Moduł 5: Walidacja

Przykładowa aplikacja składa się obecnie z trzech warstw: bardzo ubogiego "widoku", warstwy „usług biznesowych” (pobierających i zapisujących dane) oraz warstwy dostępu do bazy danych. Każda z warstw realizuje konkretne zadania i nie wykracza po nie (prezentując przy tym bardzo dobrą separację: warstwa usług jest w pełni niezależna od przyjętej strategii dostępu do bazy danych). Są jednak wymagania, które w sposób wertykalny przecinają całą aplikację i muszę być zrealizowane na każdym etapie przetwarzania. Taką funkcjonalnością jest na przykład walidacja, którą zajmiemy się właśnie teraz. Walidacja, czyli sprawdzenie poprawności przesyłanych danych. Każda warstwa samodzielnie powinna zweryfikować czy dane które otrzymuje są zgodne z oczekiwaniami (z kontraktem) oraz w przypadku błędnych danych, powinna zasygnalizować klientowi odpowiedni błąd. Jest to pewne rozszerzenie podejścia programowania defensywnego, które nakazuje sprawdzenie poprawności danych przed przystąpieniem do operacji na nich. Spring Framework oferuje w tym miejscu bardzo bogaty zestaw narzędzi wspierających właśnie taką zaawansowaną weryfikację danych.

Bean Validation

Nie jest zaskoczeniem, że istnieje co najmniej kilka sposobów walidacji obiektów dostępnych we frameworku. Jako że autorzy frameworku bardzo duży nacisk kładą na kompatybilność wstecz, można skorzystać ze wszystkich historycznie dostępnych metod walidacji obiektów. Skupimy się jednak na najnowszej i najłatwiejszej metodzie, czyli implementacji standardu Java Bean Validation, opisanego w JSR-303. Spring wspiera ten standard, chociaż nie jest go referencyjną implementacją. Do poprawnego działania wymaga obecności np. referencyjnej implementacja standardu, tj. biblioteki hibernate-validator

JSR-303 standaryzuje sposób deklarowania reguł walidacyjnych na platformie Java. Odbywa się to poprzez adnotację, którymi oznaczamy pola modelu i na tym poziomie definiowane są reguły, które element modelu musi spełnić. W praktyce wygląda to niezwykle prosto i przykładowa, używana w aplikacji klasa modelu, przybierze następującą postać.

```
@Entity
@Table (name = "tasks")
public  class Task {

	@Id
	@GeneratedValue
	@Column(name = "task_id")
	@NotNull
	private long taskId;

	@Temporal(TemporalType.DATE)
	@Column(name = "startdate")
	@NotNull
	@Future
	private Date date;

	@Basic
	@NotNull
	@Min(value = 5*60*1000)
	private int duration;

	@NotNull
	@Size(min = 1, max = 50)
	private String title;

	@NotNull
	@Size(min = 1)
	private String description;

	//..
}
```

Jak widać na przykładzie, nie ma przeszkód aby łączyć w jednej klasie modelu adnotację dotyczące bazy danych, jak i walidacji (a czasem nawet mapowanie obiektów na dokumenty XML). Niewątpliwą zaletą takiego podejścia jest przechowywanie efektywne, ponowne wykorzystanie już istniejącego kodu oraz przechowywanie wszystkich meta danych obiektów wraz z samym obiektem. Poprawia to znacznie czytelność kodu. Niestety ma też minusy – z czasem klasy potrafią urosnąć do monstrualnych rozmiarów, ich czytelność maleje, a narastają problemy z utrzymaniem takich klas. Wciąż jednak deklaratywna walidacja poprzez adnotację jest daleko czytelniejsza niż seria zagnieżdżonych wartości *if* i *else*.

Tak zmodyfikowana klasa może być użyta do walidacji. W pierwszy kroku należy w pliku konfiguracyjnym wskazać, który walidator (udostępniany przez framework) pragniemy użyć, a następnie wstrzyknąć go do odpowiedniego serwisu.

```
<bean                                                    id="validator"
class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
```

W tym momencie Spring daje nam wybór, czy korzystamy ze standardowego Bean Validation API i wstrzykujemy instancję `javax.validation.Validator` czy korzystamy z klasy dostarczonych przez framework (`org.springframework.validation.Validator`). Jako że szkolenie dotyczy Springa omówimy interfejs frameworka. Aby jednak standardom stało się zadość, użycie podstawowego API wyglądać będzie następująco.

```
	public void addTask(Date startDate, int duration, String title,
			String description) throws Exception {

		Task t = new Task.Builder().withTitle(title)
				.withDescription(description).withDate(startDate)
				.withDuration(duration).build();

		Set<ConstraintViolation<Task>> errors = validator.validate(t);

		if (errors.size() > 0) {
			System.out.println("Validation errors");
			for (ConstraintViolation<Task> e : errors)
				System.out.println(" * " + e.getMessage());
		}

		sessionFactory.getCurrentSession().save(t);
	}
```

Przed zapisaniem do bazy obiekt Task został sprawdzony pod kątem poprawności i ewentualne błędy zostały wyświetlone w konsoli. Istotne jest, że użycie standardowego API wymaga dołączenia do projektu walidatora zgodnego z API (np. modułu hibernate-validator). W przeciwnym wypadku walidacja nie zadziała, mimo że przywołana w konfiguracji klasa implementuje interfejs `javax.validation.Validator`.

