Modul Pelatihan

Akses Database dengan Spring

Mengakses database menggunakan Spring-JDBC

Version:
1.9

Last Updated:
5 September 2012

Akses Database dengan Spring

Endy Muhardin Ifnu Bima Martinus Ady Jimmy Rengga
Adi Sulistiono

5 September 2012

Contents

1 Akses database dengan Spring 2.5

Spring 2.5 baru saja keluar. Rilis kali ini membawa penambahan fitur yang cukup signifikan di sisi konfigurasi. Dalam Spring yang baru ini, kita bisa mengkonfigurasi aplikasi melalui annotation. Suatu hal yang sangat bermanfaat untuk mengurangi jumlah baris kode XML kita.

Figure 1: Logo Spring Framework

Sebetulnya tidak ada yang salah dengan XML. Walaupun demikian, ada beberapa hal yang menurut saya kurang tepat kalau dikonfigurasi melalui XML, diantaranya:

- konfigurasi transaction
- deklarasi bean standar

Konfigurasi transaction biasanya tergantung dari kode program yang ingin bertransaction. Bila kita konfigurasi di XML, maka untuk memikirkan satu logika akses database, kita harus melihat di dua tempat yang berbeda; file java dan file XML. Menurut pendapat saya, fitur declarative transaction walaupun kelihatannya mirip konfigurasi, tapi pada dasarnya adalah logika aplikasi. Tempatnya bukan di konfigurasi XML, tapi di kode Java.

Di Spring, kita harus mendaftarkan object aplikasi kita ke dalam object ApplicationContext agar bisa dikelola oleh Spring. Pada rilis sebelumnya, pendaftaran ini dilakukan dalam file XML. Cara ini memiliki incremental cost yang tinggi. Bila kita

punya 100 object yang ingin dikelola, maka kita harus punya 100 deklarasi di konfigurasi XML Spring. Sekarang kita bisa menandai object yang akan dikelola Spring melalui annotation. Jadi walaupun ada 100 object, konfigurasi XML kita tidak bertambah.

Ok, cukup berteori. Saatnya melihat contoh kode.

1.1 Domain Model

Pada artikel kali ini, kita akan membuat kode akses database untuk class `Person`. Class ini tidak istimewa, cuma POJO biasa dengan tiga property: id, name, dan email. Berikut kode program `Person.java`.

```
package tutorial.spring25.model;
public class Person {
  private Long id;
  private String name;
  private String email;
}
```

Jangan lupa membuat getter dan setter.

1.2 Skema Database

Class ini akan kita simpan di database dalam tabel bernama `T_PERSON`. Berikut definisinya untuk database MySQL.

```
create table T_PERSON (
  id BIGINT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(255),
  email VARCHAR(255)
);
```

1.3 Interface Akses Database

Operasi database yang akan kita buat dijelaskan oleh interface `PersonDao`, sebagai berikut.

```
package tutorial.spring25.dao
public interface PersonDao {
  public List<person> getAll();
  public Person getById(Long id);
  public void save(Person p);
}
```

Untuk tahap pertama, kita akan lihat cara mengakses database dengan JDBC helper yang disediakan Spring. Akses database dengan Hibernate akan dijelaskan pada artikel terpisah.

1.4 Implementasi Akses Database

Berikut adalah kerangka implementasi `PersonDao` dengan JDBC helper dari Spring. Kita simpan di file bernama `PersonDaoSpringJdbc.java`

```
package tutorial.spring25.dao.springjdbc;

@Repository(”personDao”)
@Transactional(readOnly=true)
public class PersonDaoSpringJdbc implements PersonDao {
    @Autowired
    public void setDataSource(final DataSource dataSource) {

    }

    @Override
    public List<person> getAll() {
        return null;
    }
```

```
    @Override
    public Person getById(final Long id) {
        return null;
    }

    @Override
    @Transactional(readOnly=false)
    public void save(final Person person) {

    }
}
```

Ada beberapa hal yang baru pada kode di atas. Kita melihat ada annotation `@Repository`, `@Transactional`, dan `@Autowired`.

