WebSystique
learn together

Spring4 + Hibernate4 + MySQL+ Maven Integration example (Annotations+XML)

August 20, 2014 websystiqueadmin

In this tutorial , we will integrate Spring 4 with `Hibernate 4` using annotation based configuration. We will develop a simple java application , creating hibernate entities, saving data in `MySQL database` , performing database CRUD operations within `transaction` , and learn how different layers interacts with each-other in typical enterprise application, all using `annotation based configuration` . We will also see corresponding XML configuration side-by-side for comparison.

For Spring MVC based application, checkout Spring4 MVC and Hibernate integration.

Following technologies being used:

- Spring 4.0.6.RELEASE
- Hibernate Core 4.3.6.Final

Search …

Recent Posts

Maven Installation and Setup (Windows + Unix)

Spring 4 + Quartz Scheduler Integration Example

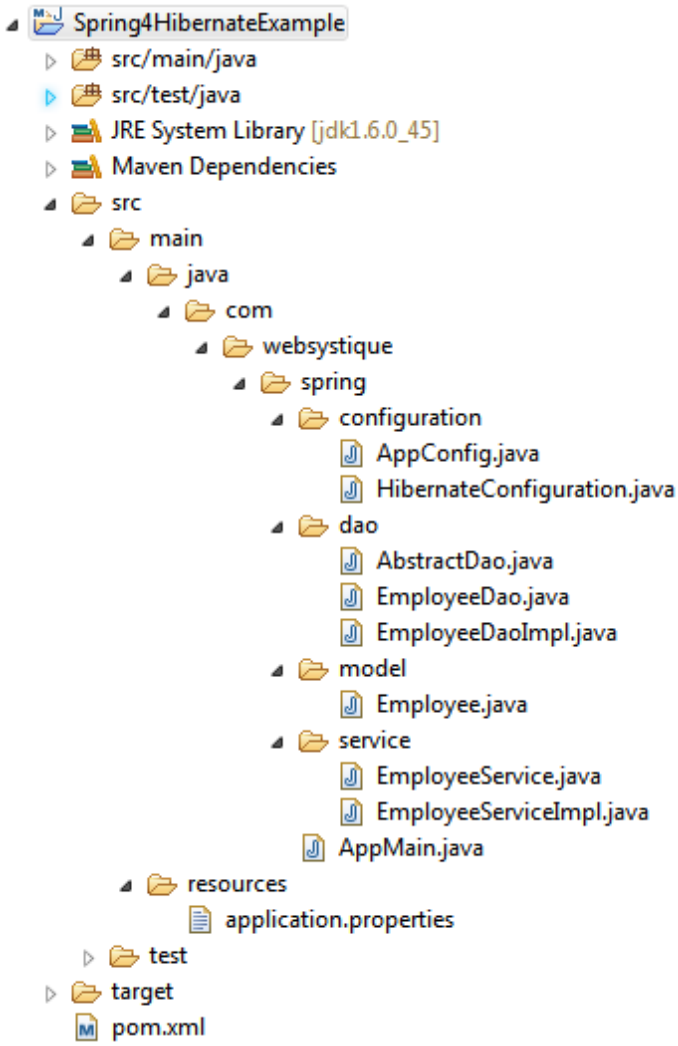
Spring Job Scheduling with @Scheduled & @EnableScheduling Annotations

Spring Job Scheduling using TaskScheduler (XML Config)

Spring DI & Beans Auto-wiring using @Autowired, @Qualifier & @Resource Annotations

- MySQL 5.1.31
- Joda-time 2.3
- Maven 3
- JDK 1.6
- Eclipse JUNO Service Release 2

Project directory structure



Let's now add the content mentioned in above structure explaining each in detail.