Ta sama operacja przy użyciu komponentów Springa wyglądać będzie odrobinę inaczej. Inny jest interfejs klasy walidatora, a także wymagane jest przygotowanie a priori struktury danych dla ewentualnych błędów. Użyta została klasa `BindException`, która jest domyślną implementacją interfejsu `Error` (o którym więcej za chwilę). Klasa ta standardowo inicjowana jest przez framework i w przypadku takich prostych zastosowań wydaje się to być nieco na wyrost. Niemniej jednak, np. w aplikacji webowej taka konstrukcja okazuje się być o wiele bardziej elastyczna niż standardowe `ConstraintViolation`.

```
	public void addTask(Date startDate, int duration, String title,
			String description) throws Exception {

		Task t = new Task.Builder().withTitle(title)
				.withDescription(description).withDate(startDate)
				.withDuration(duration).build();

		BindException errors = new BindException(t, "task");
		validator.validate(t, errors);

		if (errors.hasErrors()) {
			System.out.println("Validation errors:");
			for (Object e : errors.getAllErrors()) {
				System.out.println(" * " + e);
			}
		}

		sessionFactory.getCurrentSession().save(t);
	}
```

Ostatnią istotną z punktu widzenia programisty kwestią jest to, że w momencie dołączenia do projektu komponentów odpowiedzialnych za walidację, uruchamiane są wewnętrzne mechanizmy Springa, które uniemożliwiają zapisanie niepoprawnych danych do bazy. Mimo, że po wykryciu błędów walidacji wywołanie metody nie zostało przerwane, Spring sam zadbał o to aby niepoprawne dane nie zostały zapisane i poprzez odpowiednie klasy nasłuchujące zwrócił wyjątek. Nastąpiło to

niezależnie od tego czy użyta została standardowa biblioteka JSR-303 czy komponenty związane z frameworkiem.

Błędy walidacji i interfejs Error

W sytuacji, gdy użyte zostały narzędzia do walidacji dostarczane przez framework, konieczne stało się przygotowanie struktury danych, w której przechowywane i przekazywane były błędy walidacji. Rzadko kiedy w aplikacjach spotyka się tak explicite wykorzystanie walidacji dostarczanej przez Springa. Tak jak wspominałem, w prostszych przypadkach wykorzystuje się standardowe podejście oferowane przez standard Bean Validation. Walidacja z wykorzystaniem interfejsów frameworka okazuje się być szczególnie pomocna np. przy obsłudze formularzy na stronach internetowych. Wykorzystywane wtedy są mechanizmy DataBinding i Spring samodzielnie zajmuje się tworzeniem obiektów, wypełnianiem wartości oraz walidacją. Niemniej jednak, niezależnie od wybranego sposobu walidacji, błędy zawsze zwracane są w tej takiej samej strukturze, gdzie kluczem jest nazwa atrybutu (patrząc od głównego korzenia). Na tej podstawie możliwe jest odczytanie wyniku walidacji (komunikat o błędzie), jak i wartości, która zostały odrzucone.

We wszystkich powyższych przykładach informacje o błędach były na sztywno zapisane w kodzie programu. Nie jest to podejście zalecanie, szczególnie gdy planujemy aby tworzony system wspierał więcej niż jeden język. W takiej sytuacji, istnieje możliwość wykorzystania interfejsu `MessageBundle` i przechowania treści wiadomości w zewnętrznym pliku, posługując się w kodzie jedynie kluczem. Aby móc skorzystać z takiej własności frameworka musimy kolejny komponent w konfiguracji tworzonej aplikacji.

```
<bean                                                    id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <property name="basename" value="classpath:messages" />
    <property name="defaultEncoding" value="UTF-8"/>
</bean>
```

Powyższa instrukcja informuje framework, że przygotowane są wersje językowe, których należy użyć w aplikacji. Poprzez atrybuty *basename*, wskazujemy lokalizację plików z tłumaczeniami. Jest to bazowa nazwa pliku, gdzie dla każdego kolejnego języka dodawana jest odpowiednia końcówka (np. `messages_en.properties`, `messages_pl.properties`).

Jeżeli korzystamy z webowego frameworka (np. Spring MVC) to odpowiednie wartości mogą być wybierane automatycznie, na bazie danych przekazywanych przez przeglądarkę (takich jak domyślna lokalizacja). Jeżeli – tak jak w przypadku przykładowej aplikacji - praca związana z wyświetlaniem informacji wykonywana jest ręcznie, przez programistę, wykorzystać należy komponent `MessageSource`.

```
@Resource
MessageSource messages;

//..

String englishMessage = messages.getMessage(
      e.getCode() + "." + e.getObjectName() + "." + e.getField(),
      null, Locale.ENGLISH);
System.out.println(" * " + englishMessage);

String polishMessage = messages.getMessage(
      e.getCode() + "." + e.getObjectName() + "." + e.getField(),
      null, new Locale("PL"));
System.out.println(" * " + polishMessage);
```

Konstrukcja pliku message_xx.properties jest dowolna, ważne jest jedynie aby klucze zgadzały się na linii aplikacji, plik z tłumaczeniami. Konwencja sugeruje wykorzystanie nazewnictwa [nazwa_błedu].[obiekt].[atrybut_obiektu] (i taka konwencja wykorzystywana jest np. w aplikacjach webowych). Dlatego też w powyższym przykładzie plik wyglądać będzie następująco (odpowiednio dla języka angielskiego i polskiego).

```
Future.task.date = Invalid date; expected date from the future
Future.task.date = Nieprawid\u0142owa data.
```