Annotation `@Repository` memberi tahu pada Spring bahwa class ini adalah salah satu `@Component` dalam aplikasi kita. Semua @Component akan dipindai pada waktu inisialisasi dan kemudian diregistrasi ke dalam object `ApplicationContext` milik Spring. Selain `@Repository`, `@Component` juga memiliki turunan `@Service` dan `@Controller`. `@Service` biasanya digunakan untuk menandai class-class facade atau business delegate. Sedangkan `@Controller` digunakan untuk aplikasi web. `@Service` dan `@Controller` akan kita bahas di artikel terpisah.

Annotation `@Transactional` menandakan bahwa semua method dalam class ini akan dijalankan dalam transaksi database. Kita memberikan nilai `readOnly=true` pada deklarasi class, menandakan bahwa secara default transaksi hanya digunakan untuk mengambil data dari database. Perhatikan method save. Pada method ini, kita akan memasukkan atau mengubah data dalam database. Untuk satu method ini, kita membutuhkan transaksi yang tidak readOnly. Karena itu, kita override konfigurasi default dengan cara memberikan annotation `@Transactional(readOnly=false)`.

1.5 Konfigurasi Spring Framework

Sekarang mari kita lihat konfigurasi Application Context. File ini disave dengan nama `applicationContext.xml`.

```
<?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:p="http://www.springframework.org/schema/p"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/aop http://www.springframework.org/schema/a
    http://www.springframework.org/schema/beans http://www.springframework.org/schema/bea
    http://www.springframework.org/schema/context http://www.springframework.org/schema/c
    http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/s
    http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spr

    <context:property-placeholder location="classpath:jdbc.properties"/>
    <context:annotation-config/>
    <context:component-scan base-package="tutorial.spring25"/> <!-- tidak perlu deklarasi
    <tx:annotation-driven /> <!-- tidak perlu deklarasi transaction setting per method --

    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource"
        destroy-method="close"
        p:driverClassName="${jdbc.driver}"
        p:url="${jdbc.url}"
        p:username="${jdbc.username}"
        p:password="${jdbc.password}" />

    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
        p:dataSource-ref="dataSource" />

</beans>
```

Seperti kita lihat di atas, kita tidak lagi membutuhkan deklarasi untuk object personDao seperti pada Spring sebelumnya. Kita juga tidak perlu membuat konfigurasi transaksi untuk masing-masing method dalam PersonDao. Sebagai gambaran, kita menghilangkan beberapa baris yang seperti ini.

```
<bean id="personDaoImpl" class="tutorial.spring25.dao.springjdbc.PersonDaoSpringJdbc">
    <property name="dataSource" ref="dataSource"></property>
</bean>

<bean id="personDao" class="org.springframework.transaction.interceptor.TransactionProxyF
    <property name="target" ref="personDao"></property>
    <property name="transactionManager" ref="transactionManager"></property>
    <property name="transactionAttributes">
        <props>
            <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="save*">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>
```

Semakin banyak class DAO kita, deklarasinya juga akan semakin banyak. Pada aplikasi skala menengah, jumlah DAO bisa mencapai ratusan. Bisa dibayangkan dampaknya terhadap file xml tersebut. Dengan mencantumkan satu baris seperti ini,

```
<context:component-scan base-package="tutorial.spring25"/>
```

Spring dapat secara otomatis memeriksa seluruh package tutorial.spring dan mendaftarkan semua class yang memiliki annotation `@Component`, `@Repository`, `@Service`, dan `@Controller`.

Kita sudah lihat bagaimana keseluruhan kode program ditulis. Kecuali implementasi sebenarnya tentu saja. Sebelum melihat secara detail bagaimana kode program untuk `INSERT` dan `SELECT`, terlebih dulu kita lihat bagaimana class `PersonDaoSpringJdbc` ini digunakan.

1.6 Automated Testing

Daripada menggunakan cara yang kurang berwawasan (menggunakan method main), saya akan mengambil pendekatan yang lebih berpendidikan, yaitu menggunakan Unit Test. Lihat artikel saya tentang Unit Test dan Integration Test untuk memahami kode berikut.

