# <http://www.codingpedia.org/ama/spring-mybatis-integration-example/>

**<https://xebia.com/blog/jpa-implementation-patterns-data-access-objects/>**

# Spring MyBatis integration example

Contents

* [1. Why Mybatis?](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "1_Why_Mybatis)
  + [1.1. What is MyBatis?](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "11_What_is_MyBatis)
* [2. Spring MyBatis interaction](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "2_Spring_MyBatis_interaction)
  + [2.1. What is MyBatis-Spring?](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "21_What_is_MyBatis-Spring)
  + [2.2. Installation](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "22_Installation)
  + [2.3. Spring application context setup](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "23_Spring_application_context_setup)
  + [2.4. Injecting Mappers](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "24_Injecting_Mappers)
    - [2.4.1. Service Layer](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "241_Service_Layer)
    - [2.4.2. DAO layer](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "242_DAO_layer)
      * [2.4.2.1. Register mapper](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "2421_Register_mapper)
        + [2.4.2.1.1. With XML Config](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "24211_With_XML_Config)
  + [2.5. MyBatis configuration XML](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "25_MyBatis_configuration_XML)
    - [2.5.1. typeAliases](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "251_typeAliases)
    - [2.5.2. mappers](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "252_mappers)
  + [2.6. Mapper XML Files](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "26_Mapper_XML_Files)
    - [2.6.1. select](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "261_select)
    - [2.6.2. Result Map](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "262_Result_Map)
      * [2.6.2.1. id & result](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "2621_id_result)
* [3. Summary](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "3_Summary)
* [4. Resources](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "4_Resources)

## 1. Why Mybatis?

Short answer: simple, lightweight, open source, dynamic sql and sql control, previous iBATIS knowledge. Now let me elaborate a little bit on the subject. Back in the old days of Podcastmania.ro, see [Story of Podcastpedia.org](http://www.codingpedia.org/ama/story-of-podcastpedia-org/" \o "Story of Podcastpedia.org" \t "http://www.codingpedia.org/ama/spring-mybatis-integration-example/_blank), I used my own MVC like framwork based on servlets to develop the web application and plain old [JDBC](http://en.wikipedia.org/wiki/Java_Database_Connectivity" \o "Wikipedia - JDBC" \t "http://www.codingpedia.org/ama/spring-mybatis-integration-example/_blank) to access the database. After “upgrading” to Spring MVC, I started using Spring’s JdbcTemplate for database access, which removed some of the boilerplate code. Later I got involved in projects where database access occured via iBATIS – Hibernate was there for a long time, but because of legacy reasons and no database normalization whatsoever, iBATIS was the optimal choice. By about the same time MyBatis had been just launched, so I read the documentation, did a pilot, liked it and switched from Spring’s JdbcTemplate to MyBatis. In the mean time I’ve been working on projects with Hibernate and JPA 2.0 with Hibernate used for persistence, so I’d say I have a pretty good overview on the most popular Java Persistence Frameworks. You have currently four major options:

* JPA/Hibernate
* myBatis (former iBatis)
* Spring JDBC
* JDBC

*IMG_257Source code for this post is available on [Github](https://github.com/Codingpedia/podcastpedia) -****podcastpedia.org****is an open source project.*

Every approach has its pros and cons, but if I had to choose all over again a technology for Podcastpedia.org, I would choose MyBatis. You can find lots of resources on the web, lots of reviews, tutorials, pros and cons for using one technology over the other and so on – do your research and find out what is best persistence solution for your context. I also advise watching the following video, unfortunately the sound is pretty bad, but the content is really interesting :

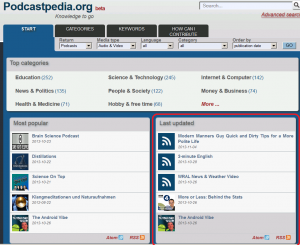
### 1.1. What is MyBatis?

