Moduł 4: Warstwa bazy danych

W poprzednich dwóch częściach składaliśmy aplikację z komponentów, wydzielając obszary odpowiedzialne za realizację poszczególnych wymagań biznesowych. Wszystkie dotychczasowe komponenty działały (w pewnym uproszczeniu) w jednej warstwie; niemalże bezpośrednio wywoływaliśmy usługi realizujące poszczególne funkcjonalności, bez pośrednictwa interfejsu użytkownika. Nie sposób jest jednak zamknąć wszystkie operacje w jednej warstwie – zresztą byłoby to niewskazane. Projektując aplikację, staramy się rozdzielać pewne obszary funkcjonalne. Pierwszym takim obszarem będzie baza danych, którą zajmiemy się teraz. Kolejnym może być interfejs użytkownika, lecz metodami umożliwiania użytkownikowi dostępu do aplikacji (np. poprzez wymyślny interfejs) zajmiemy się później.

Nie będę starał się tutaj odpowiedzieć, dlaczego rozdzielać warstwy aplikacji – zdecydowanie wykracza to poza zakres i tematykę tego szkolenia. Zresztą, w kolejnych modułach odpowiedź na to pytanie zacznie nasuwać się samoistnie, szczególnie w części poświęconej testowaniu.

Konfiguracja bazy danych

Konfiguracja dostęp u do bazy danych w aplikacji opartej o Spring Framework wygląda podobnie do sposobu konfigurowania wszystkich innych komponentów, w końcu klasy realizujące komunikację z bazą danych to także komponenty Spring. Jednakże aby w pełni zrozumieć istotę konfiguracji źródła danych, musimy wspomnieć do plikach konfiguracyjnych, które przez poprzednie dwa moduły tak skrzętnie pomijaliśmy. Dotychczas całość konfiguracji ograniczała się do dwóch wpisów w pliku XML, a wszystkie inne relacje definiowaliśmy na poziomie pisanych przez nas komponentów. Teraz niestety tak nie da się zrobić, komponenty realizujące dostęp do danych są już napisane, są elementami frameworka, a pozostała do wykonania praca sprowadza się do odpowiedniej inicjacji tych komponentów. Niestety, wykracza to poza proste skanowanie pakietów i wymaga większego nakładu pracy konfiguracyjnej.

Jak to z reguły ma miejsce w Springu, konfiguracja może odbywać się na więcej niż jeden sposób: poprzez pliki XML oraz poprzez JavaConfig. Jako że każda aplikacja bazująca na Spring Framework wymaga choćby minimalnego pliku XML, wykorzystajmy ten już istniejący. W dotychczasowych przykładach utworzony został prosty plik konfiguracyjny i kilka komponentów, które umożliwiały dodawania, pobieranie zadań oraz wyświetlenie zadań w konsoli.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">

       <context:component-scan base-package="pl.devcastzone.spring" />
       <context:property-placeholder
              location ="classpath:date-format.properties" />

</beans>
```

Jako że komponenty realizujące dostęp do bazy danych są już napisane, musimy je tylko skonfigurować. Najprostszym sposobem jest komunikacja z bazą danych bezpośrednio poprzez JDBC, z wykorzystaniem komponentu `DataSource`. Spring umożliwia komunikację z bazą danych poprzez szereg komponentów o różnym stopniu skomplikowania: od bezpośredniej komunikacji z wykorzystaniem JDBC, poprzez szereg dodatkowych poziomów abstrakcji, umożliwiających separacje programisty od bazy danych (JDBC, bezpieczny wątkowo JDBCTemplate, JPA, Hibernate jak też inne systemy ORM, na różnego rodzaju data mapperach skończywszy – np. iBatis). Wszystkie te sposoby zostaną omówione w obrębie niniejszego modułu. Zacznijmy jednak od sytuacji najprostszej: bezpośrednie wykorzystanie źródła danych (`DataSource`).

Podstawowy obiekt DAO

Jako bazy danych użyjemy HSQLdb – czyli bazy danych typu embedded. Dołączenie jej do projektu ogranicza się do dodania jednej biblioteki (hsqldb.jar) bądź dodania nowej zależności w pliku konfiguracyjnym Maven. Konfiguracja takiej bazy danych wyglądać będzie następująco

```
       <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
              destroy-method="close">
              <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
              <property name="url"
                     value ="jdbc:hsqldb:file:target/localdb/testdb" />
              <property name="username" value="sa" />
              <property name="password" value="" />
       </bean>
```

Tak przygotowane źródło danych możemy użyć w aplikacji. Jeżeli uzupełnimy bazę danych to odwołanie się do źródła danych wyglądać będzie następująco

```
create  table tasks (
   task_id bigint generated by default as identity,
   title varchar(255),
   description varchar(255),
   startdate datetime,
   duration bigint,
   primary  key (task_id)
)

insert   into tasks values (1, 'Database task', 'A hour task for preparing the
database', '2012-01-22', 3600000)
insert into tasks values (2, 'Programming task', 'A hour task for implementing
database access', '2012-01-23', 3600000)
```

Zmodyfikowany TasksService, odczytujący dane z bazy danych, powinien wyglądać następująco:

```
@Component
public  class TasksService {

	@Resource
	DataSource dataSource;

	//..

	public List<Task> getTasks() throws Exception {
		Connection connection = dataSource.getConnection();
		PreparedStatement statement = connection
				.prepareStatement("SELECT * FROM TASKS");

		statement.execute();
		ResultSet resultSet = statement.getResultSet();

		List<Task> tasks = new ArrayList<Task>();
		while (resultSet.next()) {
			Task t = new Task();
			t.setTitle(resultSet.getString("TITLE"));
			t.setDescription(resultSet.getString("DESCRIPTION"));
			t.setDate(resultSet.getDate("STARTDATE"));
			t.setDuration(resultSet.getInt("DURATION"));
			tasks.add(t);
		}

		connection.close();
		return tasks;
	}

}
```

Analogiczną metodą możemy próbować zapisać dane do bazy.

