|  |
| --- |
| Training: Spring Databases |
| Workbook  (edition 1.0, February, 2017) |

|  |  |
| --- | --- |
| Training: Spring Framework | |
|  |  |
| The authors: | Evgeniy Krivosheev (EKrivosheev@luxoft.com) Dariya Linnik (DLinnik@luxoft.com)  Maxim Potanin ([MPotanin@luxoft.com](mailto:MPotanin@luxoft.com))  Vyacheslav Yakovenko ([VYakovenko@luxoft.com](mailto:VYakovenko@luxoft.com))  Cătălin Tudose ([ftudose@luxoft.com](mailto:ftudose@luxoft.com)) |
| Product version: | Spring 5.x |
|  |  |

**Exercise 1**  
**Using JDBC in Spring when handling data**

## Project

lab-1-jdbc

## Duration

1 hour

## Objectives

Learn how to use JDBC with Spring. Learn to perform basic JDBC operations (SELECT, INSERT, UPDATE) via Spring.

## Description

Spring offers auxiliary API to work with JDBC.

To work with JDBC via Spring, it is necessary to configure DataSource. This is done through application-context.xml.

The HSQLDB is used as a database.

## Objectives

1. Retrieve and print complete country list using CountryDao. *All fields of the* Country *class are filled correctly?* To check, use the JdbcTest test, the testCountryList method.
2. Correct the CountryRowMapper so that all the Country fields would be filled.
3. Retrieve and print complete list of countries whose names begin with A letter. To check, use the JdbcTest test, the testCountryListStartsWithA method.
4. Change the name of any country in the database. To check, use the JdbcTest test, the testCountryChange method (the reference country shall be changed in this test).

## Detailed guidelines

### Part 1: Retrieving the complete country list

1. Open the project lab-1-jdbc;
2. Open the file with the application-countext.xml configuration. Analyze configuration of every bean.
3. Open the CountryDao class. Implement the getCountryList()method so that it would return the complete country list. Use the existing methods as an example.
4. For that, retrieve JdbcTemplate and invoke the query(String s, RowMapper rowMapper)method.
5. As a sql query the GET\_ALL\_COUNTRIES\_SQL can be used.
6. As a RowMapper: COUNTRY\_ROW\_MAPPER.
7. Open the JdbcTest test, execute the testCountryList method.
8. *Did the test run correctly? Why?*

### Part 2: Correcting RowMapper

1. Open the CountryRowMapper class. *What has to be done in order to have all the* Country *fields filled?*
2. Specify codeName for Country in the mapRow method, similarly to id and name.
3. Again, execute the JdbcTest test, the testCountryList method.

### Part 3: Retrieving list of countries whose name begin with A letter.

1. Open the CountryDao class, the getCountryListStartWith()method. *What is* NamedParameterJdbcTemplate*? How does it differ from* JdbcTemplate*?*
2. Open the JdbcTest test, the testCountryListStartsWithA method. *Does the test run correctly?*
3. Now, try to annotate the @DirtiesContext annotation for some tests (mixing it up). *How tests run now? Why?*

### Part 4: Changing the country name

1. Open the CountryDao class.
2. Implement the method for changing country names with known codename: updateCountryName. You can use UPDATE\_COUNTRY\_NAME\_SQL\_1, UPDATE\_COUNTRY\_NAME\_SQL\_2 as a SQL query.

You are going to have the following:

getJdbcTemplate().execute(  
UPDATE\_COUNTRY\_NAME\_SQL\_1 + newCountryName + "'" + UPDATE\_COUNTRY\_NAME\_SQL\_2 + codeName + "'");

1. To execute the JdbcTest test, the testCountryChange method, create a country with new name and known codeName.
2. How can you avoid invoking countryDao.loadCountries(); in the @Before method by changing the application-context.xml and JdbcTest? What are the advantages of this approach? (Additional information needed for answering the question: p. 13.8 Embedded database support, Spring Reference);

**Exercise 2**  
**Using ORM in Spring when handling data**

## Project

lab-2-orm

## Duration

1 hour

## Objectives

Learn how to use ORM via Spring through the example of JPA (Hibernate v.4).

