config)

- Spring & Struts Integration (XML)
- Spring & Struts Integration (Java config)
- 14 Tips for Writing Spring MVC Controller

Understand Spring Data JPA with Simple Example

Written by Nam Ha Minh
Last Updated on 24 June 2019 | Print Email

In this tutorial, you will learn how to get started with Spring Data JPA step-by-step through a very simple example. No heavy-weight XML or magic Spring Boot stuffs. Just plain Spring way to keep things as simple as possible.

By completing this tutorial, you will be able to understand how to configure a Spring application to use Spring Data JPA, and how simple it is in writing code for manipulating data with Spring Data JPA.

In the sample project below, we will be using Java 8, Eclipse IDE, Hibernate ORM, Spring framework with Spring Data JPA, MySQL database, MySQL Connector Java as JDBC driver.

Suppose that our Java application needs to manage data of the following table:

```
mysql> desc customer;
+-----------+-------------+------+-----+---------+----------------+
| Field     | Type        | Null | Key | Default | Extra          |
+-----------+-------------+------+-----+---------+----------------+
| id        | int(11)     | NO   | PRI | NULL    | auto_increment |
| firstname | varchar(45) | NO   |     | NULL    |                |
| lastname  | varchar(45) | NO   |     | NULL    |                |
+-----------+-------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)
```

You can use the following MySQL script to create this table:

```
CREATE TABLE `customer` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `firstname` varchar(45) NOT NULL,
  `lastname` varchar(45) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=7 DEFAULT CHARSET=utf8;
```

And let create a simple Maven project in Eclipse.

1. Configure Dependencies in Maven

Open the **pom.xml** file of the project to specify the required dependencies inside the `<dependencies>` section.

Since we use the core of Spring framework with support for Spring Data JPA, add the following XML:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>5.1.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
    <version>2.1.4.RELEASE</version>
</dependency>
```

As you can see, we use Spring 5. And for Hibernate framework, we use only its core ORM - so add the following dependency information:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.1.Final</version>
</dependency>
```

And the last dependency we need is JDBC driver for MySQL:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.14</version>
</dependency>
```

Save the `pom.xml` file, and Maven will automatically download all the required JAR files.

2. Configure Database Connection Properties in persistence.xml

Since Hibernate is used as the provider of JPA (Java Persistence API), we need to specify the database connection properties in the **`persistence.xml`** file which is created under the `META-INF` directory which is under the `src/main/resources` directory.

Here's the content of the **`persistence.xml`** file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
          http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
    version="2.1">

    <persistence-unit name="TestDB">
        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql:///
            <property name="javax.persistence.jdbc.user" value="root" />
            <property name="javax.persistence.jdbc.password" value="password"
            <property name="javax.persistence.jdbc.driver" value="com.mysql.j
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
        </properties>
    </persistence-unit>

</persistence>
```

Modify the JDBC URL, user and password accordingly with your MySQL server. Note that the name of the `persistence-unit` element will be used later in the code.

3. Configure EntityManagerFactory and TransactionManager

Here, we will use Java-based configuration with annotations for a simple Spring application. Create the `AppConfig` class with the following code:

```java
package net.codejava.spring;

import javax.persistence.EntityManagerFactory;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalEntityManagerFactoryBean;

@Configuration
@EnableJpaRepositories(basePackages = {"net.codejava.spring"})
public class AppConfig {
    @Bean
    public LocalEntityManagerFactoryBean entityManagerFactory() {
        LocalEntityManagerFactoryBean factoryBean = new LocalEntityManagerFac
        factoryBean.setPersistenceUnitName("TestDB");

        return factoryBean;
    }

    @Bean
    public JpaTransactionManager transactionManager(EntityManagerFactory ent
        JpaTransactionManager transactionManager = new JpaTransactionManager(
        transactionManager.setEntityManagerFactory(entityManagerFactory);

        return transactionManager;
    }
}
```

As you can see, two annotations are specified before the class:

```
@Configuration
@EnableJpaRepositories(basePackages = {"net.codejava.spring"})
```

The `@Configuration` annotation tells Spring to process this class as the source of configuration. And the `@EnableJpaRepositories` annotation tells Spring to scan for repository classes under the package `net.codejava.spring`, which we will create one in the next section.