So now that I have made my choice, let’s see first what is MyBatis. Well according to the official website, “MyBatis is a first class persistence framework with support for custom SQL, stored procedures and advanced mappings. MyBatis eliminates almost all of the JDBC code and manual setting of parameters and retrieval of results. MyBatis can use simple XML or Annotations for configuration and map primitives, Map interfaces and Java POJOs (Plain Old Java Objects) to database records.”

Enough with talking, let’s focus now on the main topic of this post, which is to find out how Spring and MyBatis can interact.

## 2. Spring MyBatis interaction

For the sake of simplicity, in this post I will present a simple example, which explains what needs to be implemented and configured to retrieve the newest(recently updated) podcasts from the database via MyBatis with Spring. You can experience the end result “live” by visiting the [homepage of Podcastpedia.org](https://github.com/Codingpedia/podcastpedia/" \o "Last updated podcasts from Podcastpedia.org" \t "http://www.codingpedia.org/ama/spring-mybatis-integration-example/_blank):

[](http://www.codingpedia.org/wp-content/uploads/2013/10/last_updated_podcasts.png)

Print screen home page Podcastpedia.org

If you are familiar with iBATIS (predecessor of MyBatis), you might know that until version 3, the Spring Framework provided direct integration with iBATIS SQL Maps in terms of resource management, DAO implementation support, and transaction strategies. But by the time iBATIS became MyBatis, Spring 3 development was already over, and the Spring team did not want to release with code based on a non-released version of MyBatis, official Spring support would have to wait. Given the interest in Spring support for MyBatis, the MyBatis community decided it was time to reunite the interested contributors and add Spring integration as a community sub-project of MyBatis instead. This is how MyBatis-Spring project was born, which is also used throughout [Podcastpedia.org](https://github.com/Codingpedia/podcastpedia" \o "Podcastpedia.org, knowledge to go" \t "http://www.codingpedia.org/ama/spring-mybatis-integration-example/_blank)

### 2.1. What is MyBatis-Spring?

MyBatis-Spring integrates MyBatis seamlessly with Spring. This library allows MyBatis to participate in Spring transactions, takes care of building MyBatis mappers and SqlSessions and inject them into other beans, translates MyBatis exceptions into Spring DataAccessExceptions, and finally, it lets you build your application **code free** of dependencies on MyBatis, Spring or MyBatis-Spring.

### 2.2. Installation

To use the MyBatis-Spring module, you just need to include the mybatis-spring-x.x.x.jar file and its dependencies in the [classpath](http://en.wikipedia.org/wiki/Classpath_(Java)" \o "Wikipedia classpath java" \t "http://www.codingpedia.org/ama/spring-mybatis-integration-example/_blank).

If you are using Maven just add the following dependency to your pom.xml:

<!-- MyBatis integration -->

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis-spring</artifactId>

<version>1.2.1</version>

</dependency>

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>3.2.3</version>

</dependency>

### 2.3. Spring application context setup

To use MyBatis with Spring you need at least two things defined in the Spring application context: an SqlSessionFactory and at least one mapper interface.

In MyBatis-Spring, an SqlSessionFactoryBean is used to create an SqlSessionFactory. Every MyBatis application centers around an instance of SqlSessionFactory. You use the SqlSessionFactory to create an SqlSession. Once you have a session, you use it to execute your mapped statements, commit or rollback connections and finally, when it is no longer needed, you close the session. With MyBatis-Spring you don’t need to use SqlSessionFactorydirectly because your beans can be injected with a thread safe SqlSession that automatically commits, rollbacks and closes the session based on Spring’s transaction configuration.

