Programowanie aplikacji w Java

Maciej Gowin

Zjazd 9 - dzień 1

Spring: zakresy ziaren

Zakresy ziaren (ang. bean scopes) definiują cykl życia dla ziaren. Do tej pory było to dla nas transparentne i używaliśmy ziaren z domyślną konfiguracją.

Spring definiuje 6 poziomów, z czego 4 ostatnie tylko dla aplikacji webowych:

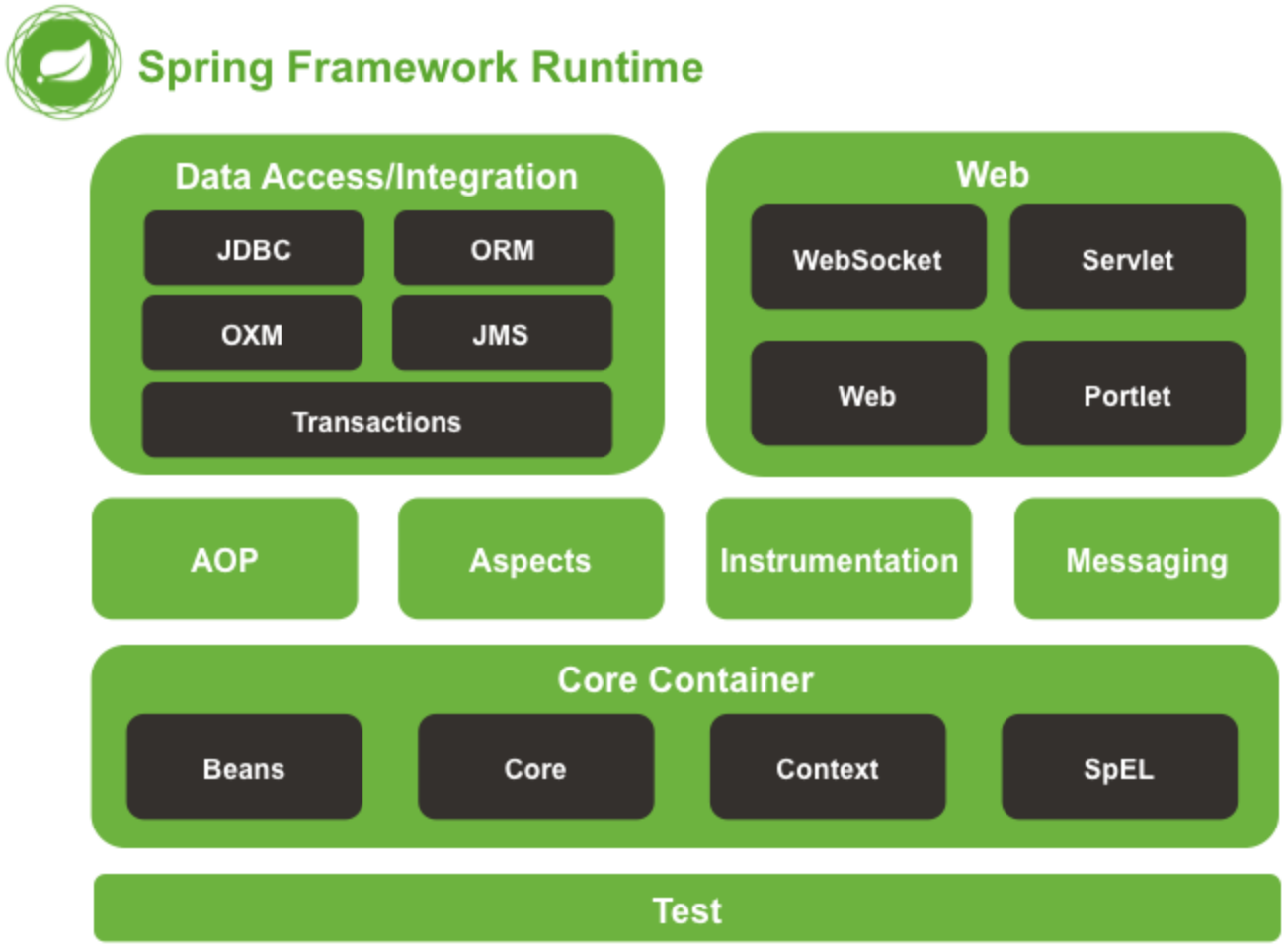
- singleton (domyślny): kontener tworzy jedną instancję ziarna, wszystkie zapytania zwrócą ten sam obiekt.
- prototype: kontener tworzy nową instancję ziarna, wszystkie zapytania zwrócą nowy obiekt.
- request: kontener tworzy instancję ziarna dla każdego zapytania,
- session: kontener tworzy instancję ziarna dla każdej sesji,
- application: kontener tworzy instancję ziarna dla całego cyklu życia obiektu `ServletContext`,
- websocket: kontener tworzy instancję ziarna dla każdej sesji WebSocket.