When a repository class is found, Spring will generate an appropriate proxy class at runtime to provide implementation details. So the `@EnableJpaRepositories` annotation is required to enable Spring Data JPA in a Spring application.

And in this configuration class, we create two important beans: `LocalEntityManagerFactoryBean` and `JpaTransactionManager`.

The first one sets up an `EntityManagerFactory` to work with the persistence unit named `TestDB`.

And the second one sets up a transaction manager for the configured `EntityManagerFactory`, in order to add transaction capababilities for respositories. Since we're creating a simple example, we don't use the `@EnableTransactionManagement` annotation.

4. Code Model Class

Create the `Customer` class with the following code:

```
package net.codejava.spring;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;

    protected Customer() {
    }

    @Override
    public String toString() {
        return "Customer [firstName=" + firstName + ", lastName=" + lastName
    }

    // getters and setters are not shown for brevity
}
```

As you can see, this domain model class is mapped to the table `customer` in the database using the annotations `@Entity`, suppose that the table has the same name as the class name.

The `@Id` and `@GeneratedValue` annotations map the field `id` to the primary key column of the table. Suppose that all the fields of the class have same name as the column names in the database table.

5. Code Repository Interface

This is the most interesting part. A repository interface leverages the power of Spring Data JPA. Instead of writing boilerplate code for a generic DAO class (as we would normally do with Hibernate/JPA without Spring Data JPA), we just declare a simple interface like this:

```
1  package net.codejava.spring;
2
3  import java.util.List;
4
5  import org.springframework.data.repository.CrudRepository;
6
7  public interface CustomerRepository extends CrudRepository<Customer, Long> {
8      List<Customer> findByLastName(String lastName);
9  }
```

As you can see, this interface extends the `CrudRepository` - which is a special interface defined by Spring Data JPA. The type parameter `<Customer, Long>` specifies the type of the domain model class is `Customer` and the type of the primary key is `Long`.

The `CrudRepository` interface defines common CRUD operations like `save()`, `findAll()`, `findById()`, `delete()`, `count()`... Here are the list of methods defined by this interface:

All Methods	Instance Methods	Abstract Methods
Modifier and Type	Method and Description	
long	**count**() Returns the number of entities available.	
void	**delete**(T entity) Deletes a given entity.	
void	**deleteAll**() Deletes all entities managed by the repository.	
void	**deleteAll**(Iterable<? extends T> entities) Deletes the given entities.	
void	**deleteById**(ID id) Deletes the entity with the given id.	
boolean	**existsById**(ID id) Returns whether an entity with the given id exists.	
Iterable<T>	**findAll**() Returns all instances of the type.	
Iterable<T>	**findAllById**(Iterable<ID> ids) Returns all instances of the type with the given IDs.	
Optional<T>	**findById**(ID id) Retrieves an entity by its id.	
<S extends T> S	**save**(S entity) Saves a given entity.	
<S extends T> Iterable<S>	**saveAll**(Iterable<S> entities) Saves all given entities.	

The interesting thing here is, we don't have to code any implementations for the `CustomerRepository` interface. We just use the methods defined in the CrudRepositoryinterface which is the super interface of CustomerRepository. At runtime, Spring Data JPA generates the implementation class that takes care all the details.

Note that in the `CustomerRepository` interface, we can declare `findByXXX()` methods (`XXX` is the name of a field in the domain model class), and Spring Data JPA will generate the appropriate code:

```
1  List<Customer> findByLastName(String lastName);
```

This will find all customers whose last name matches the specified `lastName` in the method's argument. Very convenient!

Spring Data JPA also provides the `JpaRepository` interface which extends the `CrudRepository` interface. `JpaRepository` defines methods that are specific to JPA.

6. Code Service Class

write a class to make use of the `CustomerRepository` as follows:

```
package net.codejava.spring;

import java.util.List;
import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service("customerService")
public class CustomerService {
    @Autowired
    private CustomerRepository repository;

    public void test() {
        // Save a new customer
        Customer newCustomer = new Customer();
        newCustomer.setFirstName("John");
        newCustomer.setLastName("Smith");

        repository.save(newCustomer);

        // Find a customer by ID
        Optional<Customer> result = repository.findById(1L);
        result.ifPresent(customer -> System.out.println(customer));

        // Find customers by last name
        List<Customer> customers = repository.findByLastName("Smith");
        customers.forEach(customer -> System.out.println(customer));

        // List all customers
        Iterable<Customer> iterator = repository.findAll();
        iterator.forEach(customer -> System.out.println(customer));

        // Count number of customer
        long count = repository.count();
        System.out.println("Number of customers: " + count);
    }
}
```

As you can see, this class is annotated with the @Service annotation, so Spring framework will create an instance of this class as a managed bean in the application context.