To configure the factory bean, put the following in the Spring XML configuration file:

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

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

<property name="configLocation" value="classpath:config/mybatisV3-config.xml"/>

</bean>

Notice that the SqlSessionFactory requires a DataSource. This can be any DataSource and should be configured just like any other Spring database connection. For [Podcastpedia.org](https://github.com/Codingpedia/podcastpedia" \o "Podcastpedia.org, knowledge to go" \t "http://www.codingpedia.org/ama/spring-mybatis-integration-example/_blank) the DataSource is configured via [JNDI](http://en.wikipedia.org/wiki/Java_Naming_and_Directory_Interface" \o "Wikipedia - JNDI" \t "http://www.codingpedia.org/ama/spring-mybatis-integration-example/_blank):

<!-- ========================= DATASOURCE DEFINITION via JNDI ========================= -->

<!-- When resourceRef is true, the value of jndiName will be prepended with

server’s JNDI directory. Consequently, the actual name used will be

java:comp/env/jdbc/pcmDB. -->

<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean" scope="singleton">

<property name="jndiName" value="java:comp/env/jdbc/pcmDB" />

<property name="resourceRef" value="true" />

</bean>

See my post [Tomcat JDBC Connection Pool configuration for production and development](http://www.codingpedia.org/ama/tomcat-jdbc-connection-pool-configuration-for-production-and-development/" \o "Tomcat JDBC Connection Pool configuration for production and development" \t "http://www.codingpedia.org/ama/spring-mybatis-integration-example/_blank), to find out how the database resource is configured for production and development/testing environments.

The location parameter specifies the location of the MyBatis config file. You will see in a coming section, [MyBatis configuration XML,](http://www.codingpedia.org/ama/spring-mybatis-integration-example/" \l "mybatis_config_file) how this file looks like.

### 2.4. Injecting Mappers

Rather than code data access objects (DAOs) manually using SqlSessionDaoSupport or SqlSessionTemplate, Mybatis-Spring can create a thread safe mapper that you can inject directly into other bean. I like to call mappers also DAOs.

#### 2.4.1. Service Layer

I will start at the Service Layer, which lies before the DAO layer. The service layer implementation class – StartPageServiceImpl, which loads the newest podcasts in the model to be displayed on the homepage, will be injected the PodcastDao mapper:

<!-- =================== Service beans configuration ================== -->

<bean id="startPageService" class="org.podcastpedia.service.impl.StartPageServiceImpl">

<property name="podcastDao" ref="podcastDao"/>

</bean>

Once the mapper is injected it can then be used in the business logic:

public class StartPageServiceImpl implements StartPageService {

private static final Integer NUMBER\_OF\_PODCASTS\_IN\_CHART = 5;

private static Logger LOG = Logger.getLogger(StartPageServiceImpl.class);

private PodcastDao podcastDao;

public void setPodcastDao(PodcastDao podcastDao) {

this.podcastDao = podcastDao;

}

@Cacheable(value="newestAndRecommendedPodcasts", key="#root.method.name")

public List<Podcast> getNewestPodcasts() {

List<Podcast> newestPodcasts = podcastDao.getNewestPodcasts(NUMBER\_OF\_PODCASTS\_IN\_CHART);

for(Podcast p : newestPodcasts) {

Episode lastEpisode = episodeDao.getLastEpisodeForPodcast(p.getPodcastId());

p.setLastEpisode(lastEpisode);

}

return newestPodcasts;

}

..............

}

Notice that there are no SqlSession or MyBatis references at the service layer. Nor is there any need to create, open or close the session, MyBatis-Spring will take care of that. We’ll see in the next section how I implemented and configured the PodcastDao.

**Note:** If you want to find out how the @Cacheable annotation is configured, visit my post [Spring caching with Ehcache](http://www.codingpedia.org/ama/spring-caching-with-ehcache/" \o "Codingpedia.org - Spring caching with Ehcache" \t "http://www.codingpedia.org/ama/spring-mybatis-integration-example/_blank).

#### 2.4.2. DAO layer

##### 2.4.2.1. Register mapper

You can register the mapper either using a classical XML configuration or the new 3.0+ Java Config (a.k.a. @Configuration). I prefer the classical way with XML configuration, as I like to hold the configuration as separate as possible from the code:

###### 2.4.2.1.1. With XML Config

The mapper/dao is registered to Spring by including a MapperFactoryBean in the application context XML config:

<!-- =============== MyBATIS beans configuration ================== -->

<bean id="podcastDao" class="org.mybatis.spring.mapper.MapperFactoryBean">

<property name="sqlSessionFactory" ref="sqlSessionFactory"/>

<property name="mapperInterface" value="org.podcastpedia.dao.PodcastDao" />

</bean>

The Mapper BeanFactory can be set up with a SqlSessionFactory, as here, or a pre-configured SqlSessionTemplate. You saw how the SqlSessionFactory was configured in the section Installation/Spring application context setup. You can also download the complete Spring configuration file for the DAO layer.

