



Codingpedia.org
SHARING CODING KNOWLEDGE



Spring MyBatis integration example

🕒 2013/10/30

📁 java, spring

🔗 framework, hibernate, iBatis, integration, MyBatis, persistence

👤 Adrian Matei

Like

4

G+1

7

Contents [hide]

1. Why Mybatis?
 - 1.1. What is MyBatis?
2. Spring MyBatis interaction
 - 2.1. What is MyBatis-Spring?
 - 2.2. Installation
 - 2.3. Spring application context setup
 - 2.4. Injecting Mappers
 - 2.4.1. Service Layer
 - 2.4.2. DAO layer
 - 2.4.2.1. Register mapper
 - 2.4.2.1.1. With XML Config
 - 2.5. MyBatis configuration XML
 - 2.5.1. typeAliases
 - 2.5.2. mappers
 - 2.6. Mapper XML Files
 - 2.6.1. select
 - 2.6.2. Result Map
 - 2.6.2.1. id & result
3. Summary
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](#), I used my own MVC like framwork based on servlets to develop the web application and plain old **JDBC** 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 occurred 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



Source code for this post is available on [Github](#) - **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 :



15 May 2012 : Rethinking Persistence: JPA/Hibernate & myBatis Compared

from [Nimret Sandhu](#) PLUS

1:46:19 |

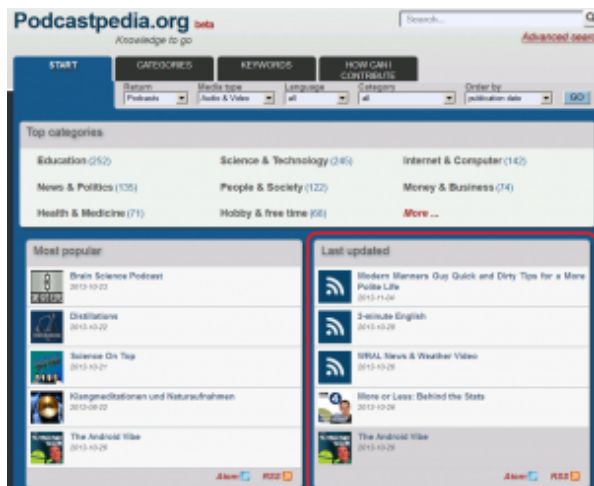
1.1. What is MyBatis?

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](#):



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

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 `SqlSession`s and inject them into other beans, translates MyBatis exceptions into Spring `DataAccessException`s, 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`.

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

```
1 <!-- MyBatis integration -->
2 <dependency>
3   <groupId>org.mybatis</groupId>
4   <artifactId>mybatis-spring</artifactId>
5   <version>1.2.1</version>
6 </dependency>
7 <dependency>
8   <groupId>org.mybatis</groupId>
9   <artifactId>mybatis</artifactId>
10  <version>3.2.3</version>
11 </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 **SqlSessionFactory** directly 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:

```
1 <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
2   <property name="dataSource" ref="dataSource" />
3   <property name="configLocation" value="classpath:config/mybatisV3-config.xml" />
4 </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 the **DataSource** is configured via **JNDI**:

```
1 <!-- ===== DATASOURCE DEFINITION via JNDI ===== -->
2 <!-- When resourceRef is true, the value of jndiName will be prepended with
3 server's JNDI directory. Consequently, the actual name used will be
4 java:comp/env/jdbc/pcmDB. -->
5 <bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean" scope="singleton">
6   <property name="jndiName" value="java:comp/env/jdbc/pcmDB" />
7   <property name="resourceRef" value="true" />
8 </bean>
```

See my post [Tomcat JDBC Connection Pool configuration for production and development](#), 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](#), 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:

```
1 <!-- ===== Service beans configuration ===== -->
2 <bean id="startPageService" class="org.podcastpedia.service.impl.StartPageServiceImpl">
3     <property name="podcastDao" ref="podcastDao"/>
4 </bean>
```

