

Projekt:
gra „Milionerzy”
Opis funkcjonalny serwera

Funkcjonalność serwera przygotował:
Kamil Marnik

Tarnów, 2019

Spis treści:

1. Wykorzystane narzędzia
2. Opis funkcjonalny klient
3. Opis funkcjonalny serwera

1. Wykorzystane narzędzia

W celu zaimplementowania serwera dla powyższego projektu, wykorzystano język programowania *Java* wraz z frameworkiem *Spring*. Natomiast przy tworzeniu aplikacji klienckiej, wykorzystano również język *Java*, a dla graficznego interfejsu – *JavaFX*.

W obu przypadkach, wykorzystano *IDE: IntelliJ IDEA*. Do współpracy wykorzystany został system hostingowy *bitbucket*. Do testów poprawności działania serwera, skorzystano z aplikacji *Postman*.

2. Opis funkcjonalny klienta

UWAGA:

W poniższym pliku została opisana funkcjonalność serwera.
W celu analizy działalności klienta zapraszam do dokumentacji pod linkiem:

https://bitbucket.org/JWarchol_1/project26_client.git

3. Opis funkcjonalny serwera

https://github.com/kamilmaestro/project26_server

3.1 Ogólny wgląd na pracę serwera

Swoistym “sercem” serwera jest klasa kontrolera, która odpowiada za kontrolę przepływu danych oraz najważniejsze funkcje, to jest wywoływanie odpowiednich metod oraz wysłanie JSON’a.

Ścieżką dla naszego serwera będzie:

<http://127.0.0.1:8080/api/milion/question/{id}>, gdzie `{id}` jest zależne od numeru pytania, które ma być w danym momencie wyświetlane w aplikacji Milionerzy. Wartość *id* jest pobierana z adresu, a następnie dokonuje się jej sprawdzenia. Poprzez metodę *isIdCorrect* z klasy *QandAServiceImpl* otrzymujemy wczesną informację na temat poprawności wprowadzonego *id*. W przypadku niezgodności ze specyfikacją rzucony jest błąd *ResourceNotFound*.

```
@RequestMapping(value = "/milion/question/{id}", method = RequestMethod.GET)
public ResponseEntity<?> sendQandA(@PathVariable("id") Long id) throws ResourceNotFound {

    if(!qandAService.isIdCorrect(id)) throw new ResourceNotFound();

    setUTF();

    QuestionDto questionDto = new QuestionDto();
    qandAService.readQandA(id, questionDto);

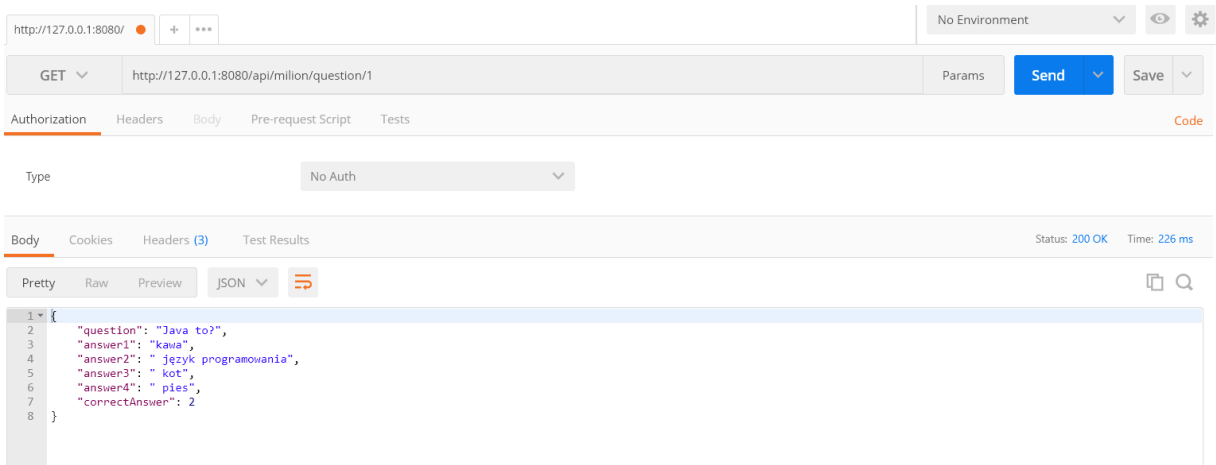
    LOGGER.info("Question id: {}", id);
    LOGGER.info("Question: {}", questionDto.getQuestion());
    LOGGER.info("Nr of correct answer: {}", questionDto.getCorrectAnswer());

    return qandAService.isEmpty(questionDto) ?
        new ResponseEntity<>(new ErrorDto("File doesn't contain correct/any content!"), HttpStatus.NOT_FOUND) :
        new ResponseEntity<>(questionDto, HttpStatus.OK);
}
```

