Spring Web

Zaawansowane techniki programowania

Wprowadzenie

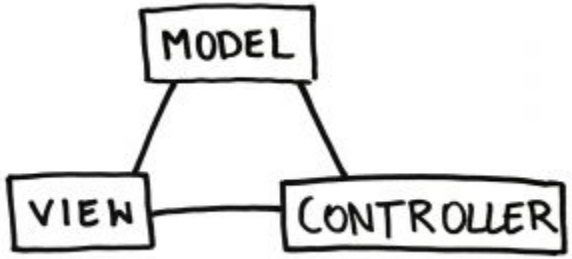
W prezentacji przedstawiony zostanie wzorzec projektowy MVC na przykładzie jego użytku we frameworku Spring. MVC jest jednym z najczęściej stosowanych wzorców projektowych w aplikacjach webowych.
Framework Spring pozwala na implementację MVC w bardzo szybki sposób, bez konieczności konfigurowania środowiska pod jego konkretne wdrożenie.

MVC - Model View Controller

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 od modelu danych oraz interfejsu użytkownika na osobne komponenty, a Spring umożliwia połączenie ich razem.

Cały wzorzec MVC składa się z trzech elementów

- Model – dane i powiązania między danymi
- View – interfejs, wyświetla dane
- Controller – logika aplikacji – odbiera żądania od użytkownika oraz decyduje o przebiegu programu



Hello, World with MVC

```
1  package com.example.chapter6;
2  import org.springframework.http.HttpStatus;
3  import org.springframework.http.MediaType;
4  import org.springframework.http.ResponseEntity;
5  import org.springframework.stereotype.Controller;
6  import org.springframework.web.bind.annotation.GetMapping;
7  import org.springframework.web.bind.annotation.ResponseBody;
8
9  @Controller
10 public class GreetingController {
11   @GetMapping(path = "/greeting", produces = {
12     MediaType.TEXT_PLAIN_VALUE
13   })
14   @ResponseBody
15   public ResponseEntity <String> greeting() {
16     return new ResponseEntity <> ("Hello, World!", HttpStatus.OK);
17   }
18 }
```

Metody HTTP


GET
/pet/{petId}
Find pet by ID
PUT
/pet
Update an existing pet
DELETE
/pet/{petId}
Deletes a pet
POST
/pet/{petId}/uploadImage
uploads an image

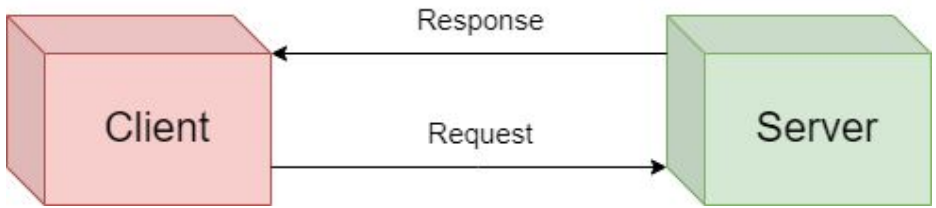
REST (ang. *Representational State Transfer*)

Przykładowy URL:

- **http://api.bandgateway.com/songs/*<song_id>***

Endpoint

Punkt końcowy (ang. “*Endpoint*”) w architekturze REST jest jednym z końców kanału komunikacyjnego między systemami. Służy do przesyłania i pobierania danych za pomocą wcześniej wspomnianych metod http.