The field CustomerRepositoryrepository is annotated with the `@Autowired` annotation so Spring Data JPA will automatically inject an instance of CustomerRepositoryinto this service class.

And finally, code the `test()` method demonstrates some usages of the CustomerRepository.

7. Code Test Program for Spring Data JPA

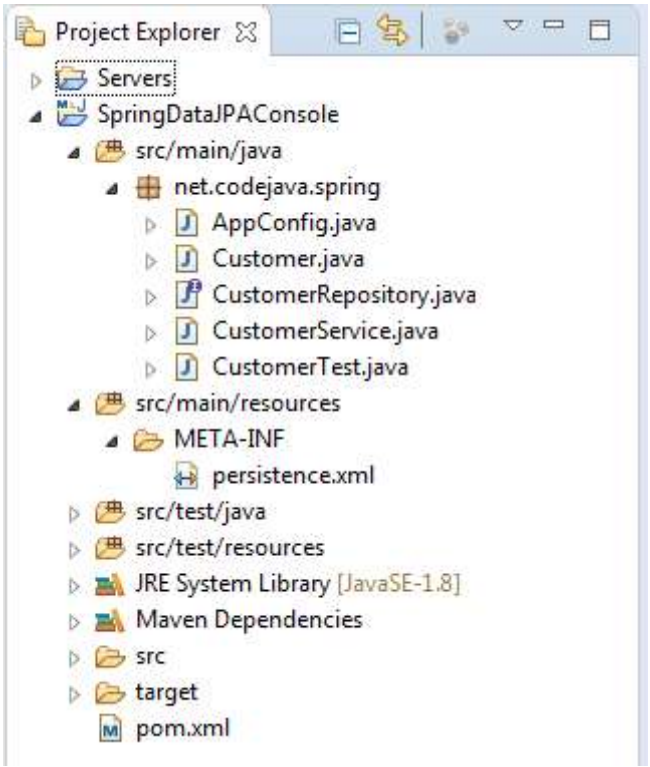
And finally, write a simple test program as follows:

```
package net.codejava.spring;

import org.springframework.context.annotation.AnnotationConfigApplicationCont

public class CustomerTest {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext appContext = new AnnotationConfigA
        appContext.scan("net.codejava.spring");
        appContext.refresh();

        CustomerService customerService = (CustomerService) appContext.getBea
        customerService.test();

        appContext.close();
    }

}
```

This program bootstraps Spring framework to scan classes in the `net.codejava.spring` package. Then it gets the `CustomerService` bean and invoke its `test()` method.

Run this program as a normal Java application and observe the result.

For your reference, the project structure looks like this:



And you can also download the sample project in the attachment section below.

How to get started with Spring Data JPA

References:

- Spring Data JPA Project
- Spring Data JPA Reference Documentation
- CrudRepository Javadoc
- JpaRepository Javadoc

Related Spring and Database Tutorials:

- Spring MVC with JdbcTemplate Example
- How to configure Spring MVC JdbcTemplate with JNDI Data Source in Tomcat
- Spring and Hibernate Integration Tutorial (XML Configuration)
- Spring MVC + Spring Data JPA + Hibernate - CRUD Example

Other Spring Tutorials:

- Understand the core of Spring framework
- Understand Spring MVC
- Understand Spring AOP
- Spring MVC beginner tutorial with Spring Tool Suite IDE
- Spring MVC Form Handling Tutorial
- Spring MVC Form Validation Tutorial
- 14 Tips for Writing Spring MVC Controller
- Spring Web MVC Security Basic Example (XML Configuration)

About the Author:

Nam Ha Minh is certified Java programmer (SCJP and SCWCD). He started programming with Java in the time of Java 1.4 and has been falling in love with Java since then. Make friend with him on Facebook.

Attachments:

SpringDataJPASimpleExample.zip **[Sample Project Code for Spring Data JPA]** **18 kB**

Add comment

Name E-mail