## Description

Spring supports various ORM using one approach. Using ORM via Spring facilitates testing, exceptions handling, and resource management. Spring supports following ORM: JPA, Hibernate, JDO, iBATIS SQL Maps, and others.

We will use Entity classes annotated through javax.persistence (JPA v.2.0: JSR-317; JPA v.2.1: JSR-338) in the example.

## Tasks

1. Create and configure LocalContainerEntityManagerFactoryBean on the basis of existing DataSource. Specify lab.model.Country as @Entity.
2. Create mapping for the Country class.
3. Implement the country save method save(Country country) in the CountryJpaDaoImpl class. Save the country in the database. Check with the CountryDaoImplTest test, testSaveCountry().
4. Implement the method of retrieving all countries, the getAllCountries()method, in the CountryJpaDaoImpl class. Retrieve the list of all countries. Check using the CountryDaoImplTest test, testGetAllCountries ().
5. Implement the method of retrieving country by its name in the CountryJpaDaoImpl class. Retrieve a country by its name. Check using the CountryDaoImplTest test, testGetCountryByName (String name).

## Detailed guidelines

### Part 1: Creating and configuring LocalContainerEntityManagerFactoryBean.

1. Open the file with the application-context.xml configuration.
2. Create the lcemf bean. Indicate LocalContainerEntityManagerFactoryBean.as the bean class.
3. Specify for the bean dataSource, persistenceUnitName, persistenceProviderClass.

You are going to have following configuration:

<bean id=*"myEmf"* class=*"org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"*>

<property name=*"persistenceUnitName"* value=*"springframework.lab.orm.jpa"*/>

<property name=*"dataSource"* ref=*"dataSource"* />

<property name=*"persistenceProviderClass"* value=*"org.hibernate.jpa.HibernatePersistenceProvider"* />

</bean>

Specify in which packages Spring Framework shall lookup for components (@Entity, @Repository, etc.). For that, add the following string in the application-context.xml:

<context:component-scan base-package=*"com.luxoft.springdb.lab2.model, com.luxoft.springdb.lab2.dao"* />

### Part 2: Creating mapping of the Country class.

1. Open the Country class. Add the @Entity and the @Table(name = "COUNTRY") annotations before class declaration.
2. Add the  
   @Id  
   @Column(nullable = false)   
   annotations before defining the getId()method.
3. Add the @Column annotation before defining the getName()method.
4. Add the @Column(name="code\_name") annotation before defining the getCodeName()method.
5. At the end you are going to have the code with following strings:

@Entity

@Table(name = "COUNTRY")

public class Country implements Serializable {

private int id;

private String name;

private String codeName;

public Country() { }

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

public int getId() {

return id;

}

…

public void setId(int id) {this.id = id;}

@Column(name = "NAME")

public String getName() {

return name;

}

…

@Column(name = "CODE\_NAME")

public String getCodeName() {

return codeName;

}

…

### Part 3: Saving a country

1. Open the CountryJpaDaoImpl class. *Why this class inherits* AbstractJpaDao *and implements the* CountryDao *interface?*
2. *Where from and in what way the EntityManagerFactory (emf variable) is obtained?*
3. In the save method retrieve EntityManager, the emf.createEntityManager()method.
4. In entityManager invoke the persist() method **within programmatic transactions**, and define the saved object as a parameter.
5. Verify whether the implementation is correct or not using the CountryDaoImplTest, testSaveCountry()test.

