Testy integracyjne warstwy dostępu do danych w środowisku Java z wykorzystaniem Spring Framework

Opracowanie:

Mateusz Koza

Modyfikacje:

Radosław Adamus

1. Wstęp

Testowanie integracyjne to sposób sprawdzenia poprawności zachowania kodu, przy uwzględnieniu rzeczywistej komunikacji tego kodu z jego zewnętrznymi zależnościami. W odróżnieniu od testowania jednostkowego nie stosujemy zaślepek aby uniezależnić testowany kod od zewnętrznego środowiska, ale wpływamy na nie z pełną świadomością, tworząc tak zwane efekty uboczne. Przykładem może być test repozytorium, w którym fizycznie łączymy się z bazą danych, dokonujemy zapisu, weryfikujemy jej stan, wpływając tym samym na otoczenie. Do innych przykładów zewnętrznych zależności należą np.: system plików, czas, kolejki komunikatów, serwery (mailowe, FTP), usługi Internetowe itd.

Do wyzwań jakie związane są z testami integracyjnymi należą:

- „kruchość" – test polega na działaniu zewnętrznych zasobów – nie ma gwarancji, że wszystko będzie poprawnie skonfigurowane – dlatego błąd testu nie zawsze oznacza problem z implementacją.

- „czas wykonania" – test integracyjny wymaga manipulacji zewnętrznymi zasobami co może być czasochłonne.

Z tego te powodu testy integracyjne powinny być oddzielone od testów jednostkowych. Testy jednostkowe z założenia powinny zawsze wykonywać się poprawnie i szybko. W przypadku testów integracyjnych wymagania te mogą nie być spełnione.

Wykonywanie testu w kontekście zależności wymaga dużej uwagi w zakresie przygotowania środowiska testowego. W takich przypadkach warto pomyśleć o zastosowaniu kontekstów testowych, które pozwolą kontrolować powstałe efekty uboczne, zapewniać każdemu testowi odpowiednie, oczekiwane przez niego „czyste" środowisko, w którym może zachowywać się w przewidziany sposób, niezależnie od innych testów i przede wszystkim, niezależnie od środowiska produkcyjnego.

Dlatego w przypadku każdej większej aplikacji, warto wprowadzić konfigurowalne tryby, które można dowolnie ustawić przed uruchomieniem. Odpowiednio skonfigurowania aplikacja będzie się np. zachowywać w inny sposób w trybie produkcyjnym, a w inny w trybie testowym.

2. Przykład środowiska testowego w technologii Spring

Aby pokazać jak działa tryb testowy w praktyce, opracujemy prosty projekt oparty o szkielet aplikacyjny Spring, który skonfigurujemy tak, aby w zależności od wybranego trybu łączył się z określoną bazą danych. Do testów integracyjnych działających na bazie danych wykorzystamy system zarzadzania bazą danych H2[1]. Dodatkowo napiszemy prosty test integracyjny potwierdzający, że wszystko działa jak należy. Do obiektowej komunikacji z wbudowaną bazą H2 wykorzystamy technologię JPA2, a żeby przyspieszyć proces implementacji dostępu do danych, skorzystamy z mechanizmu repozytoriów Spring Data.

2.1 Projekt

Wstępna wersja projektu dostępna jest tutaj https://github.com/mto-lab/lab5_2/ .

W pierwszej kolejności należy dodać wymagane zależności:

Spring: `<spring-version>4.1.6.RELEASE</spring-version>`

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring-version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>${spring-version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>${spring-version}</version>
</dependency>
```

[1] Baza pamięciowa, której konfiguracja w środowisku Spring jest prosta a która ma duże możliwości w zakresie uruchamiania w trybie kompatybilności z wieloma produkcyjnymi systemami np.: jdbc:h2:mem:test;MODE=Oracle

```
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>${spring-version}</version>
            <scope>test</scope>
        </dependency>
```

Persistency:

```
    <spring-data-version>1.7.2.RELEASE</spring-data-version>

    <hibernate-version>4.3.8.Final</hibernate-version>

    <h2-version>1.4.186</h2-version>
```

```
        <dependency>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-jpa</artifactId>
            <version>${spring-data-version}</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-entitymanager</artifactId>
            <version>${hibernate-version}</version>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <version>${h2-version}</version>
    </dependency>
```

Zależności dotyczą wymaganych bibliotek kontekstu Spring oraz modułów tx i orm (komunikacja z bazą danych). Moduł spring-test pozwala na opracowywanie testów integracyjnych w środowisku JUnit.

Pozostałe zależności związane są z obsługą danych.

2.2 Konfiguracja aplikacji

Do poprawnego działania w środowisku kontenera Spring wymagane jest skonfigurowanie tzw. kontekstu aplikacji (w ramach którego rozstrzygane będą zależności). Jeżeli potrzebne są specyficzne mechanizmy (np. trwałość danych) należy dodatkowo zdefiniować odpowiedni kontekst (w przypadku trwałości będzie to persistence context).

2.2.1 Konfiguracja kontekstu aplikacji

Główny kontekst aplikacji stanowi klasa `edu.iis.mto.integrationtest.config.ApplicationConfig`.

Wymagane adnotacje dla klasy:

