Wykonali:
Gabriela Jasnosz
Artur Hamernik
Kamil Budzik
Dominik Irytowski
Kamil Gabrysiak
Patryk Kropisz

Raport do prezentacji Spring Web

Projekt 2

tryb niestacjonarny, I rok, I semestr

(2) W prezentacji przedstawiony zostanie wzorzec projektowy MVC na przykładzie jego użytku we frameworku Spring.
MVC jest jednym z najczęściej stosowanych wzorców projektowych w aplikacjach webowych.

(3) Jak widzimy na bardzo prostej ilustracji - wzorzec nie jest skomplikowany. Architektura MVC ułatwia tworzenie aplikacji webowych poprzez podzielenie ich struktury na trzy główne luźno powiązane ze sobą moduły. Dzięki temu w prosty sposób rozdzielana jest logika biznesowa, interfejs oraz model danych na osobne komponenty, a Spring umożliwia połączenie ich razem.

Wyróżniamy w nim trzy elementy:

model – jest odpowiedzialny za przechowywanie informacji, które będziemy przesyłać między serwerem, a klientem;

widok – definiuje sposób reprezentacji danych, ostatecznie jest to plik HTML wyświetlany użytkownikowi, jednak zanim do tego dojdzie zostaje zazwyczaj wypełniony odpowiednimi danymi z modelu;

kontroler – tutaj zawiera się cała logika naszej aplikacji, jest odpowiedzialny za uzupełnienie modelu oraz zwrócenie klientowi odpowiedniego widoku.

(4) W przedstawionym fragmencie kodu znajduje się bardzo zwięzła implementacja kontrolera z jednym endpointem typu GET. Adnotacja @Controller wskazuje, że dana klasa jest kontrolerem w kontekście modelu MVC.

(5) Zrzut ekranu przedstawia fragment dokumentacji API wygenerowany popularnym narzędziem “Swagger”.

Zawarte są na nim cztery najczęściej spotykane metody HTTP. Ich działanie wygląda następująco:

GET - służy do pobierania zasobu z określonego adresu URL. Żądanie tego typu jest “bezpieczne” - stan pobieranego zasobu nie powinien zostać zmieniony przez tą metodę.

POST - służy do przesyłania żądań tworzących nowy zasób na serwerze. Razem z żądaniem transmitowany jest obiekt w ustalonej reprezentacji.

PUT - używany do aktualizacji całości wybranego zasobu. Jeśli wskazany zasób nie istnieje, jest on tworzony, a metoda powinna zwrócić kod 201 - Created.

DELETE - służy do usunięcia wybranego zasobu z serwera. Każde kolejne wywołanie tej metody powinno zwracać kod 404 Not Found - zakładając, że pierwsze żądanie się powiodło.

OPTIONS - używany do zapytań CORS.

(6) REST w przeciwieństwie do SOAP nie jest protokołem, a stylem architektury, opartym na protokole HTTP. Powstał w późnych latach dziewięćdziesiątych.

REST wykorzystuje wszystkie wcześniej przedstawione metody HTTP.

Popatrzmy na przykładowy URL zasobu przedstawiony na ekranie. Część “<song_id”> jest dodatkową ścieżką wskazującą na konkretny zasób.

Nie sprecyzowanie song_id powinno skutkować operacjami na wszystkich obiektach typu “song”.

W takim razie wykonanie metody HTTP “GET” dla URL “/songs” powinno pobrać wszystkie zasoby typu “song” z kolei GET dla URL “/songs/1” powinien pobrać piosenkę o wskazanym ID.

Analogiczne wyniki powinniśmy obserwować dla innych metod HTTP.

(7) Punkty końcowe zapewniają łatwą i szybką komunikację między serwisami.

Aplikacje klienckie łączą się z aplikacjami serwerowymi pobierając i wysyłając dane. Razem punkty końcowe tworzą api (ang. Application Programming Interface). Wymiana danych między serwisami zachodzi za pomocą zapytań (ang. “requests”) co z kolei sprawia że serwis tworzy odpowiedź (ang. “response”).

(8) Tworzenie punktu końcowego przy pomocy spring web jest bardzo proste. Należy zdefiniować klasę, która będzie służyła nam jako kontroler poprzez użycie adnotacji @Controller. Adnotacja @Controller jest bardziej szczegółową wersją adnotacji @Component, tworzącą warstwę prezentacji, w której tworzymy punkty końcowe.