Step 1: Update pom.xml to include required dependencies

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.
```

Configuration

```
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.websystique.spring</groupId>
    <artifactId>Spring4HibernateExample</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>

    <name>Spring4HibernateExample</name>

    <properties>
        <springframework.version>4.0.6.RELEASE</springfr
        <hibernate.version>4.3.6.Final</hibernate.versio
        <mysql.version>5.1.31</mysql.version>
        <joda-time.version>2.3</joda-time.version>
    </properties>

    <dependencies>

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

        <!-- Hibernate -->
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>${hibernate.version}</version>
        </dependency>

        <!-- MySQL -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId
            <version>${mysql.version}</version>
        </dependency>

        <!-- Joda-Time -->
        <dependency>
            <groupId>joda-time</groupId>
```

```
            <artifactId>joda-time</artifactId>
            <version>${joda-time.version}</version>
        </dependency>

        <!-- To map JodaTime with database type -->
        <dependency>
            <groupId>org.jadira.usertype</groupId>
            <artifactId>usertype.core</artifactId>
            <version>3.0.0.CR1</version>
        </dependency>

    </dependencies>

</project>
```

Spring, Hibernate & MySQL dependencies are pretty obvious. We have also included joda-time as we will use joda-time library for any date manipulation. usertype-core is included to provide the mapping between database date-type and joda-time LocalDate.

Step 2: Configure Hibernate

`com.websystique.spring.configuration.HibernateConfiguration`

```
package com.websystique.spring.configuration;

import java.util.Properties;

import javax.sql.DataSource;

import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Auto
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentS
import org.springframework.context.annotation.Configurat
import org.springframework.context.annotation.PropertySo
import org.springframework.core.env.Environment;
import org.springframework.jdbc.datasource.DriverManager
import org.springframework.orm.hibernate4.HibernateTrans
import org.springframework.orm.hibernate4.LocalSessionFa
import org.springframework.transaction.annotation.Enable

@Configuration
@EnableTransactionManagement
@ComponentScan({ "com.websystique.spring.configuration"
@PropertySource(value = { "classpath:application.propert
public class HibernateConfiguration {

    @Autowired
    private Environment environment;
```

```
    @Bean
    public LocalSessionFactoryBean sessionFactory() {
        LocalSessionFactoryBean sessionFactory = new Loc
        sessionFactory.setDataSource(dataSource());
        sessionFactory.setPackagesToScan(new String[] {
        sessionFactory.setHibernateProperties(hibernateP
        return sessionFactory;
     }

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverM
        dataSource.setDriverClassName(environment.getReq
        dataSource.setUrl(environment.getRequiredPropert
        dataSource.setUsername(environment.getRequiredPr
        dataSource.setPassword(environment.getRequiredPr
        return dataSource;
    }

    private Properties hibernateProperties() {
        Properties properties = new Properties();
        properties.put("hibernate.dialect", environment.
        properties.put("hibernate.show_sql", environment
        properties.put("hibernate.format_sql", environme
        return properties;
    }

    @Bean
    @Autowired
    public HibernateTransactionManager transactionManage
       HibernateTransactionManager txManager = new Hiber
       txManager.setSessionFactory(s);
       return txManager;
    }
}
```

`@Configuration` indicates that this class contains one or more bean methods annotated with `@Bean` producing beans manageable by spring container. In our case, this class represent hibernate configuration.
`@ComponentScan` is equivalent to `context:component-scan base-package="..."` in xml, providing with where to look for spring managed beans/classes.
`@EnableTransactionManagement` is equivalent to Spring's tx:* XML namespace, enabling Spring's annotation-driven transaction management capability.
`@PropertySource` is used to declare a set of properties(defined in a properties file in application classpath) in Spring run-time `Environment`, providing flexibility to have different values in different application environments.

Method `sessionFactory()` is creating a `LocalSessionFactoryBean`, which exactly mirrors the XML based configuration : We need a dataSource and hibernate properties (same as hibernate.properties). Thanks to `@PropertySource`, we can externalize the real values in a .properties file, and use Spring's `Environment` to fetch the value corresponding to an item. Once the SessionFactory is created, it will be injected into Bean method `transactionManager` which may eventually provide transaction support for the sessions created by this sessionFactory.

Below is the properties file used in this post.

Below is the properties file used in this post.
`/src/main/resources/application.properties`