```
@Configuration //wskazuje, że klasa zawiera konfigurację bean'ów Spring
@ComponentScan("edu.iis.mto.integrationtest.repository") //ustala kontekst poszukiwania komponentów
@Import(value = { PersistenceConfig.class })  //klasa związana z konfiguracją bean'ów, która importujemy
```

Klasa powinna posiadać statyczną metodę adnotowaną `@Bean` i zwracającą typ `PropertyPlaceholderConfigurer`. Konfiguracja obiektu w kodzie metody polega na określeniu ścieżki do

pliku zasobów i powinna wyglądać następująco:

```
PropertyPlaceholderConfigurer resolver = new PropertyPlaceholderConfigurer();
resolver.setLocation(new ClassPathResource(ModeUtils.getMode().getModeName() + "-
persistence.properties"));
```

Fragment 1 konfiguracja obiektu **PropertyPlaceholderConfigurer**

Konfigurujemy tutaj bean (PropertyPlaceholderConfigurer), który służy do konfigurowania aplikacji. Można dzięki niemu wczytywać do aplikacji informacje z plików konfiguracyjnych typu klucz-wartość (w Javie standardowo rozszerzenie .properties), co pozwala zmieniać jej parametry bez potrzeby kompilowania całego projektu. Ustawianie lokacji pliku przechowującego konfigurację bazy danych korzysta z aktualnego trybu aplikacji. Dzięki temu w zależności od trybu działania aplikacji możliwe jest łączenie się z różnymi bazami danych.

Kod klasy *ModeUtils* znajduje się z projekcie laboratoryjnym.

2.2.2 Konfiguracja kontekstu trwałości

Konfiguracja kontekstu związanego z trwałością reprezentowana jest przez klasę

edu.iis.mto.integrationtest.config.PersistenceConfig.

Wymagane adnotacje dla klasy:

```
@Configuration //wskazuje, że klasa zawiera konfigurację bean'ów Spring
@EnableTransactionManagement //umożlwia konfigurowanie mechanizmów transakcji z wykorzystaniem adnotacji
@EnableJpaRepositories(basePackages = {"edu.iis.mto.integrationtest.repository"}) //określa gdzie szukać
klas definiujących repozytoria Spring Data.
```

Aby umożliwić konfigurację aplikacji z poziomu pliku właściwości pola klasy powinny być mapowane na klucze z pliku konfiguracyjnego o nazwie [tryb]-persistence.properties (gdzie przedrostek [tryb] określa tryb uruchomienia aplikacji) z wykorzystaniem adnotacji `@Value`. Np.: mapowanie klucza database.url przedstawia się następująco:

```
@Value("${database.url}")
private String databaseUrl;
```

Klasa musi definiować następujące bean'y Spring (przez metody adnotowane @Bean):

1. `@Bean(name = "dataSource")` – metoda powinna zwracać „źródło danych" - typ javax.sql.DataSource. Źródło danych powinno być utworzone na podstawie informacji z pliku konfiguracyjnego (zmapowanych na pola klasy) oraz może zostać wypełnione domyślnymi danymi (np. dla potrzeb testów) z wykorzystaniem obiektu typu

org.springframework.jdbc.datasource.init.DatabasePopulator. Zakładając, że schemat danych znajduje się w odrębnym pliku oraz, że domyślne dane zależne są od trybu uruchomienia aplikacji kod metod może wyglądać następująco[2]

[2] W przypadku kopiowania kodu należy pamiętać o samodzielnym zdefiniowaniu poprawnych wartości stałych ***SQL_SCHEMA_SCRIPT_PATH, SQL_FOLDER_NAME*** *oraz* ***DATA_SCRIPT_FILENAME_SUFFIX***

```
@Bean(name = "dataSource")
public DataSource getDataSource() {
        DataSource dataSource = createDataSource();
        DatabasePopulatorUtils.execute(createDatabasePopulator(), dataSource);
        return dataSource;
}

private DatabasePopulator createDatabasePopulator() {
        ResourceDatabasePopulator databasePopulator = new ResourceDatabasePopulator();
        databasePopulator.setContinueOnError(true);
        databasePopulator.addScripts(new ClassPathResource(
                        SQL_SCHEMA_SCRIPT_PATH), new ClassPathResource(SQL_FOLDER_NAME
                        + ModeUtils.getMode().getModeName()
                        + DATA_SCRIPT_FILENAME_SUFFIX));
        return databasePopulator;
}

private SimpleDriverDataSource createDataSource() {
        SimpleDriverDataSource simpleDriverDataSource = new SimpleDriverDataSource();
        Class<? extends Driver> driverClass = getDriverClass();
        simpleDriverDataSource.setDriverClass(driverClass);
        simpleDriverDataSource.setUrl(databaseUrl);
        simpleDriverDataSource.setUsername(databaseUser);
        simpleDriverDataSource.setPassword(databasePassword);
        return simpleDriverDataSource;
}

