Persistence with Spring

By Baeldung

TABLE OF CONTENTS

1: SPRING WITH MAVEN

2: HIBERNATE 3 WITH SPRING

3: HIBERNATE 4 WITH SPRING

4: SPRING 4 AND JPA WITH HIBERNATE

5: INTRODUCTION TO SPRING DATA JPA

6: THE DAO WITH SPRING AND HIBERNATE

7: THE DAO WITH JPA AND SPRING

8: SIMPLIFY THE DAO WITH SPRING AND JAVA GENERICS

9: TRANSACTIONS WITH SPRING 4 AND JPA

CHAPTER 1

Persistence with Spring

1: SPRING WITH MAVEN

1. Overview

This section illustrates how to set up the Spring dependencies via Maven. The latest Spring releases can be found on Maven Central.

2. Basic Spring Dependencies with Maven

Spring is designed to be highly modular – using one part of Spring should not and does not require another. For example, the basic Spring Context can be without the Persistence or the MVC Spring libraries.

Let's start with a very basic Maven setup which will only use the spring-context dependency:

```
<properties>
   <org.springframework.version>3.2.8.RELEASE</org.springframework.version>
   <!-- <org.springframework.version>4.0.2.RELEASE</org.springframework.version> -->
</properties>
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-context</artifactId>
   <version>${org.springframework.version}</version>
   <scope>runtime</scope>
</dependency>
```

This dependency – spring-context – defines the actual Spring Injection Container and has a small number of dependencies: spring-core, spring-expression, spring-aop and spring-beans. These augment the container by enabling support for some of the core Spring technologies: the Core Spring utilities, the Spring Expression Language (SpEL), the Aspect Oriented Programming support and the JavaBeans mechanism.

Note the we're defining the dependency in the runtime scope – this will make sure that there are no compile time dependencies on any Spring specific APIs. For more advanced usecases, the runtime scope may be removed from some selected Spring dependencies, but for simpler projects, there is no need to compile against Spring to make full use of the framework.

Also note that, starting with Spring 3.2, there is no need to define the CGLIB dependnecy (now upgraded to CGLIB 3.0) – it has been repackaged (the all *net.sf.cglib* package is now *org.springframework.cglib*) and inlined directly within the spring-core JAR (see the JIRA for additional details).

3. Spring Persistence with Maven

Let's now look at the persistence Spring dependencies – mainly *spring-orm*:

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

This comes with Hibernate and JPA support – such as *HibernateTemplate* and *JpaTemplate* – as well as the a few additional, persistence related dependencies: *spring-jdbc* and s*pring-tx*.

The JDBC Data Access library defines the Spring JDBC support as well as the *JdbcTemplate*, and *spring-txre* presents the extremely flexible Transaction Management Abstraction.

4. Spring MVC with Maven

To use the Spring Web and Servlet support, there are two dependencies that need to be included in the *pom*, again in addition to the core dependencies from above:

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

The *spring-web* dependency contains common web specific utilities for both Servlet and Portlet environments, while spring-webmvc enables the MVC support for Servlet environments.

Since spring-webmvc has *spring-web* as a dependency, explicitly defining *spring-web* is not required when using *spring-webmvc*.

5. Spring Security with Maven

Security Maven dependencies are discussed in depth in the Spring Security with Maven section.

6. Spring Test with Maven

The Spring Test Framework can be included in the project via the following dependency:

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

As of Spring 3.2, the Spring MVC Test project, which started as a standalone project available on github, has been included into the core Test Framework – and so including the spring-test dependency is enough.

Note that for older applications still on Spring 3.1 and below, the older standalone Maven dependency still exists and can be used for almost identical results. The dependency is not on Maven Central however, so using it will require adding a custom repository to the pom of the project.

7. Using Milestones

The release version of Spring are hosted on Maven Central. However, if a project needs to use milestone versions, then a custom Spring repository needs to be added to the pom:

```
<repositories>
   <repository>
      <id>repository.springframework.maven.milestone</id>
      <name>Spring Framework Maven Milestone Repository</name>
      <url>http://repo.spring.io/milestone/</url>
   </repository>
</repositories>
```

One this repository has been defined, the project can define dependencies such as:

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-core</artifactId>
   <version>3.2.0.RC2</version>
</dependency>
```

8. Using Snapshots

Similar to milestons, snapshots are hosted in a custom repository:

```
<repositories>
   <repository>
      <id>repository.springframework.maven.snapshot</id>
      <name>Spring Framework Maven Snapshot Repository</name>
      <url>http://repo.spring.io/snapshot/</url>
   </repository>
</repositories>
```

Once the SNAPSHOT repository is enabled in the pom, the following dependencies can be referenced:

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-core</artifactId>
   <version>3.3.0.BUILD-SNAPSHOT</version>
</dependency>
```

As well as – for 4.x:

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-core</artifactId>
   <version>4.0.3.BUILD-SNAPSHOT</version>
</dependency>
```

9. Conclusion

This section discusses the practical details of using Spring with Maven. The Maven dependencies presented here are of course some of the major ones, and there are several others that may be worth mentioning and have not yet made the cut.

Nevertheless this should be a good starting point for using Spring in a project.

CHAPTER 2

Persistence with Spring

2: HIBERNATE 3 WITH SPRING

1. Overview

This section will focus on setting up Hibernate 3 with Spring – we'll look at how to use both XML and Java configuration to set up Spring with Hibernate 3 and MySQL.

2. Java Spring Configuration for Hibernate 3

Setting up Hibernate 3 with Spring and Java config is straightforward:

```
import java.util.Properties;
import javax.sql.DataSource;
import org.apache.tomcat.dbcp.dbcp.BasicDataSource;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.core.env.Environment;
import org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor;
import org.springframework.orm.hibernate3.HibernateTransactionManager;
import org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import com.google.common.base.Preconditions;

 @Configuration

@EnableTransactionManagement
@PropertySource({ "classpath:persistence-mysql.properties" })
@ComponentScan({ "org.baeldung.spring.persistence" })
public class PersistenceConfig {