The second parameter – mapperInterface – sets the mapper interface of the MyBatis mapper. Note that the mapper class specified **must** be an interface, NOT an actual implementation class:

package org.podcastpedia.dao;

import java.util.Date;

import java.util.List;

import java.util.Map;

import org.podcastpedia.domain.Comment;

import org.podcastpedia.domain.Podcast;

import org.podcastpedia.domain.Tag;

/\*\*

\* Interface for database access

\*

\* @author ama

\*

\*/

public interface PodcastDao {

/\*\*

\* Returns the newest podcasts (ORDER BY last\_updated DESC)

\*

\* @param numberOfPodcasts (number of podcasts to be returned)

\* @return

\*/

public List<Podcast> getNewestPodcasts(Integer numberOfPodcasts);

....................

}

If the PodcastDao had a corresponding MyBatis XML mapper file in the same classpath location as the mapper interface, it would have been parsed automatically by the MapperFactoryBean. But because they are different places in the classpath locations, the parameter had to be specified.

Let’s see now how the sql statement is configured and implemented in MyBatis.

### 2.5. MyBatis configuration XML

As mentioned in the beginning of the post the SqlSessionFactoryBean has the configLocation parameter that defines where the MyBatis configuration resides. Here is an extract from the configuration file that is relevant for the example presented here:

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE configuration

PUBLIC "-//[mybatis.org//DTD](http://mybatis.org//DTD) Config 3.0//EN"

"<http://mybatis.org/dtd/mybatis-3-config.dtd>">

<configuration>

<typeAliases>

<typeAlias type="org.podcastpedia.domain.Podcast" alias="Podcast"/>

.......

</typeAliases>

<mappers>

<mapper resource="maps/PodcastMapper.xml" />

........

</mappers>

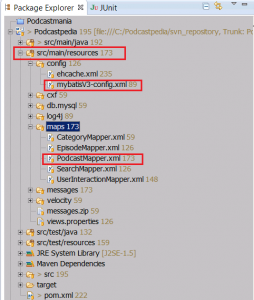
</configuration>

#### 2.5.1. typeAliases

A type alias is simply a shorter name for a Java type. It’s only relevant to the XML configuration and simply exists to reduce redundant typing of fully qualified classnames which include package names. As you’ll see in the following section, when referencing a org.podcastpedia.domain.Podcast object, I will just use “Podcast”.

#### 2.5.2. mappers

Configuration elements which tell MyBatis where to find the mappers. Java doesn’t really provide any good means of auto-discovery in this regard, so the best way to do it is to simply tell MyBatis where to find the mapping files. You can use class path relative resource references, or literal, fully qualified url references (including file:/// URLs):

[](http://www.codingpedia.org/wp-content/uploads/2013/10/MyBatis-files.png)

MyBatis configuration file and mappers location

**Note:** There are many other parameters you can set up in the configuration file for MyBatis. See the [Configuration](http://mybatis.github.io/mybatis-3/configuration.html" \o "MyBatis configuration" \t "http://www.codingpedia.org/ama/spring-mybatis-integration-example/_blank) documentation page for more details.

### 2.6. Mapper XML Files

The true power of MyBatis is in the Mapped Statements. This is where the magic happens. For all of their power, the Mapper XML files are relatively simple. Certainly if you were to compare them to the equivalent JDBC code, you would immediately see a savings of 95% of the code. MyBatis was built to focus on the SQL, and does its best to stay out of your way.