Aby utworzyć punkt końcowy definiujemy metodę oraz typ, jaki zwraca. Może to być typ prymitywny lub obiekt. W przypadku obiektu zostanie on skonwertowany i zwrócony, w domyślnej konfiguracji, aplikacji klienckiej w postaci obiektu JSON (ang. “JavaScript Object Notation”). Następnie definiujemy za pomocą jakiej metody http oraz jaką ścieżkę URL (ang. “Uniform Resource Locators”) aplikacja wysyłająca zapytanie ma użyć do uzyskania potrzebnego zasobu. W pokazanym przykładzie jest to metoda GET ze ścieżką “/artists/{artist}/songs/{name}”. Adnotacja @ResponseBody sprawia, że wartość zwracana przez metodę getSong() będzie przypisana do sekcji body odpowiedzi HTTP. @PathVariable umożliwia odczytanie wartości przesyłanych przez aplikację kliencką w ścieżce URL pomiędzy nawiasami klamrowymi.

W ciele metody wykonywane są wszelkie potrzebne operacje przeprowadzane w aplikacji serwerowej.

(9) Konfiguracja spring pozwala na tworzenie obiektów w czasie uruchomienia aplikacji (ang. “runtime”). Metoda oznaczona adnotacją @Bean zwraca obiekt zarządzany przez Spring Context. Tak stworzone obiekty możemy w łatwy sposób używać w aplikacji bez konieczności tworzenia nowych instancji interesującej nas klasy. Koncepcja ta nazywa się wstrzykiwaniem zależności (ang. “dependency injection”). Jak można zauważyć na slajdzie przedstawiony jest częściowy (początkowy) cykl życia @Bean.

- Stworzenie kontenera w którym “przechowywane” będą beany
- Tworzenie instancji każdego obiektu bean zdefiniowanego w pliku konfiguracyjnym
- Wstrzykiwanie zależności

(10) Wyjaśnijmy w jaki sposób tworzyć konfiguracje.

Należy oznaczyć klasę adnotacją @Configuration pozwala na tworzenie obiektów klas, podczas wywołania metody która należy do obiektu klasy, jako singleton

Włączamy konfiguracje Spring MVC. Adnotacja @EnableWebMvc jest używana do uruchomienia Spring MVC. Spring MVC wspiera wcześniej wspomniane adnotacje np. @Controller

Należy wskazać pakiety do skanowania (domyślnie spring skanuje wszystkie pakiety będące niżej w hierarchii).@ComponentScan jest używany do określenia jakie pakiety będą skanowane w poszukiwaniu klas używających adnotacji springowych np. @Controller, @Service, @Repository lub bardziej ogólną @Component.

Jak widać na załączonym przykładzie klasa konfiguracyjna implementuje interfejs WebMvcConfigurer który pozwala na nadpisanie implementacji metody configureViewResolvers.

Aby stworzyć obiekt singleton klasy ViewResolver należy zdefiniować beana. Robimy to używając adnotacji @Bean nad metodą która konfiguruje nowo utworzony obiekt klasy ViewResolver

(12) Na slajdzie widzimy kontroler ze zmienną @PathVariable podobną do tej, którą widzieliśmy wcześniej, akceptującej wymagane pole nazwy.

Nasza metoda zwraca tutaj String zamiast ResponseEntity, ponieważ zamierzamy przekazać z powrotem nazwę szablonu, który chcemy obsłużyć w tym żądaniu. Nasza metoda akceptuje klasę Model, do której wrzucamy wartość przekazaną do nazwy ścieżki.

(13) W powyższym kodzie używamy funkcji automatycznego generowania, aby nasz model miał dwie nazwy atrybutów, jedną o nazwie „helloWorld", a drugą „threadbareLoaf".

(14) ModelAndView jest posiadaczem zarówno Modelu, jak i Widoku w webowej strukturze MVC. Te dwie klasy są różne; ModelAndView zawiera tylko oba, aby kontroler mógł zwrócić zarówno model, jak i widok w jednej wartości zwracanej. Klasa ModelAndView obsługuje również alternatywny sposób dodawania atrybutów. Aby dodać atrybuty, możemy użyć metody addObject(). Wtedy możemy po prostu zastąpić "model.put" za "model.addObject"

(15) Zajmowanie się przypadkami błędów, gdy się pojawią, jest kluczowe dla

budowy poprawnie działającej aplikacji internetowej. Mając to na uwadze, przyjrzyjmy się kilku sposobom, dzięki którym można poinformować użytkowników naszej aplikacji o błędzie.