```
	public void addTask(Date startDate, int duration, String title,
			String description) throws Exception {
		Connection connection = dataSource.getConnection();
		PreparedStatement statement = connection
			.prepareStatement("INSERT INTO TASKS" +
				" (title, description, startdate, duration)" +
				" VALUES (?, ?, ?, ?)");
		statement.setString(1, title);
		statement.setString(2, description);
		statement.setDate(3, new java.sql.Date(startDate.getTime()));
		statement.setInt(4, duration);
		statement.execute();

		connection.close();
	}
```

Nie ulega jednak kwestii że taka metoda dostępu nie jest najlepsza. Jak widać już na powyższym przykładzie, zmusza ona programistę do pamiętania o szeregu dodatkowych czynności (zamykanie i otwieranie połączenia do bazy danych, obsłudze wyjątków `SQLException`), nie wiadomo jak kod się zachowa w sytuacji gdy będzie działać w wielowątkowej aplikacji, a na koniec (z czysto ludzkiego punktu widzenia) zaprezentowany kod nie wygląda intuicyjnie i niejednoznacznie wyraża wolę programisty. Sens pobierania danych z bazy jest oddany, ale ilość dodatkowego kodu powoduje że procedura staje się nieczytelna. Wykorzystajmy zatem dodatkowy komponent oferowany przez Spring Framework, ułatwiający komunikacje z bazą danych: `JdbcTemplate`.

Użycie komponentu JDBC Template

Jak widać było w pierwszych przykładach, wszystkie zadania związane z połączeniem do bazy danych leżały w gestii programisty; było to:

- Konfiguracja połączenia do bazy danych (lokalizacja źródła danych, nazwa użytkownika i hasło itd);
- Otwieranie połączenia do bazy danych;
- Przygotowanie zapytania do bazy oraz podanie parametrów;
- Przygotowanie i wywołanie zapytania;
- Pobranie wyników oraz iteracja po nich (z wykorzystaniem ResultSet);
- Obsługa błędów i transakcji (tego elementu jeszcze nie poruszaliśmy, zakładając, że wszystko się powiodło – wrócimy do tego tematu w dalszej części modułu);
- Zamknięcie połączenia do bazy danych.

Jak widać, kroków było sporo, z czego większość jest powtarzalna i aż prosi się o zastosowanie pewnych wzorców i uproszczeń. Właśnie w tym celu używa się komponentu do obsługi JDBC dostarczanych przez framework. Większość z wymienionych powyżej operacji odbywa się wtedy automatycznie (wykonuje ją framework) ograniczając rolę programisty to rzeczy najważniejszych: konfiguracji parametrów połączenia do bazy danych (konfiguracji `DataSource`) oraz pisania zapytań SQL i ich parametryzowanie. Liczba rzeczy, o których należy pamiętać zmniejsza się z początkowych ośmiu kroków – do dwóch.

Konfiguracja źródła danych

Klasa `JdbcTemplate` jest bezpieczna wątkowo, tzn. że wystarczy skonfigurować jedną instancję komponentu, aby móc z powodzeniem wykorzystywać go w całej aplikacji. Możemy usprawnić istniejącą aplikację i zamiast bezpośredniego korzystania ze źródła danych, użyjmy `JdbcTemplate`.

Przede wszystkim, nasze zmiany nie pociągają za sobą konieczności modyfikowania pliku konfiguracyjnego XML – definicja komponentu `DataSource` pozostaje tak jak była. Przestajemy jedynie korzystać bezpośrednio z bazy i wprowadzamy pośrednika:

```
@Component
public  class TasksService {

	private JdbcTemplate jdbcTemplate;

	@Autowired
	public TasksService(DataSource dataSource) {
		this.jdbcTemplate = new JdbcTemplate(dataSource);
	}

	//..

}
```

Samo użycie klasy `JdbcTemplate` także nie powinno nastręczać problemów; wykorzystanie jej do pobrania zadań z bazy wygląda daleko bardziej intuicyjnie niż poprzednia wersja wymagająca otwarcia połączenia do bazy, a następnie pracy na obiekcie `ResultSet`.