Here I will present a snippet from the PodcastMapper.xml that shows how the PodcastDao interface method is mapped:

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE mapper PUBLIC "-//[mybatis.org//DTD](http://mybatis.org//DTD) Mapper 3.0//EN"

"<http://mybatis.org/dtd/mybatis-3-mapper.dtd>">

<mapper namespace="org.podcastpedia.dao.PodcastDao">

<!-- result maps -->

<resultMap id="podcastsMap" type="Podcast" >

<id column="podcast\_id" property="podcastId" />

<result column="url" property="url" />

<result column="rating" property="rating" />

<result column="numberRatings" property="number\_ratings" />

<result column="number\_visitors" property="numberOfVisitors" />

<result column="DESCRIPTION" property="description" />

<result column="PODCAST\_IMAGE\_URL" property="urlOfImageToDisplay" />

<result column="TITLE" property="title" />

<result column="last\_episode\_url" property="lastEpisodeMediaUrl" />

<result column="title\_in\_url" property="titleInUrl" />

<result column="publication\_date" property="publicationDate"/>

</resultMap>

<select id="getNewestPodcasts" resultMap="podcastsMap" parameterType="Integer">

SELECT

PODCAST\_ID,

URL,

NUMBER\_VISITORS,

DESCRIPTION,

PODCAST\_IMAGE\_URL,

TITLE,

last\_episode\_url,

title\_in\_url,

publication\_date

FROM

podcasts

WHERE

availability=200

ORDER BY publication\_date DESC

limit 0, #{value};

</select>

.....

</mapper>

Notice the mapper’s namespace value at line 5 – org.podcastpedia.dao.PodcastDao and the id of the <select> element at line 21. If you combine, it results org.podcastpedia.dao.PodcastDao.getNewestPodcasts which is exactly the method of the interface defined at the DAO layer.

#### 2.6.1. select

The select statement is one of the most popular elements that you’ll use in MyBatis. Putting data in a database isn’t terribly valuable until you get it back out, so most applications query far more than they modify the data. For every insert, update or delete, there is probably many selects. This is one of the founding principles of MyBatis, and is the reason so much focus and effort was placed on querying and result mapping.

The statement exposed here is called getNewestPodcasts (same as corresponding method in the PodcastDao interface), takes a parameter of type Integer(same as input of the corresponding method in PodcastDao interface) and has a resultMap parameter of value "podcastsMap", which is a named reference to an external resultMap.

#### 2.6.2. Result Map

The resultMap element is the most important and powerful element in MyBatis. It’s what allows you to do away with 90% of the code that JDBC requires to retrieve data from ResultSets, and in some cases allows you to do things that JDBC does not even support. In fact, to write the equivalent code for something like a join mapping for a complex statement could probably span thousands of lines of code. The design of the ResultMaps is such that simple statements don’t require explicit result mappings at all, and more complex statements require no more than is absolutely necessary to describe the relationships.

Let’s talk a little about the structure of the resultMap. The top element has an id(=”podcastMap”) which uniquely identifies the select in the mapper (namespace), and is referenced in the **select**, and a type(=”Podcast”), which is a fully qualified Java class name, or a type alias (this is the case here, "Podcast" being the type alias defined in the MyBatis configuration file)

##### 2.6.2.1. id & result

<id column="podcast\_id" property="podcastId" />

<result column="url" property="url" />

These are the most basic of result mappings. Both id, and column map a single column value to a single property or field of a simple data type (String, int, double, Date, etc.).

The only difference between the two is that id will flag the result as an identifier property to be used when comparing object instances. This helps to improve general performance, but especially performance of caching and nested result mapping (i.e. join mapping).

What happens in this map is that some of the columns of the PODCASTS table are mapped to the properties of the Podcast bean:

* the column attribute represents the column name from the database, or the aliased column label – this is the same string that would normally be passed to resultSet.getString(columnName).

· the property attribute represents the field or property to map the column result to. If a matching JavaBeans property exists for the given name, then that will be used. Otherwise, MyBatis will look for a field of the given name. In both cases you can use complex property navigation using the usual dot notation. For example, you can map to something simple like: username, or to something more complicated like: address.street.number.