```
jdbc.driverClassName = com.mysql.jdbc.Driver
jdbc.url = jdbc:mysql://localhost:3306/websystique
jdbc.username = myuser
jdbc.password = mypassword
hibernate.dialect = org.hibernate.dialect.MySQLDialect
hibernate.show_sql = false
hibernate.format_sql = false
```

Corresponding XML based Hibernate configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/bean
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-inst
        xmlns:context="http://www.springframework.org/sc
        xmlns:tx="http://www.springframework.org/schema/
        xmlns:aop="http://www.springframework.org/schema
        xsi:schemaLocation="http://www.springframework.o
                            http://www.springframework.o
                            http://www.springframework.o
                            http://www.springframework.o


    <context:property-placeholder location="classpath:ap

    <context:component-scan  base-package="com.websystiq

    <tx:annotation-driven transaction-manager="transacti

    <bean id="dataSource" class="org.springframework.jdb
        <property name="driverClassName" value="${jdbc.d
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username
```

```
        <property name="password" value="${jdbc.password

    </bean>

    <bean id="sessionFactory" class="org.springframework
        <property name="dataSource" ref="dataSource"/>
        <property name="packagesToScan">
            <list>
                <value>com.websystique.spring.model</val
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">${hibernat
                <prop key="hibernate.show_sql">${hiberna
                <prop key="hibernate.format_sql">${hiber
            </props>
        </property>
    </bean>

    <bean id="transactionManager"  class="org.springfram
        <property name="sessionFactory" ref="sessionFact
    </bean>

    <bean id="persistenceExceptionTranslationPostProcess
        class="org.springframework.dao.annotation.Persis

</beans>
```

Step 3: Configure Spring

`com.websystique.spring.configuration.AppConfig`

```
package com.websystique.spring.configuration;

import org.springframework.context.annotation.ComponentS
import org.springframework.context.annotation.Configurat

@Configuration
@ComponentScan(basePackages = "com.websystique.spring")
public class AppConfig {

}
```

In our simple example, this class is empty and only reason for it's existence is `@ComponentScan` which provides beans auto-detection facility. You may completely remove above configuration and put the component scan logic in application context level (in Main). In full-fledged applications, you may find it handy to configure some beans

(e.g. messageSource, PropertySourcesPlaceHolderConfigurer) in Configuration class.

Corresponding XML based Spring Configuration

```
<beans xmlns="http://www.springframework.org/schema/bean
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
    xmlns:context="http://www.springframework.org/schema
    xsi:schemaLocation="http://www.springframework.org/s
    http://www.springframework.org/schema/context http:/

    <context:component-scan base-package="com.websystiqu

</beans>
```

As for as Annotation based configuration goes, this is all we need to do. Now to make the application complete, we will add service layer, dao layer, Domain object, sample database schema and run the application.

Step 4: Add DAO Layer

`com.websystique.spring.dao.AbstractDao`

```
package com.websystique.spring.dao;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Auto

public abstract class AbstractDao {

    @Autowired
    private SessionFactory sessionFactory;

    protected Session getSession() {
        return sessionFactory.getCurrentSession();
    }

    public void persist(Object entity) {
        getSession().persist(entity);
    }

    public void delete(Object entity) {
        getSession().delete(entity);
    }
}
```

Notice above, that SessionFactory we have created earlier in step 2, will be auto-wired here.This class serve as base class for database related operations.