   @Autowired
   private Environment env;
```

```
@Bean
public AnnotationSessionFactoryBean sessionFactory() {
  AnnotationSessionFactoryBean sessionFactory = new AnnotationSessionFactoryBean();
  sessionFactory.setDataSource(restDataSource());
  sessionFactory.setPackagesToScan(new String[] { “org.baeldung.spring.persistence.model” });
  sessionFactory.setHibernateProperties(hibernateProperties());

  return sessionFactory;
}

 @Bean

public DataSource restDataSource() {
  BasicDataSource dataSource = new BasicDataSource();
  dataSource.setDriverClassName(env.getProperty(“jdbc.driverClassName”));
  dataSource.setUrl(env.getProperty(“jdbc.url”));
  dataSource.setUsername(env.getProperty(“jdbc.user”));
  dataSource.setPassword(env.getProperty(“jdbc.pass”));

  return dataSource;
}

 @Bean
@Autowired

public HibernateTransactionManager transactionManager(SessionFactory sessionFactory) {
  HibernateTransactionManager txManager = new HibernateTransactionManager();
  txManager.setSessionFactory(sessionFactory);

  return txManager;
}
```

```
  @Bean

  public PersistenceExceptionTranslationPostProcessor exceptionTranslation() {
    return new PersistenceExceptionTranslationPostProcessor();
  }

  Properties hibernateProperties() {
    return new Properties() {
      {
        setProperty(“hibernate.hbm2ddl.auto”, env.getProperty(“hibernate.hbm2ddl.auto”));
        setProperty(“hibernate.dialect”, env.getProperty(“hibernate.dialect”));
      }
    };
  }
}
```

Compared to the XML Configuration – described next – there is a small difference in the way one bean in the configuration access another. In XML there is no difference between pointing to a bean or pointing to a bean factory capable of creating that bean. Since the Java configuration is type-safe – pointing directly to the bean factory is no longer an option – we need to retrieve the bean from the bean factory manually:

```
txManager.setSessionFactory(sessionFactory().getObject());
```

3. XML Spring Configuration for Hibernate 3

Similarly, we can set up Hibernate 3 with XML config as well:

```
<?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:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.1.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.1.xsd">

  <context:property-placeholder location="classpath:persistence-mysql.properties" />

  <bean id="sessionFactory"
   class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan" value="org.baeldung.spring.persistence.model" />
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.hbm2ddl.auto">${hibernate.hbm2ddl.auto}</prop>
        <prop key="hibernate.dialect">${hibernate.dialect}</prop>
      </props>
    </property>
  </bean>

  <bean id="dataSource"
   class="org.apache.tomcat.dbcp.dbcp.BasicDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.user}" />
    <property name="password" value="${jdbc.pass}" />
  </bean>

  <bean id="txManager"
   class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
  </bean>

  <bean id="persistenceExceptionTranslationPostProcessor"
   class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>

</beans>
```

Then, this XML file is bootstrapped into the Spring context using a *@Configuration* class:

```
@Configuration
@EnableTransactionManagement
@ImportResource({ "classpath:persistenceConfig.xml" })
public class PersistenceXmlConfig {
   //
}
```

For both types of configuration, the JDBC and Hibernate specific properties are stored in a properties file:

```
# jdbc.X
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/spring_hibernate_dev?createDatabaseIfNotExist=true
jdbc.user=sectionuser
jdbc.pass=sectionmy5ql
# hibernate.X
hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
hibernate.show_sql=false
hibernate.hbm2ddl.auto=create-drop
```

4. Spring, Hibernate and MySQL

The example above uses MySQL 5 as the underlying database configured with Hibernate – however, Hibernate supports several underlying SQL Databases.

4.1. The Driver

The Driver class name is configured via the jdbc.driverClassName property provided to the DataSource.

In the example above, it is set to *com.mysql.jdbc.Driver* from the *mysql-connector-java* dependency we defined in the pom, at the start of the section.

4.2. The Dialect

The Dialect is configured via the hibernate.dialect property provided to the Hibernate *SessionFactory*.

In the example above, this is set to *org.hibernate.dialect.MySQL5Dialect* as we are using MySQL 5 as the underlying Database. There are several other dialects supporting MySQL:

- org.hibernate.dialect.MySQL5InnoDBDialect – for MySQL 5.x with the InnoDB storage engine
- org.hibernate.dialect.MySQLDialect – for MySQL prior to 5.x
- org.hibernate.dialect.MySQLInnoDBDialect – for MySQL prior to 5.x with the InnoDB storage engine
- org.hibernate.dialect.MySQLMyISAMDialect – for all MySQL versions with the ISAM storage engine

Hibernate supports SQL Dialects for every supported Database.

5. Usage

At this point, Hibernate 3 is fully configured with Spring and we can inject the raw Hibernate *SessionFactorydirectly* whenever we need to:

```
public abstract class FooHibernateDAO{

  @Autowired
  SessionFactory sessionFactory;

  ...

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

6. Maven

To add the Spring Persistence dependencies to the pom, please see the Spring with Maven example – we'll need to define both *spring-context* and *spring-orm*.

Continuing to Hibernate 3, the Maven dependencies are simple:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>3.6.10.Final</version>
</dependency>
```

Then, to enable Hibernate to use its proxy model, we need *javassist* as well:

```
<dependency>
  <groupId>org.javassist</groupId>
  <artifactId>javassist</artifactId>
  <version>3.18.2-GA</version>
</dependency>
```

We're going to use MySQL as our DB for this section, so we'll also need:

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.32</version>
  <scope>runtime</scope>
</dependency>
```

And finally, we will not be using the Spring datasource implementation – the *DriverManagerDataSource*; instead we'll use a production ready connection pool solution – Tomcat JDBC Connection Pool:

```
<dependency>
   <groupId>org.apache.tomcat</groupId>
   <artifactId>tomcat-dbcp</artifactId>
   <version>7.0.55</version>
</dependency>
```

7. Conclusion

In this section, we configured Hibernate 3 with Spring – both with Java and XML configuration. e configured Hibernate 3 with Spring – both with Java and XML configuration.