Ini adalah kerangka class test untuk `PersonDaoSpringJdbc`. Class ini dibuat dengan menggunakan JUnit 4. Isinya masih belum lengkap. Kita akan lengkapi sambil jalan.

```
package test.spring25.dao.springjdbc
public class PersonDaoSpringJdbcTest {

  private static final ApplicationContext applicationContext;
  private static final DataSource dataSource;
  private static final PersonDao personDao;

  @BeforeClass public static void init(){}
  @Before public void resetDatabase(){}

  @Test public void testGetById(){}
  @Test public void testGetAll(){}
  @Test public void testSave(){}
}
```

Seperti kita lihat, kita sudah menggunakan annotation untuk menandai method test dan inisialisasi. Penjelasan tentang JUnit 4 akan dibahas pada artikel terpisah.

Sekarang kita lihat isi masing-masing method. Method `init` isinya seperti ini.

```
@BeforeClass public static void init(){
  ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
  dataSource = (DataSource) ctx.getBean("dataSource");
  personDao = (PersonDao) ctx.getBean("personDao");
}
```

Method `resetDatabase` dijalankan sebelum masing-masing test method. Fungsinya untuk menghapus isi tabel `T_PERSON` dan mengisi sampel data sesuai yang kita inginkan. Ini dilakukan menggunakan DBUnit. Isinya sebagai berikut

```
@Before public void resetDatabase() throws Exception {
  final Connection conn = ds.getConnection();
  DatabaseOperation.CLEAN_INSERT.execute(new DatabaseConnection(conn), new FlatXmlDataSet
  conn.close();
}
```

1.7 Sample Data

Method di atas akan menggunakan sampel data yang ada di file `person.xml`. Isinya seperti ini,

```
<dataset>
  <T_PERSON
    id="100"
    name="Endy Muhardin"
    email="endy.muhardin@gmail.com"
  />
</dataset>
```

Cukup satu record saja.

1.8 Implementasi Query Database

Pertama kali, kita akan implementasi method `getById`. Isi testnya tidak rumit. Cukup jalankan method `getById` dan periksa hasilnya.

```
@Test public void testGetById() throws Exception {
  Person endy = personDao.getById(100L);
  assertEquals("Endy Muhardin", endy.getName());
  assertEquals("endy.muhardin@gmail.com", endy.getEmail());
}
```

Implementasi `getById` dalam `PersonDaoSpringJdbc` seperti ini.

```
public Person getById(Long id) {
  return simpleJdbcTemplate.queryForObject("select * from T_PERSON where id=?", new Perso
}
```

Cukup satu baris saja.

1.9 Mapping dari `ResultSet` menjadi `Person`

Method ini membutuhkan class `PersonMapper` untuk mengkonversi object `ResultSet` menjadi object `Person`. Class ini dibuat menjadi static final inner class dalam `PersonDaoSpringJdbc`.

```
public class PersonDaoSpringJdbc implements PersonDao {
  private static final class PersonMapper implements ParameterizedRowMapper<Person>{
    @Override
    public Person mapRow(final ResultSet rs, final int rowNum) throws SQLException {
      final Person result = new Person();
      result.setId(rs.getLong("id"));
      result.setName(rs.getString("name"));
      result.setEmail(rs.getString("email"));
      return result;
    }
  }
}
```

1.10 Implementasi Query lainnya

Selanjutnya, kita akan implementasikan method `getAll`. Berikut test methodnya.

```
@Test public void testGetAll() throws Exception {
  List<Person> result = personDao.getAll();
  assertEquals(1, result.size());
  Person endy = result.get(0);
  assertEquals("Endy Muhardin", endy.getName());
  assertEquals("endy.muhardin@gmail.com", endy.getEmail());
}
```

Dan ini implementasi dari method `getAll`.

```
public List<Person> getAll() {
  return simpleJdbcTemplate.query("select * from T_PERSON", new PersonMapper(), new HashM
}
```