`com.websystique.spring.dao.EmployeeDao`

```
package com.websystique.spring.dao;

import java.util.List;

import com.websystique.spring.model.Employee;

public interface EmployeeDao {

    void saveEmployee(Employee employee);

    List<Employee> findAllEmployees();

    void deleteEmployeeBySsn(String ssn);

    Employee findBySsn(String ssn);

    void updateEmployee(Employee employee);
}
```

`com.websystique.spring.dao.EmployeeDaoImpl`

```
package com.websystique.spring.dao;

import java.util.List;

import org.hibernate.Criteria;
import org.hibernate.Query;
import org.hibernate.criterion.Restrictions;
import org.springframework.stereotype.Repository;

import com.websystique.spring.model.Employee;

@Repository("employeeDao")
public class EmployeeDaoImpl extends AbstractDao impleme

    public void saveEmployee(Employee employee) {
        persist(employee);
    }

    @SuppressWarnings("unchecked")
    public List<Employee> findAllEmployees() {
        Criteria criteria = getSession().createCriteria(
        return (List<Employee>) criteria.list();
```

```
    }

    public void deleteEmployeeBySsn(String ssn) {
        Query query = getSession().createSQLQuery("delet
        query.setString("ssn", ssn);
        query.executeUpdate();
    }


    public Employee findBySsn(String ssn){
        Criteria criteria = getSession().createCriteria(
        criteria.add(Restrictions.eq("ssn",ssn));
        return (Employee) criteria.uniqueResult();
    }

    public void updateEmployee(Employee employee){
        getSession().update(employee);
    }

}
```

Step 5: Add Service Layer

`com.websystique.spring.service.EmployeeService`

```
package com.websystique.spring.service;

import java.util.List;

import com.websystique.spring.model.Employee;

public interface EmployeeService {

    void saveEmployee(Employee employee);

    List<Employee> findAllEmployees();

    void deleteEmployeeBySsn(String ssn);

    Employee findBySsn(String ssn);

    void updateEmployee(Employee employee);
}
```

`com.websystique.spring.service.EmployeeServiceImpl`

```
package com.websystique.spring.service;

import java.util.List;
```

```
import org.springframework.beans.factory.annotation.Auto
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transa

import com.websystique.spring.dao.EmployeeDao;
import com.websystique.spring.model.Employee;

@Service("employeeService")
@Transactional
public class EmployeeServiceImpl implements EmployeeServ

    @Autowired
    private EmployeeDao dao;

    public void saveEmployee(Employee employee) {
        dao.saveEmployee(employee);
    }

    public List<Employee> findAllEmployees() {
        return dao.findAllEmployees();
    }

    public void deleteEmployeeBySsn(String ssn) {
        dao.deleteEmployeeBySsn(ssn);
    }

    public Employee findBySsn(String ssn) {
        return dao.findBySsn(ssn);
    }

    public void updateEmployee(Employee employee){
        dao.updateEmployee(employee);
    }
}
```

Most interesting part above is `@Transactional` which starts a transaction on each method start, and commits it on each method exit (or rollback if method was failed due to an error). Note that since the transaction are on method scope, and inside method we are using DAO, DAO method will be executed within same transaction.

Step 6: Create Domain Entity Class(POJO)

Let's create the actual Employee Entity itself whose instances we will be playing with in database.

`com.websystique.spring.model.Employee`

```
package com.websystique.spring.model;

import java.math.BigDecimal;

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

import org.hibernate.annotations.Type;
import org.joda.time.LocalDate;

@Entity
@Table(name="EMPLOYEE")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "NAME", nullable = false)
    private String name;

    @Column(name = "JOINING_DATE", nullable = false)
    @Type(type="org.jadira.usertype.dateandtime.joda.Per
    private LocalDate joiningDate;

    @Column(name = "SALARY", nullable = false)
    private BigDecimal salary;

    @Column(name = "SSN", unique=true, nullable = false)
    private String ssn;

    public int getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public LocalDate getJoiningDate() {
        return joiningDate;
    }

    public void setJoiningDate(LocalDate joiningDate) {
        this.joiningDate = joiningDate;
```

```
    }

    public BigDecimal getSalary() {
        return salary;
    }

    public void setSalary(BigDecimal salary) {
        this.salary = salary;
    }

    public String getSsn() {
        return ssn;
    }

    public void setSsn(String ssn) {
        this.ssn = ssn;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + id;
        result = prime * result + ((ssn == null) ? 0 : s
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (!(obj instanceof Employee))
            return false;
        Employee other = (Employee) obj;
        if (id != other.id)
            return false;
        if (ssn == null) {
            if (other.ssn != null)
                return false;
        } else if (!ssn.equals(other.ssn))
            return false;
        return true;
    }

    @Override
    public String toString() {
        return "Employee [id=" + id + ", name=" + name +
                + joiningDate + ", salary=" + salary + "
    }



}
```