CHAPTER 3

Persistence with Spring

3: HIBERNATE 4 WITH SPRING

1. Overview

This section will focus on setting up Hibernate 4 with Spring – we'll look at how to configure Spring with Hibernate 4 using both Java and XML Configuration. Parts of this process are of course common to the Hibernate 3 section.

2. Maven

To add the Spring Persistence dependencies to the project *pom.xml*, please see the section focused on the Spring and Maven dependencies.

Continuing with Hibernate 4, the Maven dependencies are simple:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>4.3.6.Final</version>
</dependency>
```

Then, to enable Hibernate to use its proxy model, we need javassist as well:

```
<dependency>
  <groupId>org.javassist</groupId>
  <artifactId>javassist</artifactId>
  <version>3.18.2-GA</version>
</dependency>
```

And since we're going to use MySQL for this section, we'll also need:

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.32</version>
  <scope>runtime</scope>
</dependency>
```

And finally, we are using a proper connection pool instead of the dev-only Spring implementation – *theDriverManagerDataSource*. We're using here the Tomcat JDBC Connection Pool:

```
<dependency>
   <groupId>org.apache.tomcat</groupId>
   <artifactId>tomcat-dbcp</artifactId>
   <version>7.0.55</version>
</dependency>
```

3. Java Spring Configuration for Hibernate 4

To use Hibernate 4 in a project, a few things have changed on the configuration side when moving from a Hibernate 3 setup.

The main aspect that is different when upgrading from Hibernate 3 is the way to create the SessionFactory with Hibernate 4.

This is now done by using the *LocalSessionFactoryBean* from the hibernate4 package – which replaces the older AnnotationSessionFactoryBean from the hibernate3 package.
The new *FactoryBean* has the same responsibility – it bootstraps the *SessionFactory* from annotation scanning. This is neccessary because, starting with Hibernate 3.6, the old *AnnotationConfiguration* was merged into *Configuration* and so the new Hibernate 4*LocalSessionFactoryBean* is using this new *Configuration* mechanism.

[box type=”info” size=”medium” style=”rounded” border=”full” icon=”none”]

It is also worth noting that, in Hibernate 4, the *Configuration.buildSessionFactory* method and mechanism have also been deprecated in favour of *Configuration.buildSessionFactory(ServiceRegistry)* – which the Spring *LocalSessionFactoryBean* is not yet using.

[/box]

The Spring Java Configuration for Hibernate 4:

```
import java.util.Properties;
import javax.sql.DataSource;
import org.hibernate.SessionFactory;
import org.apache.tomcat.dbcp.dbcp.BasicDataSource;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.core.env.Environment;
import org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor;
import org.springframework.orm.hibernate4.HibernateTransactionManager;
import org.springframework.orm.hibernate4.LocalSessionFactoryBean;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import com.google.common.base.Preconditions;

@Configuration
@EnableTransactionManagement
@PropertySource({ “classpath:persistence-mysql.properties” })
@ComponentScan({ “org.baeldung.spring.persistence” })
public class PersistenceConfig {

  @Autowired
  private Environment env;

  @Bean
  public LocalSessionFactoryBean sessionFactory() {
    LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
    sessionFactory.setDataSource(restDataSource());
    sessionFactory.setPackagesToScan(new String[] { “org.baeldung.spring.persistence.model” });
    sessionFactory.setHibernateProperties(hibernateProperties());
```

```
      return sessionFactory;
   }

   @Bean
   public DataSource restDataSource() {
      BasicDataSource dataSource = new BasicDataSource();
      dataSource.setDriverClassName(env.getProperty("jdbc.driverClassName"));
      dataSource.setUrl(env.getProperty("jdbc.url"));
      dataSource.setUsername(env.getProperty("jdbc.user"));
      dataSource.setPassword(env.getProperty("jdbc.pass"));

      return dataSource;
   }

   @Bean
   @Autowired
   public HibernateTransactionManager transactionManager(SessionFactory sessionFactory) {
      HibernateTransactionManager txManager = new HibernateTransactionManager();
      txManager.setSessionFactory(sessionFactory);

      return txManager;
   }

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

   Properties hibernateProperties() {
      return new Properties() {
         {
            setProperty("hibernate.hbm2ddl.auto", env.getProperty("hibernate.hbm2ddl.auto"));
            setProperty("hibernate.dialect", env.getProperty("hibernate.dialect"));
            setProperty("hibernate.globally_quoted_identifiers", "true");
         }
      };
   }
}
```

4. XML Spring Configuration for Hibernate 4

Simillary, Hibernate 4 can be configured with XML as well:

```
<?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:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.1.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.1.xsd">

  <context:property-placeholder location="classpath:persistence-mysql.properties" />

  <bean id="sessionFactory"
   class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan" value="org.baeldung.spring.persistence.model" />
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.hbm2ddl.auto">${hibernate.hbm2ddl.auto}</prop>
        <prop key="hibernate.dialect">${hibernate.dialect}</prop>
      </props>
    </property>
  </bean>

  <bean id="dataSource"
   class="org.apache.tomcat.dbcp.dbcp.BasicDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.user}" />
    <property name="password" value="${jdbc.pass}" />
  </bean>

  <bean id="transactionManager"
   class="org.springframework.orm.hibernate4.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
  </bean>

  <bean id="persistenceExceptionTranslationPostProcessor"
   class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>

</beans>
```

To bootstrap the XML into the Spring Context, we can use a simple Java Configuration file if the application is configured with Java configuration:

```
@Configuration
@EnableTransactionManagement
@ImportResource({ “classpath:hibernate4Config.xml” })
public class HibernateXmlConfig{
   //
}
```

Alternativelly we can simply provide the XML file to the Spring Context, if the overall configuration is purely XML.