```
	public List<Task> getTasks() throwsException {
		List<Map<String, Object>> queryForList = jdbcTemplate
				.queryForList("SELECT * FROM TASKS");

		List<Task> tasks = new ArrayList<Task>();
		for (Map<String, Object> r: queryForList) {
			Task t = new Task();
			t.setTitle((String)r.get("TITLE"));
			t.setDescription((String)r.get("DESCRIPTION"));
			t.setDate((Date) r.get("STARTDATE"));
			t.setDuration(((Long)r.get("DURATION")).intValue());
			tasks.add(t);
		}

		return tasks;
	}
```

Komponentu `JdbcTemplate` można także użyć do zapisania danych do bazy; korzystając z metody update zdecydowanie upraszcza się logikę zapisu do bazy. Napisana poprzednio metoda addTask() wyglądać będzie następująco.

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

		jdbcTemplate.update("INSERT INTO TASKS" +
			" (title, description, startdate, duration)" +
			" VALUES (?, ?, ?, ?)",
			title,
			description,
			new java.sql.Date(startDate.getTime()),
			duration);
	}
```

Jak widać na powyższych dwóch przykładach, skorzystanie z zestawu narzędzi do JDBC znacząco ułatwia realizowanie komunikacji z bazą danych. Nadal korzystamy z zapytań SQL, jednakże nie ma już konieczności pisania kodu obsługującego połączenie, który z punktu widzenia logiki biznesowej jest całkowicie zbędny. Takie podejście doskonale prezentuje wartości, które leżą u podstaw całego framworka, czyli zwiększenie efektywności programisty poprzez przeniesienie uwagi na czynności istotne z punktu widzenia aplikacji (logika, realizacja funkcjonalności). Narzędzia, które zwalniają programistę z pisania kodu nadmiarowego, niepotrzebnego z punktu widzenia logiki aplikacji, są wpisane w ekosystem Spring Framework i `JdbcTemplate` jest tego najlepszym przykładem. Co więcej, mniejsza ilość kodu to także mniejszy obszar potencjalnych błędów i problemów, zmniejsza ilość kodu potrzebnego do przetestowania. Nasze rozwiązanie staje się solidniejsze, ponieważ ma bardzo dobrą bazę w postaci stabilnego i dobrze przetestowanego frameworku.

Rodzina klas Spring Framework JDBC

`JdbcTemplate` to najpopularniejszy sposób dostępu do bazy danych, ale nie jedyny. Na bazie tej klasy utworzona została cała rodzina pomocniczych komponentów o bardziej specyficznym zastosowaniach, oferujących lepszy komfort pracy bądź zwiększające czytelność kodu. Jeżeli zamiast znaków zapytania („?”) pragniemy nazywać parametry zapytania, możemy skorzystać z `NamedParameterJdbcTemplate` lub `SimpleJdbcTemplate`, które łączy w sobie najpopularniejsze metody dwóch pierwszych komponentów.

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

		Map<String, Object> map = new HashMap<String, Object>();
		map.put("title", title);
		map.put("desc", description);
		map.put("date", new java.sql.Date(startDate.getTime()));
		map.put("duration", duration);

		jdbcTemplate.update("INSERT INTO TASKS" +
				" (title, description, startdate, duration)" +
				" VALUES (:title, :desc, :date, :duration)",
				map);
	}
```

W sytuacji gdzie aplikacje będzie wykonywała wiele zapisów do bazy, można skorzystać z `SimpleJdbcInsert`, które minimalizując ilość potrzebnego kodu do wykonania zapytania, opierając się na meta danych dostarczanych przez bazę danych. Wykorzystując `SimpleJdbcInsert` należy szczególną uwagę zwrócić na nazwy parametrów; aby nazwy kluczy w mapie odpowiadały nazwie kolumn w bazie danych – taka jest konwencja wykorzystywana do budowania zapytań.

```
@Component
public  class TasksService {

	private SimpleJdbcTemplate jdbcTemplate;
	private SimpleJdbcInsert insertTask;

	@Autowired
	public TasksService(DataSource dataSource) {
		this.jdbcTemplate = new SimpleJdbcTemplate(dataSource);
		this.insertTask = new SimpleJdbcInsert(dataSource)
			.withTableName("tasks");

	}

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

		Map<String, Object> map = new HashMap<String, Object>();
		map.put("title", title);
		map.put("description", description);
		map.put("startdate", new java.sql.Date(startDate.getTime()));
		map.put("duration", duration);

		insertTask.execute(map);
	}

	//..

}
```

Mapowanie obiektowo relacyjne w Springu

Alternatywą do bezpośredniego operowania na danych za pomocą języka SQL jest mapowanie obiektowo relacyjne (ang. Object Relational Mapping – ORM). Wykorzystanie takiego podejścia umożliwia programiście pozostanie w dziedzinie obiektów, podczas gdy wszystkie operacje bazodanowe będą wykonane automatycznie przez framework, który całkowicie zwolni nas z pisania zapytań w języku SQL. Wprowadzenie silników ORM miało przede wszystkim umożliwić wyraźnie podzielenie aplikacji na warstwy, unikając mieszania kodu odpowiedzialnego za współpracę z bazą danych z kodem biznesowym. W sytuacji gdy całość operacji bazodanowych zamknięta jest w niezależnym frameworku, programista przestaje operować w domenie SQL, skupiając się wyłącznie na warstwie obiektów. Reprezentacja pobieranych danych to obiekty a nie tablice, mapy – co

pozwala znacząco uprościć model domenowy aplikacji oraz znacznie zmniejsza ilość kodu niezbędnego do napisania. Pozwala skupić się nad logiką biznesową oraz modelu obiektowym, a nie myśleć kategoriami encji i relacji bazodanowych, których trudniej jest użyć do modelowania domeny biznesowej.

Nie jest to miejsce, aby omawiać wady i zalety silników ORM, jednakże kilka słów wprowadzenia dla osób niemających dotychczas do czynienia z tą technologia. Z początku, może się wydawać, że automatyczne tworzenie zapytań SQL jest niebezpiecznie i może być mało optymalne. Nic bardziej mylnego, w przypadku ogólnym, zapytania te będą daleko bardziej wydajne niż te napisane przez statystycznego programistę. Dodatkowo framework posiada szereg mechanizmów i algorytmów optymalizujących i cechujących, co dodatkowo, pozytywnie wpływa na wydajność. Bardzo często, jest to wystarczające, aby pokryć narzut spowodowany przez dodatkową warstwę abstrakcji.

Obiekty i encje

Pierwszy krokiem, jeszcze poprzedzającym konfigurację samego narzędzia, należy odpowiednio przygotować obiekty które będą odzwierciedlały byty bazodanowe. Nie jest konieczne tworzenie nowych obiektów, możemy w tym celu, za pomocą adnotacji, dekorować już istniejące klasy modelu (np. klasę Task, wykorzystywaną w przykładzie). Wykorzystywane adnotacje wchodzą w skład JPA i są standardem języka Java opisanym w JSR-220 oraz JSR-317 (JPA 2.0). Zmodyfikowana klasa Task wygląda następująco.

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

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

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

	@Basic
	private int duration;
	private String title;
	private String description;

	//..

}
```