Believe it or not, that’s it. It might seem like a lot just do a simple SELECT, but once you learn the basics it becomes really simple and you get some powerful stuff, some of which I will present in a future post. And also is important to mention that the learning curve is pretty small.

## 3. Summary

Well, that’s it. You’ve learned about MyBatis, how it integrates with Spring via MyBatis-Spring, how to configure MyBatis in Spring’s application context and how to use it in a Spring application.

*IMG_260Source code for this post is available on [Github](https://github.com/Codingpedia/podcastpedia) -****podcastpedia.org****is an open source project.*

## 4. Resources

1. [MyBatis](http://mybatis.github.io/mybatis-3/" \o "MyBatis" \t "http://www.codingpedia.org/ama/spring-mybatis-integration-example/_blank)
2. [MyBatis-Spring](http://mybatis.github.io/spring/index.html" \o "MyBatis-Spring" \t "http://www.codingpedia.org/ama/spring-mybatis-integration-example/_blank)
3. [Spring Data Access with JDBC](http://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/jdbc.html" \o "Data access with JDBC" \t "http://www.codingpedia.org/ama/spring-mybatis-integration-example/_blank)
4. [MyBatis-Blog](http://blog.mybatis.org/" \o "MyBatis blog" \t "http://www.codingpedia.org/ama/spring-mybatis-integration-example/_blank)

# **[JPA implementation patterns: Data Access Objects](https://xebia.com/blog/jpa-implementation-patterns-data-access-objects/)**

by [Vincent Partington](https://xebia.com/blog/author/vpartington/)

IMG_256March 9th, 2009

IMG_257[65 comments](https://xebia.com/blog/jpa-implementation-patterns-data-access-objects/" \l "comments)

The [JPA](http://java.sun.com/developer/technicalArticles/J2EE/jpa/), short for Java Persistence API, is part of the Java EE 5 specification and has been implemented by [Hibernate](http://www.hibernate.org/397.html), [TopLink](http://www.oracle.com/technology/products/ias/toplink/jpa/index.html), [EclipseLink](http://www.eclipse.org/eclipselink/), [OpenJPA](http://openjpa.apache.org/), and a number of other [object-relational mapping](http://en.wikipedia.org/wiki/Object-relational_mapping) (ORM) frameworks. Because JPA was originally designed as part of the [EJB 3.0 specification](http://java.sun.com/products/ejb/docs.html), you can use it within an EJB 3.0 application. But it works equally well outside of EJB 3.0, for example [in a Spring application](http://static.springframework.org/spring/docs/2.5.x/reference/orm.html" \l "orm-jpa). And when even Gavin King, the designer of Hibernate, recommends using JPA in the second edition of Hibernate in Action, a.k.a. [Java Persistence with Hibernate](http://www.manning.com/bauer2/), it’s obvious that JPA is here to stay.

Once you [get over your fear of annotations](https://xebia.com/blog/2009/01/19/maybe-annotations-arent-that-bad-after-all/) ;-), you find that there is plenty of literature out there that explains the objects and methods within the API, the way these objects work together and how you can expect them to be implemented. And when you stick to hello-world-style programs, it all seems pretty straight forward. But when you start writing your first ****real****application, you find that things are not so simple. The abstraction provided by JPA is pretty[leaky](http://www.joelonsoftware.com/articles/LeakyAbstractions.html) and has ramifications for larger parts of your application than just your [Data Access Objects](http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html) (DAO’s) and your domain objects. You need to make decisions on how to handle transactions, lazy loading, detached object (think web frameworks), inheritance, and more. And it turns out that the books and the articles don’t really help you here.  
  
At least, that is what I discovered when I really started using JPA for the first time. In the coming weeks, I would like to discuss the choices I came up against and the decisions I made and why I made them. When I’m done, we’ll have a number of what I would like to not-too-modestly call JPA implementation patterns.