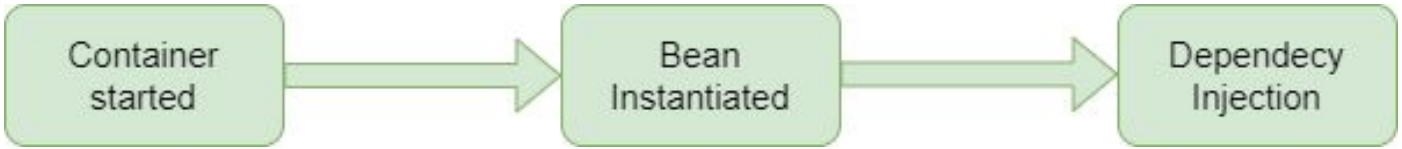


Tworzenie punktu końcowego w spring web

```
1  @Controller
2  public class GetSongsController {
3    @Autowired
4    MusicService service;
5    @GetMapping("/artists/{artist}/songs/{name}")
6    @ResponseBody
7    public ResponseEntity < Song > getSong(
8      @PathVariable("artist") final String artist,
9      @PathVariable("name") final String name
10   ) {
11     String artistDecoded = URLDecoder.decode(artist, StandardCharsets.UTF_8);
12     String nameDecoded = URLDecoder.decode(name, StandardCharsets.UTF_8);
13     Song song = service.getSong(artistDecoded, nameDecoded);
14     return new ResponseEntity < > (song, HttpStatus.OK);
15   }
16 }
```

Konfiguracja

W frameworku spring istnieje możliwość tworzenia lub nadpisywania istniejących konfiguracji obiektów tworzonych w czasie uruchamiania aplikacji między innymi za pomocą adnotacji. Tworząc klasę z adnotacją @Configuration, jesteśmy w stanie tworzyć metody używające adnotacji @Bean pozwalające na stosowanie koncepcji wstrzykiwania zależności (ang. "*Dependency Injection*"). Poniżej można zauważyć częściowy schemat pokazujący cykl życia metody oznaczonej jako @Bean.



Tworzenie konfiguracji w spring

```
1  @Configuration
2  @EnableWebMvc
3  @ComponentScan(basePackages = {
4    "com.bsg5.chapter6",
5    "com.bsg5.chapter3.mem03"
6  })
7  public class GatewayAppWebConfig implements WebMvcConfigurer {
8    @Override
9    public void configureViewResolvers(ViewResolverRegistry registry) {
10     registry.viewResolver(jtwigViewResolver());
11   }
12   @Bean
13   public ViewResolver jtwigViewResolver() {
14     JtwigViewResolver viewResolver = new JtwigViewResolver();
15     viewResolver.setPrefix("web:/WEB-INF/templates/");
16     viewResolver.setSuffix(".jtwig.html");
17     return viewResolver;
18   }
19 }
```

Templates and Models

W Springu są dostępne trzy klasy, których możemy użyć do przeniesienia danych do naszego widoku z klas kontroli: Model, ModelMap i ModelAndView. Poniżej przyjrzyjmy się jak wygląda użycie wspomnianych klas.

Model

Model (M w MVC) jest interfejsem mapy, który pozwala na całkowitą abstrakcję technologii widoku. Za pomocą MVP możemy zintegrować się bezpośrednio z technologiami renderowania opartymi na szablonach, takimi jak JSP, Velocity i Freemarker, lub bezpośrednio generować XML, JSON, Atom i wiele innych typów treści. Mapa modelu jest po prostu przekształcana do odpowiedniego formatu, takiego jak atrybuty żądania JSP, model szablonu Velocity.

```java
package com.bsg5.chapter6;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
@Controller
public class GreetingWithModelController {
  @GetMapping("/greeting/{name}") public String greeting(@PathVariable(name = "name") String name, Model
      model) {
    model.addAttribute("name", name);
    return "greeting";
  }
}
```

```html
< !DOCTYPE html >
< html >
< head >
< title > Hello, {{name}}</title>
</head >
< body >
< p > Hello, {{name}} < /p>
</body >
< /html>
```

ModelMap

Drugą metodą przekazywania danych modelu do naszego widoku jest ModelMap. Ta metoda umożliwia łączenie wywołań łańcuchowych i obsługuje automatycznie generowane nazwy atrybutów na podstawie wartości

```
1  public String greeting(ModelMap map) {
2    map.addAttribute("helloWorld");
3    map.addAttribute("threadbareLoaf");
4    return "greeting";
5  }
```

ModelAndView

Ostatnią metodą przekazywania danych modelu jest ModelAndView. Jest to klasa do zwracania zarówno modelu, jak i widoku w jednym wywołaniu. Podstawowym posiadaczem danych modelu jest ModelMap, a widokiem może być widok np. String.

```
1  public ModelAndView greeting() {
2    Map < String, String > model = new HashMap < > ();
3    model.put("helloWorld", "helloWorld");
4    model.put("threadbareLoaf", "threadbareLoaf");
5    return new ModelAndView("greeting", model);
6  }
```

Error handling

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.