Jeżeli poprzestaniemy tylko na oznaczeniu klasy adnotacjami JPA, nic nie zmieni się w sposobie działania aplikacji. Adnotacje JPA są nieinwazyjne, tzn. ich wprowadzenie nie zmienia sposobu działania aplikacji i dotychczasowe metody komunikacji z bazą danych nie muszą być zmieniane. Pokazane w przykładzie adnotację są minimalne, aby JPA zadziałało w kontekście przykładowej aplikacji, wykraczają one jednak poza minimalny zestaw adnotacji konieczny to traktowania klasy jako encji. Pełne omówienie JPA wykracza daleko poza zakres niniejszego szkolenia.

Kolejnym krokiem będzie konfiguracja `EntityManagera`

Konfiguracja EntityManagera

JPA jest standardem, zbiorem adnotacji i interfejsów, które następnie są implementowane przez frameworking ORM: Hibernate, EclipseLink (referencyjna implementacja JPA 2.0), TopLink, OpenJPA.

W przypadku ogólnym możemy korzystać z dobrodziejstw JPA bez konieczności wiązania się z konkretną implementacją; wykorzystując kilka pośredniczących komponentów dostarczanych przez Springa, kod aplikacji będzie wolny od konkretnych implementacji – bazując tylko na standardowych interfejsach. Wszystkie odniesienia do Hibernate pozostaną na poziomie plików konfiguracyjnych.

Należy jednak pamiętać, że JPA jest standardem pomyślanym dla serwerów aplikacji, gdzie źródło danych (`PersistenceUnit`) definiowane i zarządzane jest przez serwer, zmuszając aplikacje do korzystania z już istniejącej jednostki i pobierania jej z drzewa JNDI. Jednakże w przypadku naszej aplikacji jest to trochę bardziej skomplikowane – nie korzystamy z drzewa JNDI. Dlatego zamiast odniesienia do lokalizacji źródła danych w drzewie JNDI należy samodzielnie wskazać implementację JPA, która będzie wykorzystywana, a następnie ręcznie skonfigurować połączenie ze źródłem danych. Spowoduje to pewne zmiany w plikach konfiguracyjnych.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
		http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
	version="2.0">
	<persistence-unit name="pu" transaction-type="RESOURCE_LOCAL">
		<provider>org.hibernate.ejb.HibernatePersistence</provider>
	</persistence-unit>
</persistence>
```

```
<bean                                                                    id="myEmf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
	<property name="dataSource" ref="dataSource" />
	<property name="persistenceXmlLocation"
		value ="classpath:META-INF/persistence.xml"></property>
</bean>
```

Tak przygotowane pliki konfiguracyjne spowodują że w momencie budowania kontenera utworzona zostanie fabryka `EntityManagerFactory`, którą następnie wykorzystamy w serwisie, znacznie upraszczając pobieranie danych z bazy, za pomocą standardowych metod JPA.

```
@Component
public  class TasksService {

    @PersistenceContext
    private  EntityManager entityManager;

    //..

       public List<Task> getTasks() throws Exception {
              List<Task> tasks = entityManager.createQuery("from Task",
                            Task.class).getResultList();
              return tasks;
       }

}
```

Zastosowanie i ograniczenie się do standardowej biblioteki JPA i EntityManager'a ma swoje ograniczenia, jest szereg metod i adnotacji, z których nie możemy skorzystać. W przypadku tak małej aplikacji może to być przerost formy nad treścią. W sytuacji jednak, gdyby aplikacja miała zostać uruchomiona na serwerze aplikacji (np. JBoss), mała tylko zmiana konfiguracji pozwoli na zastosowanie bazy danych dedykowanej serwerowi aplikacji. Sposób, w jaki taka zmiana mogłaby się odbyć zostanie omówiony w dalszych modułach szkolenia, przy okazji dokładnego omawiania plików

konfiguracyjnych XML. Wtedy też zostaną pokazane sposoby dzielenia plików konfiguracyjnych na mniejsze części, aby umożliwić elastyczne wdrażanie aplikacji w zależności od środowiska.

Konfiguracja Hibernate

Hipotetyczna elastyczność, opisana w poprzednim punkcie, bardzo często jest zbędna. Budując aplikację, tworzymy ją pod kątem konkretnego środowiska, z założeniem że nigdy nie konieczności przenoszenia jej pomiędzy różnymi serwerami (bądź w ogóle korzystania z serwera aplikacji). Zastosowanie komponentu `EntityManager` jest zbędną elastycznością i skoro pod spodem i tak korzystamy np. z Hibernate (jako frameworka ORM), to przywołajmy go bezpośrednio. Uprości to znacznie konfigurację (przede wszystkim poprzez likwidację dodatkowego pliku konfiguracyjnego persistence.xml). Hibernate jest trochę bardziej explicite, łącznie z koniecznością wskazywania encji (lub pakietów, które należy w poszukiwaniu encji przeskanować).