Spring: moduły

Framework Spring składa się z wielu modułów realizujących dane zadania.

Moduły mogą zostać dodane w razie potrzeb i w wielu przypadkach są niezależne.

Spring: moduły



Źródło: https://spring.io

Spring: moduły

Oprócz głównych modułów wchodzących w skład centralnego projektu Spring Framework, środowisko Spring definiuje szereg rozwiązań pogrupowanych w projekty:

- Spring Framework
- Spring Boot
- Spring Cloud
- Spring Cloud Data Flow
- Spring Data
- Spring Integration
- Spring Batch
- Spring Security

Spring: moduły

Warto zauważyć, że wiele modułów tworzy warstwę abstrakcji opartą o istniejące już rozwiązania.

Dla przykładu moglibyśmy użyć biblioteki Hibernate bezpośrednio, definiując ziarna na podstawie wbudowanych klas Hibernate.

Spring wspiera integrację z JPA przy użyciu Hibernate za pomocą biblioteki `spring-data-jpa` dodając własne rozwiązania ułatwiające użycie JPA oraz jego konfigurację.

Spring Boot

Podczas tworzenia aplikacji przy użyciu frameworka Spring wykonujemy szereg powtarzalnych kroków, które prowadzą nas do stworzenia działającego rozwiązania.

Korzystając z przykładu prostej aplikacji webowej opartej o framework Spring, konieczne było:

- skonfigurowanie serwera Apache Tomcat,
- zdefiniowanie bibliotek Spring w pliku `pom.xml`,
- zainicjalizowanie `DispatcherServlet`,
- skonfigurowanie kontekstu,
- zdefiniowanie podstawowych ziaren dostarczających obsługi JSP.

Rozwiązaniem problemu powtarzalnych czynności jest Spring Boot. Spring Boot pozwala na tworzenie automatycznie skonfigurowanych aplikacji opartych na podejściu `convention-over-configuration`.

Spring Boot

Spring Boot jest rozszerzeniem frameworka Spring dostarczającym domyślnie skonfigurowane rozwiązanie pozwalające na łatwe definiowanie szablonów aplikacji

Dzięki temu możemy skoncentrować się na istotnej logice biznesowej, oddelegowując mniej istotne aspekty infrastruktury do frameworka.

Co ważne, choć Spring Boot dostarcza domyślnych konfiguracji systemu, pozwala również na pełną kontrolę aplikacji oraz nadpisywanie konfiguracji w razie potrzeb.

Spring Boot

Spring Boot charakteryzuje się następującymi funkcjonalnościami:

- łatwe tworzeni aplikacji *standalone*,
- wbudowane serwery aplikacji Tomcat oraz Jetty (brak konieczności uruchamiania plików WAR, uruchamianie serwera wraz z aplikacją poprzez uruchomienie pliku JAR),
- szereg modułów typu *starter* pozwalających na typową konfigurację projektu,
- automatyczna konfiguracja aplikacji, jeżeli jest to możliwe,
- wsparcie dla aplikacji korporacyjnych działających na środowiskach produkcyjnych poprzez moduły metryk, sprawdzania stanu aplikacji (ang. health check),
- pełne wsparcie dla konfiguracji bez konieczności użycia XML.