### Part 4: Retrieving the list of all countries

1. Open the getAllCountries() method in the CountryJpaDaoImpl class.
2. Retrieve EntityManager, invoke the createQuery method. Define query row and Country.class as a parameter. Verify whether the implementation is correct or not using the CountryDaoImplTest, testGetAllCountries ()test.
3. You are going to have a code like that:

List<Country> countryList = null;

EntityManager em = emf.createEntityManager();

if (em != null) {

countryList =

em.createQuery("from Country", Country.class)

.getResultList();

em.close();

}

return countryList;

### Part 5: Retrieving a country by its name

1. Open the getCountryByName(String name) method in the CountryJpaDaoImpl class.
2. Create a query for lookup and pass it to the retrieved EntityManager as follows:

Country country = em

.createQuery("SELECT c FROM Country c WHERE c.name LIKE :name",

Country.class).setParameter("name", name)

.getSingleResult();

1. Verify whether the implementation is correct or not using the CountryDaoImpTest, testGetCountryByName()test.
2. As a result, you will have a code like that:

EntityManager em = emf.createEntityManager();

Country country = null;

if (em != null) {

country = em

.createQuery("SELECT c FROM Country c WHERE c.name LIKE :name",

Country.class).setParameter("name", name)

.getSingleResult();

em.close();

}

return country;

Additional questions:

1. *What other strategies for retrieving primary key can be used for entities?* (@GeneratedValue(strategy = GenerationType.AUTO))
2. Switch root logger to the DEBUG mode (for that, edit the file log4j.properties)
3. *Try out various strategies for retrieving primary key. What is the difference between them?*

**Exercise 3**  
**Transaction management in Spring**

## Project

lab-3-tx

## Duration

1 hour

## Objectives

Learn how to use declarative and programmatic transaction management in Spring.

## Description

Spring allows for declarative and programmatic transaction management. It is also possible in Spring to integrate with abstractions for accessing data via ORM or JDBC. Here, transaction management is exemplified through JDBC.

## Guidelines

1. Create TransactionManager based on DataSource. Using annotations define declarative transactions support.
2. Define the countryService bean. Make all CountryServiceImpl class’ methods transactional. Specify different values of propagation for methods according to methods names (for example, for the getAllCountriesRequired()method the Propagation.REQUIRED value should be specified).
3. Test different propagation variants for the CountryService methods using the DeclarativeTransactionTestTest test. Compare behavior of methods that have been invoked within and out of the transaction at different propagation methods.

## Detailed guidelines

### Part 1: Creating TransactionManager

1. Open the file with the application-context.xml configuration.
2. Add following configuration:

<context:annotation-config/>

<context:component-scan base-package=*"com.luxoft.springdb.lab3.service"* />

What is the purpose of each string?

1. Create the transactionManager bean for the org.springframework.jdbc.datasource class.  
   DataSourceTransactionManager. Specify the reference to DataSource.
2. Add the <tx:annotation-driven /> string that activates annotation-based declarative transaction management.

### Part 2: Configuring CountryServiceImpl

1. Create the countryService bean for the CountryServiceImpl class. Specify a reference to dao.
2. Open the CountryServiceImpl class. Add the @Transactional string before class definition. All the class methods will become transactional with default settings.
3. To change settings of specific methods, it is necessary to add the   
   @Transactional(readOnly = false, propagation = Propagation.REQUIRED) definition before method declaration. Specify relevant definitions for all methods like getAllCountriesRequired(), changing only propagation value according to method name.

### Part 3: Testing declarative transaction definition

1. *How should method behave with* propagation REQUIRED *that was invoked within and out of the transaction? With* propagation MANDATORY*?*
2. Verify whether the implementation is correct or not using the DeclarativeTransactionTest test.

### Part 4: Testing programatic transaction definition

1. *Define and configure the* programmaticTransactionCountryService *bean into* application.context.xml.
2. Verify whether the implementation is correct or not using the ProgrammaticTransactionTest test.

**Exercise 4**

**Build a Spring Databases application using Apache Derby**

## Project

lab-4-apache-derby

## Duration

3 hours

## Objectives

Learn how to build a Spring databases application using Apache Derby.