```
<bean                                                          id="mySessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
	<property name="dataSource" ref="dataSource" />
	<property name="packagesToScan" value="pl.devcastzone.spring.model" />
	<property name="hibernateProperties">
		<value>
			hibernate.dialect=org.hibernate.dialect.HSQLDialect
		</value>
	</property>
</bean>
```

```
@Component
public class TasksService {

	@Resource
	private SessionFactory sessionFactory;

	//..

	public List<Task> getTasks() throws Exception {
		List<Task> tasks = (List<Task>) sessionFactory.openSession()
				.createQuery("from Task").list();
		return tasks;
	}

}
```

W analogiczny sposób możemy zapisać dane do bazy danych, także wykorzystując sesję Hibernate.

```
	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();

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

Warto jednak zwrócić uwagę na użycie metody openSession(), które zawsze tworzy nową sesję, bazującą na istniejącym połączeniu do bazy danych. Jeżeli spróbowalibyśmy zrobić to samo z wykorzystaniem EntityManager'a (metody persist() a następnie flush()) to niestety nie powiedzie się to; zwrócony zostanie wyjątek *TransactionRequiredException: no transaction is in progress.*

Analogiczny problem pojawi się gdy zamiast tworzyć za każdym razem nowe sesje Hibernate, postanowimy używać już istniejącej, pobierając ją za pomocą metody getCurrentSession(): *HibernateException: No Hibernate Session bound to thread, and configuration does not allow creation of non-transactional one here.*

Transakcje

No więc właśnie – transakcje. Omawiając silniki ORM i pokazując, w jaki sposób można z nich korzystać za pośrednictwem Springa, zupełnie pominąłem metodę dodającą nowe zadanie do bazy danych. Zostało to zrobione specjalnie, aby nie robić zamieszania i nie wprowadzać nowych pojęć zbyt szybko. Przez cały moduł omawiając wszystkie przykłady ani razu nie pojawiła się kwestia zarządzania transakcjami. Dopiero na samym końcu doszliśmy do sytuacji gdzie transakcje stały się koniecznością.

Transakcje w kontekście aplikacji nie różnią się niczym od pojęcia które jest dobrze znane w domenie baz danych. Podobnie jak w przypadku baz danych, transakcje można scharakteryzować jako ACID:

- Atomowe (atomic) - transakcja jest elementem niepodzielnym - albo udaje się w całości albo wcale,
- Spójne (consistent) – transakcja nie narusza integralności danych, przed i po wykonaniu transakcji system pozostaje spójny,
- Izolowane (isolated) - jeżeli dwie transakcje wykonywane są równolegle to nie widzą zmian przez siebie wprowadzanych,
- Trwałe (durable) - w przypadku nagłej awarii, system jest w stanie udostępnić spójne i nienaruszone dane, niezależnie od momentu, w którym transakcja została przerwana.

Różnica polega na tym, że operacje objęte transakcją nie muszą być operacjami na bazie danych, ale może to być także zapis do pliku, wywołanie zewnętrznego zasobu poprzez web service itd.
Jeżeli spojrzeć na nieco zmodyfikowany pierwszy przykład, można zobaczyć, że pomimo wystąpienia błędu podczas przetwarzania (wyrzucony został wyjątek) – rekord został dodany do bazy danych. Jest to zachowanie jak najbardziej prawidłowe, ponieważ nigdzie nie zostało zaznaczone, że zapisanie rekordu do bazy ma być uzależnione od jakiejkolwiek innej operacji.

Transakcje w kontekście aplikacji oferowane przez framework wykraczają poza typowe transakcje bazodanowe, niejako je rozszerzając. Używając managera transakcji udostępnianego przez Spring zyskujemy jednolite i wygodne API do zarządzania transakcjami po stronie aplikacji, ale także łączymy to z transakcją bazodanową (lub jakąkolwiek inną, która może istnieć).

```
@Component
public  class TasksService {

	private SimpleJdbcInsert jdbcInsert;

	@Autowired
	public TasksService(DataSource dataSource) {
		jdbcInsert = new SimpleJdbcInsert(dataSource)
			.withTableName("TASKS");
	}

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

		Map<String, Object> params = new HashMap<String, Object>();
		params.put("title", title);
		params.put("description", description);
		params.put("startdate", new java.sql.Date(startDate.getTime()));
		params.put("duration", duration);

		jdbcInsert.execute(params);

		throw new RuntimeException("ERROR!");

	}

	//..
}
```

Transakcje aplikacyjne służą właśnie do łączenia kilku operacji i uzależniają końcowy wynik przetwarzania od powodzenia wszystkich operacji objętych transakcją. Powyższa operacja przebiegła jak najbardziej poprawnie z punktu widzenia bazy danych – operacja zapisu zakończyła się sukcesem. Błąd wystąpił po stronie aplikacji, która w żaden sposób nie dbała o spójność wykonywanych operacji.