Spring Boot

W skład Spring Boot wchodzi szereg modułów. Do najistotniejszych zaliczymy:

- `spring-boot`: centralna biblioteka wspierająca inne części Spring Boot, takie jak: wsparcie dla aplikacji *standalone*, wbudowane kontenery aplikacji webowych, wsparcie konfiguracji czy inicjalizacja kontekstu aplikacji.
- `spring-boot-autoconfigure`: wsparcie automatycznej konfiguracji na podstawie dostępnych modułów w ścieżce przeszukiwanie oraz domyślnych ustawień. Głównym zadaniem auto-konfiguracji jest automatyczna dedukcja ziaren, które mogą być potrzebne w aplikacji. Przykładowo podczas dodania zależności do bazy danych auto-konfiguracja postara się automatycznie skonfigurować połączenie z bazą danych na podstawie z góry ustalonych zmiennych.

Spring Boot

W skład Spring Boot wchodzi szereg modułów. Do najistotniejszych zaliczymy:

- `spring-boot-starters`: startery ułatwiają ładowanie zależności i modułów na podstawie ich przeznaczenia, dostarczają pogrupowanych modułów, dzięki czemu możemy zaimportować jeden moduł zawierający wszystkie konieczne zależności. Dla przykładu `spring-boot-starter-web` zawiera wszystkie zależności do rozpoczęcia pracy nad aplikacją webowa oparta o Spring Boot.
- `spring-boot-cli`: wsparcie aplikacji opartych o CLI (Command Line Interface) oraz aplikacji *standalone*.

Spring Boot

W skład Spring Boot wchodzi szereg modułów. Do najistotniejszych zaliczymy:

- `spring-boot-actuator`: wsparcie dla monitoringu, analizy środowiska uruchomieniowego dla aplikacji oraz innych metadanych związanych z aplikacją.
- `spring-boot-actuator-autoconfigure`: wsparcie dla auto-konfiguracji monitoringu oraz bibliotek powiązanych.
- `spring-boot-test`: wsparcie dla testów.
- `spring-boot-test-autoconfigure`: wsparcie dla auto-konfiguracji testów.

Spring Boot

W skład Spring Boot wchodzi szereg modułów. Do najistotniejszych zaliczymy:

- `spring-boot-loader`: biblioteka wspierająca budowanie wykonywalnych plików `jar` wraz z ich zależnościami. Nie jest używana bezpośrednio, ale przez Maven plugin taki jak: `spring-boot-maven-plugin`.
- `spring-boot-devtools`: dodatkowe narzędzia programistyczne, które mogą zostać użyte podczas fazy rozwoju oprogramowania.

Spring Boot w praktyce: SpringApplication.run

Inicjalizacja oraz uruchomienie aplikacji Spring przy użyciu Spring Boot w domyślnej konfiguracji ogranicza się do kilku linii kodu.

Główną ideą Spring Boot jest dostarczenie domyślnych opcji pozwalających na pozbycie się powtarzalnych kroków.

Uruchomienie aplikacji odbywa się poprze wywołanie `SpringApplication.run`.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class);
    }
}
```

Programowanie: przykład 84

Prosta konfiguracji aplikacji webowej Spring Boot.

Spring Boot w praktyce: @SpringBootApplication

`@SpringBootApplication` jest adnotacją, która powoduje automatyczne dodanie następujących adnotacji:

- `@Configuration`: oznaczenie klasy jako klasy konfiguracyjnej będącej źródłem definicji ziaren dla kontenera (kontekstu aplikacji).
- `@EnableAutoConfiguration`: aktywowanie auto-konfiguracji, na podstawie której Spring Boot będzie w stanie zdefiniować domyślną konfigurację systemu oraz aktywować domyślne zachowania. Dla przykładu: w przypadku dodania zależności `spring-webmvc` zostanie automatycznie skonfigurowany `DispatcherServlet`.
- `@ComponentScan`: konfiguracja automatycznego skanowania w poszukiwanie komponentów, serwisów, konfiguracji oraz innego typu elementów definiujących ziarna.

Spring Boot w praktyce: spring-boot-starter-web

Użyliśmy *startera* dla aplikacji webowych: `spring-boot-starter-web`. Dostarcza on wszystkich niezbędnych zależności koniecznych do budowania oraz uruchomienia aplikacji webowych.

Dodatkowo zależność ta posiada domyślne auto-konfiguracje, które pozwalają na łatwe uruchomienie aplikacji.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.6.5</version>
</dependency>
```