@SuppressWarnings("unchecked")
private Class<? extends Driver> getDriverClass() {
        try {
                Class<?> driverClass = Class.forName(databaseDriverClass);
                if (Driver.class.isAssignableFrom(driverClass)) {
                        return (Class<? extends Driver>) driverClass;
                }else {
                        LOGGER.error("database driver class is not the SQL driver ");
                        return null;
                }
        } catch (ClassNotFoundException e) {
                LOGGER.error("database driver class not found", e);
                return null;
        }
}
```

Fragment 2: Kod konfiguracji źródła danych

2. Bean’y Spring wymagane przez JPA:

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
        LocalContainerEntityManagerFactoryBean entityManagerFactoryBean = new
        LocalContainerEntityManagerFactoryBean();
        entityManagerFactoryBean.setDataSource(getDataSource());
        entityManagerFactoryBean.setPackagesToScan("edu.iis.mto.integrationtest.model");
        entityManagerFactoryBean .setJpaVendorAdapter(new HibernateJpaVendorAdapter());
        return entityManagerFactoryBean;
}

@Bean(name = "transactionManager")
public PlatformTransactionManager annotationDrivenTransactionManager() {
        return new JpaTransactionManager();
}

@Bean
public PersistenceExceptionTranslationPostProcessor exceptionTranslation() {
        return new PersistenceExceptionTranslationPostProcessor();
}
```

Fragment 3 Konfiguracja bean'ów Spring związanych z wykorzystaniem technologii JPA

2.3 Utworzenie modelu i repozytorium danych

Model danych wykorzystujący JPA to klasa reprezentująca strukturę danych wyposażona w adnotacje pozwalające na mapowanie pomiędzy klasą a tabelą w bazie danych.

W przykładzie wykorzystana jest następująca klasa modelu (edu.iis.mto.integrationtest.model.Person).

Uwaga: wymagane adnotacje (np. Entity, Table, itd.) pochodzą z pakietu javax.persistence.

```
@Entity
@Table(name = "person")
public class Person {

    @Id
    @Column(unique = true, nullable = false)
    private Long id;

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;
//Getters and setters, default constructor, toString override
}
```

Fragment 4 Klasa modelu danych w kontekście JPA

Korzystając z technologii spring-data, utworzenie prostego repozytorium, pozwalającego na interakcję z bazą danych jest tak proste jak utworzenie interfejsu (podstawowe operacje CRUD dla modelu są generowane automatycznie) (edu.iis.mto.integrationtest.repository.PersonRepository):

```
public interface PersonRepository extends JpaRepository<Person, Long> {

}
```

Fragment 5 Wykorzystanie Spring-data do definicji repozytorium

W projekcie znajduje się przykładowa klasa uruchomieniowa dla aplikacji (edu.iis.mto.integrationtest.main.ApplicationMain).

3. Testy integracyjne

Skonfigurowana w ten sposób aplikacja daje nam duże możliwości w zakresie implementacji testów integracyjnych komponentów. Dobrą praktyką jest utworzenie bazowej klasy testu, definiującego podstawowe elementy dla potrzeb testów integracyjnych.

```
@ContextConfiguration(classes = { ApplicationConfig.class }) //kontekst Spring
@RunWith(SpringJUnit4ClassRunner.class) // uruchamianie z wykorzystaniem specyficznego wykonawcy
public class IntegrationTest {

	@BeforeClass
	public static void setUpMode() {
		ModeUtils.setMode(Mode.TEST);
	}
}
```

Fragment 6 Klasa bazowa testów integracyjnych

Projekt laboratoryjny zawiera szkic implementacji testu integracyjnego `PersonRepositoryIntegrationTest` wykorzystującej bazową klasę z Fragment 6.

Po skonfigurowaniu projektu (zgodnie z opisem w niniejszym dokumencie), należy dodać adnotację `@Autowired` do pola personRepository testu `PersonRepositoryIntegrationTest`. Spowoduje to, że zależność ta zostanie automatycznie wstrzyknięta (and. dependency injection) zgodnie z konfiguracją kontekstu Spring.

4. Zadania:

1. Uzupełnij projekt zgodnie z opisem zawartym w niniejszym dokumencie.
2. Odkomentuj testy w klasie `PersonRepositoryIntegrationTest` i uruchom je. Następnie przeanalizuj przyczynę ich niepowodzenia i popraw projekt.
3. Opracuj testy pozostałych operacji CRUD dla modelu.
4. Dodaj do interfejsu PersonRepository metodę

```
public List<Person> findByFirstNameLike(String firstName);
```

Opracuj test sprawdzający jej działanie.