Aby zabezpieczyć się przed takimi problemami, Spring umożliwia zarządzanie transakcjami po stronie aplikacji (w analogiczny sposób jak odbywa się to w kontenerach EJB). Spring oferuje dość szeroki zestaw narzędzi umożliwiających duża granulację transakcji i bardzo dużą dowolność w wykorzystaniu transakcji. Na początku skupimy się na najprostszym przypadku, czyli adnotacji `@Transactional`, która umożliwia objęcie procesu (lub jego fragmentu) transakcją zarządzaną przez framework. Samo oznaczenie metody wspomnianą adnotacją, obejmuje ją transakcją i nie pozwala na częściowe zakończenie operacji w przypadku wystąpienia błędu. Poprawna konfiguracja sprowadza się do poinformowania kontenera o wykorzystaniu transakcji opartych o adnotację oraz połączeniu managera transakcji z odpowiednim źródłem danych (aby operacje bazodanowe nie zostały zakończone w momencie wystąpienia problemu po stronie aplikacji). Po uzupełnieniu pliku konfiguracyjnego można zacząć stosować adnotację `@Transactional` mając pewność, że operacja przerwana błędem zostanie poprawnie wycofana.

```
	<tx:annotation-driven transaction-manager="txManager" />

	<bean id="txManager"

	class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
		<property name="dataSource" ref="dataSource" />
	</bean>
```

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

      Map<String, Object> params = new HashMap<String, Object>();
      params.put("title", title);
      params.put("description", description);
      params.put("startdate", new java.sql.Date(startDate.getTime()));
      params.put("duration", duration);

      jdbcInsert.execute(params);

      throw new RuntimeException("ERROR!");

}
```

Dokładne zrozumienia sposobu działania transakcji wymaga wprowadzenia nowego pojęcia – aspektów i interceptorów. Aspektami zajmiemy się w kolejnych modułach, więc na razie poprzestaniemy na minimalnej ilości nowych informacji, bezpośrednio związanych z obsługą bazy danych. Dokładniejsze omówienie różnorakich sposobów na konfigurowanie transakcji nastąpi w części odnoszącej się do plików konfiguracyjnych XML, w jednym z dalszych modułów.

Transakcje i JPA

W odróżnieniu od poprzednich przykładów opartych o czyste niskopoziomowe JDBC, gdzie skorzystanie z transakcji zarządzanych przez framework było opcjonalne – aplikacja potrafiła działać także bez tego, JPA wymaga istnienia transakcji (przy czym operacje odczytu z oczywistych względów jej nie potrzebują). Zarządzanie transakcjami opisuje kolejny standard – JTA. Nie będziemy tutaj jednak go omawiać, skupimy się w jak skonfigurować framework, aby zarządzał transakcjami. Także w tym przypadku oprzemy się o adnotację `@Transactional`, którą oznaczymy metody korzystające z bazy danych. Gdy nie użyjemy adnotacji i spróbujemy zapisać dane to zwrócony zostanie wyjątek – co widzieliśmy na przykładzie z `EntityManagerem`. Jeżeli decydujemy się na użycie Hibernate'a to bez jesteśmy ograniczeni do mało efektywnego tworzenia sesji dla każdej operacji bazodanowej. Utworzenie transakcji, jeżeli korzystamy z Hibernate'a jest niezwykle proste i przebiega w analogiczny sposób co w przypadku połączenia JDBC – wystarczające jest uzupełnienie pliku konfiguracyjnego o następujący wpis.

```
<tx:annotation-driven transaction-manager="txManager" />

<bean id="txManager"