Spring Boot w praktyce: tomcat-embed-jasper

W podstawowej konfiguracji Spring Boot oraz `spring-boot-starter-web` nie jest rozszerzony o wsparcie dla JSP. Aby dodać obsługę JSP, posłużymy się zależnością:

```
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
    <version>9.0.60</version>
    <scope>provided</scope>
</dependency>
```

Spring Boot w praktyce: tomcat-embed-jasper

Dzięki użyciu zależności wspierającej JSP oraz auto-konfiguracji ziarno definiujące `ViewResolver` zostało domyślnie utworzone.

Aby było to możliwe auto-konfiguracja automatycznie rozpoznaje zmienne konfiguracyjne wymagane do inicjalizacji ziarna:

```
spring.mvc.view.prefix = /WEB-INF/jsp/
spring.mvc.view.suffix = .jsp
```

Spring w praktyce: application.properties

Spring automatycznie ładuje plik `applition.properties` ze ścieżki przeszukiwania, jeżeli inny nie został zdefiniowany.

W pliku tym umieszczamy szereg wartości dla zmiennych, które:

- są wbudowane we sam framework Spring,
- są wymagane przez biblioteki, z których korzystamy
- sami definiujemy.

Istnieje szereg sposobów, aby dostarczyć plik konfiguracyjny z zewnątrz. Najczęściej spotykanym jest jego zainicjalizowanie z poziomu uruchomienia programu:

```
java -jar application.jar --spring.config.location=file:///Users/home/config/dev.properties
```

Spring Boot w praktyce: spring-boot-maven-plugin

Aby zbudować aplikację posiadającą wszystkie zależności dla typowej aplikacji webowej, wybraliśmy pakowanie typu `war`.

Oczywiście, podobnie jak w przypadku pakowania typu `jar`, plik wynikowy nie będzie posiadał wszystkich koniecznych zależności, aby uruchomić go z poziomu: `java -jar`.

Aby dodać kontekst pakowania aplikacji Spring Boot ze wszystkimi zależnościami, użyjemy *plugin*:

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <version>2.2.2.RELEASE</version>
</plugin>
```

Spring Boot w praktyce: spring-boot-maven-plugin

Dzięki rozszerzeniu procesu budowania o `spring-boot-maven-plugin` możemy uruchomić aplikację z poziomu konsoli oraz projektu Maven poprzez:

```
mvn spring-boot:run
```

Lub też utworzyć plik wynikowy, który będzie zawierał wszystkie zależności poprzez przepakowanie standardowo utworzonego pliku archiwum:

```
mvn clean package spring-boot:repackage
```

Spring Boot w praktyce: spring-boot-maven-plugin

Istnieje też możliwość rozszerzenia standardowego pakowania o automatyczne wykonanie kroku *repackage* poprzez dodanie do `pom.xml`:

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <version>2.2.2.RELEASE</version>
    <executions>
        <execution>
            <goals>
                <goal>repackage</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Po zmianie konfiguracji plik archiwum zostanie automatycznie przeładowany podczas:

```
mvn clean package
```

Spring w praktyce: @ResponseBody

Do tej pory metody zwracające odpowiedź dla danego zapytania zwracały nazwę widoku lub obiekt go reprezentujący. Jest to domyślne zachowanie.

W przypadku, gdy chcemy, żeby wartość zwracana była faktyczną wartością wysyłaną jako odpowiedź zapytania użyjemy adnotacji `@ResponseBody`.

> Wykorzystanie @ResponseBody jest szczególnie przydatne przy tworzeniu serwisów RESTful, które pomijają standardowego warstwę widoku.

Spring w praktyce: @ResponseBody

`@ResponseBody` pozwala na zdefiniowanie ciała zwracanego w postaci ciągu znaków:

```
@GetMapping("/")
@ResponseBody
public String root() {
    return "root";
}
```

Spring w praktyce: @ResponseBody

Co więcej, `@ResponseBody` pozwala na zdefiniowanie ciała zwracanego w postaci obiektu:

```
@GetMapping("/test")
@ResponseBody
public TestDto test() {
    return TestDto.builder().test("test").build();
}
```

```
@Setter
@Getter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class TestDto {

    private String test;
}
```

Spring w praktyce: @RequestBody

Adnotacja `RequestBody` jest analogiczną adnotacją służącą do definiowania ciała w przesyłanym żądaniu. Model ten również może być oparty o `String` lub obiekt.

```
@PostMapping("/")
@ResponseBody
public String postRoot(@RequestBody String root) {
    return "post-" + root;
}

@PostMapping("/test")
@ResponseBody
public TestDto postTest(@RequestBody TestDto testDto) {
    return TestDto.builder().test("post-" + testDto.getTest()).build();
}
```

Programowanie: przykład 85

Prosta konfiguracji aplikacji webowej Spring Boot z prostymi serwisami niedefiniującymi widoków JSP.

Spring Boot w praktyce: spring-boot-starter-web

W przypadku aplikacji webowej, która zwraca dane tekstowe w postaci ciągu znaków, JSON czy też XML nie jest konieczne dostarczenie obsługi JSP. Dzięki temu pozbywamy się zależności oraz katalogu `webapps`. Co więcej, możemy budować archiwum oparte o format `jar`.

W przypadku złożonych obiektów zostały one:

- zserializowane do formatu JSON
- zdeserializowane z formatu JSON.

Zależność `spring-boot-starter-web` zawiera w sobie bibliotekę `jackson` wraz z automatyczną konfiguracją pozwalającą za serializację i deserializację obiektów dla zapytań i odpowiedzi HTTP.

Spring Boot w praktyce: spring-boot-starter-web

W przypadku użycia samego `spring-webmvc` bez wsparcia Spring Boot konieczne byłoby dodanie zależności dla `jackson`:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.13.2</version>
</dependency>
```

Oraz konfigurację konwersji wiadomości dla obiektów do JSON przy pomocy `jackson`:

```
@Bean
public MappingJackson2HttpMessageConverter mappingJackson2HttpMessageConverter() {
    return new MappingJackson2HttpMessageConverter(
        new Jackson2ObjectMapperBuilder().failOnEmptyBeans(false).build());
}
```

RESTful Services

REST jest stylem architektonicznym usprawniającym wymianę danych pomiędzy serwisami. Jego głównym celem jest ustalenie sposobu komunikacji, która za założenia powinna być uniwersalna oraz prosta w konfiguracji.

Podejście to opiera się na formacie danych, który jest używany do ich reprezentacji. Dzięki temu nie jest istotne to, przy jakiego użyciu rozwiązania został on zaimplementowany.

Serwisy webowe dostarczające API. Aby serwis REST mógł zostać określony mianem RESTful, musi spełniać szereg reguł.

RESTful Services: reguły

1. Architektura klient-serwer (ang. client–server architecture)

Rozdzielenie aplikacji na komponenty uruchamiane niezależnie. Odseparowanie warstwy widoku oraz interfejsu użytkownika od warstwy dostarczającej dane.

2. Bezstanowość (ang. statelessness)

Każde zapytanie wysyłane do serwisu nie jest zależne od zapytania poprzedzającego, czego efektem jest brak stanu.

RESTful Services: reguły

3. Wsparcie pamięci podręcznej (ang. cacheability)

Odpowiedzi zapytań mogą zostać przechowywane w pamięci podręcznej. Na poziomie odpowiedzi definiowane jest to, czy dane mogą zostać przechowane w pamięci podręcznej, a jeżeli tak, to w jaki sposób.
Mechanizm ten może byc definiowany na poziomie klienta lub serwera.