For both types of configuration, the JDBC and Hibernate specific properties are stored in a properties file:

```
# jdbc.X
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/spring_hibernate_dev?createDatabaseIfNotExist=true
jdbc.user=sectionuser
jdbc.pass=sectionmy5ql

# hibernate.X
hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
hibernate.show_sql=false
hibernate.hbm2ddl.auto=create-drop
```

5. Spring, Hibernate and MySQL

The Drivers and Dialects supported by Hibernate have been extensively discussed for Hibernate 3 – and everything still applies for Hibernate 4 as well.

6. Usage

At this point, Hibernate 4 is fully configured with Spring and we can inject the raw Hibernate SessionFactorydirectly whenever we need to:

```
public abstract class BarHibernateDAO{

   @Autowired
   SessionFactory sessionFactory;

   ...

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

An important note here is that this is now the recommended way to use the Hibernate API – the older*HibernateTemplate* is no longer included in the new *org.springframework.orm.hibernate4* package as it shouldn't be used with Hibernate 4.

7. Conclusion

In this section, we configured Spring with Hiberate 4 – both with Java and XML configuration.

CHAPTER 4

Persistence with Spring

4: SPRING 4 AND JPA WITH HIBERNATE

1. Overview

This is section shows how to set up Spring with JPA, using Hibernate as a persistence provider.

2. The JPA Spring Configuration with Java

To use JPA in a Spring project, the EntityManager needs to be set up.

This is the main part of the configuration – and it is done via a Spring factory bean – either the *simplerLocalEntityManagerFactoryBean* or the more flexible LocalContainerEntityManagerFactoryBean. The latter option is used here, so that additional properties can be configured on it:

```
@Configuration
@EnableTransactionManagement
public class PersistenceJPAConfig{

  @Bean
  public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean em = new LocalContainerEntityManagerFactoryBean();
    em.setDataSource(dataSource());
    em.setPackagesToScan(new String[] { “org.baeldung.persistence.model” });

    JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
    em.setJpaVendorAdapter(vendorAdapter);
    em.setJpaProperties(additionalProperties());

    return em;
  }
  @Bean
  public DataSource dataSource(){
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql://localhost:3306/spring_jpa");
    dataSource.setUsername( "sectionuser" );
    dataSource.setPassword( "sectionmy5ql" );
    return dataSource;
  }

  @Bean
  public PlatformTransactionManager transactionManager(EntityManagerFactory emf){
    JpaTransactionManager transactionManager = new JpaTransactionManager();
    transactionManager.setEntityManagerFactory(emf);

    return transactionManager;
  }

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

  Properties additionalProperties() {
    Properties properties = new Properties();
    properties.setProperty("hibernate.hbm2ddl.auto", "create-drop");
    properties.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQL5Dialect");
    return properties;
  }
}
```

Also, note that, before Spring 3.2, cglib had to be on the classpath for Java *@Configuration* classes to work; to better understand the need for *cglib* as a dependency, see this discussion about the cglib artifact in Spring.

3. The JPA Spring Configuration with XML

The same Spring Configuration with 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:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-3.2.xsd">

  <bean id="myEmf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan" value="org.baeldung.persistence.model" />
    <property name="jpaVendorAdapter">
      <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
    </property>
    <property name="jpaProperties">
      <props>
        <prop key="hibernate.hbm2ddl.auto">create-drop</prop>
        <prop key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>
      </props>
    </property>
  </bean>

  <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:3306/spring_jpa" />
    <property name="username" value="sectionuser" />
    <property name="password" value="sectionmy5ql" />
  </bean>

  <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="myEmf" />
  </bean>
  <tx:annotation-driven />

  <bean id="persistenceExceptionTranslationPostProcessor"
    class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor" />

</beans>
```

There is a relatively small difference between the way Spring is configured in XML and the new Java based configuration – in XML, a reference to another bean can point to either the bean or a bean factory for that bean. In Java however, since the types are different, the compiler doesn't allow it, and so the *EntityManagerFactory* is first retrieved from it's bean factory and then passed to the transaction manager:

txManager.setEntityManagerFactory(this.entityManagerFactoryBean().getObject());

4. Going full XML-less

Usually JPA defines a persistence unit through the *META-INF/persistence.xml* file. Starting with Spring 3.1, thepersistence.xml is no longer necessary – the *LocalContainerEntityManagerFactoryBean* now supports a*'packagesToScan'* property where the packages to scan for *@Entity* classes can be specified.

This file was the last piece of XML to be removed – now, JPA can be fully set up with no XML.

4.1. The JPA Properties

JPA properties would usually be specified in the *persistence.xml* file; alternatively, the properties can be specified directly to the entity manager factory bean:

factoryBean.setJpaProperties(this.additionalProperties());

As a side-note, if Hibernate would be the persistence provider, then this would be the way to specify Hibernate specific properties.

5. The Maven configuration

In addition to Spring Core and persistence dependencies – show in detail in the Spring with Maven section – we also need to define JPA and Hibernate in the project, as well as a MySQL connector:

```xml
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>4.3.5.Final</version>
  <scope>runtime</scope>
</dependency>

<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.30</version>
  <scope>runtime</scope>
</dependency>
```

Note that the MySQL dependency is included as a reference – a driver is needed to configure the datasource, but any Hibernate supported database will do.

6. Conclusion

This section illustrated how to configure JPA with Hibernate in Spring using both XML and Java configuration.
We also discussed how to get rid of the last piece of XML usually associated with JPA – the *persistence.xml*. The final result is a lightweight, clean DAO implementation, with almost no compile-time reliance on Spring.

CHAPTER 5

Persistence with Spring

5: INTRODUCTION TO SPRING DATA JPA

1. Overview

This section will focus on introducing Spring Data JPA into a Spring 4 project and fully configuring the persistence layer.

2. The Spring Data generated DAO – No More DAO Implementations

As discussed in an earlier section, the DAO layer usually consists of a lot of boilerplate code that can and should be simplified. The advantages of such a simplification are many: a decrease in the number of artifacts that need to be defined and maintained, consistency of data access patterns and consistency of configuration.

Spring Data takes this simplification one step forward and makes it possible to remove the DAO implementations entirely – the interface of the DAO is now the only artifact that need to be explicitly defined.