class="org.springframework.orm.hibernate3.HibernateTransactionManager">
      <property name="sessionFactory" ref="mySessionFactory" />
</bean>
```

Adnotacja @Transactional

Wykorzystana w poprzednich przykładach adnotacja `@Transactional`, umożliwiała deklaratywne objęcie fragmentu systemu transakcją. Podejście zaproponowane przez Spring Framework znacząco różni się od modelu np. JEE, gdzie wszystkie publiczne metody są objęte transakcją z definicji i programista wprowadza zmiany, jeżeli domyślna konfiguracja mu nie odpowiada. W Springu, brak deklaracji transakcji oznacza jej brak. Samo objęcie metody transakcją odbywa się poprzez dodanie

do kontekstu komponentu `TransactionManager` oraz wspomnianą powyżej adnotację. Sama adnotacja przyjmuje szereg parametrów które umożliwiają dalsze konfigurowanie transakcji:

- Value: opcjonalna nazwa kwalifikatora managera transakcji który ma zostać użyty (jeżeli występuje więcej niż jeden).
- Propagation: propagacja transakcji, definiuje sposób zachowania w przypadku gdy transakcja została już rozpoczęta. Domyślna wartość to `REQUIRED`
- Isolation: poziom izolacji transakcji, określa sposób komunikacji pomiędzy transakcjami, czy możliwe jest odczytywanie tymczasowych danych (ang. *dirty read*), z jeszcze niezamkniętych transakcji. Domyślna wartość to `DEFAULT` (odpowiadająca domyślnej strategi na poziomie źródła danych JDBC).
- ReadOnly: określa czy transakcja jest tylko odczytująca czy obejmuje zarówno operacje odczytu jak i zapisu
- Timeout: określony w sekundach maksymalny czas trwania transakcji.
- Rollback / NoRollback: seria atrybutów, które definiują które wyjątki (wymienione poprzez typ lub nazwę) powodują lub nie cofnięcie transakcji.

Propagacja transakcji

Spring wspiera wszystkie typy transakcji znane z EJB, a także nieznacznie je rozszerza:

- REQUIRED; wartość domyślna dla kontenera. Jeżeli klient wywoła metodę biznesową oznaczoną atrybutem REQUIRED to operacja zawsze będzie objęta transakcją. Będzie to nowo utworzona transakcja (w momencie gdy klient nie jest nią objęty) lub wykorzystana będzie transakcja już istniejąca
- NOT_SUPPORTED; klient nie obsługuje transakcji. Jeżeli klient jest takową już objęty, to na czas wywołania metody transakcja jest zawieszona i po przetwarzaniu wznowiona.
- SUPPORTS; komponent zachowuje się poprawnie zarówno z jak i bez transakcji. Atrybut SUPPORTS powoduje dwoiste zachowanie się komponentu. W przypadku aktywnej transakcji, zachowanie jest takie samo jak w przypadku atrybutu REQUIRED. Jeżeli natomiast transakcja nie jest aktywna - NOT_SUPPORTED.
- REQUIRES_NEW; dla każdego wywołania metody tworzona jest nowa transakcja. Jeżeli transakcja istnieje, jest zawieszona a następnie wznowiona po wywołaniu metody.
- MANDATORY; transakcja musi być aktywna. Jeżeli takowa istnieje, zachowanie się kontenera jest analogiczne do sytuacji z atrybutem REQUIRED. Jeżeli transakcji nie ma, zgłaszany jest błąd i wyrzucany jest wyjątek
- NEVER; sytuacja odwrotna do powyższej. Metoda nie może być wywołana w obrębie transakcji. Jeżeli transakcja nie istnieje - zachowanie jest takie samo jak w przypadku atrybutu NOT_SUPPORTED. W innym przypadku zgłaszany jest błąd.

Dodatkowym atrybutem jest NESTED, która powoduje utworzenie zagnieżdżonej transakcji, z wieloma punktami zapisu (ang. *save points*). Ta transakcja nie ma swojego odpowiednika w EJB, ponieważ jest wspierana tylko i wyłącznie przez źródła danych oparte o JDBC 3.0. Użycie takie transakcji pozwala na częściowe nawrócenie transakcji (ang. *rollback*) do najbliższego wewnętrznego punktu zapisu, podczas gdy z zewnętrznej perspektywy transakcja wciąż jest aktywna i działająca.

Inne sposoby dostępu do bazy (iBatis)

Wprowadzanie kolejnej warstwy w aplikacji nie odbywa się niestety bez dodatkowych kosztów. Kolejna warstwa, mimo że znacznie uprasza samą aplikację, wymaga więcej zasobów, zwiększa czas

uruchamiania się aplikacji. Podejmując decyzję o np. rezygnacji z `JdbcTemplate` i wykorzystaniu JPA, należy brać pod uwagę nie tylko samą wygodę programisty (ta bowiem nie podlega dyskusji), ale także implikacje niefunkcjonalne, takie jak szybkość działania (np. wydajność generowanych zapytań SQL), czas startu aplikacji. Okazuje się bowiem, że pomiędzy JDBC i JPA istnieje jeszcze kilka alternatywnych podejść, które zostaną omówione w kolejnych przykładach. Mam tutaj na myśli alternatywne sposoby dostępu do bazy danych; alternatywne w tym sensie niebycia standardem języka Java. Opisane powyżej przypadki reprezentują dwie skrajności: przechowywania zapytań SQL bezpośrednio w kodzie aplikacji, jak i całkowite oderwanie aplikacji od konkretnego typu bazy danych poprzez wejście na wyższy poziom abstrakcji i wykorzystanie mapowania obiekt – relacja. Stosunkowo często programista potrzebuje rozwiązania pośrodku: mapowania konkretnych wyrażeń SQL (napisanych przez programistę) na obiekty, bez konieczności korzystania – przy okazji nie przechowując tych zapytań bezpośrednio w kodzie aplikacji.

Takim złotym narzędziem może być iBatis, które mapuje wynik zapytania SQL na konkretną klasę, przy czym całość konfiguracji leży po stronie programisty i nie ma tutaj żadnej magii, którą stara się oferować JPA. Nim jednak skorzystamy z zewnętrznej biblioteki, warto spojrzeć jakie możliwość oferuje sam framework – często one okazują się być wystarczające.

Mapowanie zapytań na obiekty

Zacznijmy od klasy `MappingSqlQuery`, dostępnej w samym frameworku. Spring udostępnia bazową klasę ułatwiająca mapowanie relacji na obiekty oraz kojarzenie encji (obiektu) z typowymi dla niego zapytaniami SQL (coś na wzór Named Queries dostępnego później w JPA). I tak, rozszerzając klasę `MappingSqlQuery`, umożliwiamy całkowite rozłączenie kodu SQL od logiki aplikacji.