4. System wielowarstwowy (ang. layered system)

Podczas komunikacji klienta z serwisem nie jest on w stanie jawnie stwierdzić czy komunikuje się z serwerem docelowym, czy serwerem pośredniczącym.
Komunikacja ta powinna być dla niego transparentna, a sposób odpowiedzi jednolity.

RESTful Services: reguły

5. Jednolity interfejs (ang. uniform interface)

Serwis powinien zwracać dane w jednolitym formacie, który jest zrozumiany przez wszystkich (np. format JSON).
Dobrze zaprojektowany interfejs jest w stanie zaspokoić potrzeby wszystkich klientów (np. strona WWW, aplikacja mobilna).

6. Kod na żądanie (ang. code on demand, opcjonalne)

Serwis może wysyłać do klienta fragmenty kodu, który może zostać później uruchomiony po jego stornie.

RESTful Services: HTTP

W praktyce najpopularniejszą implementacją serwisów RESTful są serwisy oparte o protokół HTTP.

Działają się one na zasobach opisywanych przez URI, na których możemy wykonywać operacje przy użyciu metod HTTP.

Do opisu formantu, który jest używany do wymiany danych, stosujemy typy MIME (ang. media type, MIME types) przesyłane w nagłówkach HTTP.

Informacje o statusie przetworzenia zapytania są przesyłane przy pomocy kodów HTTP.

RESTful Services: HTTP

Każda z metod HTTP ma swoje znaczenie w kontekście zasobów, na których operuje. Możemy zauważyć, że realizują one podstawowe operacje CRUD.

Metod	**Opis**
GET	pobranie zasobu
POST	stworzenie zasobu
PUT	aktualizacja lub zmiana stanu zasobu
DELETE	usunięcie zasobu

RESTful Services: przykład

Zarządzenie zespołami.

Operacja	Opis
GET /teams	pobranie zespołów
POST /teams	stworzenie zespołu
PUT /teams/{teamId}	aktualizacja zespołu
DELETE /teams/{teamId}	usunięcie zespołu

RESTful Services: przykład

Zarządzenie graczami w kontekście zespołu.

Operacja	Opis
GET /teams/{teamId}/players	pobranie graczy
POST /teams/{teamId}/players	stworzenie gracza
PUT /teams/{teamId}/players/{playerId}	aktualizacja gracza
DELETE /teams/{teamId}/players/{playerId}	usunięcie gracza

RESTful Services: dobre praktyki

Do reguł opisujących RESTful możemy jeszcze dodać kilka dobrych praktyk.

- zasoby powinny być definiowane w postaci rzeczowników, a nie czasowników,
- nazwy zasobów opisujemy przy pomocy liczby mnogiej,
- każde z API powinno być wersjonowane, co ułatwia późniejszy rozwój oprogramowania.

Programowanie: przykład 86

Aplikacja w Spring Boot do zarządzania użytkownikami przy użyciu podejścia RESTful.

Spring w praktyce: ResponseEntity

Do tej pory zwracaliśmy pełny obiekt w metodzie obsługującej zapytanie. Spring automatycznie definiuje kod `200` dla poprawnego zapytania. Problem pojawił się w momencie braku zasobu i obsłudze błędu `RuntimeException`.

Spring domyślnie zrealizuje obsługę błędu oraz zdefiniuje ciało odpowiedzi.

```
{
    "timestamp": "2022-04-04T09:16:07.981+00:00",
    "status": 500,
    "error": "Internal Server Error",
    "path": "/customers/123"
}
```

Spring w praktyce: ResponseEntity

Dzięki `ResponseEntity` możemy w pełni sterować odpowiedzią, nie tylko definiując jej ciało, ale pełne parametry, w tym kod odpowiedzi.