In order to start leveraging the Spring Data programming model with JPA, a DAO interface needs to extend the JPA specific Repository interface – *JpaRepository*. This will enable Spring Data to find this interface and automatically create an implementation for it.
By extending the interface we get the most relevant CRUD methods for standard data access available in a standard DAO out of the box.

3. Custom Access Method and Queries

As discussed, by implementing one of the *Repository* interfaces, the DAO will already have some basic CRUD methods (and queries) defined and implemented.

To define more specific access methods, Spring JPA supports quite a few options – you can:

- simply define a new method in the interface
- provide the actual JPQ query by using the @Query annotation
- use the more advanced Specification and Querydsl support in Spring Data
- define custom queries via JPA Named Queries

The third option – the Specifications and Querydsl support – is similar to JPA Criteria but using a more flexible and convenient API – making the whole operation much more readable and reusable. The advantages of this API will become more pronounced when dealing with a large number of fixed queries that could potentially be more concisely expressed through a smaller number of reusable blocks that keep occurring in different combinations.

This last option has the disadvantage that it either involves XML or burdening the domain class with the queries.

3.1. Automatic Custom Queries

When Spring Data creates a new *Repository* implementation, it analyses all the methods defined by the interfaces and tries to automatically generate queries from the method names. While this has some limitations, it is a very powerful and elegant way of defining new custom access methods with very little effort.

Let's look at an example: if the managed entity has a name field (and the Java Bean standard getName andsetName methods), we'll define the findByName method in the DAO interface; this will automatically generate the correct query:

```
public interface IFooDAO extends JpaRepository< Foo, Long >{
  Foo findByName( String name );
}
```

This is a relatively simple example; a much larger set of keywords is supported by query creation mechanism.

In the case that the parser cannot match the property with the domain object field, the following exception is thrown:

```
java.lang.IllegalArgumentException: No property nam found for type class org.rest.model.Foo
```

3.2. Manual Custom Queries

Let's now look at a custom query that we will define via the *@Query* annotation:

```
@Query("SELECT f FROM Foo f WHERE LOWER(f.name) = LOWER(:name)")
Foo retrieveByName(@Param("name") String name);
```

For even more fine grained control over the creation of queries, such as using named parameters or modifying existing queries, the reference is a good place to start.

4. Transaction Configuration

The actual implementation of the Spring Data managed DAO is indeed hidden, since we don't work with it directly. However – it is a simple enough implementation – the *SimpleJpaRepository* – which defines transaction semantics using annotations.

More explicitly – a read only *@Transactional* annotation is used at the class level, which is then overridden for the non read-only methods. The rest of the transaction semantics are default, but these can be easily overridden manually per method.

4.1. Exception Translation is alive and well

The question is now – since we're not using the default Spring ORM templates *(JpaTemplate,HibernateTemplate)* – are we loosing exception translation by using Spring Data JPA?-Are we not going to get our JPA exceptions translated to Spring's *DataAccessException* hierarchy?

Of course not – exception translation is still enabled by the use of the *@Repository* annotation on the DAO. This annotation enables a Spring bean postprocessor to advice all *@Repository* beans with all *thePersistenceExceptionTranslator* instances found in the Container – and provide exception translation just as before.

The fact that exception translation is indeed active can easily be verified with an integration test:

```
@Test(expected = DataIntegrityViolationException.class)
public void givenFooHasNoName_whenInvalidEntityIsCreated_thenDataException() {
   service.create(new Foo());
}
```

Keep in mind that exception translation is done through proxies – in order for Spring to be able to create proxies around the DAO classes, these must not be declared final.

5. Spring Data Configuration

To activate the Spring JPA repository support with an XML configuration – we'll use the jpa namespace and specify the package where to DAO interfaces are located:

```
<?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:jpa="http://www.springframework.org/schema/data/jpa"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
    http://www.springframework.org/schema/data/jpa
    http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

<jpa:repositories base-package="org.rest.dao.spring" />

</beans>
```

Starting with Spring Data 1.4, we can do the same with Java-only configuration:

```
@EnableJpaRepositories(basePackages = "org.baeldung.persistence.dao")
public class PersistenceConfig { ... }
```

6. The Spring Java or XML configuration

We already discussed in great detail how to configure JPA in Spring in a previous section. Spring Data also takes advantage of the Spring support for the JPA *@PersistenceContext* annotation which it uses to wire the*EntityManager* into the Spring factory bean responsible with creating the actual DAO implementations –JpaRepositoryFactoryBean.

In addition to the already discussed configuration, we also need to include the Spring Data XML Config – if we are using XML:

```
@Configuration
@EnableTransactionManagement
@ImportResource( “classpath*:*springDataConfig.xml” )
public class PersistenceJPAConfig{
   ...
}
```

7. The Maven dependency

In addition to the Maven configuration for JPA defined in a previous section, the *spring-data-jpa* dependency is added:

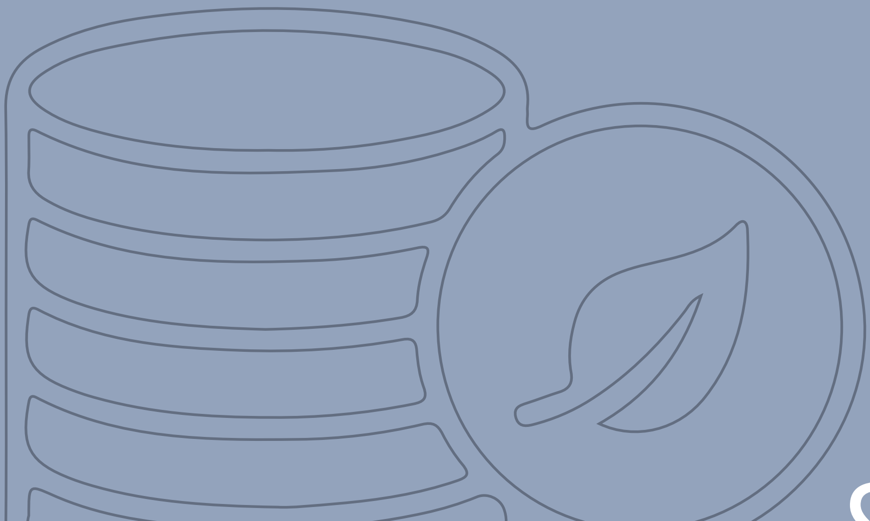
```
<dependency>
   <groupId>org.springframework.data</groupId>
   <artifactId>spring-data-jpa</artifactId>
   <version>1.6.0.RELEASE</version>
</dependency>
```

8. Conclusion

This section covered the configuration and implementation of the persistence layer with Spring 4, JPA 2 and Spring Data JPA (part of the Spring Data umbrella project), using both XML and Java based configuration.