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

```
1 public class StartPageServiceImpl implements StartPageService {
2
3     private static final Integer NUMBER_OF_PODCASTS_IN_CHART = 5;
4     private static Logger LOG = Logger.getLogger(StartPageServiceImpl.class);
5     private PodcastDao podcastDao;
6
7     public void setPodcastDao(PodcastDao podcastDao) {
8         this.podcastDao = podcastDao;
9     }
10
11     @Cacheable(value="newestAndRecommendedPodcasts", key="#root.method.name")
12     public List<Podcast> getNewestPodcasts() {
13         List<Podcast> newestPodcasts = podcastDao.getNewestPodcasts(NUMBER_OF_PODCASTS_IN_C
14         for(Podcast p : newestPodcasts) {
15             Episode lastEpisode = episodeDao.getLastEpisodeForPodcast(p.getPodcastId());
16             p.setLastEpisode(lastEpisode);
17         }
18         return newestPodcasts;
19     }
20 }
21 .....
22 }
```

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](#).

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:

```

1 <!-- ===== MyBatis beans configuration ===== -->
2 <bean id="podcastDao" class="org.mybatis.spring.mapper.MapperFactoryBean">
3     <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
4     <property name="mapperInterface" value="org.podcastpedia.dao.PodcastDao" />
5 </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:

```

1 package org.podcastpedia.dao;
2
3 import java.util.Date;
4 import java.util.List;
5 import java.util.Map;
6
7 import org.podcastpedia.domain.Comment;
8 import org.podcastpedia.domain.Podcast;
9 import org.podcastpedia.domain.Tag;
10
11 /**
12  * Interface for database access
13  *
14  * @author ama
15  *
16  */
17 public interface PodcastDao {
18
19     /**
20      * Returns the newest podcasts (ORDER BY last_updated DESC)
21      *
22      * @param numberOfPodcasts (number of podcasts to be returned)
23      * @return
24      */
25     public List<Podcast> getNewestPodcasts(Integer numberOfPodcasts);
26
27     .....
28 }

```

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:

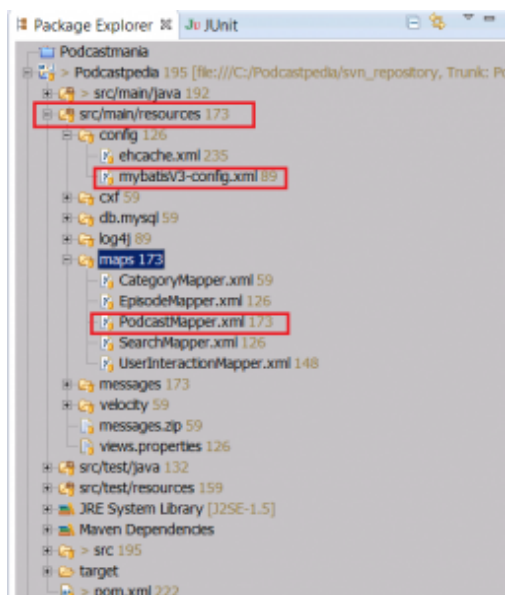
```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration
3     PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6 <configuration>
7     <typeAliases>
8         <typeAlias type="org.podcastpedia.domain.Podcast" alias="Podcast"/>
9         .....
10    </typeAliases>
11    <mappers>
12        <mapper resource="maps/PodcastMapper.xml" />
13        .....
14    </mappers>
15 </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):



*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](#) 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:

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
4
5  <mapper namespace="org.podcastpedia.dao.PodcastDao">
6      <!-- result maps -->
7      <resultMap id="podcastsMap" type="Podcast" >
8          <id column="podcast_id" property="podcastId" />
9          <result column="url" property="url" />
10         <result column="rating" property="rating" />
11         <result column="numberRatings" property="number_ratings" />
12         <result column="number_visitors" property="numberOfVisitors" />
13         <result column="DESCRIPTION" property="description" />
14         <result column="PODCAST_IMAGE_URL" property="urlOfImageToDisplay" />
15         <result column="TITLE" property="title" />
16         <result column="last_episode_url" property="lastEpisodeMediaUrl" />
17         <result column="title_in_url" property="titleInUrl" />
18         <result column="publication_date" property="publicationDate"/>
19     </resultMap>
20
21     <select id="getNewestPodcasts" resultMap="podcastsMap" parameterType="Integer">
22         SELECT
23             PODCAST_ID,
24             URL,
25             NUMBER_VISITORS,
26             DESCRIPTION,
27             PODCAST_IMAGE_URL,
28             TITLE,
29             last_episode_url,
30             title_in_url,
31             publication_date
32         FROM
33             podcasts
34         WHERE
35             availability=200
36         ORDER BY publication_date DESC
37         limit 0, #{value};
38     </select>
39     ....
40 </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 **ResultSet**s, 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 **ResultMap**s 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

```
1 <id column="podcast_id" property="podcastId" />
2 <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.



Source code for this post is available on [Github](#) - **podcastpedia.org** is an open source project.

4. Resources

1. [MyBatis](#)
2. [MyBatis-Spring](#)
3. [Spring Data Access with JDBC](#)
4. [MyBatis-Blog](#)




Adrian Matei

Creator of [Podcastpedia.org](#) and [Codingpedia.org](#), computer science engineer, husband, father, curious and passionate about science, computers, software, education, economics, social equity, philosophy - but these are just outside labels and not that important, deep inside we are all just consciousness, right?




Adrian's favorite Spring and Java books



[Spring Recipes](#)
Gary Mak, Daniel R...
[Best Price \\$1.48](#)
or Buy New [\\$32.33](#)
[Buy from amazon.com](#)

Privacy Information



[Spring in Action](#)
Craig Walls
[Best Price \\$7.58](#)
or Buy New
[Buy from amazon.com](#)

Privacy Information



[Effective Java](#)
Joshua Bloch
[Best Price \\$17.01](#)
or Buy New [\\$32.59](#)
[Buy from amazon.com](#)

Privacy Information




[Click here](#)
Privacy Information

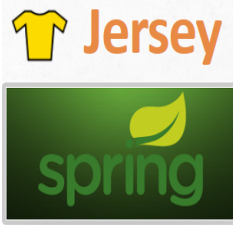
Related Posts:



Java Persistence
Example with



Autocomplete search box with jQuery and



RESTful Web Services Example in



How to setup multiple data

About Adrian Matei

Creator of Podcastpedia.org and Codingpedia.org, computer science engineer, husband, father, curious and passionate about science, computers, software, education, economics, social equity, philosophy. [View all posts by Adrian Matei](#) →

3 Comments

Codingpedia

 Login ▾ Recommend Share

Sort by Best ▾



Join the discussion...

**Abdul** • 19 days ago

This is really useful as I am using iBATIS in my project.

 |  • Reply • Share >**guoy** • a year agowhy not use `org.mybatis.spring.mapper.MapperScannerConfigurer` |  • Reply • Share >**André Bedregal** • 2 years ago

Thank you, Adrian.

Your post was very useful to me.

 |  • Reply • Share > Subscribe Add Disqus to your site Add Disqus Add Privacy

Sponsored

Looking For a Job? Don't Say These 8 Words

The Naked CEO

Find out why more than 20 million people can't stop playing this game

Soldiers: Free Online Game

New Golf Wedge is Changing the Game for Everyone

xE1 Golf

These Lawnmower Fails Are Totally Ridiculous!

The Viral Lane

15 Most INSANE Pictures Of The Amazon

TravelTips4Life

20 Facts You Never Knew About Antarctica

BestPictureBlog