(16) Ponieważ klasa pochodna zawsze posiada klasę bazową jako szablon, konieczne jest zainicjalizowanie klasy bazowej jako pierwszy krok w konstruowaniu obiektu pochodnego. Domyślnie, jeśli nie zostanie wywołane super, Java użyje domyślnego (pozbawionego parametrów) konstruktora do stworzenia klasy bazowej. Jeśli chcemy, aby użyty został inny konstruktor, musimy użyć super, aby przekazać żądane parametry i wywołać właściwy konstruktor.

W przypadku niestandardowych wyjątków tak jak nasz, używamy super do inicjalizacji komunikatu o błędzie wyjątku; przekazując komunikat do konstruktora klasy bazowej, klasa bazowa zajmie się pracą związaną z prawidłowym ustawieniem komunikatu.

Wyjątek z powyższego listingu jest dość standardowy przyjmuje tylko komunikat błędu. Można go dostosować do swoich potrzeb, dodając kody błędów lub inne dane, jeśli uznamy to za konieczne.

Najciekawszę rzeczą tutaj jest sposób, w jaki jest ten wyjątek jest obsługiwany , a umożliwia nam to adnotacja @ExceptionHandler. Wskazuje ona na miejsce, w którym ten wyjątek ma być obsłużony jeśli zostanie rzucony.

(17) W fragmencie kodu w powyższym listingu pokazujemy handler dla naszego niestandardowego wyjątku ArtistNotFoundException.

Jest to klasa komponentu więc należy ją oznaczyć adnotacją @Component, albo jedną z jej specjalizacji, czyli np. @Service, @Repository, czy @Controller w tym wypadku. Wstrzykiwanie zależności przez konstruktor odbywa się w Springu w wersji 4.2 i nowszych automatycznie. Wcześniej należało wskazać, że chcemy do konstruktora wstrzyknąć zależność przy pomocy adnotacji @Autowired.

Adnotacja @ExceptionHandler powie Springowi, że jeśli wyjątek ArtistNotFoundException zostanie rzucony to powinien zostać obsłużony przez metodę handleCustomException w klasie GetArtistsExceptionController. Używa ona obiektu ModelAndView, który poznaliśmy w poprzednim podrozdziale i wypełnia kod statusu 404, ponieważ jest to wyjątek typu "Not Found", komunikat błędu z wyjątku, a sam obiekt jest tworzony z widokiem “error”.

(18) Co się stanie, gdy jednak zdarzy się coś innego i jest to wyjątek, którego nie uwzględniliśmy? Możemy zdefiniować wyjątek "catch-all", jak w fragmencie kodu powyższym listingu. W tym wypadku w adnotacji @ExceptionHandler jak i w parametrze jaki przyjmuje metoda handleAllExceptions użyliśmy klasy Exception po której dziedziczą wszystkie wyjątki. Co sprawia, że jakikolwiek wyjątek by nie był rzucony to zostanie obsłużony przez tą metodę.

Domyślnie będzie on zwracał 500, ponieważ mógł zostać rzucony gdzieś poza naszym kodem i jest to prawdopodobnie uzasadniony błąd serwera, o którym możemy poinformować użytkownika.

(19) Ponieważ używamy metod usługowych z rozdziału 3, a one nie rzucają wyjątków ani nie zwracają nullów, do celów testowych nasza nasza następna metoda na każde zapytanie GET pod endpointem /artists/{artysta} będzie zwracać błąd "404 Not Found".

(20) Nasze definicje ModelAndView w dwóch metodach obsługi wyjątków naszego kontrolera na poprzednich slajdach odwołują się do widoku o nazwie "error". Przyjrzyjmy się naszemu szablonowi błędu. Pokazany szablon jest prosty; buduje stronę błędu i pobiera dane z naszego kontrolera w przypadku, gdy coś pójdzie nie tak. W tytule strony pojawi się kod błędu, a w body paragraf z kodem błędu i komunikatem błędu.