### **Do we really need a DAO?**

So, let’s start with the thing you would probably write first in your JPA application: the data access object (DAO). An interesting point to tackle before we even start is [whether you even need a DAO when using JPA](http://www.infoq.com/news/2007/09/jpa-dao). The conclusion of that discussion more than a year ago was “It depends” and while it is very hard to argue with such a conclusion :-), I would like to stick with the idea that a DAO does have its place in a JPA application. Arguably it provides only a thin layer on top of JPA, but more importantly making a DAO per entity type gives you these advantages:

* Instead of having to pick the right [EntityManager](http://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html) method every time you want to store or load data, you decide which one to use once and you and your whole team can easily stick to that choice.
* You can disallow certain operations for certain entity types. For example, you might never want your code to remove log entries. When using DAO’s, you just do not add a remove method to your LogEntry DAO.
* Theoretically, by using DAO’s you could switch to another persistence system (like plain JDBC or iBATIS). But because JPA is such a leaky abstraction I think that that is not realistically possible for even a slightly complex application. You do get a single point of entry where you can add tracing features or keep performance statistics.
* You can centralize all the queries on a certain entity type instead of scattering them through your code. You could use [named queries](http://java.sun.com/javaee/5/docs/api/javax/persistence/NamedQuery.html) to keep the queries with the entity type, but you’d still need some central place where the right parameters are set. Putting both the query, the code that sets the parameters, and the cast to the correct return type in the DAO seems a simpler thing to do. For example:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public List<ChangePlan> findExecutingChangePlans() {        Query query = entityManager.createQuery(              "SELECT plan FROM ChangePlan plan where plan.state = 'EXECUTING'");        return (List<ChangePlan>) query.getResultList();  } |

So when you decide you are going to use DAO’s, how do you go about writing them? The highlighted (in bold) comment in the Javadoc for Spring’s [JpaTemplate](http://static.springframework.org/spring/docs/2.5.x/api/org/springframework/orm/jpa/JpaTemplate.html) seems to suggest that there’s not much point in using that particular class, which also makes [JpaDaoSupport](http://static.springframework.org/spring/docs/2.5.x/api/org/springframework/orm/jpa/support/JpaDaoSupport.html)superfluous. Instead you can write your JPA DAO as a POJO using the [@PersistenceContext](http://java.sun.com/javaee/5/docs/api/javax/persistence/PersistenceContext.html)annotation to get an EntityManager reference. It will work in an EJB 3.0 container and it will work in Spring 2.0 and up if you add the [PersistenceAnnotationBeanPostProcessor](http://static.springframework.org/spring/docs/2.5.x/api/org/springframework/orm/jpa/support/PersistenceAnnotationBeanPostProcessor.html) bean to your Spring context.

### **The type-safe generic DAO pattern**

Because each DAO shares a lot of functionality with the other DAO’s, it makes sense to have a base class with the shared functionality and then subclass from that for each specific DAO. There are [a](http://www.ibm.com/developerworks/java/library/j-genericdao.html) [lot](http://www.greggbolinger.com/blog/2008/04/17/1208457000000.html) [of](http://forum.springsource.org/showthread.php?t=66139) [blogs](https://xebia.com/blog/2009/02/07/acessing-generic-types-at-runtime-in-java/) out there about such a type-safe generic DAO pattern and you can even download [some code from Google Code](http://code.google.com/p/generic-dao/). When we combine elements from all these sources, we get the following JPA implementation pattern for DAO’s.

### **The entity class**

Let’s say we want to persist the following Order class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | *@Entity*  *@Table*(name = "ORDERS")  public class Order {  *@Id*  *@GeneratedValue*  private int id;  private String customerName;  private Date date;    public int getId() { return id; }  public void setId(int id) { this.id = id; }    public String getCustomerName() { return customerName; }  public void setCustomerName(String customerName) { this.customerName = customerName; }    public Date getDate() { return date; }  public void setDate(Date date) { this.date = date;}  } |

Don’t worry too much about the details of this class. We will revisit the specifics in other JPA implementation patterns. The @Table annotation is there because ORDER is a reserved keyword in SQL.