W kontrolerze dbamy również, aby JSON miał możliwość wyświetlania polskich znaków (metoda *setUTF* umożliwia korzystanie z systemu kodowania Unicode UTF-8).

Wywołuje się główną metodę serwisu odpowiedzialną za odczytywanie pytań oraz odpowiedzi, a na końcu zwraca się:

- JSON'a wraz z wartościami:



- Wyjątek mówiący o niekompletności danych wraz z kodem błędu *404 Not Found*:



3.2 Cykl życia danych

Klasa *QuestionDto* służy jako klasa przechowująca zmienne, które są przesyłane przez cały “cykl” Springa, poprzez *QuestionDao*, czyli pierwotny dostęp do “bazy danych”(to jest klasy *Dto*). Otrzymujemy ilość pytań w pliku, a także numer pytania. Następnie dane przesyłane są do serwisu *QandAServiceImpl*, gdzie dokonywane jest odczytanie danych z plików txt, w których przetwarzamy pytania wraz z odpowiedziami.

```
@Override
public void readQandA(Long questionId, QuestionDto questionDto) {
    String line;

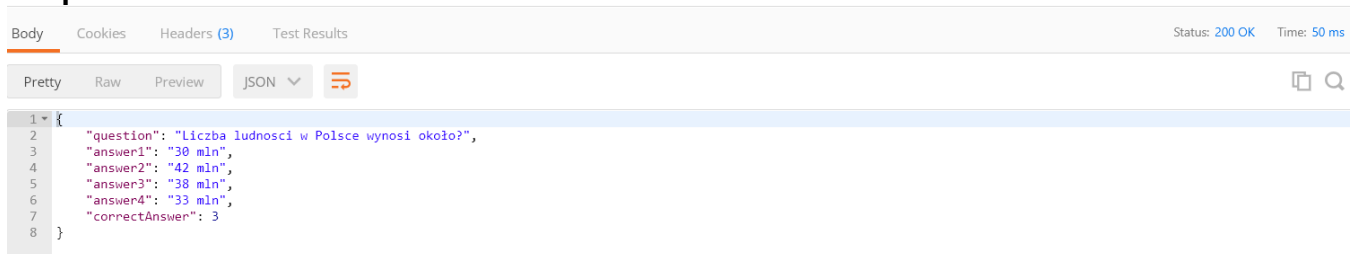
    try{
        int randNrQuestion = questionDAO.randQuestionNr(questionId);

        BufferedReader reader = new BufferedReader(new FileReader( fileName: Long.toString(questionId) + ".txt"));
        try {
            while ((line = reader.readLine()) != null) {
                if (line.startsWith(Integer.toString(randNrQuestion))) {
                    setVariables(questionId, line, questionDto);
                }
            }
        } finally {
            reader.close();
        }
    } catch (FileNotFoundException ex) {
        LOGGER.error("File not found!");
    } catch (IOException ex) {
        LOGGER.error("I can not read this file!");
    } catch (Exception ex) {
        LOGGER.error("File is empty!");
    }
}
```

Kolejna metoda *setVariables* ustawia te dane przygotowując je do wysłania poprzez JSON'a. Informacje przechowywane są w 12 plikach txt, numerowo odpowiadając kolejnym pytaniom, przechodząc przez repozytorium i serwis, są zapisywane, a na końcu wysyłane za pomocą kontrolera.