Naszym pierwszym zadaniem jest zbudowanie własnego wyjątku, który będzie po prostu rozszerzeniem RuntimeException i wyeksponowaniem jednej z metod.

Wyjątek niestandardowy

Poniższy wyjątek jest bardzo zbliżony do standardowego i można go dostosować do swoich potrzeb, dodając kody błędów lub inne dane, jeśli uznamy to za konieczne. Sposób, w jaki jest ten wyjątek obsługiwany jest adnotacja @ExceptionHandler. Wskazuje ona na miejsce, w którym ten wyjątek ma być obsłużony jeśli zostanie rzucony.

```
1 public class ArtistNotFoundException extends RuntimeException {
2   private static final long serialVersionUID = 1462190646166272903 L;
3   public ArtistNotFoundException(String message) {
4     super(message);
5   }
6 }
```

Handler wyjątków niestandardowych

W naszym fragmencie kodu pokazujemy handler dla naszego niestandardowego wyjątku ArtistNotFoundException. Adnotacja @ExceptionHandler powie Springowi, że jeśli taki wyjątek zostanie rzucony to powinien zostać obsłużony tutaj i takiego widoku powinien szukać.

```
1  @Controller
2  public class GetArtistsExceptionController {
3    @Autowired
4    MusicService service;
5    @ExceptionHandler(ArtistNotFoundException.class)
6    public ModelAndView handleCustomException(ArtistNotFoundException ex) {
7      ModelAndView model = new ModelAndView("error");
8      model.addObject("message", ex.getMessage());
9      model.addObject("statusCode", 404);
10     return model;
11   }
12 }
```

Handler wszystkich wyjątków

Co się stanie, gdy jednak zdarzy się coś innego i jest to wyjątek, którego nie uwzględniliśmy? Możemy zdefiniować wyjątek "catch-all". Domyślnie będzie on zwracał 500, ponieważ mógł zostać rzucony gdzieś poza naszym kodem.

```
1  @Controller
2  public class GetArtistsExceptionController {
3    @Autowired
4    MusicService service;
5    @ExceptionHandler(Exception.class)
6    public ModelAndView handleAllExceptions(Exception ex) {
7      ModelAndView model = new ModelAndView("error");
8      model.addObject("message", ex.getMessage());
9      model.addObject("statusCode", 500);
10     return model;
11   }
12 }
```

Testowy kontroler

W celach testowych nasza nasza następna metoda na każde zapytanie GET pod endpointem /artists/{artysta} będzie zwracać błąd "404 Not Found".

```
1  @Controller
2  public class GetArtistsExceptionController {
3    @Autowired
4    MusicService service;
5    @GetMapping("/artists/{artist}")
6    @ResponseBody
7    public ResponseEntity < Artist > getSong(
8      @PathVariable("artist") final String artist
9    ) {
10     throw new ArtistNotFoundException("Artist with name " + artist + "
11       not found ");
12     }
13   }
```

Szablon błędu

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 prostą stronę błędu i pobiera dane z naszego kontrolera w przypadku, gdy coś pójdzie nie tak.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title>Error {{ statusCode }}</title>
5  </head>
6  <body>
7      <p>
8          An error has occurred with status: {{ statusCode }}
9          and message: {{ message }}
10 </p>
11 </body>
12 </html>
```

Podsumowanie

W tym rozdziale zobaczyliśmy, jak zbudować bardziej funkcjonalną aplikację internetową, poprzez usunięcie wszystkich ręcznych inwokacji i konwersji oraz servletów z naszej aplikacji. Zobaczyliśmy w jaki sposób radzić sobie z błędami, które mogą wystąpić w trakcie działania aplikacji za pomocą wyjątków. Poznaliśmy również tajniki MVC oraz budowę oraz funkcjonowanie stylu architektonicznego REST.

Wykonali:

- Gabriela Jasnosz
- Artur Hamernik
- Kamil Budzik
- Dominik Irytowski
- Kamil Gabrysiak
- Patryk Kropisz