The various method of defining more advanced custom queries are discussed, as well as configuration with the new jpa namespace and transactional semantics. The final result is a new and elegant take on data access with Spring, with almost no actual implementation work.

CHAPTER 6



Persistence with Spring

6: THE DAO WITH SPRING AND HIBERNATE

1. Overview

This section will show how to implement the DAO with Spring and Hibernate. For the core Hibernate configuration, see the sections about Hibernate 3 and Hibernate 4 with Spring.

2. No More Spring Templates

Starting Spring 3.0 and Hibernate 3.0.1, the Spring HibernateTemplate is no longer necessary to manage the Hibernate Session. It is now possible to make use of contextual sessions – sessions managed directly by Hibernate and active throughout the scope of a transaction.

As a consequence, it is now best practice to use the Hibernate API directly instead of the *HibernateTemplate*, which will effectively decouple the DAO layer implementation from Spring entirely.

2.1. Exception Translation without the HibernateTemplate – alive and well

Exception Translation was one of the responsibilities of *HibernateTemplate* – translating the low level Hibernate exceptions to higher level, generic Spring exceptions.

Without the template, this mechanism is still enabled and active for all the DAOs annotated with the*@Repository* annotation. Under the hood, this uses a Spring bean postprocessor that will advice all *@Repositorybeans* with all the *PersistenceExceptionTranslator* found in the Spring context.

One thing to remember is that exception translation is done through proxies; in order for Spring to be able to create proxies around the DAO classes, these must not be declared **final.**

2.2. Hibernate Session management without the Template

When Hibernate support for contextual sessions came out, the *HibernateTemplate* essentially became obsolete; in fact, the javadoc of the class has been updated with this advice (bold from the original):

Note: As of Hibernate 3.0.1, transactional Hibernate access code can also be coded in plain Hibernate style. Hence, for newly started projects, consider adopting the standard Hibernate3 style of coding data access objects instead, based on {@link org.hibernate. SessionFactory#getCurrentSession()}.

---

3. The DAO

We'll start with the base DAO – an abstract, parametrized DAO which supports the common generic operations and is meant to be extended for each entity:

```
public abstract class AbstractHibernateDAO< T extends Serializable >{
  private Class< T > clazz;

  @Autowired
  private SessionFactory sessionFactory;

  public void setClazz( final Class< T > clazzToSet ){
    clazz = clazzToSet;
  }

  public T findOne( final long id ){
    return (T) getCurrentSession().get( clazz, id );
  }
  public List< T > findAll(){
    return getCurrentSession()
     .createQuery( “from “ + clazz.getName() ).list();
  }

  public void save( final T entity ){
    getCurrentSession().persist( entity );
  }

  public T update( final T entity ){
    return (T) getCurrentSession().merge( entity );
  }

  public void delete( final T entity ){
    getCurrentSession().delete( entity );
  }
  public void deleteById( final long id ){
    final T entity = findOne( id);
    delete( entity );
  }

  protected final Session getCurrentSession(){
    return sessionFactory.getCurrentSession();
  }
}
```

A few aspects are interesting here – as discussed, the abstract DAO does not extend any Spring template (such as *HibernateTemplate*). Instead, the Hibernate *SessionFactory* is injected directly in the DAO, and will have the role of the main Hibernate API, through the contextual Session it exposes:

```
this.sessionFactory.getCurrentSession();
```

Also, note that the Class of the entity is passed in the constructor to be used in the generic operations.

Now, let's look at an example implementation of this DAO, for a *Foo* entity:

```
@Repository
public class FooDAO extends AbstractHibernateDAO< Foo > implements IFooDAO{

   public FooDAO(){
      setClazz(Foo.class );
   }
}
```

4. Conclusion

This section covered the configuration and implementation of the persistence layer with Hibernate and Spring 4, using both XML and Java based configuration.

The reasons to stop relying on templates for the DAO layer was discussed, as well as possible pitfalls of configuring Spring to manage transactions and the Hibernate Session. The final result is a lightweight, clean DAO implementation, with almost no compile-time reliance on Spring.

CHAPTER 7

Persistence with Spring

7: THE DAO WITH JPA AND SPRING

1. Overview

This section will show how to implement the DAO with Spring and JPA. For the core JPA configuration, see the section about JPA with Spring.

2. No More Spring Templates

Starting with Spring 3.1, the *JpaTemplate* and the corresponding *JpaDaoSupport* have been deprecated in favor of using the native Java Persistence API.

Also, both of these classes are only relevant for JPA 1 (from the *JpaTemplate* javadoc): Note that this class did not get upgraded to JPA 2.0 and never will.

As a consequence, it is now best practice to use the Java Persistence API directly instead of the *JpaTemplate*.

2.1. Exception Translation without the Template

One of the responsibilities of *JpaTemplate* was exception translation – translating the low level exceptions into higher level, generic Spring exceptions.

Without the template, exception translation is still enabled and fully functional for all DAOs annotated with*@Repository*. Spring implements this with a bean postprocessor which will advice all *@Repository* beans with all the *PersistenceExceptionTranslator* found in the Container.

It is also important to note that the exception translation mechanism uses proxies – in order for Spring to be able to create proxies around the DAO classes, these must not be declared final.