Daje nam to możliwość usunięcia kodu, który generował błąd w przypadku braku zasobu.

```
@GetMapping("/customers/{id}")
public ResponseEntity<?> getCustomer(@PathVariable("id") Integer id) {
    return customerService.getCustomer(id)
        .map(this::asDto)
        .map(customer -> new ResponseEntity<>(customer, HttpStatus.OK))
        .orElseGet(() -> new ResponseEntity<>(HttpStatus.NOT_FOUND));
}
```

Spring w praktyce: ResponseStatus

Jeżeli zależy nam jedynie na nadpisaniu statusu odpowiedzi, możemy użyć adnotacji `@ResponseStatus` na poziomie metody.

```
@PostMapping("/customers")
@ResponseStatus(HttpStatus.CREATED)
public CustomerDto addCustomer(@RequestBody CustomerDto customerDto) {
    return asDto(customerService.addCustomer(
                customerDto.getFirstName(), customerDto.getLastName()));
}
```

Programowanie: przykład 87

Użycie nadpisanych odpowiedzi przy użyciu `@ResponseEntity` oraz `@ResponseStatus`.

Spring w praktyce: @ControllerAdvice

Adnotacja `@ControllerAdvice` jest specjalnym typem `@Component`, która definiuje ziarno powalające na automatyczne opakowanie ziaren `@Controller` w dodatkową logikę, która będzie spójna dla każdego z nich.

Przykładem takiej wspólnej logiki może być wsparcie dla obsługi błędów.

Spring w praktyce: @ExceptionHandler

Domyślnie Spring Boot definiuje własną generyczną odpowiedź dla zaistniałych błędów.

```
{
    "timestamp": "2022-04-04T09:16:07.981+00:00",
    "status": 500,
    "error": "Internal Server Error",
    "path": "/customers/123"
}
```

Przy użyciu adnotacji `@ExceptionHandler` na poziomie metody zdefiniowanej w ziarnie `@ControllerAdvice` możemy zdefiniować obsługę interesujących nas błędów oraz przetłumaczyć każdy z nich na odpowiednią odpowiedź HTTP.

Spring w praktyce: @ExceptionHandler

Każda z adnotacji `@ExceptionHandler` definiuje błąd, który zamierzamy obsłużyć.

```
@RestControllerAdvice
public class ApplicationControllerAdvice {

    @ExceptionHandler(MyCustomResourceNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public ErrorDto handle(MyCustomResourceNotFoundException ex) {
        return ErrorDto.builder().message(ex.getMessage()).build();
    }
}
```

Spring w praktyce: ResponseEntityExceptionHandler

Spring MVC definiuje szereg błędów stricte związanych z działaniem `DisptcherServlet`. Dla przykładu:

`HttpRequestMethodNotSupportedException` wystąpi w momencie, gdy wykonamy zapytanie przy użyciu metody HTTP, która nie jest wspierana.

W takim przypadku możemy zdefiniować `@ExceptionHandler`, dla każdego z błędów lub też wykorzystać domyślną obsługę błędów opisaną w `ResponseEntityExceptionHandler`.

Spring w praktyce: ResponseEntityExceptionHandler

W przypadku gdy nasze ziarno opisane jako `@ControllerAdvice` dziedziczy po `ResponseEntityExceptionHandler` automatycznie załadowane zostaną definicje `@ExceptionHandler` z klasy `ResponseEntityExceptionHandler`.

Co więcej, możemy nadpisywać metody z `ResponseEntityExceptionHandler` obsługujące dany błąd.

```
@Override
protected ResponseEntity<Object> handleHttpRequestMethodNotSupported(
        HttpRequestMethodNotSupportedException ex, HttpHeaders headers,
        HttpStatus status, WebRequest request) {
    super.handleHttpRequestMethodNotSupported(ex, headers, status, request);
    return new ResponseEntity<>(
                ErrorDto.builder().message("Method not allowed").build(),
                HttpStatus.METHOD_NOT_ALLOWED);
}
```

Programowanie: przykład 88

Obsługa błędów.

Spring w praktyce

Obsługa błędów w Springu jest najlepszym przykładem tego, że framework ten pozwala na bardzo rozbudowaną konfigurację projektów oraz używanie wielu mechanizmów, które rozwiązują dany problem.

Podczas tworzenia aplikacji opartych o framework Spring istnieje wiele opcji pozwalających na rozwiązanie tego samego problemu.