```
public  class TaskMappingQuery extends MappingSqlQuery<Task> {

	public TaskMappingQuery(DataSource ds) {
		super(ds, "select * from tasks");
		compile();
	}

	@Override
	protected Task mapRow(ResultSet rs, int rowNumber) throws SQLException {
		Task t = new Task.Builder()
			.withId(rs.getLong("task_id"))
			.withTitle(rs.getString("title"))
			.withDescription(rs.getString("description"))
			.withDate(rs.getDate("startdate"))
			.withDuration(rs.getInt("duration"))
		.build();

		return t;
	}

}
```

Tak utworzoną klasę możemy z wykorzystać do pobierania danych z bazy.

```
@Component
public  class TasksService {

	TaskMappingQuery query;

	@Autowired
	public void setDataSource(DataSource dataSource) {
	    this.query = new TaskMappingQuery(dataSource);
	}

	//..

	public List<Task> getTasks() throws Exception {
		return query.execute();
	}
}
```

Jak widać na powyższym przykładzie, kod serwisu znacznie się uprościł – zbliżając się poziomu czytelności oferowanego przez silniki ORM (np. Hibernate).

Wykorzystanie iBatis

Potężniejszym i bardziej elastycznym narzędziem jest iBatis. Jednakże jego konfiguracja wymaga kilku dodatkowych zabiegów. Przede wszystkim musimy dołączyć do projektu nowe biblioteki związane z samym SQL mapperem, który następnie musi zostać skonfigurowany.

Obiekt będący reprezentacją relacji jest już utworzony, przy okazji przykładów z ORM, jak i wcześniejszych `SqlQuery` – możemy więc go ponownie wykorzystać. Pozostaje konfiguracja mapowania, którą umieszczamy w pliku [nazwa_encji].xml – czyli w naszym przypadku Task.xml. Kolejnym kroku definiujemy samego mappera w pliku sqlmap-config.xml i łączymy go ze Springiem, dodając odpowiedni wpisy w pliku konfiguracyjnym XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap
	PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
	"http://ibatis.apache.org/dtd/sql-map-2.dtd">

<sqlMap namespace="Task">

	<resultMap id="result" class="pl.devcastzone.spring.model.Task">
		<result property="taskId" column="TASK_ID" columnIndex="1" />
		<result property="title" column="TITLE" columnIndex="2" />
		<result property="description" column="DESCRIPTION"
			columnIndex="3" />
		<result property="date" column="STARTDATE" columnIndex="4" />
		<result property="duration" column="DURATION" columnIndex="5" />
	</resultMap>

	<select id="getTasks" resultMap="result">
		select * from TASKS
	</select>

	<insert id="insertTask">
		insert into TASKS (TITLE, DESCRIPTION, STARTDATE, DURATION)
		values (#title#, #description#, #date#, #duration#)
	</insert>

</sqlMap>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMapConfig
      PUBLIC "-//iBATIS.com//DTD SQL MAP Config 2.0//EN"
      "http://ibatis.apache.org/dtd/sql-map-config-2.dtd">

<sqlMapConfig>
      <sqlMap resource="META-INF/sqlmap/Task.xml" />
</sqlMapConfig>
```

```
<bean id="sqlMapClient"
      class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
      <property name="configLocation"
            value ="classpath:META-INF/sqlmap/sqlmap-config.xml" />
      <property name="dataSource" ref="dataSource" />
</bean>
```

Z tak przygotowaną aplikacją można zacząć modyfikację serwisu.

```
@Component
public  class TasksService {

      @Autowired
      SqlMapClient sqlMapClient;

      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();

            sqlMapClient.insert("insertTask", t);
      }

      public List<Task> getTasks() throws Exception {
            return (List<Task>) sqlMapClient.queryForList("getTasks", null);
      }

}
```

Zaletą iBatis, jest to, że wszystkie zapytania do bazy danych przechowywane są w zewnętrznym pliku konfiguracyjnym. Dzięki temu możliwe jest dostosowanie aplikacji do drobnych zmian w strukturze bazy danych, bez konieczności ponownej kompilacji całej aplikacji. W praktyce jednak, okazuje się, że bardzo często taka kompilacja jest konieczna, a zysk z rozdzielenia konfiguracji na kilka plików okazuje się być iluzoryczny. Najlepszym tego przykładem jest EJB i platforma JEE, która z biegiem czasu całkowicie odchodzi o zewnętrznych plików konfiguracyjnych, które okazały się być przerostem formy nad treścią. Oczywiście nie twierdzę, że narzędzia typu iBatis są zupełnie niepotrzebne; uważam że wprowadzając nową bibliotekę do systemu konieczne jest dokładne przeanalizowanie przypadków użycia oraz upewnienie się że funkcjonalność której potrzeba nie istnieje już wewnątrz frameworku.

Słowem podsumowania

W niniejszym module przedstawiłem dwa aspekty frameworku: dostęp do bazy danych oraz konfigurację transakcji, a zarazem pokazałem na przykładach jak wielka jest elastyczność frameworka i jak każdą rzecz można zrobić na więcej niż jeden sposób – w zależności od tego jakie są założenia

projektu. Nie trzeba wizji rozwiązania dostosowywać do aktualnych możliwości (jak ma to miejsce np. w EJB), a framework jest narzędziem wspomagającym, a nie dominującym.