### **The DAO interfaces**

First we define a generic DAO interface with the methods we’d like all DAO’s to share:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public interface Dao<K, E> {        void persist(E entity);        void remove(E entity);        E findById(K id);  } |

The first type parameter, K, is the type to use as the key and the second type parameter, E, is the type of the entity. Next to the basic persist, remove, and findById methods, you might also like to add a List findAll() method. But like the entity class itself, we will revisit the DAO methods in later JPA implementation patterns.

Then we define one subinterface for each entity type we want to persist, adding any entity specific methods we want. For example, if we’d like to be able to query all orders that have been added since a certain date, we can add such a method:

|  |  |
| --- | --- |
| 1  2  3 | public interface OrderDao extends Dao<Integer, Order> {  List<Order> findOrdersSubmittedSince(Date date);  } |

### **The base DAO implementation**

The third step is to create a base JPA DAO implementation. It will have basic implementation of all the methods in the standard Dao interface we created in step 1:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public abstract class JpaDao<K, E> implements Dao<K, E> {  protected Class<E> entityClass;    *@PersistenceContext*  protected EntityManager entityManager;    public JpaDao() {  ParameterizedType genericSuperclass = (ParameterizedType) getClass().getGenericSuperclass();  this.entityClass = (Class<E>) genericSuperclass.getActualTypeArguments()[1];  }    public void persist(E entity) { entityManager.persist(entity); }    public void remove(E entity) { entityManager.remove(entity); }    public E findById(K id) { return entityManager.find(entityClass, id); }  } |

Most of the implementation is pretty straight forward. Some points to note though:

* The constructor of the JpaDao includes [the method proposed by my colleague Arjan Blokzijl](https://xebia.com/blog/2009/02/07/acessing-generic-types-at-runtime-in-java/) to use reflection to get the entity class.
* The @PersistenceContext annotation causes the EJB 3.0 container or Spring to inject the entity manager.
* The entityManager and entityClass fields are protected so that subclasses, i.e. specific DAO implementations, can access them.

### **The specific DAO implementation**

And finally we create such a specific DAO implementation. It extends the basic JPA DAO class and implements the specific DAO interface:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class JpaOrderDao extends JpaDao<Integer, Order> implements OrderDao {  public List<Order> findOrdersSubmittedSince(Date date) {  Query q = entityManager.createQuery(  "SELECT e FROM " + entityClass.getName() + " e WHERE date >= :date\_since");  q.setParameter("date\_since", date);  return (List<Order>) q.getResultList();  }  } |

### **Using the DAO**

How you get a reference to an instance of your OrderDao depends upon whether we use EJB 3.0 or Spring. In EJB 3.0 we’d use a annotation like this:

|  |  |
| --- | --- |
| 1  2 | *@EJB*(name="orderDao")  private OrderDao orderDao; |

while in Spring we can use the XML bean files or we can use autowiring like this:

|  |  |
| --- | --- |
| 1  2 | *@Autowired*  public OrderDao orderDao; |

In any case, once we have a reference to the DAO we can use it like this:

|  |  |
| --- | --- |
| 1  2  3  4 | Order o = new Order();  o.setCustomerName("Peter Johnson");  o.setDate(new Date());  orderDao.persist(o); |

But we can also use the entity specific query we added to the OrderDao interface:

|  |  |
| --- | --- |
| 1  2  3  4 | List<Order> orders = orderDao.findOrdersSubmittedSince(date);  for (Order each : orders) {  System.out.println("order id = " + each.getId());  } |

With this type-safe DAO pattern we get the following advantages:

* No direct dependency on the JPA api from client code.
* Type-safety through the use of generics. Any casts that still need to be done are handled in the DAO implementation.
* One logical place to group all entity-specific JPA code.
* One location to add transaction markers, debugging, profiling, etc. Although as we will see later, we will need to add transaction markers in other parts of our applications too.
* One class to test when testing the database access code. We will revisit this subject in a later JPA implementation pattern.

I hope this convinces you that you do need DAO’s with JPA.

And that wraps up the first JPA implementation pattern. In the next blog of this series we will build on this example to discuss the next pattern. In the meantime I would love to hear from you how you write your DAO’s!