3.3.1 Użycie aplikacji Postman

Aplikacja pozwoliła nam na testowanie żądań i zobaczenie wartości przesyłanych w JSON'ie. Poprawne pytanie i odpowiedzi:

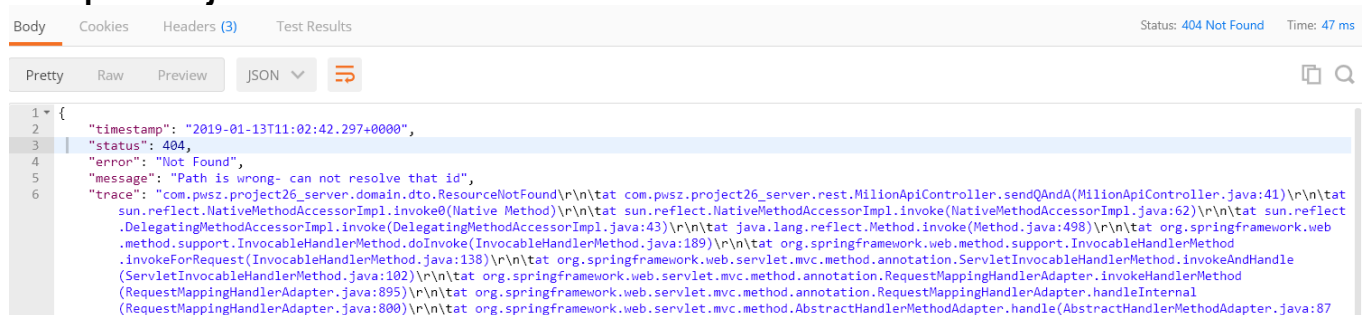


3.3.2 Błędy i wyjątki

Serwer odpowiednio reaguje na wszelakie błędy znalezione podczas zapisu czy wprowadzenia adresu. W rozdziale 3.1 pokazana została główna obsługa w kontrolerze “łapiąca” inne wyjątki, między innymi przypadek błędnego przypadku zapisu do klasy *QuestionDto* (niekompletność danych), a także sprawdzenie, na samym początku, czy w ogóle wprowadzone zostało poprawne *id* w adresie oraz wyjątków odnośnie systemu kodowania Unicode(UTF-8). Sprawdzane są również błędy wynikłe podczas odczytu pliku:

```
}
} catch (FileNotFoundException ex) {
    LOGGER.error("File not found!");
} catch (IOException ex) {
    LOGGER.error("I can not read this file!");
} catch (Exception ex) {
    LOGGER.error("File is empty!");
}
```

Wynik przykładowego błędu wraz z kodem odpowiedzi serwera *404 Not Found* i wiadomością opisującą przyczynę w aplikacji Postman:



The screenshot shows the Postman interface with the 'Body' tab selected. The response is a JSON object indicating a 404 Not Found error. The status is 404, the error is 'Not Found', and the message is 'Path is wrong- can not resolve that id'. A detailed stack trace is provided in the 'trace' field, showing the error occurred in the `sendQAndA` method of `MilionApiController` at line 41, which then propagated through the Spring MVC handler chain.

```
1 {
2   "timestamp": "2019-01-13T11:02:42.297+0000",
3   "status": 404,
4   "error": "Not Found",
5   "message": "Path is wrong- can not resolve that id",
6   "trace": "com.pwsz.project26_server.domain.dto.ResourceNotFound\r\n\tat com.pwsz.project26_server.rest.MilionApiController.sendQAndA(MilionApiController.java:41)\r\n\tat sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)\r\n\tat sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)\r\n\tat sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)\r\n\tat java.lang.reflect.Method.invoke(Method.java:498)\r\n\tat org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMethod.java:189)\r\n\tat org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:138)\r\n\tat org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:102)\r\n\tat org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandlerMethod(RequestMappingHandlerAdapter.java:895)\r\n\tat org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:800)\r\n\tat org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(AbstractHandlerMethodAdapter.java:87
```