3. The DAO

First, we'll implement the base layer for all the DAOs – an abstract class using generics and designed to be extended:

```
public abstract class AbstractJpaDAO< T extends Serializable > {

   private Class< T > clazz;

   @PersistenceContext
   EntityManager entityManager;

   public final void setClazz( Class< T > clazzToSet ){
      this.clazz = clazzToSet;
   }

   public T findOne( long id ){
      return entityManager.find( clazz, id );
   }
   public List< T > findAll(){
      return entityManager.createQuery( "from " + clazz.getName() )
       .getResultList();
   }

   public void create( T entity ){
      entityManager.persist( entity );
   }

   public T update( T entity ){
      return entityManager.merge( entity );
   }
   public void delete( T entity ){
      entityManager.remove( entity );
   }
   public void deleteById( long entityId ){
      T entity = findOne( entityId );
      delete( entity );
   }
}
```

The main interesting aspect here is the way the EntityManager is injected – using the standard*@PersistenceContext* annotation. Under the hood, this is handled by the*PersistenceAnnotationBeanPostProcessor* – which processes the annotation, retrieves the JPA entity manager from the contains and injects it.

The persistence post processor is either created explicitly by defining it in the configuration, or automatically, by defining *context:annotation-config* or *context:component-scan* in the namespace config.

Also, note that the entity Class is passed in the constructor to be used in the generic operations:

```
@Repository
public class FooDAO extends AbstractJPADAO< Foo > implements IFooDAO{

   public FooDAO(){
      setClazz(Foo.class );
   }
}
```

4. Conclusion

This section illustrated how to set up a DAO layer with Spring and JPA, using both XML and Java based configuration. We also discussed why not to use the JpaTemplate and how to replace it with the EntityManager. The final result is a lightweight, clean DAO implementation, with almost no compile-time reliance on Spring.

CHAPTER 8

Persistence with Spring

8: SIMPLIFY THE DAO WITH SPRING AND JAVA GENERICS

1. Overview

This section will focus on simplifying the DAO layer by using a single, generified Data Access Object for all entities in the system, which will result in elegant data access, with no unnecessary clutter or verbosity.

2. The Hibernate and JPA DAOs

Most production codebases have some kind of DAO layer. Usually the implementation ranges from multiple classes with no abstract base class to some kind of generified class. However, one thing is consistent – there isalways more then one – most likely, there is a one to one relation between the DAOs and the entities in the system.

Also, depending on the level of generics involved, the actual implementations can vary from heavily duplicated code to almost empty, with the bulk of the logic grouped in a base abstract class.

These multiple implementations can usually be replaced by a single parametrized DAO used in such no functionality is lost by taking full advantage of the type safety provided by Java Generics.

Two implementations of this concept are presented next, one for a Hibernate centric persistence layer and the other focusing on JPA. These implementation are by no means complete – only some data access methods are included, but they can be easily be made more thorough.

2.1. The Abstract Hibernate DAO

```
public abstract class AbstractHibernateDao< T extends Serializable > {

  private Class< T > clazz;

  @Autowired
  SessionFactory sessionFactory;

  public final void setClazz( Class< T > clazzToSet ){
    this.clazz = clazzToSet;
  }

  public T findOne( long id ){
    return (T) getCurrentSession().get( clazz, id );
  }
  public List< T > findAll(){
    return getCurrentSession().createQuery( “from “ + clazz.getName() ).list();
  }

  public void create( T entity ){
    getCurrentSession().persist( entity );
  }

  public void update( T entity ){
    getCurrentSession().merge( entity );
  }

  public void delete( T entity ){
    getCurrentSession().delete( entity );
  }
  public void deleteById( long entityId ){
    T entity = findOne( entityId );
    delete( entity );
  }

  protected final Session getCurrentSession(){
    return sessionFactory.getCurrentSession();
  }
}
```

The DAO uses the Hibernate API directly, without relying on any Spring templates (such as *HibernateTemplate*). Using of templates, as well as management of the *SessionFactor*y which is autowired in the DAO were covered in the Hibernate DAO section.

2.2. The Generic Hibernate DAO

Now that the abstract DAO is done, we can implement it just once – the generic DAO implementation will become the only implementation needed:

```
@Repository
@Scope( BeanDefinition.SCOPE_PROTOTYPE )
public class GenericHibernateDao< T extends Serializable >
  extends AbstractHibernateDao< T > implements IGenericDao< T >{
   //
}
```

First, note that the generic implementation is itself parametrized – allowing the client to choose the correct parameter in a case by case basis. This will mean that the clients gets all the benefits of type safety without needing to create multiple artifacts for each entity.

Second, notice the prototype scope of these generic DAO implementation. Using this scope means that the Spring container will create a new instance of the DAO each time it is requested (including on autowiring). That will allow a service to use multiple DAOs with different parameters for different entities, as needed.

The reason this scope is so important is due to the way Spring initializes beans in the container. Leaving the generic DAO without a scope would mean using the default singleton scope, which would lead to a single instance of the DAO living in the container. That would obviously be majorly restrictive for any kind of more complex scenario.

The *IGenericDao* is simply an interface for all the DAO methods, so that we can inject our implementation with Spring in (or in whatever is needed):

```
public interface IGenericDao<T extends Serializable> {

   T findOne(final long id);

   List<T> findAll();

   void create(final T entity);

   T update(final T entity);

   void delete(final T entity);

   void deleteById(final long entityId);
}
```

2.3. The Abstract JPA DAO

```
public abstract class AbstractJpaDao< T extends Serializable > {

   private Class< T > clazz;

   @PersistenceContext
   EntityManager entityManager;

   public void setClazz( Class< T > clazzToSet ){
      this.clazz = clazzToSet;
   }

   public T findOne( Long id ){
      return entityManager.find( clazz, id );
   }
   public List< T > findAll(){
      return entityManager.createQuery( “from “ + clazz.getName() )
       .getResultList();
   }

   public void save( T entity ){
      entityManager.persist( entity );
   }

   public void update( T entity ){
      entityManager.merge( entity );
   }

   public void delete( T entity ){
      entityManager.remove( entity );
   }
   public void deleteById( Long entityId ){
      T entity = getById( entityId );
      delete( entity );
   }
}
```

Similar to the Hibernate DAO implementation, the Java Persistence API is used here directly, again not relying on the now deprecated Spring *JpaTemplate*.