1.11 Implementasi Insert Data

Terakhir, mari kita implementasi method save. Method testnya sedikit lebih panjang, karena untuk yakin akan hasilnya, kita harus melakukan query ke database dengan JDBC murni.

```
@Test public void testSave() throws Exception {
  Person dhiku = new Person();
  dhiku.setName("Hadikusuma Wahab");
  dhiku.setEmail("dhiku@gmail.com");
  assertNull(dhiku.getId());
  personDao.save(dhiku);
  assertNotNull(dhiku.getId());

  final Connection conn = ds.getConnection();
  final PreparedStatement ps = conn.prepareStatement("select * from T_PERSON where id=?")
  ps.setLong(1, dhiku.getId());
  final ResultSet rs = ps.executeQuery();
  assertTrue(rs.next());

  assertEquals(dhiku.getName(), rs.getString("name"));
  assertEquals(dhiku.getEmail(), rs.getString("email"));

  ps.close();
  rs.close();
  conn.close();
}
```

Untungnya implementasi method save juga hanya satu baris.

```
@Transactional(readOnly=false)
public void save(final Person person) {
  person.setId(simpleJdbcInsert.executeAndReturnKey(new BeanPropertySqlParameterSource(pe
}
```

Ada beberapa hal yang perlu dijelaskan dari kode di atas. Pertama, kita perlu mengaktifkan transaksi database untuk mengubah isi database. Kita lakukan dengan `@Transactional(readOnly=false)`.

Kedua, kita bisa menggunakan fitur terbaru Spring JDBC, yaitu `SimpleJdbcInsert`. Fitur ini mampu melihat ke dalam database dan mengambil daftar nama fieldnya. Object ini diinisialisasi pada saat kita menginjeksi `DataSource`. Berikut kodenya.

```
@Autowired
public void setDataSource(final DataSource dataSource) {
  this.simpleJdbcTemplate = new SimpleJdbcTemplate(dataSource);
  this.simpleJdbcInsert = new SimpleJdbcInsert(dataSource).withTableName("T_PERSON").usin
}
```

Pada saat menginisialisasi object `SimpleJdbcInsert` kita memberi tahu Spring tentang nama tabel dan field yang isinya autogenerated, misalnya `id`.

Selama nama field dalam `T_PERSON` sama dengan nama properti di class `Person`, kita bisa menghilangkan kode untuk mapping properti ke `PreparedStatement`. Kode yang biasanya tiga baris seperti ini

```
PreparedStatement ps = conn.prepareStatement("insert into T_PERSON (name, email) values(?
ps.setString(1, person.getName());
ps.setString(2, person.getEmail());
```

Dapat direduksi menjadi satu baris seperti ini:

```
SqlParameterSource parameterSource = new BeanPropertySqlParameterSource(person);
```

Object `parameterSource` ini bisa langsung diumpankan ke `simpleJdbcInsert` seperti ini untuk melakukan insert sekaligus mengambil nilai id yang digenerate database.

```
Long newId = simpleJdbcInsert.executeAndReturnKey(parameterSource).longValue()
```

Fitur `SimpleJdbcInsert` ini sangat bermanfaat kalau entity class kita terdiri dari puluhan field. Adanya method `executeAndReturnKey` untuk mengambil autogenerated primary key dari database juga akan sangat membantu kita untuk menghilangkan perbedaan antar database. Biasanya masing-masing merek database memiliki cara yang berbeda-beda untuk mengambil nilai ini.

Sebagai contoh, bila kita lakukan secara manual untuk database MySQL, kodenya akan tampak seperti ini.

```
@Transactional(readOnly=false)
public void save(final Person person) {
  simpleJdbcInsert.execute(new BeanPropertySqlParameterSource(person));
  Long newId = simpleJdbcInsert.getJdbcOperations().queryForLong("select last_insert_id()
  person.setId(newId);
}
```

Demikianlah sekilas tentang penggunaan fitur Spring terbaru untuk mengakses database. Pada artikel selanjutnya, kita akan lihat fitur-fitur baru di sisi MVC framework.