## Description

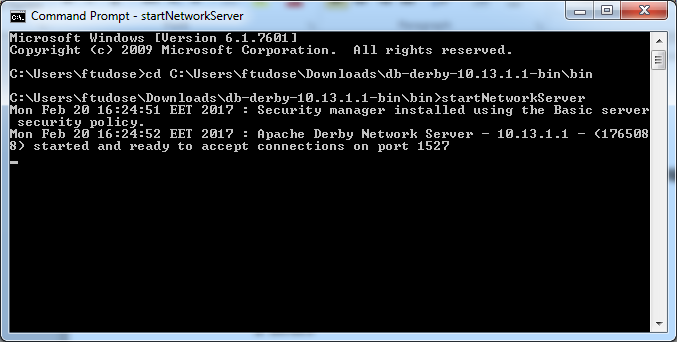
Install the Apache Derby RDBMS, configure it, create a database on it and build a Spring application to interact with it.

## Guidelines

1. Download latest Derby distribution from <https://db.apache.org/derby/derby_downloads.html> and extract it to a folder.
2. Set the DERBY\_HOME environment variable and the PATH variable according to the description here:

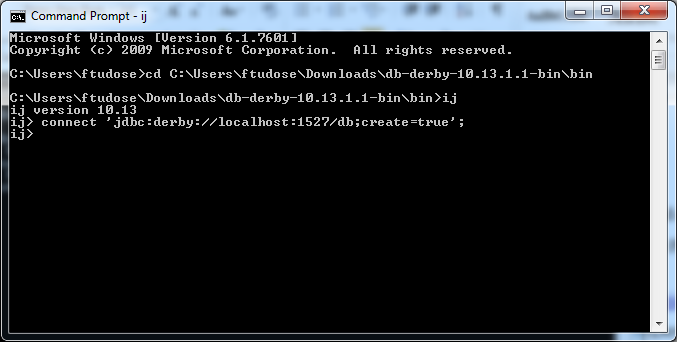
<https://db.apache.org/derby/docs/10.2/getstart/tgssetupjavaenvir.html>

1. Using a command prompt, run the C:\Users\ftudose\Downloads\db-derby-10.13.1.1-bin\bin\startNetworkServer.bat



1. In a separate command prompt, run ij.bat

Run connect 'connect 'jdbc:derby://localhost:1527/db;create=true';

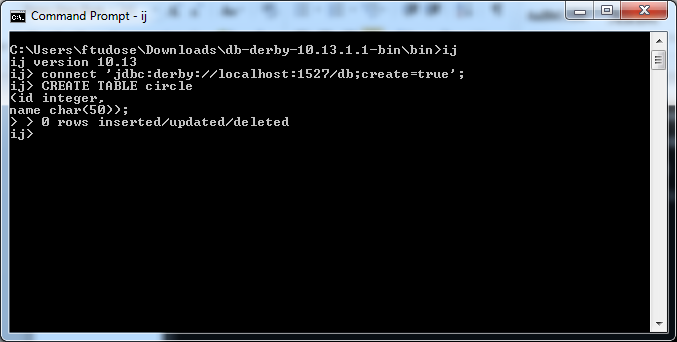


1. Create table CIRCLE:

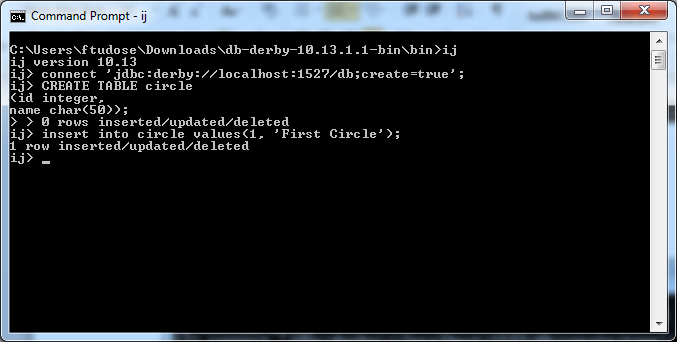
CREATE TABLE circle

(id integer,

name char(50));



1. insert into circle values(1, 'First Circle');



1. Write an application to communicate with the Apache Derby database that you have created and to be able to insert/update/delete/select rows, using the 3 ways discussed during the course: plain JDBC; Spring JDBC; Spring ORM.

There is no detailed description this time, in order to simulate a situation closer to delivery and to allow you to make the implementation the way you desire.