2.4. The Generic JPA DAO

Similar to the the Hibernate implementation, the JPA Data Access Object is straighforward as well:

```
@Repository
@Scope( BeanDefinition.SCOPE_PROTOTYPE )
public class GenericJpaDao< T extends Serializable >
 extends AbstractJpaDao< T > implements IGenericDao< T >{
   //
}
```

3. Injecting this DAO

There is now a single DAO to be injected by Spring; also, the Class needs to be specified:

```
@Service
class FooService implements IFooService{

   IGenericDao< Foo > dao;

   @Autowired
   public void setDao( IGenericDao< Foo > daoToSet ){
      dao = daoToSet;
      dao.setClazz( Foo.class );
   }

   // ...

}
```

Spring autowires the new DAO insteince using setter injection so that the implementation can be customized with the *Class* object. After this point, the DAO is fully parametrized and ready to be used by the service.

There are of course other ways that the class can be specified for the DAO – via reflection, or even in XML. My preference is towards this simpler solution because of the improved readability and transparency compared to using reflection.

4. Conclusion

This section discussed the simplification of the Data Access Layer by providing a single, reusable implementation of a generic DAO. This implementation was presented in both a Hibernate and a JPA based environment. The result is a streamlined persistence layer, with no unnecessary clutter.

CHAPTER 9

Persistence with Spring

9: TRANSACTIONS WITH SPRING 4 AND JPA

1. Overview

This section will discuss the right way to configure Spring Transactions, how to use the @ *Transactionalannotation* and common pitfalls.

For a more in depth discussion on the core persistence configuration, check out the Spring with JPA section.

There are two distinct ways to configure Transactions – annotations and AOP – each with their own advantages – we're going to discuss the more common annotation config here.

2. Configure Transactions without XML

Spring 3.1 introduces the @EnableTransactionManagement annotation to be used in on @ *Configurationclasses* and enable transactional support:

```
@Configuration
@EnableTransactionManagement
public class PersistenceJPAConfig{

  @Bean
  public LocalContainerEntityManagerFactoryBean entityManagerFactoryBean(){
    ...
  }

  @Bean
  public PlatformTransactionManager transactionManager(){
    JpaTransactionManager transactionManager = new JpaTransactionManager();
    transactionManager.setEntityManagerFactory(
     entityManagerFactoryBean().getObject() );
    return transactionManager;
  }
}
```

3. Configure Transactions with XML

Before 3.1 or if Java is not an option, here is the XML configuration, using annotation-driven and the namespace support:

```
<bean id=”txManager” class=”org.springframework.orm.jpa.JpaTransactionManager”>
  <property name=”entityManagerFactory” ref=”myEmf” />
</bean>
<tx:annotation-driven transaction-manager=”txManager” />
```

4. The *@Transactional* Annotation

With transactions configured, a bean can now be annotated with @Transactional either at the class or method level:

```
@Service
@Transactional
public class FooService {
   ...
}
```

The annotation supports further configuration as well:

- the Propagation Type of the transaction
- the Isolation Level of the transaction
- a Timeout for the operation wrapped by the transaction
- a readOnly flag – a hint for the persistence provider that the transaction should be read only
- the Rollback rules for the transaction

Note that – by default, rollback happens for runtime, unchecked exceptions only. Checked exception do not trigger a rollback of the transaction; the behavior can of course be configured with the *rollbackFor andnoRollbackFor* annotation parameters.

5. Potential Pitfalls

5.1. Transactions and Proxies

At a high level, Spring creates proxies for all the classes annotated with @Transactional – either on the class or on any of the methods. The proxy allows the framework to inject transactional logic before and after the method being invoked – mainly for starting and committing the transaction.

What is important to keep in mind is that, if the transactional bean is implementing an interface, by default the proxy will be a Java Dynamic Proxy. This means that only external method calls that come in through the proxy will be intercepted – any self-invocation calls will not start any transaction – even if the method is annotated with @*Transactional*.

Another caveat of using proxies is that only public methods should be annotated with @ Transactional – methods of any other visibilities will simply ignore the annotation silently as these are not proxied.

This section discusses further proxying pitfals in great detail here.

5.2. Changing the Isolation level

You can change transaction isolation level – as follows:

```
@Transactional(isolation = Isolation.SERIALIZABLE)
```

Note that this has actually been introduced in Spring 4.1; if we run the above example before Spring 4.1, it will result in:

org.springframework.transaction.InvalidIsolationLevelException: Standard JPA does not support custom isolation levels – use a special *JpaDialect* for your JPA implementation

5.3. Read Only Transactions

The readOnly flag usually generates confusion, especially when working with JPA; from the javadoc:

Note that this just serves as a hint for the actual transaction subsystem; it will not necessarily cause failure of write access attempts. A transaction manager which cannot interpret the read-only hint will not throw an exception when asked for a read-only transaction.

The fact is that it cannot be guaranteed that an insert or update will not occur when the readOnly flag is set – its behavior is vendor dependent whereas JPA is vendor agnostic.

It is also important to understand that the *readOnly* flag is only relevant inside a transaction; if an operation occurs outside of a transactional context, the flag is simply ignored. A simple example of that would calling a method annotated with:

```
@Transactional( propagation = Propagation.SUPPORTS,readOnly = true )
```

from a non-transactional context – a transaction will not be created and the *readOnly* flag will be ignored.

5.4. Transaction Logging

Transactional related issues can also be better understood by fine-tuning logging in the transactional packages; the relevant package in Spring is *“org.springframework.transaction”, which should be configured with a logging level of TRACE*.

6. Conclusion

We covered the basic configuration of transactional semantics using both java and XML, how to use@ *Transactional* and best practices of a Transactional Strategy. The Spring support for transactional testing as well as some common JPA pitfalls were also discussed.