This is a standard Entity class annotated with JPA annotations `@Entity`, `@Table`, `@Column` along with hibernate specific annotation `@Type` which we are using to provide mapping between database date type and Joda-Time `LocalDate`

Step 7: Create Schema in database

```
CREATE TABLE EMPLOYEE(
    id INT NOT NULL auto_increment,
    name VARCHAR(50) NOT NULL,
    joining_date DATE NOT NULL,
    salary DOUBLE NOT NULL,
    ssn VARCHAR(30) NOT NULL UNIQUE,
    PRIMARY KEY (id)
);
```

Step 8: Create Main to run as Java Application

```
package com.websystique.spring;

import java.math.BigDecimal;
import java.util.List;

import org.joda.time.LocalDate;
import org.springframework.context.annotation.Annotation
import org.springframework.context.support.AbstractAppli

import com.websystique.spring.configuration.AppConfig;
import com.websystique.spring.model.Employee;
import com.websystique.spring.service.EmployeeService;

public class AppMain {

    public static void main(String args[]) {
        AbstractApplicationContext context = new Annotat

        EmployeeService service = (EmployeeService) cont

        /*
         * Create Employee1
         */
        Employee employee1 = new Employee();
        employee1.setName("Han Yenn");
        employee1.setJoiningDate(new LocalDate(2010, 10,
        employee1.setSalary(new BigDecimal(10000));
        employee1.setSsn("ssn00000001");

        /*
         * Create Employee2
```

```
         */
        Employee employee2 = new Employee();
        employee2.setName("Dan Thomas");
        employee2.setJoiningDate(new LocalDate(2012, 11,
        employee2.setSalary(new BigDecimal(20000));
        employee2.setSsn("ssn00000002");

        /*
         * Persist both Employees
         */
        service.saveEmployee(employee1);
        service.saveEmployee(employee2);

        /*
         * Get all employees list from database
         */
        List<Employee> employees = service.findAllEmploy
        for (Employee emp : employees) {
            System.out.println(emp);
        }

        /*
         * delete an employee
         */
        service.deleteEmployeeBySsn("ssn00000002");

        /*
         * update an employee
         */

        Employee employee = service.findBySsn("ssn000000
        employee.setSalary(new BigDecimal(50000));
        service.updateEmployee(employee);

        /*
         * Get all employees list from database
         */
        List<Employee> employeeList = service.findAllEmp
        for (Employee emp : employeeList) {
            System.out.println(emp);
        }

        context.close();
    }
}
```

Note : In case you want to drop AppConfig altogether, in above main, you just have to replace

```
AbstractApplicationContext context = new AnnotationConfi
```

with

```
AnnotationConfigApplicationContext  context = new Annota
context.scan("com.websystique.spring");
context.refresh();
```

Rest of code remains same. Run above program, you will see following output

```
Employee [id=1, name=Han Yenn, joiningDate=2010-10-10, s
Employee [id=2, name=Dan Thomas, joiningDate=2012-11-11,
Employee [id=1, name=Han Yenn, joiningDate=2010-10-10, s
```

That's it.

Download Source Code

Download Now!

References

- Spring framework
- Hibernate

Related Posts:

1. **Spring 4 MVC+Hibernate 4+MySQL+Maven integration example using annotations**
2. **Spring Auto-detection & Component-scanning Example With Annotations**
3. **Spring Beans Auto-wiring using XML Configuration**
4. **Spring 4 + Quartz Scheduler Integration Example**

spring. permalink.

← Spring @Profile Example

Spring Dependency Injection with Constructor and Property Setter (XML Example) →

Ghostery blocked comments powered by Disqus.