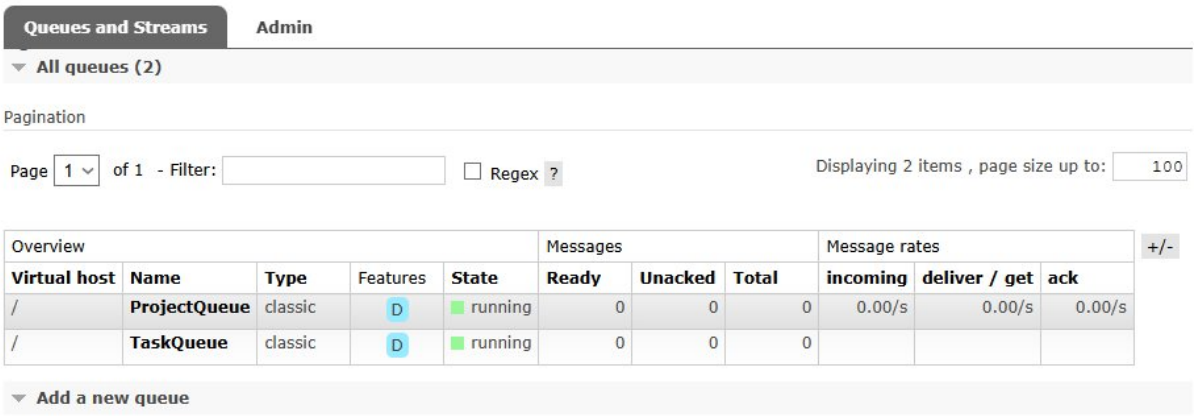


Po aktualizacji zadania zrealizowanej przez kod przytoczony poniżej, mikroserwis zadań zliczył wszystkie gotowe zadania należącego do tego projektu i przekazał je w wiadomości do kolejki *ProjectQueue* nasłuchiwanej przez mikroserwis powiadomień.

```
@PutMapping("/{id}/status")
public Task updateTaskStatus(@PathVariable Long id, TaskStatusUpdateDto dto) {
    Task updatedTask = taskService.updateTaskStatus(id, dto.getStatus());
    Long projectId = updatedTask.getProjectId();

    String completion = taskService.getProjectCompletionPercentage(projectId);
    String message = String.format(
            "Task updated: \"%s\" with status %s. Project ID: %d. Completion: %s.",
            updatedTask.getDescription(),
            updatedTask.getStatus(),
            projectId,
            completion
    );
    senderService.sendMessage(message);

    return updatedTask;
}
```

Poniżej widoczna jest kolejka *ProjectQueue* w podglądzie serwera *RabbitMQ*.

Rysunek 8. Utworzona kolejka ProjectQueue w RabbitMQ



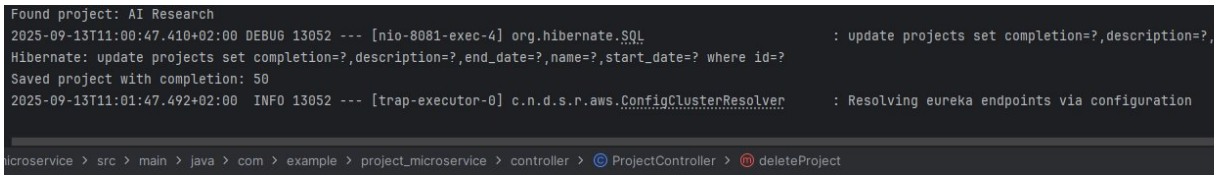
Następnie otrzymana przez mikroserwis powiadomień informacja o aktualizacji stanu projektu została przetworzona i wysłana do API aktualizującego stan pola postępu projektu w mikroserwisie projektów.

Poniżej widoczny jest kod z mikroserwisu powiadomień wysyłający żądanie do mikroserwisu projektów.

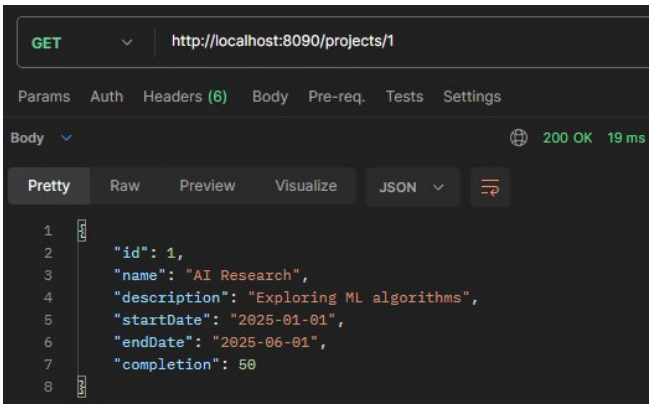
```
@RabbitListener(queues = "#{ProjectQueue.name}")
public void listen(String message) {
    System.out.println(message);

    try {
        Long projectId = parseProjectId(message);
        String completion = parseCompletion(message);

        System.out.println(projectId);
        System.out.println(completion);

        updateProjectCompletion(projectId, completion);

    } catch (Exception e) {
        System.err.println("Failed to process message: " + message);
        e.printStackTrace();
    }
}

private void updateProjectCompletion(Long projectId, String completion) {
    String url = "http://PROJECT-MICROSERVICE/projects/" + projectId +
"/completion";

    HttpEntity<Integer> request = new
HttpEntity<>(Integer.parseInt(completion.replace("%", "")));
    restTemplate.put(url, request);

    System.out.println("Updated Project " + projectId + " completion to " +
completion);
}
```

Po obsłużeniu żądania nastąpiło zmienienie stanu postępu projektu o id = 1, z poziomu 0% na 50%, gdyż 1 z 2 zadań w projekcie zostało wykonane. Poniżej widać wyniki.

Rysunek 9. Podgląd zapytania SQL aktualizującego stan ukończenia projektu wewnątrz mikroserwisu projektów

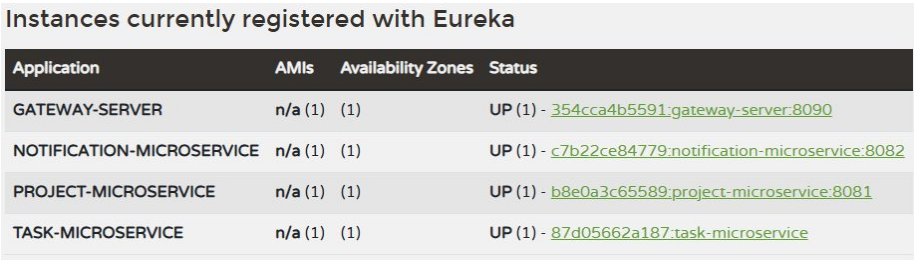


Rysunek 10. Widok stanu projektu zaktualizowanego przez komunikację międzyserwisową



Poniżej przedstawiono wszystkie 4 usługi widziane w systemie odkrywania usług *Eureka*. Widoczne są identyfikatory stworzonych kontenerów w programie *Docker*.

Rysunek 11. Podgląd wszystkich mikroserwisów i usług w Eurece



Poniżej w zrzucie ekranu z GUI *Dockera* wszystkie kontenery projektu należące kolejno do: serwera *RabbitMQ*, baz danych *tasksdb* i *projectsdb*, *Eureki*, mikroserwisów: projektów, zadań i powiadomień, i na końcu serwer *Gateway*.

Rysunek 12. Podgląd skonteneryzowanego projektu w Dockerze